Automatic generation of adaptive codes

Maxime Schmitt

Équipes
ICube **ICPS** | IRMA **MOCO**
Inria **CAMUS** et **TONUS**

Université de Strasbourg

Laboratoire d'Excellence **IRMIA**

École Doctorale Mathématiques, Sciences de l'information et de l'Ingénieur

Laboratoire des sciences de l'Ingénieur, de l'Informatique et de l'Imagerie
Institut de Recherche Mathématique Avancée de Strasbourg

THÈSE présentée par :

Maxime Schmitt

Soutenue le : **30 septembre 2019**

Pour obtenir le grade de : **Docteur de l'université de Strasbourg**
Discipline / Spécialité : **Informatique**

Génération automatique de codes adaptatifs

Automatic Generation of Adaptive Codes

THÈSE dirigée par :

M. Cédric Bastoul	Professeur à l'université de Strasbourg, France
M. Philippe Helluy	Professeur à l'université de Strasbourg, France

RAPPORTEURS :

M. Denis Barthou	Professeur à l'université de Bordeaux INP, France
M. Fabrice Rastello	Directeur de recherche chez Inria Grenoble, France

---

EXAMINATEURS :

M. Albert Cohen	Senior Researcher chez Google Paris, France
Mme Fabienne Jézéquel	Maître de conférences à l'université de Panthéon-Assas, France

Contents

List of Figures viii

List of Listings ix

List of Algorithms and Grammars xi

Remerciements xiii

Abstract xv

1 Introduction 1
- 1.1 Programming Challenges . . . 2
- 1.2 Approximate Computing . . . 4
- 1.3 Contributions . . . 5

2 Scientific Foundation and State of the Art 9
- 2.1 Numerical Analysis — Solving a System of Equations With a Computer 9
  - 2.1.1 From Maxwell's PDE to Update Equations . . . 9
  - 2.1.2 From Update Equations to Simulation Code . . . 13
  - 2.1.3 Program Optimization and Adaptive Techniques . . . 14
  - 2.1.4 Generating an Adaptive Version . . . 16
- 2.2 Approximation Techniques as Program Optimization . . . 19
  - 2.2.1 Approximate Computing Strategies . . . 19
  - 2.2.2 Characterizing Approximation Quality . . . 23
- 2.3 The Polyhedral Model . . . 23
  - 2.3.1 Iteration Domain . . . 24
  - 2.3.2 Scheduling Function . . . 25
  - 2.3.3 Optimization in the Polyhedral Realm . . . 25
  - 2.3.4 From Polyhedral Representation to Imperative Code . . . 26

3 Language Abstractions for Approximate Computing 29
- 3.1 Extending Languages to Support Approximation . . . 29
- 3.2 Static Dynamic Adaptive Strategy . . . 31
- 3.3 Programming Language Annotations for Approximate Computing . . . 33
  - 3.3.1 Annotation for Adaptation Granularity . . . 35
  - 3.3.2 Annotation for Code Transformation . . . 35
  - 3.3.3 Annotations for Adaptive Decision Making . . . 37

3.4 Adaptive Stencil Approximation . . . 41
3.4.1 Annotation for Stencil Computation . . . 41

4 Adaptive Code Generation 43
4.1 Raising Annotated Kernels . . . 43
4.2 Generating Alternatives . . . 45
4.3 Compile Time Adaptive Program Generation . . . 47
4.4 Just-In-Time Adaptive Program Generation . . . 51
4.5 Compilation With Polyhedral Optimizations . . . 56

5 Automatic Adaptive Stencil 59
5.1 Generating Stencil Alternatives . . . 59
5.2 Ordering the Stencil Alternatives . . . 63
5.3 Monitored Data Discovery . . . 66
5.4 Granularity Selection . . . 68
5.5 Discussion on Output Deviation . . . 70

6 Automatic Data Compression 73
6.1 Data Compression and Multiscale Information . . . 75
6.1.1 The Wavelet Basis . . . 75
6.1.2 Wavelet Construction Properties . . . 75
6.2 Using the Wavelet Transform for Data Compression . . . 77
6.3 Evaluation . . . 78

7 Benchmarks and Evaluation 81
7.1 Using Adaptive Code Refinement . . . 81
7.1.1 *LetItBench* Benchmark Set and Their ACR Annotations . . . 81
7.1.2 Eulerian Fluid Simulation . . . 82
7.1.3 Heat Equation . . . 85
7.1.4 Finite-Difference Time-Domain (FDTD) . . . 86
7.1.5 Game of Life . . . 88
7.1.6 K-Means Clustering . . . 91
7.1.7 Overall Performance and Overhead . . . 94
7.2 Using Adaptive Stencil Approximation . . . 95
7.2.1 Heat Solver . . . 95
7.2.2 Eulerian Fluid Simulation . . . 97
7.2.3 Game of Life . . . 98
7.2.4 Finite Difference Time Domain (FDTD) . . . 99
7.2.5 K-Means Clustering . . . 100
7.2.6 Performance Evaluation . . . 101

8 Conclusion and Perspectives 103

Bibliography 107

A Résumé en français **119**
A.1 Défis de programmation 120
A.2 Calcul approché 122
A.3 Contributions 124
A.4 Perspectives 126

List of Figures

2.1 Yee FDTD Algorithm . . . 11
2.2 FDTD Work-Sharing Example . . . 15
2.3 Magnetic Field Intensity Propagation Example . . . 17
2.4 Magnetic Field's Adaptive Grid . . . 18
2.5 Polyhedral Model Representations of Codes . . . 24
2.6 Control Overhead from a Polyhedral Code Generator . . . 28

3.1 Eulerian Fluid Simulation . . . 33
3.2 ACR Framework Compilation Process . . . 34
3.3 ACR-Defined Alternative Overview . . . 36
3.4 Data Domain and Iteration Domain . . . 39
3.5 Loop Dependencies and Valid Schedules . . . 40
3.6 Two-dimensional von Neumann stencil halo . . . 42

4.1 Compile Time Adaptive Code Generation . . . 49
4.2 From the Polyhedral Representation to Code . . . 51
4.3 Simulation Data to Alternative Iteration Domain . . . 52
4.4 "Original" Code to Adaptive "Guarded," "Compile ACR" and "JIT ACR" . . . 53
4.5 Adaptive Code Refinement JIT . . . 54
4.6 ACR Raw, Versioning and Stencil Approximation Level Computation . . . 55
4.7 Skewing to Enable Tiling . . . 57

5.1 Stencil Narrowing Strategies . . . 62
5.2 Stencil Adaptive Grading . . . 66
5.3 Grid Size Effect on Execution Time . . . 69
5.4 Stencil Granularity Empirical Selection . . . 70

6.1 Kármán vortex street . . . 74
6.2 Fourier, Gabor and Wavelet transformation time-frequency correlation . . . 75
6.3 Discrete Wavelet Transform . . . 77
6.4 Wavelet Transtorm Tree Representation . . . 78

7.1 LetItBench Generated Plot . . . 83
7.2 Heat Solver Simulation . . . 87
7.3 Game of Life Clock Automaton . . . 90
7.4 K-Means Image Segmentation . . . 92
7.5 Grid Size Influence on Heat Solver . . . 97
7.6 Grid Size Influence on Eulerian Fluid Simulation . . . 98

7.7 Grid Size Influence on Game of Life 99
7.8 Grid Size Influence on FDTD 100
7.9 Grid Size Influence on K-Means 101

List of Listings

2.1 Profiling a Program on Linux with `perf` 16
2.2 ACR Annotations Example 18
2.3 Output from a Polyhedral Code Generator 26

3.1 Parallel Reduction Using OpenMP Annotations 31
3.2 Parallel Reduction Using a Thread Spawning Library 31
3.3 Fluid Simulation Kernel 32

6.1 Example of Automatic Compression Annotation 77

7.1 Fluid Simulation Kernel 85
7.2 Heat Equation Solver Kernel 87
7.3 FDTD Kernel 89
7.4 Game of Life Kernel 91
7.5 K-Means Kernel 93

List of Algorithms and Grammars

1 Adaptive Code Refinement Grammar 38
2 Adaptive Stencil Approximation Grammar 42

3 Mark Statements With its Alternatives 44
4 Alternative application algorithm 46
5 Compile Time Adaptive Code Generation Algorithm 50

6 Example 2D stencil algorithm 60
7 Search for the biggest stencil in a statement set 61
8 Normalize per Distance Redistribution Multiplier 62
9 Generate Approximate Stencil Versions 64
10 Stencil Graded Grid Construction 65
11 Discover Write Only Arrays from the Polyhedral Representation 67
12 Discover Potential Accumulators in the Polyhedral Representation 69

Remerciements

Je tiens tout particulièrement à remercier mes directeurs de thèse Cédric Bastoul et Philippe Helluy. J'ai pris beaucoup de plaisir à apprendre les ficelles du métier de chercheur à vos côtés. J'ai pu apprécier les problèmes du point de vue d'un informaticien et d'un mathématicien. Grâce à vous, mon épopée de thèse a été des plus enrichissantes, scientifiquement et humainement.

Je souhaite remercier mes rapporteurs de thèse Denis Barthou et Fabrice Rastello pour leurs relectures et retours sur le manuscrit. Je remercie également Albert Cohen et Fabienne Jézéquel pour avoir accepté de faire partie de mon jury de thèse.

Je remercie les membres de la fabuleuse équipe `ICPS`. Yann Barsamian pour les nombreuses discussions et aventures partagées. Luke Bertot adepte du vélo-boulot et meilleur vérificateur d'information sur mobile multifonction. Bérenger Bramas qui partage avec moi le vice du thé. Arthur Charguéraud qui teste sur des doctorants, des docteurs et sans le savoir des candidats d'entreprises ses jeux éducatifs destinés à de jeunes étudiants. Philippe Clauss, le curé de la paroisse Ehrhart, qui optimise la messe en fonction du nombre de fidèles. Stéphane Genaud qui peut discuter sans interruption sur les réformes dans l'éducation nationale. Paul Godard qui donne toujours une bonne impression et est un fervent défenseur de la ligne droite. Jens Gustedt, mine d'information pour le langage C et optimisations de bas niveau, et source de doute pour les personnes devant prononcer son nom ou « `ORWL` ». Alain Ketterlin, personne à l'image absente d'Internet et ayant un savoir infini, sur tous les sujets, comment fais-tu ? Salwa Kobeissi pour ses histoires incroyables et sa famille qui a réussi à conquérir le monde. Vincent Loechner pour son aide sur de nombreux sujets et l'organisation de sorties en équipe (un coucou à Charlotte et ses blagues rigolotes). Harenome Ranaivoarivony-Razanajato d'avoir su me supporter pendant la licence, le master et la thèse. Mariem Saied pour nous avoir fait visiter tant de plantes au bord de l'eau. Éric Violard pour m'avoir encadré lors de mes stages en licence. Juan Manuel Martinez Caamaño, Marek Felsoci, Raquel Lazcano et Manuel Selva pour ces moments passés ensemble.

Merci à tous mes amis et tout particulièrement Jimmy Brunet, Jeremy Meyer et Raphael Schimchowitsch. Merci à mon « super colocataire » Dimitris Katsouris pour ces deux années passées ensemble.

Il m'aurait été difficile d'en arriver là sans le soutien de mes parents et de ma famille. Vous m'avez permis de suivre la voie qui m'intéressait et je ne pourrais jamais assez vous en remercier, je vous aime fort.

Abstract

In this thesis we introduce a new application programming interface to help developers to optimize an application with approximate computing techniques. This interface is provided as a language extension to advise the compiler about the parts of the program that may be optimized with approximate computing and what can be done about them. The code transformations of the targeted regions are entirely handled by the compiler to produce an adaptive software. The produced adaptive application allocates more computing power to the locations where more precision is required, and may use approximations where the precision is secondary. We automate the discovery of the optimization parameters for the special class of stencil programs which are common in signal/image processing and numerical simulations. Finally, we explore the possibility of compressing the application data using the wavelet transform and we use information found in this basis to locate the areas where more precision may be needed.

Résumé

Dans cette thèse nous proposons une interface de programmation pour aider les développeurs dans leur tâche d'optimisation de programme par calcul approché. Cette interface prend la forme d'extensions aux langages de programmation pour indiquer au compilateur quelles parties du programme peuvent utiliser ce type de calcul. Le compilateur se charge alors de transformer les parties du programme visées pour rendre l'application adaptative, allouant plus de ressources aux endroits où une précision importante est requise et utilisant des approximations où la précision peut être moindre. Nous avons automatisé la découverte des paramètres d'optimisation que devrait fournir l'utilisateur pour les codes à stencil, qui sont souvent rencontrés dans des applications de traitement du signal, traitement d'image ou simulation numérique. Nous avons exploré des techniques de compression automatique de données pour compléter la génération de code adaptatif. Nous utilisons la transformée en ondelettes pour compresser les données et obtenir d'autres informations qui peuvent être utilisées pour trouver les zones avec des besoins en précision plus importantes.

Chapter 1

Introduction

The modern definition of a computer is a machine that executes a sequence of instructions to carry out a computation. They exist in many formats, from small microcontrollers that power not-really intelligent toothbrushes to many powerful processors in huge data centers that deliver a multitude of contents. However, exploiting every bit of their power is a challenge for the developers. Many languages, tools, compiler techniques, or libraries have been proposed to help developers to write efficient applications. In this thesis we propose a new abstraction to achieve performance through adaptive computing, targeting heavy computation only where the application requires it the most.

Early computers were mechanical, and used for example as tools for navigation, astronomy or to keep track of time for calendars. An operator sets the input parameters of the mechanical computer, runs the mechanism to carry out the computation and reads the result on the device. Mechanical, and later electromechanical computers were suffering of slow operating time. In the early 1940s, electronic computers supplanted mechanical ones, using vacuum tubes to carry out operation at a blazing speed of 5000 additions or subtractions per second with the machine called `ENIAC`. This device was operated by women, among the first computer programmers, who had to manipulate switches and wires to configure a program into the machine. This operation could take weeks. The basis of modern computers was defined by Alan Turing [117]. He proposed a machine where the program, specified as a list of instructions, is stored on tape, allowing the machine to be easily programmable.

The advent of fast and compact computers in the 1980s opened a wide range of new opportunities in many societal areas [123]. This escalated the need for computational power and storage capacity to process data in multiple fields, e.g., scientific computing, multimedia, social media, finance, health care, etc. However, the demand for resources is outgrowing the resources that can be allocated to process them. The ratio of data traffic increase to computing power increase has been observed to be in the order of 3 in the early 2000s [64]. Therefore, approximate computing as well as approximate data storage becomes a particularly attractive solution to this ongoing phenomenon where the demand in data processing exceeds the capacity to process it. Gains proportional to the level of approximation can be expected with thorough selection of approximation strategies.

In the first section of the introduction we show that a good hardware abstraction is crucial for developers. Utilizing the full potential of today's hardware is challenging and

is even harder when the amount of data to process exceeds the capabilities of a single machine. In the second section we introduce the rational behind approximate computing and we show that using such techniques adds another layer of complexity that programmers have to handle. We close this chapter by outlining our contributions in this area.

1.1 Programming Challenges

From today's perspective, first digital computers are completely different beasts: to program them you would drill holes in a set of punched cards or tape, representing programs that the computer would execute. The program itself would most likely be written on paper, in a "high-level language" (i.e., combining readability for developers and independence from the target machine), before being transcribed on a punched card by an operator using a keypunch machine. First high level languages as Fortran, C or Cobol, to name a few, freed the developer from having to program in a hardware-dependent way. A processor has a limited set of instructions that can be executed, and a limited number of memory locations, called registers, for the operand(s) and the result of the instructions. Programming languages typically provide the following abstractions:

Unlimited Variable Count The storage place for operands and results can be named and their number is virtually unlimited.

Typed Variables The variables have types defining possible values and the set of possible operations on data of that type, avoiding a class of problem where, for example, one mistakingly try to add apples to oranges.

High-Level Data Structure An *array* allows to store multiple value of the same type and a *structure* allows to construct complex types from other types, for example a basket of apples and oranges.

Procedures A procedure is a named list of instructions. It can be called from other code locations, can take parameters as input and may return a value.

A compiler is a computer program that translates programs written in one format into another format, for example from source code in a given programming language to processor instructions. The compiler allocates the processor's limited resources to execute an equivalent version of the origin program. The transformation of a program is not unique and many semantically equivalent translation may exist. The compiler will usually take one implementation over another based on the predicted behavior of the processor and the memory system.

Early computers would take the punched cards storing the program, read the content and store it inside the computer memory, use the compiler to generate the executable and run the program until termination. The processing was inherently mono-task, one program at a time and executed until completion. Then came the storage and terminal revolution, going from punched card to floppy disks and from paper to program written on a computer equipped with a screen, a keyboard and a text editor program. The task of the compiler remaining unchanged.

Processors with ever-increasing operation speed were built until they hit some physical limits, where the heat to dissipate and power drawn were too high to continue in that direction. Multiple optimizations were added to processors to increase their performance independently to their clock speed:

Cache Accesses to memory (RAM) is slow compared to the processor speed (currently $\approx 400\times$). Hence, a fast memory (currently ≈ 1 to $50\times$ slower than the CPU) is added inside the processor to compensate for the RAM accesses.

Superscalar The processor possesses multiple copies of functional units (e.g., adders, multipliers, etc.) allowing it to execute two or more instructions at a time.

Pipelining Processor instructions can be subdivided into smaller tasks to allow instructions level parallelism. For example, if we want to make a smoothie, the instruction can be divided into fruit peeling followed by the mixer. To make many smoothies: as soon as the fruits are peeled, mix them and start peeling new fruits for the next smoothie in parallel, repeat. Doing it sequentially, we would always wait two working cycles to have a new smoothie. Pipelined we initially wait two working cycles for the first one, and every working cycle after that we obtain a new smoothie for our greatest pleasure.

Out-of-Order An instruction may have to wait for the result of another one. Waiting for the result "stalls" the instruction in the pipeline (previous point), creating waiting cycles instead of executing instructions in parallel. Out of order processors reorder the sequence of instructions to reduce the occurrences of pipeline stalls.

Vectorization A vector is a list of elements of a given size. A vector instruction takes a vector and executes the same instruction to all the elements of the vector in parallel. For example, doing 8 additions at the same time instead of 8 consecutive add instructions.

Multi-Core Can execute two or more programs in parallel by duplicating the computation and control units.

Caches, pipelines and out-of-order optimizations have no visible interface in high-level programming languages. They are architectural optimizations implemented inside the processor to maximize its resource usage. An optimizing compiler should generate a specialized code for each different processor architecture.

Using vectorization and multi-core processors efficiently is at the responsibility of the programmer. Compilers can vectorize and even exploit multithreading automatically, but only if the data structures and instructions are simple enough to be analyzed by the compiler. In general, especially for multithreading, parallelizing or helping the compiler to parallelize has to be done manually. Developing parallel programs is complex and error-prone, the programmer has to take special care with parallel data access, synchronization, deadlocks, etc.

The following list gives an overview of parallel architecture challenges: **Non-uniform memory access (NUMA)**, where more than one multi-core processor and accelerator

(specialized fast computation unit) are present inside the same computer. Each processor or accelerator comes with a dedicated memory bank. The access to another processor's memory is slower than its own memory. Programs not aware of the difference in the memory latency run slower. **Computer Clusters** where multiple machines are interconnected with a network to provide more computing power. Dedicated programming models with different levels of abstraction help the development of parallel programs on clusters.

Allowing an easy access to these resources for the majority of the developers is the goal of the programming abstractions and extensions. For example, OpenMP is an extension to the C/C++ and Fortran programming languages that provides a syntax to easily express parallel, multithreaded regions and tasks [39, 20].

1.2 Approximate Computing

In the previous section we introduced along the main programming challenges, the function of a compiler and the central role that plays a programming language to hide the complexity of the hardware and to provide different ways of thinking about application development. Approximate computing is a technique to exploit an application error resilience to gain in performance. In this section we introduce approximate computing techniques and preconize the use of abstractions to ease the development process.

The term *approximation* comes from the Latin *proximus*, meaning close or near. Consistently, in approximate computing we study functions that are close to the desired result while being simpler to define or to compute. In a nutshell, approximate computing trades precision for performance. Approximate computing techniques are well suited for various kinds of applications:

Noisy input systems. Storage and analysis of noisy data necessitate a lot of resources to extract the relevant data. Each analog sensor is prone to interferences and comes with a specification of its capabilities and error margin. It is non-trivial to extract the signal from the background noise. Therefore, analysis of numerous noisy data streams is a challenge which can be addressed by using approximation techniques.

Error-resilient applications. In some cases, e.g. computer vision or sound synthesis, the program output quality is nearly impacted if we allow a small error margin. The quality of the result is bounded and approximation techniques can leverage the application full potential while sticking to an acceptable output quality.

Iterative applications. Some commonly used algorithms for scientific simulation or machine learning use iterative methods in order to reduce the problem complexity. The runtime of those programs can greatly be reduced by allowing approximate portions of code inside each iteration.

To use approximate computing in an application, three main possibilities are available. Given that the developers are comfortable with approximation techniques, the first option is to write the application from the start with approximation in mind. This requires a high level of expertise of the application algorithms and in approximation theory. The second method uses domain specific libraries that already implement the approximation. With the last option, the developer provides code annotations for the compiler

and lets the framework generate the approximate version automatically. In this thesis, we propose a new way relying on language extensions to enable a new compiler optimization based on approximate techniques.

Traditionally, a compiler must comply with the semantics of the input program and cannot alter the precision of the result on its own. Therefore, the language extensions introduced in this thesis provide, explicitly, the ability for the compiler to generate modified precision code. This extension allows alterations of the code, to partially or even totally replace the original computation. We only consider programs that run on a general purpose processor and we do not take into account specialized hardware, e.g., approximate arithmetical and logical unit or memory storage which can be used to reduce energy consumption, memory access time and computation time or resource sharing [25, 94, 130]. This specialized hardware is much less common than general purpose processors while our purpose is to provide language extensions for a wide range of problems and target platforms.

Adaptive techniques belong to a family of approximate computing methods which use approximation "intelligently" to target the computational resources in locations where it matters. Such techniques originate from the observation that the computation's precision can be expressed as a factor of the data that is being processed. For example, when someone drives a car in a straight direction, the driver's attention will mainly be directed in front of the car and when located at an intersection the attention is focused on the sides of the vehicle. The same principle can be applied to computer programs where we can use a precise version in important locations (i.e., important data) and approximations otherwise to speed up the computation.

1.3 Contributions

In this thesis we introduce a new application programming interface to help developers to optimize an application with approximate computing techniques. This interface is provided as a language extension to advise the compiler about the parts of the program that may be optimized with approximate computing and what can be done about them. The code transformations of the targeted regions are entirely handled by the compiler to produce an adaptive software. The produced adaptive application allocates more computing power to the locations where more precision is required, and may use approximations where the precision is secondary. We automate the discovery of the optimization parameters for the special class of stencil programs which are common in signal/image processing and numerical simulations. Finally, we explore the possibility of compressing the application data using the wavelet transform and we use information found in this basis to locate the areas where more precision may be needed.

Specifying Adaptive Information

An application's development usually follows three phases: (1) implementation of the algorithms to solve a problem, (2) validation of the result and (3) optimization of the program to scale with problem sizes and obtain a result within a realistic time frame. Managing approximate optimizations during the optimization phase usually requires the developer to rethink the data structures and tweak the algorithms accordingly. We believe

that these optimizations should be carried out by the compiler instead of the developer. This should increase the overall productivity and reduce the amount of bugs that could be introduced during this phase. We introduce an extension to high-level programming languages to provide relevant information to the compiler to generate an adaptive version of the program [103, 100, 99]. The relevant information includes (see Chapter 3):

Alternative Code Transformations Alternatives define transformations that modify the code to generate an approximate version. The compiler provides predefined transformations that the developer can use. Alternatively, the users can write several approximate versions of their code and provide them to the compiler.

Data Monitoring To be able to select between the different code versions at runtime, the developer must indicate how to extract, process and interpret the data. The monitoring annotation tells the compiler which data structures are relevant to retrieve meaningful information and how to process the raw values into an approximation level metric.

Granularity Adaptive techniques apply the approximations locally (with respect to either the data space or the computation space), where an approximate version is good enough to achieve a computation with minimal deviation of the result. In our case, the granularity defines a Cartesian grid over the monitored data, specifying the level at which the approximations are applied. We call one element of this grid a *cell*.

Strategy The strategy links the approximation level extracted by the monitoring to the alternatives. Multiple alternatives can be linked to the same approximation level if their composition is legal. The strategy defines the approximate code versions which are generated by the compiler at compile time or specialized version generated during the execution of the program.

Adaptive Code Generation

Automatically generating the adaptive code requires the annotations provided by the user (see Chapter 4) or the adaptive information extracted from a stencil (see Chapter 5). We use the polyhedral model to abstract the kernel to optimize into a mathematical representation and to perform our analyses and code transformations. This allows our technique to be more generally applicable as we are not bound by a specific programming language. We also greatly benefit from existing techniques to minimize the overhead of the generated code.

Firstly, to generate an adaptive application, the annotated kernel is transposed into the polyhedral model. The code is analyzed to assess whether a compile time version can be generated or not. If that is the case, the approximate versions are generated and integrated into the executable. The data is then partitioned to generate the cells used by the monitoring. The monitoring is finally integrated into the cells and the mechanism to select the code version is combined to the application's kernel.

When the dependencies forbid the generation of a compile time version, we rely on a runtime to generate the adaptive code dynamically. The runtime monitors the application data and generates specialized versions of the kernel using polyhedral techniques

and just in time compilation before providing it to the application. This lowers the additional runtime costs of the approximate version selection to the minimum.

Extracting Adaptive Information

Stencil codes are a class of programs where data are updated using the values of their neighbor according to a fixed pattern. We automated the search of the aforementioned adaptive information, easing the work of developers for stencil kernels [101] (see Chapter 5). One unique parameter remains to help the compiler by reducing the search domain: the threshold defining the limit when using an approximate version of the code is considered to be acceptable. The remaining adaptive information is extracted by stencil code analysis.

Data Compression

The annotation-based adaptive code generation alters the computation of the application kernel. We investigated data compression methods to complement the approximate computation (see Chapter 6). We rely on the wavelet transform which carries interesting properties for adaptive codes [102]:

Multiscale Information The wavelet transform breaks the signal down into several localized frequencies. With this information we can localize the areas where more precision is required, as the monitoring does for adaptive computation.

Data Compression The wavelet transform breaks the signal down into "approximations" and the "details" required to reconstruct the signal. The small details, with a value close to zero, can be nullified. Nullifying the details produces an approximation of the signal but allows for a good compression of the data.

Interpolation The approximation generated by the wavelet transform produces values that are nearly those of the studied function. Hence, the values generated by the decomposition can be used as is without the need of doing the reverse the transformation beforehand.

Experimental Study

We built a benchmark set from existing applications to evaluate our approach. Those benchmarks spans from numerical simulation to data processing and are representative of applications resilient to approximation. Our results show that, at the price of inserting few annotations, thanks to our language extensions and our compiler approach, a developer can exploit approximations to significantly improve performance with low deviation from the original result (see Chapter 7).

This thesis introduces an application programming interface to generate an adaptive version of a program from annotations provided by the developer. Our goal is to allow experts and non-experts in the approximate computing field to use additional optimization

methods based on approximation without needing an extensive application rewriting. In our approach, the adaptive information is inserted as optional annotations for the compiler. The compiler generates the adaptive program and selects the relevant approximate versions at execution time from information gathered from the application's data. Data compression techniques complement the approximate code versions. The remainder of this thesis exposes how we proceed.

Chapter 2

Scientific Foundation and State of the Art

In this chapter we introduce the useful background and notations for the remainder of our work on exploiting approximation through a compiler approach. In Section 2.1 we present how a typical numerical simulation problem is translated into a code and how it may be optimized using approximations. Section 2.2 follows with a state of the art of existing approximation techniques. Our compiler framework builds on the polyhedral model which is introduced in Section 2.3. This model is a mathematical abstract representation of codes, which defines the application domain that we can handle directly and provides a framework for dependency analysis, verified code transformation and low-control-overhead code generation.

2.1 Numerical Analysis — Solving a System of Equations With a Computer

The field of numerical analysis aims at finding an accurate approximation of solutions to problems which are too complex to solve precisely. In this section we emphasize the applicability of our compiler assisted approximation technique on Maxwell's equations of electromagnetism [114, 111]. Our journey starts with the introduction of the context and the partial differential equations (PDE), and we will work our way towards a `C` implementation of the 1 dimensional case. Finally, we will show a preview of our annotations (detailed in Chapter 3) to make the application react to the simulation variations, i.e., be adaptive.

2.1.1 From Maxwell's PDE to Update Equations

Maxwell's equations (2.1, 2.2) are used to simulate physics phenomena in a wide range of applications, for example antennas in electronic devices, radars, fast data transfer medium and many medical applications such as cancer detection or body exposure to microwave radiation. The equation we are interested here in solving is the Maxwell's curl equation in linear, isotropic, nondispersive and lossy material, i.e., materials having field-independent, direction-independent and frequency-independent electric and magnetic

properties and where the magnetic and electric field can lose energy while heat is generated in the material.

$$\frac{\partial \boldsymbol{H}}{\partial t} = -\frac{1}{\mu}\nabla \times \boldsymbol{E} - \frac{1}{\mu}\left(\boldsymbol{M}_{\text{source}} + \sigma^* \boldsymbol{H}\right) \tag{2.1}$$

$$\frac{\partial \boldsymbol{E}}{\partial t} = \frac{1}{\varepsilon}\nabla \times \boldsymbol{H} - \frac{1}{\varepsilon}\left(\boldsymbol{J}_{\text{source}} + \sigma \boldsymbol{E}\right) \tag{2.2}$$

where

Symbol	Physical Meaning	Unit
$\boldsymbol{H}$	magnetic field	ampere / meter
$\boldsymbol{E}$	electric field	volt / meter
μ	magnetic permeability	henry / meter
ε	electric permittivity	farads / meter
$\boldsymbol{J}_{\text{source}}$	electric source	ampere / meter^2
$\boldsymbol{M}_{\text{source}}$	magnetic source	ampere / meter^2
σ	electric conductivity	siemen / meter
σ^*	equivalent magnetic loss	ohm / meter

and $\nabla \times F$ is the curl operator, representing infinitesimal rotation of a field F.

Equation 2.1 derives from Faraday's law of induction and dictates how a magnetic field interacts with an electric circuit to produce an electromotive force. This equation reads as follows: a time varying magnetic field is related to a spatially varying electric field plus magnetic sources and losses, i.e., a change over time of the magnetic field induces a current in the medium. Similarly, Equation 2.2 derives from Ampère's law and relates how a time varying electric field is related to a spatially varying magnetic field.

Equations 2.1 and 2.2 are partial differential equations of time and space and cannot be implemented as a computer simulation in this form. We need to use a numerical scheme that is a good approximation of the equations and choose a discrete data representation so it can be represented inside a computer's memory. In the following implementation, we will use the scheme introduced by Kane Yee [129]. He proposed a finite-difference scheme, i.e. an approximation of the derivative in the form of a difference $f(x+b)-f(x+a)$, where both electric and magnetic field are solved simultaneously. The electric and magnetic field representation is shown in Figure 2.1. Each discrete value of a field is surrounded by two values of the other but the values are computed at time steps which are $\frac{\Delta t}{2}$ apart. This peculiar placement of the fields in time and space results from the application of finite-difference in time and space.

Finite-difference is a mathematical expression to approximate the derivative of a function. To understand how this works, let us use the Tailor's series expansion of a function of space and time $u(i,t)$, also denoted $u|_i^t$, for the space point i at time t for $i + \frac{h}{2}$ in Equation 2.3 and $i + \frac{h}{2}$, where h is a value chosen small enough to reduce the error, in

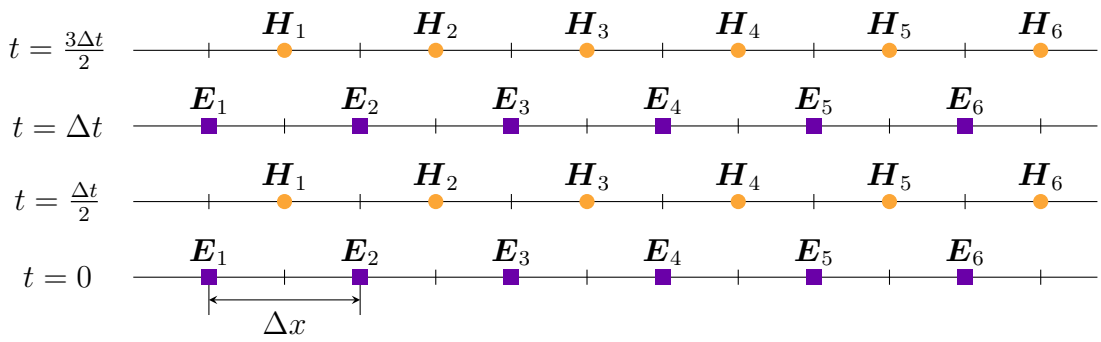


Figure 2.1 – Yee's algorithm chart in one dimension. The electric field $\boldsymbol{E}$ and magnetic field $\boldsymbol{H}$ are physically interleaved by a distance of $\frac{\Delta x}{2}$. The electric and magnetic field at a time t is computed using the magnetic and electric field at the time $t - \frac{\Delta t}{2}$, respectively.

Equation 2.4:

$$u\left(i+\frac{h}{2}\right)\Big|^t = u|_i^t + \frac{h}{2}\frac{\partial u}{\partial x}\Big|_i^t + \frac{\left(\frac{h}{2}\right)^2}{2!}\frac{\partial^2 u}{\partial x^2}\Big|_i^t + \frac{\left(\frac{h}{2}\right)^3}{3!}\frac{\partial^3 u}{\partial x^3}\Big|_i^t + \cdots \tag{2.3}$$

$$u\left(i-\frac{h}{2}\right)\Big|^t = u|_i^t - \frac{h}{2}\frac{\partial u}{\partial x}\Big|_i^t + \frac{\left(\frac{h}{2}\right)^2}{2!}\frac{\partial^2 u}{\partial x^2}\Big|_i^t - \frac{\left(\frac{h}{2}\right)^3}{3!}\frac{\partial^3 u}{\partial x^3}\Big|_i^t + \cdots \tag{2.4}$$

If we subtract these two Tailor series as shown in Equation 2.5 and rearrange the terms, we obtain Equation 2.6. This equation is known as the central-difference formula of order $O(h^2)$ because the added error, when ignoring the remaining terms of the Tailor series, goes to zero in the same manner as h^2. This method can also be applied to do a central-difference on the time dimension.

$$u\left(i+\frac{h}{2}\right)\Big|^t - u\left(i-\frac{h}{2}\right)\Big|^t = h\frac{\partial u}{\partial x}\Big|_i^t + \frac{2\left(\frac{h}{2}\right)^3}{3!}\frac{\partial^3 u}{\partial x^3}\Big|_i^t + \cdots \tag{2.5}$$

$$\frac{\partial u}{\partial x}\Big|_i^t = \left[\frac{u\left(i+\frac{h}{2}\right) - u\left(i-\frac{h}{2}\right)}{h}\right]^t + O\left(h^2\right) \tag{2.6}$$

We now have a formula to approximate the derivative of a function which only requires nearby function values provided that we pick them close enough to where we compute the derivative (error in $O(h^2)$). Yee applies this central-difference formula to Maxwell's Equations 2.1 and 2.2. First let us rewrite these two equations in the Cartesian coordinate system by applying the curl operator of a field $\nabla \times F$. Equations 2.1 and 2.2 yield 2.7 and 2.8, respectively, and a, b and c are oriented in the same direction as the three basis vectors.

$$\frac{\partial \boldsymbol{H}_x}{\partial t} = \frac{1}{\mu}\left[\frac{\partial \boldsymbol{E}_y}{\partial z} - \frac{\partial \boldsymbol{E}_z}{\partial y} - (\boldsymbol{M}_{\text{source}_x} + \sigma^* \boldsymbol{H}_x)\right] \tag{2.7a}$$

$$\frac{\partial \boldsymbol{H}_y}{\partial t} = \frac{1}{\mu}\left[\frac{\partial \boldsymbol{E}_z}{\partial x} - \frac{\partial \boldsymbol{E}_x}{\partial z} - (\boldsymbol{M}_{\text{source}_x} + \sigma^* \boldsymbol{H}_x)\right] \tag{2.7b}$$

$$\frac{\partial \boldsymbol{H}_z}{\partial t} = \frac{1}{\mu}\left[\frac{\partial \boldsymbol{E}_x}{\partial y} - \frac{\partial \boldsymbol{E}_y}{\partial x} - (\boldsymbol{M}_{\text{source}_x} + \sigma^* \boldsymbol{H}_x)\right] \tag{2.7c}$$

$$\frac{\partial \boldsymbol{E}_x}{\partial t} = \frac{1}{\varepsilon}\left[\frac{\partial \boldsymbol{H}_z}{\partial y} - \frac{\partial \boldsymbol{H}_y}{\partial z} - (\boldsymbol{J}_{\text{source}_x} + \sigma \boldsymbol{E}_x)\right] \tag{2.8a}$$

$$\frac{\partial \boldsymbol{E}_y}{\partial t} = \frac{1}{\varepsilon}\left[\frac{\partial \boldsymbol{H}_x}{\partial z} - \frac{\partial \boldsymbol{H}_z}{\partial x} - \left(\boldsymbol{J}_{\text{source}_y} + \sigma \boldsymbol{E}_y\right)\right] \tag{2.8b}$$

$$\frac{\partial \boldsymbol{E}_z}{\partial t} = \frac{1}{\varepsilon}\left[\frac{\partial \boldsymbol{H}_y}{\partial x} - \frac{\partial \boldsymbol{H}_x}{\partial y} - (\boldsymbol{J}_{\text{source}_z} + \sigma \boldsymbol{E}_z)\right] \tag{2.8c}$$

To keep our example simple, we will continue with the lossless ($\sigma = \sigma^* = 0$) one-dimensional case, also known as x-directed, z-polarized transverse-electromagnetic (TEM) mode. It is defined by the two equations 2.9 and 2.10.

$$\frac{\partial \boldsymbol{H}_y}{\partial t} = \frac{1}{\mu}\left[\frac{\partial \boldsymbol{E}_z}{\partial x} - M_{\text{source}_y}\right] \tag{2.9}$$

$$\frac{\partial \boldsymbol{E}_z}{\partial t} = \frac{1}{\varepsilon}\left[\frac{\partial \boldsymbol{H}_y}{\partial x} - J_{\text{source}_z}\right] \tag{2.10}$$

Lets apply the central-difference approximations (Equation 2.6 and its time counterpart) to the space and time derivatives on both sides of Equation 2.9:

$$\left[\frac{\boldsymbol{H}_y\left(t + \frac{\Delta t}{2}\right) - \boldsymbol{H}_y\left(t - \frac{\Delta t}{2}\right)}{\Delta t}\right]_i = \frac{1}{\mu_i}\left(\left[\frac{\boldsymbol{E}_z\left(i + \frac{\Delta x}{2}\right) - \boldsymbol{E}_z\left(i - \frac{\Delta x}{2}\right)}{\Delta x}\right]^t - M_{\text{source}_y}\right)$$

Reordering the previous equation gives us the Update Equation 2.11 and doing the same for Equation 2.10 yields 2.12.

$$\boldsymbol{H}_y\left(t + \frac{\Delta t}{2}\right)\Big|_i = \boldsymbol{H}_y\left(t - \frac{\Delta t}{2}\right)\Big|_i + \frac{\Delta t}{\mu_i}\left(\left[\frac{\boldsymbol{E}_z\left(i + \frac{\Delta x}{2}\right) - \boldsymbol{E}_z\left(i - \frac{\Delta x}{2}\right)}{\Delta x}\right]^t - M_{\text{source}_y}\right) \tag{2.11}$$

$$\boldsymbol{E}_z\left(t + \frac{\Delta t}{2}\right)\Big|_i = \boldsymbol{E}_z\left(t - \frac{\Delta t}{2}\right)\Big|_i + \frac{\Delta t}{\varepsilon_i}\left(\left[\frac{\boldsymbol{H}_y\left(i + \frac{\Delta x}{2}\right) - \boldsymbol{H}_y\left(i - \frac{\Delta x}{2}\right)}{\Delta x}\right]^t - J_{\text{source}_z}\right) \tag{2.12}$$

Equations 2.11 and 2.12 are what we need to implement the algorithm pictured in Figure 2.1. The first equation will be used to compute the magnetic field $\boldsymbol{H}$ at time steps equal to $t_0 + (\alpha + \frac{1}{2})\Delta t$ and the electric field $\boldsymbol{E}$ at time steps equal to $t_0 + \alpha\Delta t$, $\alpha \in \mathbb{N}$. We start with the initial condition for each field $\boldsymbol{H}$ and $\boldsymbol{E}$, the materials properties ε and μ, and source functions M and J and alternate the update equation to reach a given time step.

2.1.2 From Update Equations to Simulation Code

The Equations 2.11 and 2.12 that we constructed in Section 2.1.1 to approximate Maxwell's partial differential equations may serve as the basis of a simulation program. In this section we are going to construct the following program parts: the data structure, $\boldsymbol{H}$ and $\boldsymbol{E}$ fields update functions, the initialization function, the source function and the function handling the domain borders.

Constraints on the values of Δt and Δx have to be enforced for the numerical stability of the `FDTD` method. In our implementation we use the Courant-Friedrichs-Levy (`CFL`) condition [36]:

$$S \equiv c\Delta t \sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}} \leq 1 \tag{2.13}$$

Where c is the speed of light in free space. When $\Delta x = \Delta y = \Delta z = \Delta$ we have $\frac{c\Delta t}{\Delta} \leq \sqrt{3}$ in 3D, $\frac{c\Delta t}{\Delta} \leq \sqrt{2}$ in 2D and $\frac{c\Delta t}{\Delta} \leq 1$ in 1D. To run a simulation, a value of Δx is selected, e.g., 20 discrete points for the lowest wavelength ($\lambda_{\min}/20$) and a value for Δt is computed from the previous formula for a given `CFL`. To summarize this condition: the time step should not be too large with respect to the physical discretization or we are getting an erroneous solution for the waves that travel further than one physical step during a time step.

`FDTD` Data Structure

The following data structure stores the information for the one-dimensional implementation of our example `FDTD` simulation. The electric and magnetic fields are implemented as one-dimensional arrays $\boldsymbol{H}$ and $\boldsymbol{E}$ with size `size`. Each field is zero-initialized. The sources are stored in the `f_source` data structure arrays, having size `Mnum` and `Jnum` for the magnetic and electric sources respectively. The physical properties of the medium are stored in the `permittivity` (ε_i) and `permeability` (μ_i) arrays.

```
1 struct fdtd1D {
2   uintmax_t size, Mnum, Jnum;
3   float *E, *permittivity;
4   float *H, *permeability;
5   f_source *M_sources, *J_sources;
6   float Sc, dt, dx, t, domain_size;
7 };
```

FDTD Field Update Function

The following update functions `updateH` and `updateE` derive from Equations 2.11 and 2.12. The indices of the array accesses of $\boldsymbol{H}$ and $\boldsymbol{E}$ at lines 5 and 13 follow the pattern pictured in Figure 2.1, where the fields are interleaved in space.

```
1  void updateH(struct fdtd1D *fdtd) {
2    float dtdx = fdtd->dt / fdtd->dx;
3    for (uintmax_t pos = 0; pos < fdtd->size - 1; pos++) {
4      fdtd->H[pos] = fdtd->H[pos] +
5        (fdtd->E[pos + 1] - fdtd->E[pos]) * dtdx / fdtd->permeability;
6    }
7  }
8
9  void updateE(struct fdtd1D *fdtd) {
10   float dtdx = fdtd->dt / fdtd->dx;
11   for (uintmax_t pos = 1; pos < fdtd->size; pos++) {
12     fdtd->E[pos] = fdtd->E[pos] +
13       (fdtd->H[pos] - fdtd->H[pos - 1]) * dtdx / fdtd->permittivity;
14   }
15 }
```

FDTD Simulation Loop

The `fdtd_sim` function takes care of the time dimension of the simulation solver. The central-difference approximation that is used to approximate the time derivative in Equations 2.11 and 2.12 requires the update functions to be interleaved. As depicted in Figure 2.1, $\boldsymbol{H}$ and $\boldsymbol{E}$ are not solved for the same time step because both equation require values of the other field at a time t and their own field value at time $t - \frac{\Delta t}{2}$ to compute the result at time $t + \frac{\Delta t}{2}$. Hence, composing the two functions advances the simulation by an amount of Δt, solving $\boldsymbol{H}$ at times multiple of $\frac{2t+1}{2}$ and $\boldsymbol{E}$ for times multiple of $t+1$.

```
1  void fdtd_sim(struct fdtd1D *fdtd, float t_end) {
2    while(fdtd->t < t_end) {
3      updateH(fdtd); addSourceH(fdtd); // t + dt / 2
4      updateE(fdtd); addSourceE(fdtd); // t + dt
5      fdtd->t += fdtd->dt;
6    }
7  }
```

2.1.3 Program Optimization and Adaptive Techniques

We finished the first part of our journey to an implementation of a solver for Maxwell's equations with the algorithm introduced in Section 2.1.2. Unfortunately, a solver implementation can occasionally lack applicability on concrete problems due to limited computational resources. Our solver memory and computational requirement are summarized in Table 2.1. We can observe that the limiting factor comes from the simulated physical domain size, both in memory and number of operations to perform.

Table 2.1 – Usage requirement for the `FDTD` algorithm implemented in Section 2.1.2

Resource	Usage	Order
Memory	Arrays `E`, `H`, `permittivity` and `permeability`	$\mathcal{O}(n^{\dim})$
	Structures `M_sources` and `J_sources`	$\mathcal{O}(n)$
Computation	`updateH` and `updateE` (`sizeX` add and $2 \times$ sizeX multiply)	$\mathcal{O}(n^{\dim})$
	`addSourceH` and `addSourceE`	$\mathcal{O}(n)$

Exploiting Parallelism

A first optimization approach is to distribute the work and memory among multiple workers whenever possible. There is no data dependency present inside the same update function for $\boldsymbol{H}$ and $\boldsymbol{E}$ which allows every dimensions of the `for` loop to be parallel. To update the field $\boldsymbol{E}$ at a position $\vec{p}$ we require a read and write operation on $\boldsymbol{E}$ at the same position $\vec{p}$ and up to 12 reads of the neighbors of $\vec{p}$ of $\boldsymbol{H}$. Figure 2.2 shows an example of sharing the computation and data between multiple workers. Using this method, we could scale indefinitely to any problem size providing enough resources are available. In theory, this approach scales linearly with the number of workers added.

Locating Expensive Code Regions and Becoming Adaptive

The previous optimization distributes the computation among multiple workers. The global amount of computation remains constant but we add more computing power to solve the same problem. Approximation techniques, on the other hand, can reduce the amount of computation drastically while providing a result with sufficient precision.

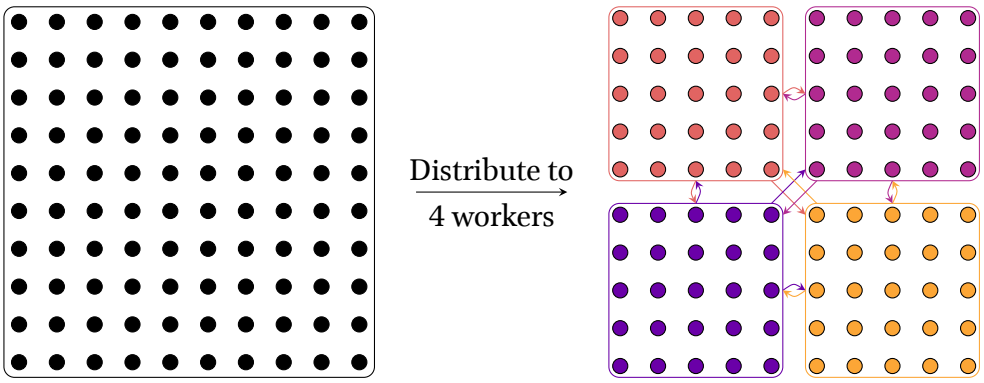


Figure 2.2 – **Work-Sharing Example** `FDTD` data $\boldsymbol{H}$ and $\boldsymbol{E}$ represented as points, initially belongs to a single worker which does all the computation on that data, left. The iterations of `updateH` and `updateE` loops can be shared between multiple workers because there is no dependency preventing it. Distributing the work equally between 4 workers can be done as shown on the right, where each box corresponds to a worker. Each worker stores the portion of the data accessed by the work they have to do. An arrow between two workers indicates that some data communication is necessary, because the workers will access neighboring data at the borders.

To have a better understanding of where optimizations such as approximation should be performed, we must locate the expensive code portions of our program. On the Linux operating system, the `perf` utility is available to profile programs [84]. Listing 2.1 shows how to accomplish a profiling session for our `FDTD` program. We can observe that most of the execution time is spent inside the two update functions `updateH` and `updateE`. We will therefore prioritize the optimization of these parts.

```
$ perf record -e cycles:up --call-graph dwarf ./fdtd
$ perf report --stdio --sort comm
Samples: 15K of event 'cycles:up'

Children      Self  Command
........  ........  .......

 100.00%   100.00%  fdtd
          |
           --99.97%--_start
                     __libc_start_main
                     main
                     |
                     |--98.89%--fdtd_sim
                     |           |
                     |           |--49.63%--updateE (inlined)
                     |           |
                     |            --49.24%--updateH (inlined)
                     |
                      --0.88%--init_fdtd_1D_medium
```

Listing 2.1 – Profiling of the FDTD application using the `perf` utility. The first command runs the `FDTD` program with the cycle hardware counter enabled. The second command uses the generated profile to print the proportion of time spent in different sections of the program.

Approximate computing is a broad domain where multiple strategies can be used. We introduce the state of the art approximate computing strategies in more depth in Section 2.2. In this section we only consider computation skipping. Knowing which computation to skip requires some domain specific knowledge. Figure 2.3 lets us visualize the magnetic field of a 2D `FDTD` simulation at two time frames. We can observe that the signal travels in space, covering previously neutral white zones and clearing the location near the black obstacle. The white zones, where the field is neutral, contain values which are close zero while the red and blue zones have positive and negative non-null values, respectively. A wave travels the space continuously, from discrete neighbor to neighbor in the simulation, therefore a neutral region will not suddenly become charged in the absence of incoming wave or source. Updating the neutral zones less frequently is the approximation opportunity we are going to exploit in such application.

2.1.4 Generating an Adaptive Version

Our strategy consists in reducing the amount of computation in the neutral zones dynamically, while the simulation runs. Implementing such strategy for our application is

straightforward and can be implemented in by adding a single line of source code before the computation inside `updateH` loop:

```
1 if (fdtd->E[pos + 1] > Ethreshold || fdtd->E[pos + 1] < -Ethreshold ||
2     fdtd->E[pos    ] > Ethreshold || fdtd->E[pos    ] < -Ethreshold)
```

This strategy results in a reduction of the computation volume by 45%, an average deviation of the result compared to the original version output by 16% and a speedup of 0.87 (a slowdown!). We are still far from the expected goal of reduced computation time and narrow results. By adding the guard before the computation we slow the flow of execution, which has to check for every value if it is going to compute the result or not. Since each iteration of the kernel is not computation heavy, here two additions, one multiplication and one division, we are essentially adding four tests for four arithmetic instructions. A guard may also prevent some compiler optimization to be performed, for example, vectorization. Profiling the application with `perf` shows that half the computation time of this kernel is shared between the actual computation and the guard.

For our optimization to be meaningful, we need to reduce the guard overhead while keeping the number of skipped computation as close as possible to 45%, which is the upper bound. Figure 2.4 shows a partitioning example of the simulation domain. The domain is partitioned in square shaped cells and we apply the same approximation to each constituent of a cell. The gray zone in this figure covers the cells where the decision to skip the computation has been made. This corresponds to the white neutral areas in Figure 2.3. Implementing such strategy is programmatically an order of magnitude harder than inserting a test as we did formerly. We need to partition the data domain, take a cell-wise decision and add a guard only when we cross a cell boundary. Furthermore, once applied, it is challenging to change the cell size or try multiple approximation strategies, because the original algorithm became indissociable from the optimization portions which makes it harder for code maintenance and further optimizations.

Compiler Based Adaptive Optimization

Instead of asking the developers to implement the adaptive optimization, our goal is to add adaptive optimization to programming languages and to rely on the compiler to perform the code transformations. Hence, the source code remains easy to maintain and

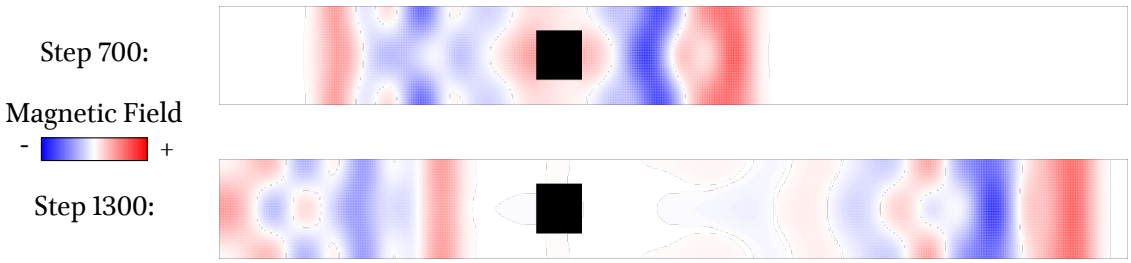


Figure 2.3 – **Magnetic Field Intensity Propagation Example** Graphical representation of the magnetic field H in a 2D `TEM Mode FDTD` simulation. The black box is an obstacle which reflects the magnetic field.

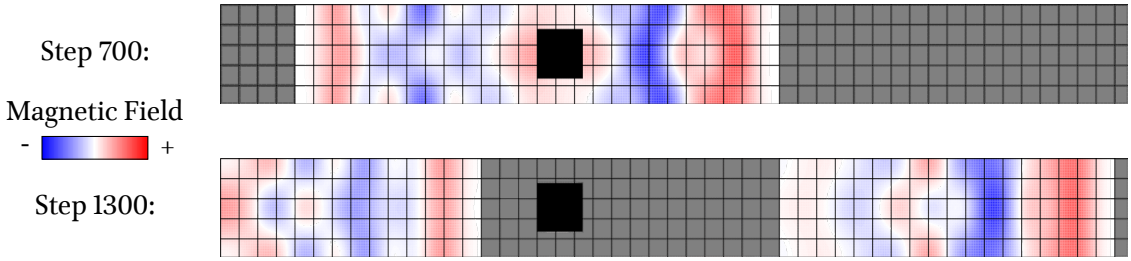


Figure 2.4 – **Magnetic Field's Adaptive Grid** Same simulation as Figure 2.3 using a grid to take an adaptive decision. The approximation decision is done for each cell (a square) represented in the figure. A gray cell indicates that the computation may be skipped for all the values present inside the square.

the developers can quickly try multiple approximation strategies without having to implement them themselves. Providing an API lowers the expertise required to use approximate computing as most of the transformations will be handled automatically. We name our transformation framework `Adaptive Code Refinement` (`ACR`). Chapter 3 presents the annotations available to the user and their semantics and Chapter 4 presents how the annotations information is used to generate an adaptive version of the program.

Listing 2.2 shows the magnetic field kernel with the `ACR` annotations added. For this kernel we use the same skipping computation strategy as before. It is defined by the `zero-compute` `alternative` at line 5. An `alternative` is an approximate version of the kernel either user-provided or generated by the compiler, and the selected one skips all the computation belonging to a cell. The `monitor` annotation at line 4 instructs the compiler to monitor the electric field at runtime using the `EtoStrategy` function to gather meaningful information which is used to provide adaptiveness to the kernel. In this example the monitor gathers a value of 0 or 1 for each cell using a threshold on the

```
1  void updateH(struct fdtd1D *fdtd) {
2    float dtdx = fdtd->dt / fdtd->dx;
3    #pragma acr grid(15)
4    #pragma acr monitor(fdtd->E[pos], min, EtoStrategy)
5    #pragma acr alternative skip(zero-compute)
6    #pragma acr strategy dynamic(1, skip)
7    for (uintmax_t pos = 0; pos < fdtd->size - 1; pos++) {
8      fdtd->H[pos] =
9        fdtd->H[pos] +
10       (fdtd->E[pos + 1] - fdtd->E[pos]) * dtdx / fdtd->permeability;
11   }
12 }
13 uint8_t EtoStrategy(float Eval) {
14   if (Eval > Ethreshold || Eval < -Ethreshold)
15     return 0;
16   else
17     return 1;
18 }
```

Listing 2.2 – **ACR Annotations Example** Using the `ACR` API to annotate the `FDTD` kernel.

intensity of the electric field. A value of 0 always maps to the original kernel and 1 can map to one of the alternatives when one or more have been defined. At line 6, the `strategy` annotation states that a cell with a monitored value of 1 can use the approximate version named "skip," the name given to the alternative line 5. Finally, the `grid` annotation at line 3 defines a cell as an agglomerate of 15 electric field values. The grid constructed by ACR is regular and multidimensional.

These annotations provide the application-specific information needed by the compiler to generate the adaptive program as shown in Figure 2.4. This annotated source code is given as input to a source to source compiler which adds the necessary components for the monitor runtime, the alternative versions and the runtime alternative selection. The kernel analysis, transformations and `alternative` generation are based on an algebraic representation of code known as the polyhedral model, which is introduced in Section 2.3. We use this representation to generate approximate version of an annotated kernel by skipping some iterations of the loop. Approximation strategies are introduced in Section 2.2.

2.2 Approximation Techniques as Program Optimization

The goal of a program optimization is to generate a more efficient yet semantically equivalent version of a program. The most common efficiency metrics are the program execution time, the power efficiency or the number of operations executed with respect to the non-optimized program. A programming language defines a semantics which formally relates the textual source code to the program that will be executed by the computer. An approximate computing transformation may break the language semantics to allow more aggressive optimizations to be performed. Approximation may therefore induce an alteration of the application output compared to what the source code expresses. In this section we present a set of approximate computing techniques and the state of the art compiler support for approximation [78].

2.2.1 Approximate Computing Strategies

Approximate computing strategies rely on relaxing the program semantics in order to optimize for a given metric. We dissociate approximate computing strategies in two categories: the ones that use a hardware feature and the ones that rely solely on software.

Hardware-Based Approximate Computing

Approximate computing at the hardware level targets efficiency in terms of execution time and power consumption. Specialized hardware implementation targets domain specific functions, e.g., trigonometric functions like sin or cos, and are optimized for a given representation of numbers, for input parameter range and output precision [65, 3]. Specialization is particularly interesting when the operation is extensively used in the application or strict deadline are expected and enforced at the hardware level. Approximate hardware implementation can practically be implemented on field-programmable gate array (FPGA) hardware [105]. Esmaeilzadeh et al. show that neural network accelerators can be used to approximating general purpose computation [45]. They trained

neural networks to replace part of imperative code by an approximate version which is 2 times faster on average and reduces the energy consumption by $3\times$.

Approximate data storage is a good solution to reduce the power consumption drastically and increase the life span of memory storage. Fang et al. work on phase change memory (PCM), where a write is expensive. They add circuitry to skip a write when the difference between the value to write and the one already in memory is small. This yields a reduction of write operation to memory cells of 34% while the approximation to the application generated image is hardly perceptible [46]. Sampson et al. work on multilevel cell (MLC) solid state memory suggest to implement a faster write and reduced power draw data write that can lead to corruption of certain data bit [95]. Errors are mitigated by packing multiple values and can be combined with error correction codes to limit the probability for multiple errors to occur on the same value and being able to recover or detect errors. They showed that wear-out cells can be recycled by using it exclusively for approximate storage. Implementation of approximate memory requires extensive hardware knowledge and, to the best of our knowledge, hardware providing such capabilities is scarce.

Software-Based Approximate Computing Strategies

Approximate computing finds its root in software as transformation that relies on computation approximation and dependency relaxation to generate faster programs. This section presents a range of techniques used to achieve approximation relying exclusively on software.

Loop perforation is a technique which optimizes loop nests, one of the most computation expensive construct in programs [106]. This optimization uses a profiling phase to find loops that account for at least 1% of the program execution time and tries multiple approximation strategies to approximate them. If a strategy fails, i.e., crashes the program or generates an output with significant errors, it is discarded and another strategy is tried. The strategies consists in changing the stride of a loop induction variable, skipping chunks of iterations at the beginning or at the end of the loop, and a random policy. The profiling phase dictates how many iterations of the loop are to be skipped in the optimized application. Sculptor is an improved version of loop perforation [68]. It allows fine grained instruction skipping instead of a complete one. The instruction are selected based on their relative expensiveness and the level of approximation they generate when skipped. This solution incorporates a runtime scheduler to select one perforated version between many. The scheduler can take a decision as a function of the function call site (using the call graph and profiling) and runtime monitoring of the error by running the original and approximated version and comparing both to compute the error rate. Misailovic et al. use a probabilistic approach to reason about *loop perforation* [77]. They recognize computational patterns where it is possible to compute the probability that the error generated by the transformation is lower than an upper bound.

Task skipping avoids the execution of some tasks to save computation at the price of some precision loss. This technique is similar to loop perforation but applied at a task level [87, 52]. This technique requires the task output data type to match its input data type, otherwise, the next task in line will receive an erroneous input. Task based programming usually tends to split a program into tasks and construct a task graph. The tasks are represented as nodes in the graph and edges stand for the data flow between the tasks. Af-

ter some graph analysis, it is possible to distribute the concurrent tasks between multiple processors or computers. Removing tasks from the graph can lead to better parallelism and reduced resource utilization. Rinard et al. showed that the computation may continue in the event of a failed or skipped task and that a probabilistic model may be used to predict the distortion of the result [87]. ApproxHadoop is an example of task skipping for the `MapReduce` paradigm, where a function is applied on a big amount of data (map) and another function summarizes the computed results (reduce) [52]. They show that their technique is capable of speedup up to $35\times$ with a target error bound of 1% by skipping the map tasks.

Function multi-versioning makes use of multiple approximate versions of a function. During the execution of the program, a runtime can select one of these versions to maximize the program speed while minimizing the generated error. The only restriction of this technique is that the different function's versions must share the same prototype. PetaBricks is a programing language and compiler which uses empirical approach to select the best kernel among a provided set of versions [4]. PetaBricks's compiler has been augmented to automatically generate approximate versions using a genetic algorithm which targets a minimum deviation of the output [5].

Memoization stores pairs of parameters and results of function calls to avoid an expensive function call when the parameters have already been encountered and the result is known. Such technique is usually found in functional language where the function calls are pure [74, 1]. A pure function always returns the same value provided identical arguments and is free of side effects. Determining the number of pairs to store, the retention policy and the data structure to store them are critical for the usage of this solution. To be profitable, the memoized function execution time must be a magnitude higher than the arguments matching mechanism and the stored pairs reused. Rahimi et al. introduce memoization at the processor level in the context of single instruction multiple data (`SIMD`) architectures [86]. They use memoization to speed up the error recovery mechanisms which are triggered by a processor timing error. Suresh et al. provide a technique to intercept dynamically linked, pure and computationally expensive functions to make them use memoization without modification of the source code [113]. They augmented the technique to handle compile time linking, pointer parameter support and hardware acceleration [112].

Memory access skipping removes data accesses in order to accelerate computation. The non-accessed values can either be interpolated using neighboring values, speculated / predicted using data regression algorithms or ignored. Accessing data from main memory may be much slower than computing a new one. Applications that process massive amount of data may reduce the processor to RAM bus contention and distributed applications may use this technique to avoid costly network transfers. Miguel et al. show that load approximations on cache misses, where the access to memory is evicted entirely, can achieve up to 28% speedups and 44% energy savings [75]. Memory access skipping is used by most of the aforementioned approximation techniques to reduce the memory footprint and in signal processing to compress video or image while maintaining a good image quality [96, 122, 62]. Precimonious is a tuning assistant that discovers the floating point precision needed within a target application [89]. By doing this, Precimonious reduces the memory accesses and increases the performance of the program when the lower precision arithmetic is selected.

Dependence relaxation can lead to better utilization of the capacity of the hardware. Helix-UP chooses to ignore dependencies and synchronizations to increase parallelism in applications [24]. Byna et al. dropped dependencies of a sequential document search algorithm to utilize the full potential of GPUs and coupled that with task skipping to achieve better performance [23].

Framework for Approximate Computing

Using the hardware and software approximate computing presented at the beginning of Section 2.2 requires a high level of expertise. In this section we present the frameworks and tools to help the developer during the optimization task.

Evaluating approximation targets in an application is a non-trivial task. Chippa et al. propose a framework to characterize the resilience of applications to approximate computing techniques [29]. This framework can help the developers to uncover the parts of an application where approximate techniques could be implemented.

Automatic optimizers analyze and transform the code with as minimal intervention of the user as possible. Sage is an automatic tool to skip costly atomic operations, pack data and relax dependencies to improve parallelism on GPUs [91]. Paraprox automatically identifies computational patterns in program and uses approximation templates to replace them [90]. The developer relies on the compiler to identify these patterns and optimize them for the target architecture automatically. Meng et al. provide an optimizer applied for the domains of recognition and data mining where a huge amount of data has to be processed [72].

An approximate hardware specification is proposed by Esmaeilzadeh et al. in the form of an instruction set architecture for other framework to build upon [44]. They support individual approximate instructions, approximate storage with approximate loads and stores. They define the interaction between the approximate and non-approximate portions of the processor. Chisel is a framework that relies on approximate computing hardware specification to provide statistical guarantees for operations and functions [76]. They provide a language construct which states the maximum error for a function and make the compiler generate approximate hardware instructions to achieve approximation.

Software-hardware whole stack optimization has been investigated by Chippa and contributors [30]. Approximate computing is considered at each level of the execution stack, at the software level where some computation is skipped, at the architecture level where multiple similar operations with reduced precision are packed together for efficiency and at the hardware level where approximate arithmetical operations are used.

Environment feedback to drive approximation can be used to tune the level of approximation dynamically. Hoffmann et al. incorporate a runtime that monitors system metrics, e.g. average load or power draw, to tune running application parameters to meet a desired throughput [60]. The parameters are provided by the user to their framework which subsequently initiates a search of the parameter space for good candidates that meets an acceptable quality of service. The system can therefore react to resource shortage by asking applications to change their level of accuracy to meet deadlines. Eon provides a language and runtime system that integrates hardware information, e.g. battery level, to tune the quality of service depending on the environment [107].

Programming language support allows developers to explicitly specify the approximation to do and let the compiler implement them and enforce the constraints. PetaBricks is a programing language which supports *polyalgorithms*, the definition of multiple algorithms computing a common output [4]. The sort algorithm is a good example where the runtime selects between multiple sort implementations (e.g. quicksort or bubblesort) the best one with respect to the data features and size. They incorporated an automatic approximate version generator in the compiler using machine learning with evolutionary algorithm [5]. They relied on the *polyalgorithm* feature to select between the generated approximate versions. The algorithm was perfected to be able to handle variations of the input data with respect to the algorithmic choice [42]. EnerJ provides type system annotations in order to reduce the overall system power draw with the help of approximation techniques [94]. EnerJ also formalizes the semantics of a program executing on approximate data type or code portions to isolate them from ordinary code sections. With Green, developers supply approximate versions and specify the maximum acceptable deviation of result for loops and procedures [7]. Their framework provides a statistical guarantee that the value computed by the application will follow the quality of service requested. Uncertain<T> provides an uncertain data type and operations to manipulate this object [22]. This data type allows the developer to immediately identify approximate computations to allow for better interpretation of the result. Operations on this data type are optimized at runtime based on statistical information of accuracy and efficiency. Rely is a programing language that provides probabilistic reliability guarantees for programs running on unreliable hardware[25]. The compiler takes as input a hardware reliability specification and user-provided function constraints, for example maximum deviation of the function result, and computes whether the program meets all the constraints on the specified hardware or not.

2.2.2 Characterizing Approximation Quality

Whenever approximate computing is used, we also need the ability to measure the precision or quality of the output. There exists multiple quality metrics available in the literature. One of the most common metrics is the relative difference or error with respect to a standard output, i.e. obtained without the use of approximate techniques [116]. Application specific metrics are available, e.g. image and video processing have peak-signal-to-noise-ratio (PSNR) and structural similarity index measure (SSIM) [2], physics-based simulations use energy conservation as a metric [49] and machine learning algorithms are validated against pre-processed datasets [19]. The developer should be able to select the best suited metrics for the considered problem.

2.3 The Polyhedral Model

The polyhedral model is an algebraic representation of programs where instances of a statement inside the application correspond to the points of a $\mathbb{Z}$-polyhedron [47, 48]. Within this model, we can represent a wide variety of programs. The model imposes some restrictions on loop bounds and data accesses: loop bounds, control statements and data accesses must be affine expressions of surrounding loop variables and constant parameters. In this work, we rely on the polyhedral model to transform and create new

optimized versions to produce approximate versions from the original code. In the following section, we introduce the loop statement representation, followed by dependency analysis and code transformation, and finish with the code generation algorithm which translates the representation back to imperative program parts.

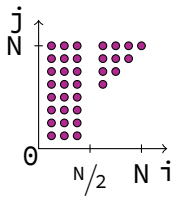
The polyhedral representation embodies the following information for each statement:

- ***The iteration domain*** of the statement, which represents the set of the various executions of the statement. The domain is empty when the statement is not part of a loop nest.
- ***The scheduling*** of the statement, which maps the statement instances to a time domain. Hence, it represents the order in which the various statement instances have to be executed with respect to each other.
- ***The array access polyhedra***, which are the read and write map functions from the iteration domain to array indices. They are used, e.g., for dependency analysis, to assert that transformations made to the schedule are valid.

2.3.1 Iteration Domain

The iteration domain corresponds to the set of coordinates of the polyhedron's points corresponding to the iterator values of a given statement. This polyhedron is constructed using a set of inequalities extracted from control statements (bounds, ifs) of the loop nest. Polyhedral frameworks include tools to extract the iteration domain from programming languages constructions [55, 12, 120]. In Figure 2.5 the C code at the top-left is executing the statement S_1 for each pair of value taken by induction variables (i, j) guarded by the inequations from the loop bounds and the conditional statement at the bottom. This union of inequations represents the iteration domain of the statement S_1.

```
for(int i = 0; i <= N; ++i)
  for(int j = 0; j <= N; ++j)
    if (2 * i > N && j >= i || 2 * i < N)
      A[i][j] = B[i]; // S1
```



$$D_{S_1}(\mathtt{N}) = \left\{ \begin{pmatrix} \mathtt{i} \\ \mathtt{j} \end{pmatrix} \middle| \begin{array}{l} 0 \leq \mathtt{i} \leq \mathtt{N} \\ 0 \leq \mathtt{j} \leq \mathtt{N} \\ 2\mathtt{i} < \mathtt{N} \end{array} \right\} \bigcup \left\{ \begin{pmatrix} \mathtt{i} \\ \mathtt{j} \end{pmatrix} \middle| \begin{array}{l} 0 \leq \mathtt{i} \leq \mathtt{N} \\ i \leq \mathtt{j} \leq \mathtt{N} \\ \mathtt{N} < 2\mathtt{i} \end{array} \right\}$$

Figure 2.5 – **Polyhedral Model Representations of Codes** Multiple abstract representation of a same iteration domain. The source code in the C language is represented at the top-left. The iteration domain is represented in the polyhedral model by a union of two sets of inequation shown at the bottom. The resulting parametrized polyhedron is pictures at the top-right. Each point of the polyhedron represents an instance of the statement S_1 and its coordinates are the values taken by the induction variables (i, j).

2.3.2 Scheduling Function

The scheduling polyhedron is a function, a linear map, from the iteration domain to a multidimensional time domain ($f : D \rightarrow S$). The polyhedron defines a total order between the statement instances. The original loop scheduling function, which corresponds to the lexicographic ordering is the identity function. In our example, each instance of (i, j) is ordered lexicographically as defined by the `for` loop semantics. An instance happens before another one, $(i, j) \prec (i', j')$, if and only if $(i < i') \vee (i = i' \wedge j < j')$. An identity scheduling function indicates that the instances execute in the same order as the dimensions of the domain polyhedron. If two statements maps to the same logical date, they can be executed in any order with respect to each other, or in parallel.

2.3.3 Optimization in the Polyhedral Realm

Dependency analysis is one essential cornerstone of the model. It is used to verify transformation's legality and to assert that the new scheduling will not modify the program semantics with respect to the original one. Dependency analysis in the polyhedral model uses a dependency polyhedra to represent the dependencies between two statements [47, 48, 11]. A dependency between two statements S and T exists if the three following properties are met: Firstly, there must exist an instance $S(\boldsymbol{i})$ and $T(\boldsymbol{j})$ that either read or write to the same memory location M. Secondly, $S(\boldsymbol{i})$ must happen-before $T(\boldsymbol{j})$, meaning that we have $\boldsymbol{i} \prec \boldsymbol{j}$ or $\boldsymbol{i} = \boldsymbol{j}$ and S appears textually before T [104]. We call $\boldsymbol{d} = \boldsymbol{j} - \boldsymbol{i}$ the distance vector of the dependence. Table 2.2 presents the four dependence types.

Table 2.2 – Dependency Types

Name	Notation	Definition
Flow	δ^f	When $S(\boldsymbol{i})$ writes and $T(\boldsymbol{j})$ reads
Anti	δ^a	When $S(\boldsymbol{i})$ reads and $T(\boldsymbol{j})$ writes
Input	δ^i	When both $S(\boldsymbol{i})$ and $T(\boldsymbol{j})$ are reading
Output	δ^o	When both $S(\boldsymbol{i})$ and $T(\boldsymbol{j})$ are writing

Code transformations are linear functions that are used to modify the scheduling polyhedra of the statements. Transformations do not modify the number of instances but their relative order with respect to each other. Thanks to the linearity of the transformations, it is possible to compose them together to create more complex ones. The following example illustrates the function to perform a loop interchange for a two-dimensional loop nest with the dimensions i and j. The matrix represents the interchange loop transformation.

$$\text{Interchange} = \left\{ \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \right\}$$

The same transformation functions are applied to the dependency polyhedron which is formerly checked for the absence of backward dependency, i.e., a value from a statement is used before it has been computed [127]. When there is no such backward dependency and the number of dependency is not changed by the transformation, the transformation

```
1 for (i=0;i<=floord(N-1,2);i++)
2   for (j=0;j<=N;j++)
3     a[i][j] = b[i];
4 for (i=ceild(N+1,2);i<=N;i++)
5   for (j=i;j<=N;j++)
6     a[i][j] = b[i];
```

(a) Lexicographical ordering schedule

```
1 for (j=0;j<=N;j++) {
2   for (i=0;i<=floord(N-1,2);i++)
3     a[i][j] = b[i];
4   for (i=ceild(N+1,2);i<=j;i++)
5     a[i][j] = b[i];
6 }
```

(b) Schedule with a loop-interchange

Listing 2.3 – **Output from a Polyhedral Code Generator** Example of the CLooG code generator output from the polyhedral representation of Figure 2.5. Both generated codes are semantically equivalent to the original version. The left version follows the same execution order as the original one but the inner 'if' guard made the generator split the outer loop in two with the appropriate bounds. In the right version the two loops have been interchanged and therefore the inner loop is split in two.

is legal. Automatic code optimization solves Diophantine equations which are specifically crafted to optimize one or multiple goals, for example to minimize value reuse distance to allow to use the cache more efficiently, move dependencies vectors to make dimensions parallel or make the inner most loop parallel and enforce stride-one memory access for vectorization [55, 21, 10].

2.3.4 From Polyhedral Representation to Imperative Code

Code generation translates the domain polyhedron and its associated schedule to a programming language representation. The task is handled by a code generator, e.g. ClooG, CodeGen or isl [9, 27, 119], that produces an imperative AST from the polyhedral representation. The generated AST scans the iteration domain following the schedule generated in the optimization phase. Listing 2.3 shows the code generated by CLooG for our example code of Figure 2.5 with two different schedules. The Listing 2.3a shows a code following the same schedule as the original program. The dimension i has been split in two loops at lines 1 and 4 corresponding to the left and right convex polyhedra pictured in Figure 2.5 (top right), respectively. Applying a loop interchange to the schedule results in the generated code in Listing 2.3b. The j loop became the outermost loop (line 1) and the i dimension is scanned within it because of the schedule reordering.

Code generation can also optimize the control overhead by lifting, whenever possible, the control statements outside loops. The outer loop splitting in Listing 2.3a or inner loop splitting in Listing 2.3b are examples of such optimization. Another one is the code at the bottom-left of Figure 2.6 which is the result of a code generation pass with CLooG on the original domain and schedule extracted from the code at the top right of the same figure. The generated code does not include empty iterations contrary to the initial version. New loop bounds are used to ensure the non-emptiness of the iteration domain. The graphical representation on the right of the figure pictures the iteration domain of the loop nest. The two lines represent the linear inequalities of the lower and upper loop bounds. The points that are present inside the cone pointing up are the valid instances. The generated loop only visits the points inside the cone whereas the original loop in visits points outside and uses the `if` guard to prevent the statement execution.

The code generation algorithm shifts the control statement of guarded empty iterations upwards, to the upper loop dimensions whenever possible, reducing the total loop overhead. This overhead could represent a significant part of the application's total computation time. Therefore, polyhedral code generation guarantees that the control overhead will be small even for complex polyhedra. Our work benefits from the optimized loop scanning generated by the code generator to produce approximate versions with less overhead (see Section 4.4).

The polyhedral model offers great flexibility at analyzing, transforming and generating efficient programs using a robust mathematical background. We decided to rely on this model to statically analyze the code that the developer marks for optimization.

```
for (int i = 0;i <= N; i++)
 for (int j = i; j <= N; j++)
  if (i + j - P >= 0)
   A[i][j] = B[i + j - P];
```

```
if ((N >= 0) && (2 * N >= P))
 for (i = max(0, P-N); i <= N; i++)
  for (j = max(i, P-i); j <= N; j++)
   A[i][j] = B[i + j - P];
```

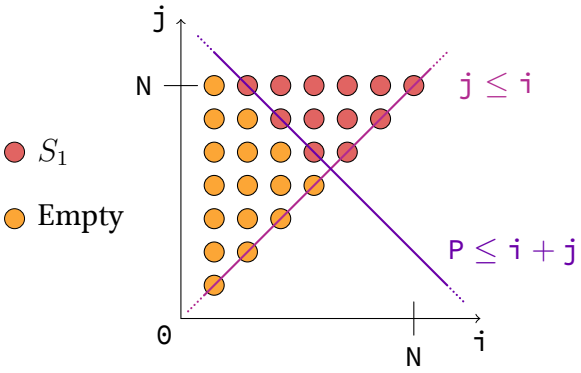


Figure 2.6 – **Control Overhead from Polyhedral Code Generator** The polyhedral model can be used to reduce the control overhead of complex loops. The code at the top-left scans the polyhedron which is represented at its right by the S_1 and `Empty` points. The code at the bottom-left is generated by CLooG and scans only the S_1 points, minimizing the overhead of visiting and checking if a computation should be performed.

Chapter 3

Programming Language Abstractions for Approximate Computing

This chapter introduces the framework interface for adaptive optimization using approximate computing. We motivate our approach based on a language extension set in Section 3.1. In Section 3.2 we provide an overview of the framework, starting with an annotated code through all the steps leading to an adaptive code. Section 3.3 follows with the semantics and grammar of our proposal: Adaptive Code Refinement (`ACR`) annotations. Finally, Section 3.4 introduces an annotation specific to stencil computations which lets the framework find the relevant parameters automatically and relies on `ACR` to generate the adaptive versions.

3.1 Extending Languages to Support Approximation

A programming language specification evolves by addition or modification of the language grammar. This specification creates a common ground that each compiler targeting a language has to respect. This process usually requires the adoption of the community and can take some time. It is sometimes better not to integrate these modifications directly in the language specification but to provide a compiler which can handle the added constructs. It allows language developers to iterate faster on the constructs, which may also break backwards compatibility.

Most compilers provide a way for developers to specify how to behave while processing the input. These constructs, called *directives* or *pragmas*, are not part of the language grammar and provide supplementary information that may be used if the compiler implements them. With directives, the compiler can issue a program with a modified behavior, for example parallelize portions of a `C/C++/Fortran` code on multicore processors using OpenMP annotations or on accelerators with OpenACC [39, 20, 80].

Both of these methods require some support from the compiler, but directives are optional and a compiler which does not implement them should process the source code without error by ignoring them. On the other hand, using an in-language construct which is not recognized by the compiler will inevitably lead to a parsing error. Hence, directives are more flexible and the source code should be compatible with a greater variety of compilers.

Language-Integrated Approximate Computing

In-language extensions add new keywords or constructions inside the language grammar. The software developer may use these new constructions to provide additional semantics to variables or code blocks. The following table shows examples of additional constructions added to programming languages.

Language Construction Type	Examples
Type system enhancement	`APPROX int a = ...` `_Atomic int a = ...`
Function output reliability	`int<0.90*R(x)> f(x){...}`
Control flow keywords	`accuracy_metric ...` `for_enough { do ... }`

In the first example, the type system was enhanced with new keywords. A defined variable can be approximated [93] or accessing the variable be atomic with respect to the C11 extensions [34], i.e. accesses are safe between concurrent threads. The second example specifies a function reliability metric stating that the output precision for the function `f` must be at least as precise at 90% of its input parameter [25]. The last example provides additional control flow constructions where the number of iterations and the stride of a `for` loop are determined by the compiler, at runtime, from an accuracy metric using information from a training dataset [5]. Related works on this subject are described in Section 2.2.

Compiler Directives and Helper Library

Optional language extensions are additions that live alongside the language grammar. The extensions are expressed as annotations, inside a comment section or with special construct that is ignored by the compiler when not recognized. These constructions use a dedicated syntax and can refer to information available inside the program source code, e.g., identifier names and values. For example, OpenMP is a language extension for the `C/C++/Fortran` languages that provides parallelization and task generation extensions [39, 20]. The code snippets in Listing 3.2 and 3.1 achieve a parallel addition of all the values returned by a function and store the result in the variable a. The first version in Listing 3.1 uses OpenMP directives while the second in Listing 3.2 uses a thread spawning library to add every element of an array.

The OpenMP version is less verbose and allows for fast analysis of the main purpose of the algorithm while the version using the library contains more implementation details due to the library interface. Internally, OpenMP may transparently use different optimized reduction strategies. Exploiting of existing libraries is not a problem when annotations are used as the compiler can issue a code targeting them. Hence, the code in Listing 3.1 may compile to the same code as in Listing 3.2.

Optional language extensions give flexibility to the developer. They can be ignored by compilers that do not implement the extensions or be disabled at compile time, e.g., using a compilation flag. On the other hand, a missing library or unknown type / construct will raise a compilation error. Specialized libraries can reduce the maintainability,

```
1 float a = 0.f;
2 #pragma omp parallel for reduction(+:a)
3 for (i = 0; i < N; ++i)
4     a += f(i);
```

Listing 3.1 – **Parallel reduction of an array using the OpenMP directives** The "omp parallel" directive states that the following loop can safely be parallelized and the "reduction(+:a)" indicates that we want to do a reduction using the addition operator on the variable `a`. The final result is stored in the said variable. The code is a valid program without the OpenMP annotation or compiler support. It will execute sequentially in that case.

```
1 void reduceChunk(id, res){
2   fst = id*tNum;
3   lst = min(fst+tNum, N)
4   res[id] = 0.;
5   for(i=fst; i<lst; ++i)
6     res[id] += f(i);
7 }
```

```
8  float a = 0.f;
9  float res[ceil(N/tNum)];
10 for(id=0; id < ceil(N/tNum); ++id)
11   thread(reduceChunk, id, res);
12 waitAllThreads();
13 for(id=0; id < ceil(N/tNum); ++id)
14   a += res[i];
```

Listing 3.2 – **Parallel reduction of an array using a thread spawning library** Each thread spawned by the code on the right is given a unique identifier `id`, and a chunk of the initial array to reduce via the `reduceChunk` function (left). The master thread waits for each spawned threads to terminate and finishes the job using the intermediary results.

debugging and optimization potential of complex programs, e.g., interprocedural, global and code inlining optimizations are not possible for dynamic libraries. Language extensions are less invasive because code transformation is done automatically by the compiler. Optional extensions may be more verbose than integrated ones, but they share a common interface between many programming languages. Therefore, optional language extensions are the most suited abstraction for our adaptive approximation API based on the language agnostic polyhedral representation of programs.

3.2 Static Dynamic Adaptive Strategy

No current programming language, to the best of our knowledge, does provide ways to express adaptive computation. While approximate computing techniques are starting to get adoption in compilers, they treat data as a black box, i.e., one entity with some parameters attached and want to minimize the error while maximizing the performance or the energy consumption (see Section 2.2.1). Our technique aims to retrofit adaptive techniques inside compilers with an interface that can be used by non-expert in approximate computing. Our proposal is the Adaptive Code Refinement framework (ACR).

An example application of ACR on a part of a fluid simulation code, i.e., a typical target for our framework, is shown in Listing 3.3 (studied in detail in Section 7.1.2). In this code, ACR annotations tell the compiler how to exploit approximation for that specific kernel. The adaptive opportunity for this application resides in the iterative solver shown

in that kernel. This solver requires a number of iterations to have a converged solution. However, this number does not need to be homogeneously high for the whole data space. As a consequence of inserting the ACR annotations, our system will dynamically monitor application's data to apply convenient approximation levels to different parts of the data space. For instance, Figure 3.1 shows a visualization of the simulation at a given point. The left side of the figure shows the density of the fluid. The right side of the figure shows the view of our compiler, which partitions the data space into cells and dynamically evaluate the approximation level of each cell. Dark cells need a low number of iterations to converge while brighter cells need a high number of iterations to converge. The ACR annotations lines 12 to 19 will be explained in depth in the remainder of that chapter.

```
1  static inline unsigned char density_val(float a) {
2    if (a < 0.1f) return 2;
3    if (a < 2.f) return 1;
4    return 0;
5  }
6
7  void lin_solve (int M, int N,
8                  float x[M][N],
9                  float x0[M][N],
10                 float a, float c) {
11
12 #pragma acr grid(50)
13 #pragma acr monitor(x[i][j], min, density_val)
14 #pragma acr alternative high(parameter, P = 6)
15 #pragma acr alternative medium(parameter, P = 4)
16 #pragma acr alternative low(parameter, P = 1)
17 #pragma acr strategy dynamic(0, high)
18 #pragma acr strategy dynamic(1, medium)
19 #pragma acr strategy dynamic(2, low)
20   for (int k=0 ; k < P ; k++ ) {
21     for (int i = 1; i < M-1; ++i)
22       for (int j = 1; j < N-1; ++j)
23         x[i][j] =
24           ( x0[i][j] +
25             a * (x[i-1][ j ] +
26                  x[i+1][ j ] +
27                  x[ i ][j-1] +
28                  x[ i ][j+1])
29           ) / c;
30     set_bnd(M, N, x);
31   }
32 }
```

Listing 3.3 – **Fluid Simulation Kernel** A Gauss-Seidel iterative solver of linear equations. Three levels of approximation are used: low, medium and high which map to modified number of iterations of the solver. The number of iterations of the solver is lowered by the runtime in the cells where the density is low or medium.

The general view of our framework is depicted in Figure 3.2 and decomposed in two sections: Firstly, we have the source to source compiler. It takes as input a source code with ACR annotations (to be detailed in Section 3.3). This source code is parsed to extract the information expressed using the annotations and stores them inside in an abstract syntax tree for later use. At the same time we extract the polyhedral model representa-

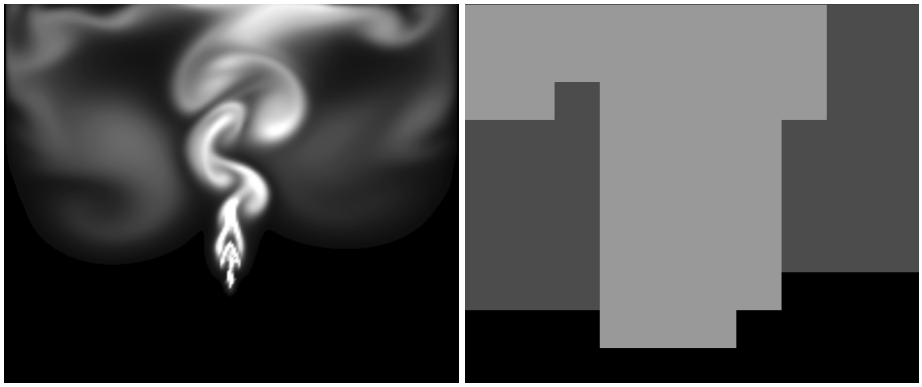


Figure 3.1 – **Eulerian Fluid Simulation** Simulation of a fluid flow with a source near the center of the simulation space. The fluid is injected with a velocity pointing top. The left image shows the density of the fluid, white being denser than black. The right image shows the adaptive grid view of the compiler from the monitoring of the data. The light-gray zone uses the original kernel, the dark-gray zone an approximate version, and the black zone an even more aggressive approximation than dark-gray.

tion of the kernel, e.g., using the clan library into an OpenScop representation [12]. The compiler generates and includes the approximate versions alongside the original version in the source code. It also adds the necessary machinery to gather runtime information and dynamically select the version to be used. The output file is entirely written in the input programming language and can be compiled to machine code.

Secondly, the executable loads the `ACR` library which provides the helper functions used by the compiler to create the adaptive version. The first time the kernel is reached, the runtime for this kernel is initialized. The data structure to store the precision required by the various parts of the data space is allocated and is transmitted to the monitor along with the address of the data to monitor. In the case of dynamic compilation, the coordinator thread is spawned to initiate the helper threads (to be detailed with code generation and runtime in Section 4.4). At the end of the initialization phase, a non-approximate kernel is executed to bootstrap the mechanism with approximation levels. For each subsequent execution of the kernel, the relevant alternative is executed and a monitoring mechanism updates its precision map. A reset of a kernel runtime can be initiated by the application with a special annotation or triggers when the program ends.

3.3 Programming Language Annotations for Approximate Computing

Programming languages are the main tools to create programs. The language syntax and external libraries provide convenient abstractions to build or optimize programs. We reviewed three methods to extend an existing language to support adaptive approximation: in-language constructs, libraries and optional language extensions. We analyzed in Section 3.1 the advantages and drawbacks between the three approaches and motivated our choice for the language extension. In the following sections we present the `ACR` annotations. The whole set of annotations is specified by Grammar 1. In a nutshell, they provide

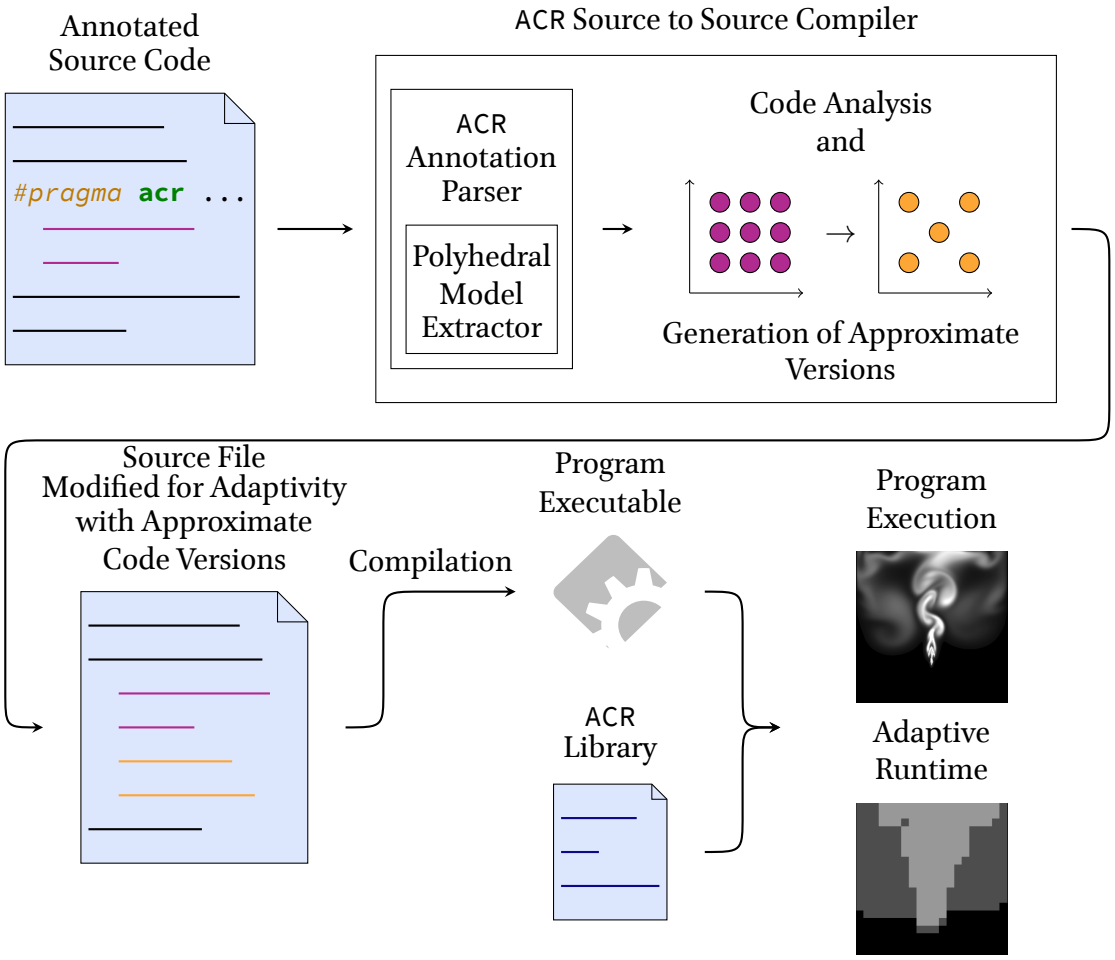


Figure 3.2 – **Compilation Process for ACR** The framework consists in a source to source compiler and a runtime library. The annotated source code is given to the compiler which extracts the annotation information and the polyhedral representation of the kernel. The compiler generates approximate versions of the kernel and adds the necessary instructions to select a version during the execution of the program. The generated adaptive source is compiled with the compiler for the programming language of the source file. An ACR helper library is loaded at runtime to provide common functions issued by the compiler to achieve adaptiveness.

the compiler with the information needed to generate an adaptive version: the granularity of the decision (Section 3.3.1), how to generate approximate versions (Section 3.3.2) and how to select where to use approximations dynamically (Section 3.3.3).

3.3.1 Annotation for Adaptation Granularity

The granularity specifies the level at which approximate code versions are selected. It corresponds to the ⟨*grid-option*⟩ grammar rule in Grammar 1. With this *grid* construct, the data array is subdivided into *cells* using a "data-celling" transformation, which is the equivalent of a tiling for the data [126, 125]. The transformation can be viewed as a function from $\mathbb{Z}^n \rightarrow \mathbb{Z}^{2n}$ where the first n dimensions are the cell index and the last n dimensions the data cells associated with that index. Hence, the technique subdivides the initial data domain into cells which can be selected individually. The size of the cell used for the data-celling technique is selected with the *grid* construct line 12 in Listing 3.3. The same size in every direction, i.e., an n-dimensional cube [38]. The effect of this annotation is visible in Figure 3.1 right. The white, gray and dark areas are composed of many smaller cells, and some of them are visible at the interface with the white region at the top left and bottom right.

A more generic meshing procedure using piecewise affine function could complement the data-celling. Piecewise affine functions allow for non-structured grids, e.g. a rectilinear or curvilinear grid, but usually requires a grid generator algorithm [115]. For the time being we only consider structured Cartesian grids in our API. A data-celling size of one is used whenever data share no similarities. The decision is then made per data basis.

3.3.2 Annotation for Code Transformation

In this section we present the language extension API to help building applications exploiting approximate computing techniques. Our API can generate various approximate versions that can be ordered from the least approximative to the most approximative. We present a tool called "Adaptive Code Refinement" (ACR) which uses a directive set to generate approximate and adaptive versions of annotated computation kernels.

In ACR, the dedicated extensions to specify a collection of means to achieve approximate computing is the *alternative* construct. Multiple approximate versions can be defined for the code of interest. The *alternative* construct is described in Grammar 1 with the ⟨*alternative-option*⟩ grammar rule. A simplified notation is:
`#pragma acr alternative` *name*`(`*type*`,` *type specific parameter*(*s*)`)`
The "*name*" input provides a name to the alternative using C identifier notation. Other constructs can reference this name to select a particular code alternative. This name is used to link with the strategy selection mechanism (Section 3.3.3). The first parameter, type, indicates which transformation is used to generate the approximate alternative code. To define an alternative, the developer may choose between the five following options:

- With *parameter*, a constant parameter used inside the target loop nest will be replaced with the value of another parameter or a constant value specified by the developer. The alternative parameter indicates a change in a parameter value. This

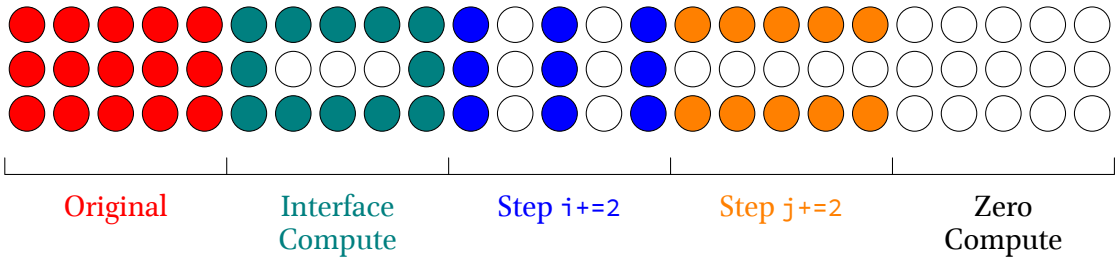


Figure 3.3 – **ACR Alternative Effects on the Polyhedral Representation** Graphical representation of a two-dimensional domain with several alternatives applied. The domain consists of five cells delimited by the black-dashed line. Each cell consists of fifteen iteration instances represented as points. An iteration is executed if the corresponding point is filled and not executed otherwise. With `interface compute` only the instructions at the borders are kept. `Zero compute` disables them all. `Step` modifies the step count for the first or second dimension depending on the position of the annotation.

can result in loop perforation when the parameter is a loop bound or loop induction increment [106]. It may also help for task skipping, when the parameter is part of guard statement and acting as an activation parameter. This condition can be optimized by the polyhedral compiler.

- With *code*, the alternative parameter defines as an external function or code block. The developer can implement multiple algorithms for a given problem and name them as different alternatives. This provides the flexibility of code multi-versioning while leaving the compiler with the care of generating the runtime mechanism. Memory access skipping can be implemented by replacing a memory access instruction with an interpolating function in the alternative code [96].

- With *zero-compute*, the target computation is removed and will not be executed. There is no alternative parameter required. This construct can be used to implement loop perforation and task skipping at the cell granularity.

- With *interface-compute*, the points of the kernel's iteration domain that do not intersect with the cell mesh are not visited, i.e. only the iterations at the border of a cell are visited. This allows stencil-like computation to be implemented in a more natural fashion (example in Section 7.2.3). This alternative can take the size of the interface as parameter.

- With *step*, the specified loop counter increment is modified according to the specified parameter (e.g. `i++` is replaced with `i+=step`). This construct can be used to implement loop perforation in a particular case of *interface-compute*.

Figure 3.3 shows a graphical representation of the last three aforementioned alternatives. These alternatives selectively reduce the amount of iterations inside the cells, i.e.

the number and position of the empty points. The *code* alternative modifies the code executed for the remaining colored instances of the cells. The *parameter* alternative applies a modified constant parameter value for every statement in the kernel.

Each alternative except *parameter* can be defined on a particular statement, code block or loop nest. The *parameter* alternative can only be positioned at the kernel top-level, because parameters are constants for the kernel's duration. By positioning the code annotation at the right place inside the kernel, the developer can selectively apply the alternatives. Generating the alternative from user transformation specification provides the compiler with a set of approximate versions to adjust the intensity of computation. This is the first step to provide adaptive capabilities to the application.

For the fluid solver application, three alternatives have been defined at lines 14–16 of Listing 3.3. To adapt the computation we choose to lower the number of iteration of the solver by positioning the `P` parameter (line 21) to values 6, 4 and 1. The names given to the alternatives reflect the relative amount of iteration between the three of them.

3.3.3 Annotations for Adaptive Decision Making

The ACR optimized kernel is composed of several *alternative* versions of the initial kernel. With the *grid* annotation we know at which granularity to take the decision, but not where to extract the information from and how we should interpret them. In this section we introduce the ACR constructs which provide the decision mechanism.

Monitoring: Getting Approximation Level for Each Cell

How to collect information from runtime data is declared to the compiler in the form of a multidimensional array access within the ⟨*monitor-option*⟩ rule. A simple example of the *monitor* extension format, as seen in Figure 3.4, is:
`#pragma acr monitor(`*data*$[f(\vec{i})]$`)`

The monitor's grammar ⟨*array-access*⟩ in Grammar 1 declares an array where $f(\vec{i})$ is its data access function. Our framework constrains the dimensions of the array specification to form an affine map between the iteration space and the data space. Hence, they must be a linear function of the parameters, constants and iterators available inside the optimized kernel. The data domain does not necessarily need to match the loop's iteration domain. For example, in Figure 3.4, the iteration and data domains do not match. Our system relies on polyhedral techniques to recover the iteration space from the data space by computing the preimage or inverse image [110]. The statement instances that access the array `A` in Figure 3.4 are pictured with the same color on the data and iteration domains.

Approximate computing techniques can be applied at different granularity levels: the same approximation everywhere, a different approximation for each operation or an in-between where we consider sets of operations with different approximation. Our goal is to provide the compiler with the information it needs to use the right approximation at the right moment during the execution of the application. Cell-level decisions implicitly require the retrieval of the most conservative level of approximation from the cell state during the program's execution. The components of each cell are aggregated using the ⟨*folding-function*⟩ attribute of the *monitor* construct. The folding function defines a mapping from multiple all the data values of a cell to a natural integer (with prototype

Grammar 1: Adaptive Code Refinement Grammar The ACR grammar declares easy to use language extensions for approximate computing. The *alternative* defines a modification of the code to express various approximate blocks or statements. The *strategy* links these alternatives to a constant value for adaptive selection. The *monitor* specifies which data will be considered to take a decision. The *grid* defines the granularity at which the decision is made.

⟨*pragma-defininion*⟩ ::= #pragma acr ⟨*acr-option*⟩

⟨*acr-option*⟩ ::= grid ⟨*grid-option*⟩
| monitor ⟨*monitor-option*⟩
| strategy ⟨*strategy-option*⟩
| alternative ⟨*alternative-option*⟩
| checker-select (⟨*checker*⟩ , sync)
| checker-select (⟨*checker*⟩ , async)

⟨*grid-option*⟩ ::= (⟨*positive-integer-list*⟩) | none

⟨*monitor-option*⟩ ::= (⟨*data-domain*⟩)
| (⟨*data-domain*⟩ , ⟨*folding-function*⟩ , ⟨*pre-processing*⟩ , ⟨*post-processing*⟩)
| (⟨*data-domain*⟩ , ⟨*folding-function*⟩ , ⟨*pre-processing*⟩ , ⟨*post-processing*⟩ , ⟨*init-fold-val*⟩)

⟨*data-domain*⟩ ::= ⟨*array-identifier*⟩ ⟨*array-dim*⟩

⟨*array-dim*⟩ ::= [⟨*induction-var*⟩ | ⟨*param-id*⟩ | ⟨*constant*⟩]
| ⟨*array-dim*⟩ [⟨*induction-var*⟩ | ⟨*param-id*⟩ | ⟨*constant*⟩]

⟨*folding-function*⟩ ::= min | max | mean | std-deriv
| ⟨*function-pointer*⟩

⟨*alternative-option*⟩ ::= ⟨*alternative-id*⟩ (parameter , ⟨*identifier*⟩ = ⟨*const-val*⟩)
| ⟨*alternative-id*⟩ (parameter , ⟨*identifier*⟩ = ⟨*identifier*⟩)
| ⟨*alternative-id*⟩ (code , ⟨*inline-code*⟩)
| ⟨*alternative-id*⟩ (code , ⟨*function-to-be-replaced*⟩ = ⟨*replacement-function*⟩)
| ⟨*alternative-id*⟩ (zero-compute)
| ⟨*alternative-id*⟩ (interface-compute , ⟨*constant*⟩)
| ⟨*alternative-id*⟩ (step , ⟨*const-val*⟩)

⟨*strategy-option*⟩ ::= dynamic (⟨*constant*⟩ , ⟨*alternative-id*⟩)
| dynamic (⟨*constant*⟩ , ⟨*constant*⟩ , ⟨*alternative-id*⟩)
| static (global , ⟨*alternative-id*⟩)
| static (⟨*area*⟩ , ⟨*alternative-id*⟩)

⟨*checker*⟩ ::= raw | versioning | stencil

```
#pragma acr monitor (A[j][i])
for (size_t i=0; i<=P; i++)
 for (size_t j=0; j<=Q; j++)
   A[j][i] = ...
```

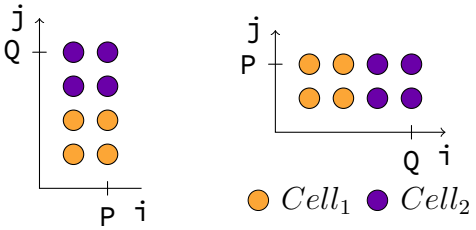


Figure 3.4 – **Data Domain and Iteration Domain** Example of a loop nest and its accompanying iteration domain (center) and data domain (right) when `P` $<$ `Q`. The iteration domain is defined by two nested loops with i being the outer dimension and j the inner dimension. Hence, the polyhedral representation at the center. The A array accesses use the iterators in the reverse order, which is equivalent to a $x = y$ line symmetry applied on the iteration domain, and creates the polyhedral representation on the right. The two colors correspond to two cells resulting from the *grid* annotation. The iterations of one color are accessing the data with the same color in the data domain.

`int fold(datatype, int)`). This integer represents the *approximation level*, i.e., the maximum level of approximation that can be used while maintaining a good quality of result. The approximation level 0 corresponds to the original kernel and subsequent positive integers maps to increasing levels of approximation. The ⟨*preprocessing-function*⟩ is applied on every component of a cell before passing the value to the folding-function. The preprocessing allows an application-specific data type to be used along with the ACR predefined *folding functions*. If not required, the preprocessing function can be set as the identity function. The ⟨*init-fold-value*⟩ may be specified to initialize the value of the fold, otherwise we use zero. This initial value allows for better flexibility with the folding function implementation and usage, e.g., use of the `min` folding function with an initial value of zero will always return zero.

Strategy: Linking Data Regions or Approximation Levels to Code Versions

Adaptiveness means the capability to choose the precision dynamically at runtime depending on the state of the data. Several *strategies* are possible and we have to select the most convenient approximate computing alternative for each data cell. Two categories of strategies can be used. The *static* category specifies non-adaptive portion of the data that will unconditionally be computed using a specified version of the code. The *dynamic* category selects a version depending on the runtime context.

With a *static* strategy, the compiler generates an optimized code at compile time with approximated computation embedded inside the kernel. The approximation remains constant during the whole application execution. The regions to be approximated are marked by the developer with the ⟨*strategy-option*⟩ *static* construct. The approximation strategy can either be global, i.e., covering the entire data space, or be defined on a localized area. In this case the area is specified with the set of iterator values corresponding to the desired data domain. We rely on the widely used isl or Omega notations [119, 66, 67]. For instance, to select half the data space of an array `data[i][j]`, where both `i` and `j` span from `0` to `N`, we may specify $[\mathrm{N}] \rightarrow \{[\mathrm{i},\mathrm{j}] : 0 <= \mathrm{i} <= \mathrm{N}/2 \, and \, 0 <= \mathrm{j} <= \mathrm{N}\}$.

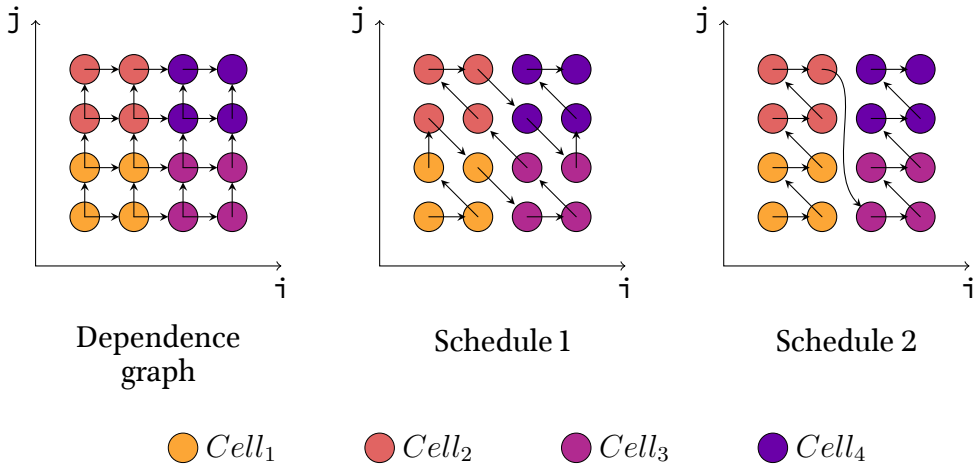


Figure 3.5 – **Loop Dependency** Example of loop carried dependency between iteration instances with two valid schedules. A point represents the execution of one statement instance. For the dependence graph, an arrow represents a dependence: some instances have to be executed before the target instances. For the schedules, the arrows represent the execution order. Schedule 1 exhibits the maximum parallelism at iteration level whereas schedule 2 focuses on parallelism at the cell level. The code generated with the first schedule requires more cell-transition checks than it would with the second one.

With *dynamic* strategies, the approximation may vary at runtime depending on the state of the data. In this case, the approximation level is mapped to an approximate code version: every cell with the same approximation level will be processed using the same approximate code version. In ACR, this adaptive capability is inspired by adaptive mesh refinement (AMR) [16, 15]. AMR technique defines a multi-level dynamically changing grid over the data domain. The deepest levels typically contain more data points than the outer levels and are located over active regions. This allows for a finer grained grid, i.e. more precise, only on the regions where it matters.

Example

In the fluid simulation example, the *monitor* construct is defined at line 13 of Listing 3.3. The annotation states that the adaptive information has to be extracted from a 2D array x of size parametrized by the lower and upper bounds of the induction variables i and j. During the program execution, the monitoring provides, for each cell, an integer value representing the approximation level allowed with information gathered from the application data. The function `density_val` is used to pre-process the raw application data to an integer with a threshold. The folding function extracts the minimum (`min`) from a cell of 50×50 values, defined by the *grid* annotation line 12. The minimum selects the most precise version from all the approximation levels of a cell. The *dynamic* option of the *strategy* annotations lines 17–19 links an approximation level with a defined *alternative*. An *alternative* can be linked by one or more *strategy* provided the following condition is met: The *code*, *zero-compute*, *interface-compute* and *step* alternatives cannot be nested when linked to the same strategy, because a composition between them would

void the semantics of individual transformation.

Thanks to polyhedral techniques, the compiler computes the affine mapping from the data space specified by the monitoring to the iteration space together with the transformation from the data space to a cell coordinate. Hence, the compiler can instrument the code to update the precision at the cell level whenever a modification of the monitored cell data occurs.

ACR does not allow the compiler to break data dependencies between the remaining iteration instances to limit the deviation of the results. When the loop carries a dependency between iterations, the potential schedules are constrained. In the context of `ACR`, this constrains the order in which the iterations are scanned. If automatic parallelization is applied, further dependency relaxation could unlock more parallelism at the price of a larger deviation. For instance, Figure 3.5 presents a dependence graph that shows a dependency carried between the monitored cells. In this example, we propose two valid schedules with different properties. A close look at the first schedule and the dependence graph reveals that the computations could be parallelized within the diagonals (wavefront parallelism [71]). This schedule, when diagonally parallelized depicts the maximum parallelism for this particular dependence graph. The second schedule, on the other hand, allows the *diagonal cells* to be parallelized (wavefront parallelism of the cells). Hence, we trade a one parallelism level for nested parallelism with better data locality. The best performing schedule depends on the computing platform properties such as cache, pipeline, memory bandwidth and prefetching algorithms.

3.4 Adaptive Stencil Approximation

With ACR, the developer inputs approximation-related information by means of annotations to let the compiler generate an adaptive program version. We studied the kind of applications where approximate computing can be beneficial and a significant portion of them are using stencil computation. A stencil is a type of computation where the update of a value depends on its neighbors according to a fixed pattern. Figure 3.6 shows an example of a two-dimensional stencil on a Cartesian grid. In this example, the neighbors at distance 1, 2 or 3 are used to update the central value.

Adaptive Stencil Approximation is a specialized application programming interface for adaptive computing targeted at stencil computation. In this section we introduce the annotation and show how to discover the granularity and discover the data of interest. The approximate stencil alternatives and strategies are discussed in Section 5.1.

3.4.1 Annotation for Stencil Computation

Using approximate computing either requires deep knowledge of the user in this field or, if automated, ask the user to delegate the transformation to the compiler. Each of these two approaches has its pros and cons. An expert developer may select and implement the best approximation technique suitable for his problem. On the other hand, an automatic method may discover potential approximation targets using binary error injection, using genetic/machine learning algorithms or using pattern matching to find a suitable approximation strategy to be applied to the original algorithm [29, 5, 90]. The goal is to lower the program's computation footprint while keeping the program's output deviation

below a user-defined threshold. The deviation is defined as the l^∞ norm of the difference between two states corresponding to the same data, i.e., $\max(|\mathbf{state}_i - \mathbf{state}'_i|)$.

Grammar 2 provides the details of the ASA annotation. Using ASA only requires one annotation with one parameter: the deviation allowed to use an approximate version. All the other parameters are extracted automatically by the compiler. The deviation parameter is application specific and, in order to be input agnostic, it must be provided by the developer. The developer knows how much approximation is reasonable to obtain meaningful application results.

Grammar 2: Adaptive Stencil Approximation grammar ASA specifies an annotation where most of the approximation-related parameters are deduced by the compiler. The remaining parameter defines the upper bound of the absolute deviation allowed between two invocation of the annotated kernel. The deviation is defined by the difference between the approximate and the original kernel results.

⟨*pragma-defininion*⟩ ::= #pragma asa (maximum-deviation)

In Chapter 5 we provide a new method that automatically extracts adaptive approximation information from an annotated portion of a program. Our method also allows interaction between the user and the compiler to select the best alternatives. Automatic stencil approximation (ASA) relies on the polyhedral model to analyze the code.

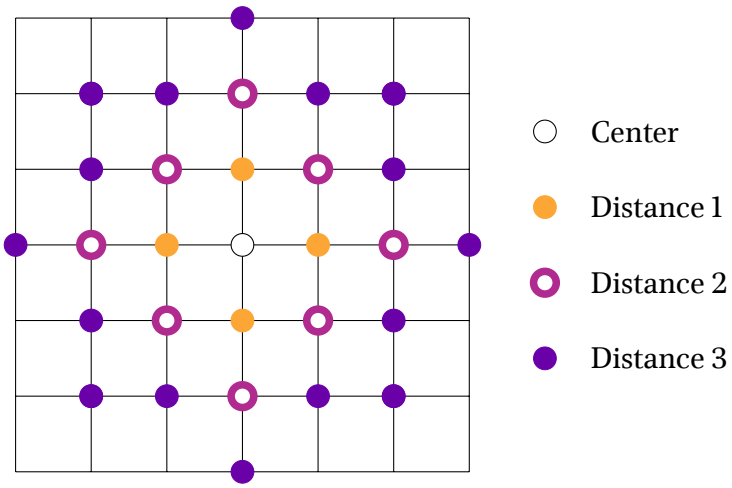


Figure 3.6 – Two-dimensional von Neumann stencil halo on a Cartesian grid.

Chapter 4

Adaptive Code Generation

An adaptive technique dynamically varies the level of precision to reduce the number of computations in locations where approximations are beneficial. Chapter 3 introduces the ACR programming interface to help developers to specify an adaptive version of their programs. In this chapter we present how to raise annotated kernels to an abstract representation including polyhedral and ACR information (Section 4.1). Then, we detail our algorithms to apply approximation related transformations (Section 4.2), to generate an adaptive version at compile time (Section 4.3) and using just-in-time compilation (Section 4.4) to generate specialized kernel versions. We explain how the *monitoring* is implemented and how the runtime-gathered information drives the approximate code selection. We close the chapter with a discussion about conjoint utilization of polyhedral code optimization and adaptive optimization (Section 4.5).

4.1 Raising Annotated Kernels

The first step of the compilation process raises the code to optimize to the polyhedral representation (see Section 2.3). Clan and pet are state-of-the-art front-end tools that transform a high-level language into an equivalent polyhedral representation [12, 120]. Clan is a standalone application and library and pet is a library integrated into clang, the `C/C++` LLVM compiler suite's front end. The output format consists in information on parameters and on every statement present in the kernel. Each statement $S_i \in S$, where S is the set of kernel statements, holds the following associated information gathered from the input program:

- The context polyhedron, which is the information gathered about the parameters
- The iteration domain
- The scheduling
- The array access polyhedra
- The statement itself, which is stored in an AST or string format. This is the part that will be copied back in place once the loop control has been generated.

During the compiler front-end execution, the language extensions are included in the program's abstract syntax tree [35]. After the AST construction, the statements in relation with an alternative are marked using a depth-first AST traversal. This is done with the recursive function in Algorithm 3, which manages a state that contains the active alternative set. Every time an alternative node is encountered, it is added to the state and will be removed before the function returns. When a node representing a statement S_i is visited, it is marked with the alternatives present in the state along with the strategies, grid and monitoring information into a data structure for future processing.

Algorithm 3: Link Alternative Annotations to Statements Mark the statements with the alternatives defined as language annotations stored in the abstract syntax tree. An alternative annotation applies to the block or statement that follows. Hence, if we encounter a statement we store the alternatives, otherwise we must keep track of the defined alternatives in a set and take care of resetting the set when a block or statement is visited.

```
Function markStmtAlts(AST, Alts)
    ACRpragmas ← ∅;
    switch type(AST) do
        case Statement do
            store_alternatives(AST, Alts);
        case Alternative Annotation as alt do
            newAltSet ← Alts ∪ alt;
            foreach Childrens of AST as chld do
                ACRpragmas ←
                  ACRpragmas ∪ markStmtAlts(chld, newAltSet);
        case Non-Alternative ACR Annotation as alt do
            ACRpragmas ← ACRpragmas ∪ alt;
        case List do
            altSet ← Alts;
            foreach Element in List as elem do
                if type(elem) is Alternative Annotation as alt then
                    altSet ← altSet ∪ alt;
                else
                    ACRpragmas ←
                      ACRpragmas ∪ markStmtAlts(elem, altSet);
                    altSet ← Alts;
        case Other do
            foreach Childrens of AST as chld do
                ACRpragmas ← ACRpragmas ∪ markStmtAlts(chld, Alts);
    return ACRpragmas
```

4.2 Generating Alternatives

Each statement is marked with the alternatives that define the transformations to be applied to generate the approximate version. In this section we introduce the code transformation algorithm used by the compile time code generator (Section 4.3) and the just-in-time generator (Section 4.4).

The user-defined alternatives drive the alteration of the computation kernels. Algorithm 4 describes how the statements are modified according to the alternative types. The function *ApplyAlt* takes a statement S_i along with an alternative definition and generates a new statement corresponding to the user alternative definition. The first two cases of the function match the parameter modification. If the user defined a constant value, the parameter value is set to the new value inside the context polyhedron. Otherwise, only the name of the parameter will be altered. The change of the code statement is handled by a replacement of the original code in the compiler internal statement data structure. For *zero-compute*, the iteration domain is set to an empty domain, i.e. corresponding to a statement that will never be executed. For *interface-compute*, to generate the grid borders, we add a new dimension D^n without any constraint and set the dimensions D^j to be a multiple of each point $a \in D^n$ times the `GridSize`. We project the newly created dimension D^n out and what remains in D^j is the multiple of the grid size. The operation is repeated with `GridSize` -1 to capture the left border of the cells. The *step* is handled in the same way as *interface-compute* but with a multiple of the step instead of the grid size. The second function, `ApplyAllAlt`, iterates over all alternatives linked to a strategy and applies the alternatives in the order in which they are defined.

The following helper functions are used by the code generation algorithm. These functions are implementations of regular search algorithms and linear algebra functions found in the literature [104].

`DimsRelatedToData` takes a domain and returns the set of dimension identifier in relation to the data. This is done by application of the preimage of the monitor array access matrix.

`AddDimEq` takes a domain D, a dimension identifier D^j and a parametric constant C. It adds a dimension D^n to D and the constraint $D^j = C \times D^n$ to D ($\mathbb{Z}^n \rightarrow \mathbb{Z}^{n+1}$).

`ProjectOut` takes a domain and set of dimension identifiers D^{id} and projects these dimensions out ($\mathbb{Z}^n \rightarrow \mathbb{Z}^{n-|D^{id}|}$).

`StrategyAlts` takes a strategy and returns the alternatives related to this strategy.

Algorithm 4 applies each transformation to the polyhedral representation. The *zero-compute*, *interface-compute* and *step* transformations reduce (or may maintain) the number of instances scanned by the loop. Hence, these transformations generate an approximate version when the *Step* is greater than the loop stride and the grid size is bigger than two for *interface-compute*. For the *parameter* transformation, we need more information than what is available at compile time to prove that it generates an approximation. The analysis can be performed at runtime, once the parameters are known, to count the number of points in the original and alternative polytopes [121, 31, 32]. The number of points gives an indication about the amount of memory accesses and computations,

which can be used to estimate the execution time. For these transformations, a warning can be issued to the user if it is proved that the iteration count of the loop is not lower than the original version, avoiding the need for debugging the adaptive program. The *code* alternative can be compared with the counting method mentioned before if both codes can be analyzed in the polyhedral model. Static complexity analysis is another option to compare the codes, however, such analysis is complex [124]. The state of the art in this domain can analyze Ocaml functions and output a multivariate resource polynomials which are functions of size parameters that depend on Ocaml types [61]. The last option is to monitor the function at runtime and to issue a warning if the supposedly approximate version takes more time to execute than the original version.

Algorithm 4: Alternative Application Algorithm Algorithm to apply the different alternatives to a statement S_i. The parameter alternatives are handled inside the context. The statement code, which is stored as a string literal or AST, is replaced if needed. **Zero** and **interface-compute** alternatives do modifications on the iteration domain for the dimensions in relation to the monitored data map function.

Function `ApplyAlt`(S_i, Alt)
- **switch** *Alt* **do**
  - **case** *parameter, Constant* **do**
    - Set parameter value in S_i context;
  - **case** *parameter, Identifier* **do**
    - Change parameter name in S_i data structure;
  - **case** *code* **do**
    - Change code statement in S_i data structure;
  - **case** *zero-compute* **do**
    - $S_i \leftarrow \emptyset$;
  - **case** *interface-compute* **do**
    - **forall** $S_i^j \in \text{DimsRelatedToData}(S_i)$ **do**
      - $S_i \leftarrow \text{AddDimEq}(S_i, S_i^j, GridSize)$;
      - $S_i \leftarrow \text{ProjectOut}(S_i, \text{LastDim})$;
      - $S_i \leftarrow \text{AddDimEq}(S_i, S_i^j, GridSize - 1)$;
      - $S_i \leftarrow \text{ProjectOut}(S_i, \text{LastDim})$;
  - **case** *step* **do**
    - Similar to *interface-compute* with *step* instead of $GridSize$;
- **return** S_i

Function `ApplyAllAlt`$(S_i, \text{Strategy})$
- **forall** $\text{Alt}_i \in \text{StrategyAlts}(\text{Strategy})$ **do**
  - $S_i \leftarrow \text{ApplyAlt}(S_i, \text{Alt}_i)$;
- **return** S_i

4.3 Compile Time Adaptive Program Generation

In Section 4.2 we introduced the algorithm used to transform the polyhedral representation of the kernel from a set of alternatives. In this section we use the annotations extracted from the kernel to generate an adaptive version of the program at compile time.

The generation of an adaptive version at compile time requires that a data-celling yields a valid schedule for the iteration domain. The algorithm follows these three steps: Firstly, the data domain is data-celled according to the grid annotation. The inverse image of the celled data domain is computed to obtain the iteration domain. Finally, the transformed iteration domain is checked for the absence of backward dependencies. If the last check is returns an empty set of backward dependencies, the adaptive version can be generated at compile time.

Algorithm 5 transforms the polyhedral representation of the kernel statements according to the strategies. The algorithm can be decomposed in three phases:

The first phase iterates through the kernel statements $S_i \in S$ and generates two statements from each original statement. The statement S'_i contains the static-defined strategy iteration domains mapped from the data space (see Section 4.2. The statement S''_i domain corresponds to the remainder of the initial statement's domain without these static areas. The alternatives are applied to S'_i without adding monitoring nor selection mechanism, locking these statement with the user selected alternative. This statement is stored in a separate set for future use. The second statement, S''_i, is stored in a statement set used in the second phase. It is worth noting that a statement with an empty domain will not generate any code and can be removed at any point of this algorithm. Figure 4.1 shows a graphical application of Algorithm 5 on an example. This first phase is represented with the partitioning of the static and dynamic domains. The static domain is transformed by the *otherCode* alternative.

The second phase handles the dynamic strategies. For the dynamic version, the goal is to generate a kernel that scans the monitored cells one independently. This allows for a code with low control overhead, e.g., the schedule 2 present in Figure 3.5. The first operation applies a data-celling transformation to the domain with respect to the dimensions used by the monitoring. A new statement is constructed from this data-celled domain by projecting out the cell dimensions of the domain. Only one point per cell remains after the projection. This domain is used to generate a guard in front of each approximate code version. These steps correspond to the bottom left transition in Figure 4.1. The approximate statements are generated with Algorithm 4 and scheduled immediately after the guard statement of the corresponding alternative. The `AddMonitoring` function ensures that all writes to the monitored data will be followed by a folding function call to update the maximum approximation level. Bottom right of Figure 4.1 shows the dynamic alternative generation with addition of the monitoring. The initial approximation level, zero by default or user-provided, is set by the guard statement before entering each cell. Once all the strategy versions along with their guards have been generated, a default guard is inserted to fall back to the unmodified statement which is executed by default.

The last phase of the algorithm deals with the static global strategies that must be enforced for the whole duration of the program. These alternatives must be applied to the previously generated statements, i.e. the union of the static area and the dynamic related statements. At the end of this last phase, the statement set is composed of all

the polyhedral representation of the static alternative, dynamic alternatives with guards, the default code path, all with the global strategy enforced. Polyhedral code generation finally generates the actual code to be compiled. In Figure 4.1, this phase is represented by the merge arrow that creates the new domain top right. This domain is later provided to the polyhedral code generator.

Figure 4.1 summarizes Algorithm 5. The static portions are separated from the dynamic ones at the start, creating the set S_i' for static and S_i'' for dynamic. The alternatives are applied to the static set. The dynamic set is data-celled and a new statement, one per cell, is created to select the alternative code versions. A parametric code is created for each cell with a different strategy and the monitoring is embedded inside the cell computation. Finally, the static and dynamic sections are merged to produce the adaptive kernel.

Algorithm 5 is accompanied by the following set of helper functions:

`StaticGlobal, StaticArea, Dynamic` respectively return the set of static global, area and dynamic strategies.

`UniverseDomain` takes a domain $S \in \mathbb{Z}^n$ and returns a domain with the same number of dimensions that has no constraint, i.e., an infinite polyhedron.

`CellDimensions` takes a domain D, a set of dimension identifiers D^{id} and a cell size l and applies a data-celling transformation on these dimensions.

`GenAlternativeGuard` takes a statement S_i and a strategy Strat_i and generates the guard statement that enters the branch if the alternative value matches the one of the folding function of the monitoring process.

`LowerCellDims` takes a domain D and a set of dimension identifiers D^{id} used for data-celling and returns the set of dimension identifiers of the cell itself and the one it encloses.

`AddMonitoring` adds the monitoring fold function to all access to the monitored array inside the statement.

`AddMonitorReset` adds the code to initialize the maximum approximation value. If not provided by the developer, defaults to lowest approximation value (original code).

```
1  #pragma acr grid(2)
2  #pragma acr monitor(B[i][j], min, foldTo1)
3  #pragma acr strategy static([i,j] : 1 < i < 4 and 1 < j < 4, otherCode)
4  #pragma acr strategy dynamic(1, bigStep)
5  for (int i = 0; i < 6; ++i) {
6    #pragma acr alternative bigStep(step, 2)
7    for (int j = 0; j < 6; ++j) {
8      #pragma acr alternative otherCode(code, { A[i][j] = B[i][j]; })
9      A[i][j] = B[i][j] * C[i][j];
10   }
11 }
```

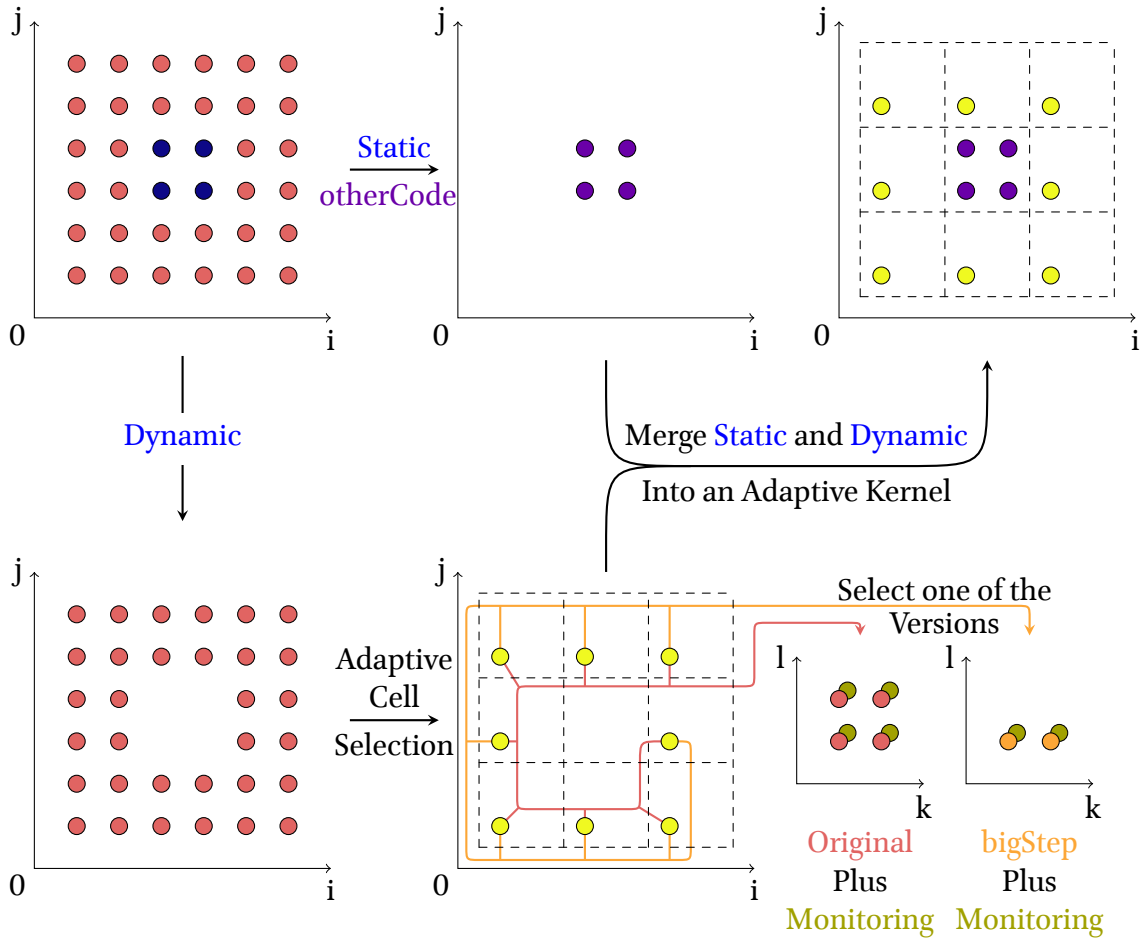


Figure 4.1 – **Compile Time Adaptive Code Generation** An adaptive version of the code snippet at the top is compiled. The code uses two *strategies*, one *static* defined at line 3 with a *code alternative* (line 8) and one dynamic (line 4) linked to a *step alternative* (line 6). The *grid* annotation defines cells of size 2. The bottom part of the figure shows the compilation phases for this top. The first phase separates the static and dynamic domains and applies the alternatives to the static part. The dynamic domain is data-celled according to the grid parameter (bottom center) and the guards are created for the cells. Each guard selects one of the specialized cell version with the information gathered by the monitoring embedded inside the cells themselves. The static and dynamic versions are merged to generate the new adaptive kernel.

Algorithm 5: Compile Time Adaptive Code Generation Algorithm Given a set of statements S and strategies Strats, this function returns a new set of statements corresponding to the annotated strategy versions with alternatives applied.

Function `CompileTimeGeneration`(S, Strats)

```
StaticState ← ∅;
LeftToDynamic ← ∅;
/* Phase 1                                                    */
forall S_i ∈ S do
    forall Strat_i ∈ StaticArea(Strats) do
        S'_i ← S_i ∩ Area(Strat_i);
        S'_i ← ApplyAllAlt(S'_i, Strat_i);
        StaticState ← StaticState ∪ S'_i;
        S''_i ← S_i − S'_i;
    LeftToDynamic ← LeftToDynamic ∪ S''_i;
/* Phase 2                                                    */
DynamicState ← ∅;
forall S_i ∈ LeftToDynamic do
    Dims ← DimRelatedToData(S_i);
    CelledDomain ← CellDimensions(S_i, Dims);
    CellDims ← LowerCellDims(S_i, Dims);
    OverCell ← ProjectOut(CelledDomain, CellDims);
    forall Strat_i ∈ Dynamic(Strats) do
        AltGuard ← GenAlternativeGuard(OverCell, Strat_i);
        AltGuard ← AddMonitorReset(AltGuard);
        AltStmt ← AddMonitoring(ApplyAllAlt(S_i, Strat_i));
        AltGuard ← ScheduleBefore(AltGuard, AltStmt);
        DynamicState ← DynamicState ∪ AltGuard ∪ AltStmt;
    S_mon ← AddMonitoring(S_i);
    DefaultGuard ← GenDefaultGuard(OverCell);
    DefaultGuard ← ScheduleBefore(DefaultGuard, S_mon);
    DynamicState ← DynamicState ∪ DefaultGuard ∪ S_mon;
/* Phase 3                                                    */
FinalStatements ← ∅;
forall S_i ∈ (DynamicState ∪ StaticState) do
    forall Strat_i ∈ StaticGlobal(Strats) do
        FinalStatements ← FinalStatements ∪ ApplyAllAlt(S_i, Strat_i);
return FinalStatements
```

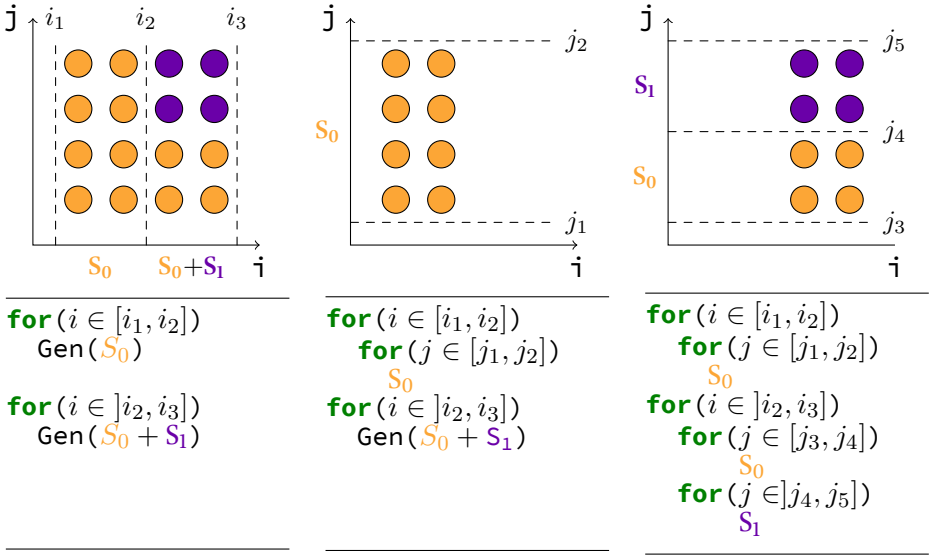


Figure 4.2 – **From the Polyhedral Representation to Code** Quilleré et al. polyhedral code generation algorithm applied to the union of two strategy statement domains [85, 9]. The dimensions are processed one after the other recursively. The first dimension to be generated is the outermost loop (left). Then the second loop dimension is generated on the last two domains (center and right). The algorithm stops when there is no remaining domain to generate.

4.4 Just-In-Time Adaptive Program Generation

The compile time code generation algorithm is well suited for kernels that can be generated using the data-celling process. However, dependencies can forbid the use of a data-celling transformation and thus generate the approximate kernel at compilation time. A small grid size may also cause extensive control overhead such as in schedule 1 of Figure 3.5, making the static code impractical. To remedy to these problems, we designed a dynamic method that generates a specialized approximate code at runtime. It is based on a runtime library that exploits cell monitoring information and polyhedral code generation techniques to generate a specialized version of the code with the lowest possible control overhead. For applications which have strongly correlated data locality, the dynamic approach can reduce the number of guards tremendously as cells neighbors may more likely share the same approximation level.

The polyhedral code generation algorithm is briefly depicted in Figure 4.2. The domain to be translated is similar to the one in Figure 3.5 and contains four cells. In this example, during the execution of the application, three of those cells situated on the bottom and left part of the domain allow a maximum approximation corresponding to the strategy S_0 and the remaining one to the strategy S_1. There is no need for extra control overhead to be generated whenever the flow of execution goes from one cell to another one sharing the same strategy. Hence, the iteration domains of the cells that share the same strategy are merged and treated as the same entity. This step is done with the information gathered by monitoring the data before the code generation algorithm is executed.

The code generation algorithm takes as argument a set of statements S in their poly-

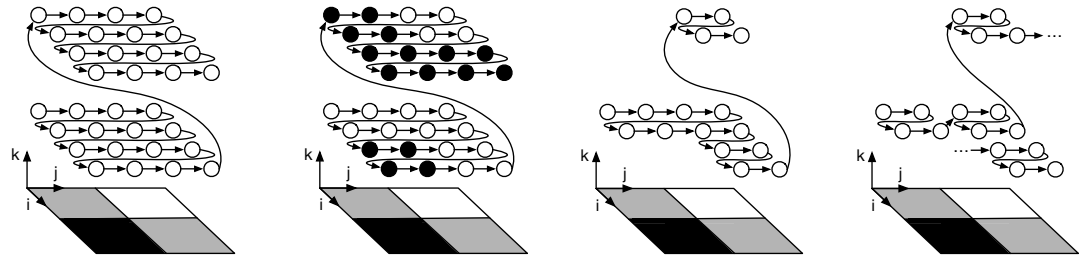
(a) Application Data Gradient (b) Data to Strategy Mapping (c) Strategy to Iteration Domain

Figure 4.3 – **Simulation Data to Alternative Iteration Domain** The data is represented as a gradient in (a). The monitor extracts the precision from the data and selects one of the tree available strategies for each cell (b). The iteration domain for each data domain that maps to the same strategy is computed, i.e., here each color in (b) corresponds to data sharing a common strategy and maps to the same color on the iteration domain (c). The three iteration domains with their respective code are used to generate the specialized kernel.

hedral representation and outputs an AST. For dynamic code generation, each statement potentially implements a different strategy. The iteration domain for a strategy is computed as the union of the domains of all the cells monitored with the same approximation level $D^{\text{Strat}_i} = \bigcup_{\text{Cell}\in\text{Strat}_i} D^{\text{Cell}}$. The loop generation algorithm acts dimension by dimension recursively. It starts with the outermost dimension where a projection is applied. This projection delimits the contiguous part of the domain in this dimension where the same statements are present. The projection over the 'i' dimension in Figure 4.2 results in two domains, one containing only the statement for the strategy S_0 and the other which contains both S_0 and S_1. In order to avoid a guard inside the loop nest to guard the execution of a given strategy, the loop is split in two parts and the algorithm used recursively on the two remaining polyhedra for the second dimension. The final result is an AST with a low control overhead compared to an equivalent compile time generated code, as presented in Section 4.3, which would contain many inner loop guards.

With dynamic code generation the iteration domain is split per strategy. Figure 4.3 shows how such partitioning is done. The application data is mapped to a strategy per cell and this information is used to reconstruct the iteration domain for each strategy. The code generator will interleave the iteration instances of the various strategies following the original code's schedule while keeping a low control overhead.

An illustration of the code generated for the compile time and runtime methods is shown in Figure 4.4. The original version of the code (a) calls a function at each iteration of a three-dimensional loop nest. Each white points of the iteration domain maps to a triplet values (i, j, k) passed to the function. The monitoring operates on 2×2 cells, on a data array related to the i and j dimensions. The cells are illustrated by the square tiling projected at the bottom of the domain. For this example we consider the following strategies: the white cell is using the original kernel, the black cell is using the *zero-*




```
ub = 1;
#pragma acr grid(2)
#pragma acr strategy
  dynamic(1, med)
#pragma acr strategy
  dynamic(2, low)
#pragma acr alternative
  med(parameter, ub=0)
#pragma acr alternative
  low(zero-compute)
for(k=0; k<=ub; k++)
 for(i=0; i<=3; i++)
  for(j=0; j<=3; j++)
   computation(k,i,j)
```

(a) Original Code

```
for(k=0; k<=1; k++)
 for(i=0; i<=3; i++)
  for(j=0; j<=3; j++)
   if(compute(k,i,j))
    computation(k,i,j)
```

(b) Guarded Code

```
for(i=0; i<=1; i++)
 for(j=0; j<=3; j++)
  computation(0,i,j)
for(i=2; i<=3; i++)
 for(j=2; j<=3; j++)
  computation(0,i,j)
for(i=0; i<=1; i++)
 for(j=2; j<=3; j++)
  computation(1,i,j)
```

(c) JIT Generated

```
for(k=0;k<cell0k;k++)
 for(i=0; i<=1; i++)
  for(j=0; j<=1; j++)
   computation(k,i,j)
for(k=0;k<cell1k;k++)
 for(i=0; i<=1; i++)
  for(j=2; j<=3; j++)
   computation(k,i,j)
for(k=0;k<cell2k;k++)
 for(i=2; i<=3; i++)
  for(j=0; j<=1; j++)
   computation(k,i,j)
for(k=0;k<cell3k;k++)
 for(i=0; i<=1; i++)
  for(j=2; j<=3; j++)
   computation(k,i,j)
```

(d) Compile Time

Figure 4.4 – **Code Generation Approaches** This figure presents the result of different code generation approaches for the original situation depicted in part (a). The top of the figure shows the code's iteration domain, where white points are executed while black points are not. Arrows represent the execution ordering. Under the iteration domain is a representation of the grid state. The approximation strategy is the following: in the black grid cell, no iteration is executed, in the gray grid cells, one iteration of the k-loop is executed and on the white grid cell, two iterations of the k-loop are executed. Parts (b), (c) and (d) show how the various approaches generate the approximated code: with an internal guard for *guarded*, perfectly matching the situation for *JIT* or on a grid-cell basis for the *compile time generator*.

compute alternative and the gray cells use the alternative *parameter* to set the number of iterations of the outermost dimension to one. The *guarded* code version (b) is a valid implementation of our extensions that, for each iteration instance, checks against the cell maximum approximation value collected by the monitoring. In this naïve implementation the control overhead is high because of the internal guard. Each dark dot of the domain corresponds to the execution of a test without useful computation. On the other hand, the dynamic version (c) has the least control overhead as the runtime generated code matches exactly the state of the program data. The last version (d) is the statically generated code for whom the cells are entirely visited in series. This version only requires four checks, here represented by the k dimension upper bound, to select the number of iterations to do in the outermost loop from zero to one. Because the iteration ordering in the statically generated version is different from the original code, it is not always possible because of data dependencies.

The dynamic method requires a just-in-time compiler to be able to generate a specialized version of the code during the execution of the program. We choose to implement

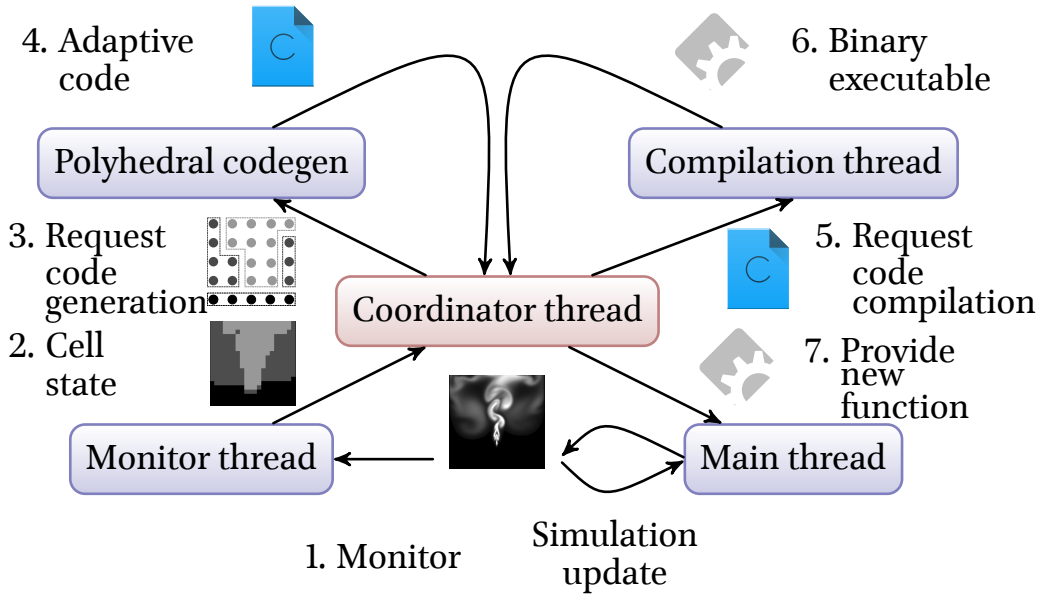


Figure 4.5 – **Adaptive Code Refinement JIT** Dynamic runtime used to generate optimized adaptive code.

the runtime as a shared library that can be loaded whenever the program encounters a dynamic optimized section. At compile time, the compiler adds a call back to the library functions responsible for the compilation management. In Figure 4.5 we disclose the implemented architecture. The runtime is separated in four distinct functional units:

The monitoring thread checks the state of the data after a kernel execution to select the most appropriate approximation strategy for each cell. Once a kernel has been fully executed, it provides the list of all the states (i.e. the suggested approximation level for each cell) to the coordinator thread.

The polyhedral code generation thread generates on demand an AST from the polyhedral representation given by the coordinator thread. Each generated statement corresponds to a strategy. The generated code has the following properties: (1) it has no costly internal tests to decide about the optimization strategy, (2) it makes more computation only where it is necessary and (3) the remaining computations are done with respect to the initial ordering to preserve validity and accuracy.

The compilation thread is responsible for generating an executable from the AST that has been generated from the polyhedral code generation unit. We choose to implement a compilation with two separate parallel paths. A fast lane using Tiny C Compiler [14] that has a fast JIT but has low code optimization capabilities and a second lane with the GCC compiler which is an order of 10-50 times slower but applies efficient code optimizations.

The coordinator thread is the central part which orchestrates the other threads. It has the responsibility to check whether a specialized code version that is used for the kernel is still valid from the current cell state obtained from the monitoring thread. The validity can be expressed in three ways:

1. **Raw checking:** For each cell, compare the maximum approximate level, i.e. the dynamic approximation level, used in the optimized kernel to the value obtained

by the monitoring thread. If the two values do not match to the same strategy, the check fails.

2. **Versioning checking:** The same as 'raw' checking but accepting monitored values that require less precise computation than the one the kernel is using. Reject cell values that require more precise versions than the one currently used by the kernel. If the percentage of values in an over-precise state exceeds a threshold, a new version is generated.

3. **Stencil checking:** The last checking is similar to 'versioning' but it also considers that the neighbors of a cell can influence the precision required for a given cell. It sets the cell approximation level to the minimum approximation strategy possible between what the cell requires and the immediately greater approximation level than the most precise of its neighbors, for each cell, before applying the 'versioning' checking algorithm.

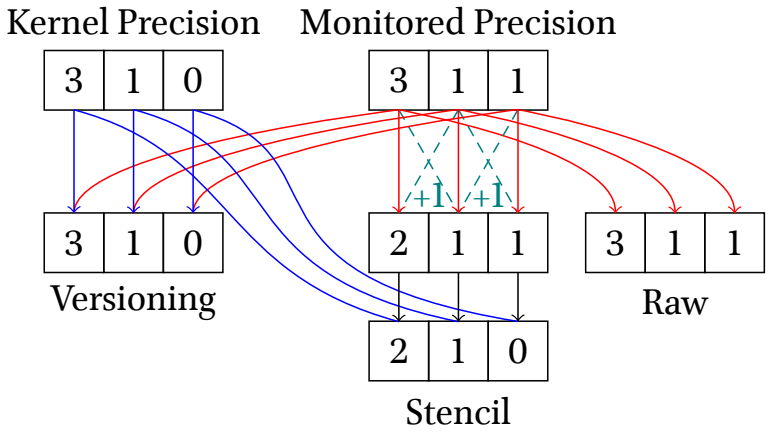


Figure 4.6 – **Coordinator Approximation Level Computation** Precision computation for the raw, versioning and stencil validity checks. The domain consists of three monitored cells. Each cell is represented by a square labeled with its precision level (the lower, the most precise). The precision of the optimized kernel is also visible. A vertex from one cell to another represents a value dependency. If multiple vertices enter the same destination cell, the lowest of the parent values is chosen. The stencil check requires an intermediate step because the neighbor values are also considered to compute a cell precision. The value from the neighbors is incremented by one (green dashed lines) before they enter the intermediate cell. This increment limits the influence of the neighbors when the stencil check is used.

The coordinator thread will use one of the three pre-defined validity checks before requesting a new version generation. An example of a validity check computation is shown in Figure 4.6. If the check fails, the monitor generates the alternative statement domain from the monitored data and sends a request to the polyhedral code generation thread. Once the AST has been generated it is directly forwarded to the compilation thread. The coordinator waits for the binary code from the compilation thread to modify the function pointer in the main executable to the newly optimized version instead of the previous one.

The coordinator thread may be configured in two modes, synchronous or asynchronous checking. Synchronous checking only allows the main application to ask for a new optimized version whenever the version is checked. If the coordinator requested a new version and the resulting binary code is not ready yet, the main application will be stalled in the meantime. This mode of operation is preferred to limit deviation of the output result. The asynchronous mode allows the application to run in parallel, with a non-appropriate approximation strategy, while the new optimized version is computed. If the main application arrives at the kernel call site before the new version is fully compiled, the code falls back to the original code version instead. Asynchronous mode may allow a less precise version to run temporarily while the monitoring and coordinator are processing the data cells. The original version can be used whenever no optimized version is available, allowing for the compilation to run concurrently to the slower but more precise version. To optimize the time required to generate new specialized code versions, the coordinator could speculatively generate new versions that could be called instead of the original version for the non-'raw' policies. The coordinator can also maintain a buffer of previously generated versions and use least recently used replacement policy whenever the buffer is full. Our runtime implements all these mechanisms. The user may select them through compilation option or through the ACR's *checker-select* extension depending on the needs (speed vs precision). Section 7.1.2 introduces an example using this annotation.

The runtime is optimized in several ways to ensure a convenient optimized code is available for the computation thread as soon as possible. Firstly, it uses threads to better utilize modern computer architecture resources, where the number of cores are legion. We believe that prioritizing the execution of the adaptive runtime on one core of the machine is not disproportionate given that the runtime is most of the time in a sleeping state, waiting for the application to finish its computation. The coordinator is awoken by the main thread, at the beginning and at the end of the kernel and if one of its managed thread finishes its task. Secondly, the coordinator thread requests two different compiled codes for the same C input: a non-optimized one which may be generated and used quickly (we use TCC, the Tiny C Compiler for this) and a better optimized code that may be available later and that will replace the non-optimized one (we used GCC with aggressive optimization options for this). Thirdly, to improve the availability of specialized version, the coordinator can preemptively request the generation of versions that do not represent the current state of the application but has a higher precision in the neighborhood of active regions, trying to predict the precision shift. This mitigates the case where no specialized version is available and stalls the application until a version is ready (synchronous) or falls back to the original and most expensive kernel while the compilation of a specialized version is done in parallel (asynchronous). We can say that higher precision computation is "safe" and does at least what was specified by the domain-specific information.

4.5 Compilation With Polyhedral Optimizations

One of the many advantages procured by the polyhedral representation of programs is the robustness of dependency analysis. A property of this model is to handle complete and exact dependency information [47, 48]. Computing the dependency polyhedron requires solving a Diophantine equation system that runs in the worst case in exponential

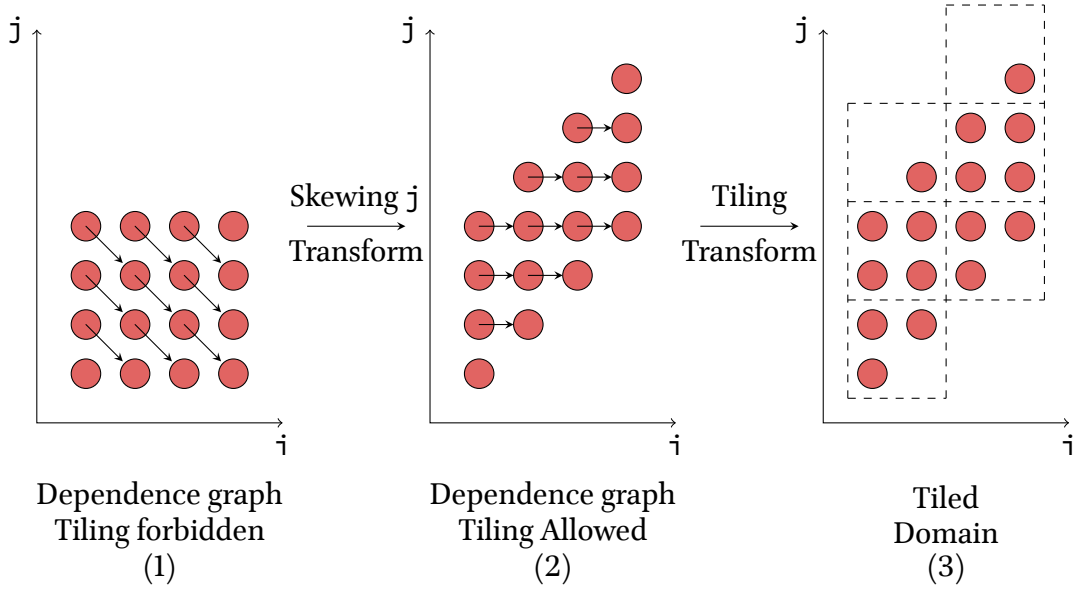


Figure 4.7 – **Tiling Transformation Enabled by a Skewing Transformation** The dependencies in (1) discard the possibility of applying a tiling transformation directly because a tile would need the value of a tile that happens after. However, by applying a skewing transformation on the `j` dimension (2) the target code can be tiled because the dependencies follow the tile execution order. A skewing is not the only transformation that works in this case, for example a loop reversal of the `j` dimension can enable the tiling with a domain that keeps its square/rectangular shape.

time. Hence, this method is mostly reserved for complex dependency analysis as required when optimizing loop nests [8].

Compilation tools have been developed to automatically optimize loop nests [47, 48, 21, 82, 81]. Those tools search for a solution to a linear programming problem. For example, finding a scheduling which minimizes the distance of reuse of written variables, i.e. optimizing for locality, or to push dependency on the lowest level of the loop, i.e. optimizing for parallelism. Such algorithm has been implemented in tools like PLUTO [21] or Traco [17].

These loop optimization techniques are orthogonal to ACR. They search for a different schedule for the loop nest that exhibits certain characteristics useful for performance. Our method focuses on slicing the input domain by alternatives and ensuring that the appropriate alternative is used. Hence, the automatic optimization tools can be used in a pre-processing stage to obtain the optimized schedule. Finding a scheduling that makes the data-celling transformation legal for generating the version at compile time as in Figure 4.7. In ACR, the original lexicographical scheduling would be replaced by the optimized one before the adaptive code generation stage. The generated code could then exhibit parallelism or better locality. Ongoing work aims at mixing polyhedral optimizations with our ACR optimization approach.

Chapter 5

Automatic Adaptive Stencil

Stencil codes are a class of iterative kernel that uses nearest-neighbor computations on a grid or graph data structure, i.e., such that the update of a point on the grid depends on the adjacent neighbor values according to a fixed pattern. Stencils are extremely commonly found in scientific computing, machine learning and image processing applications. e.g., Algorithm 6 top shows a five-point stencil kernel on a 2D Cartesian grid. The compiler can rely on a static data access analysis in order to automatically detect the most common form of stencils. In our case, we are interested in data read and write accesses for each statement of the kernel to detect stencil-like computations. We provide Algorithm 7 to identify the stencils. This algorithm builds for each written data at a given iteration, the set of read data that contributes to the computation of the written data (both inside a statement or transitively through several statements). It considers each write statement in the lexicographical order and selects the one with the most reads to the same array. It outputs the set of written + contributing data, with maximum cardinality.

In the previous chapter we presented how to generate an adaptive code from a computational kernel and user input approximation annotations. Thanks to ACR, the developer can focus on his/her algorithm and let our framework exploit his/her domain specific knowledge to transform the code. In this chapter we introduce a technique to identify automatically pertinent ACR annotations in the restricted, yet useful, class of programs based on stencil computation. We present the automatic generation of approximate stencil versions (Section 5.1), then how to order the generated versions with respect to their approximation level (Section 5.2). Discovering the relevant data to monitor is detailed in Section 5.3 and the granularity is empirically chosen as explained in Section 5.4. Finally, we discuss deviation of the results in Section 5.5.

5.1 Generating Stencil Alternatives

To generate multiple versions of a stencil, we propose the following adaptive alternative policies, which reduce the stencil computation and data accesses:

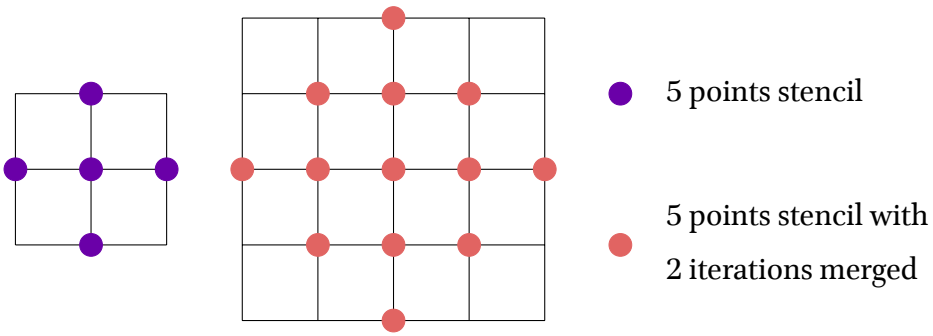
Skipping Skipping some stencil computation entirely (e.g. skip it every N iteration of the computing loop) when the precision function has monitored a low deviation.

Narrowing Reduce the size of a stencil to reduce both its calculation complexity and its memory load.

Algorithm 6: Example 2D stencil algorithm

def *Five points stencil*:
Data: Input array $I[M][N]$
Result: Output array $O[M-2][N-2]$
for $i \leftarrow 1$ **to** $M-1$ **do**
for $j \leftarrow 1$ **to** $N-1$ **do**
$O[i][j] \leftarrow I[i][j] \oplus I[i-1][j] \oplus I[i+1][j] \oplus I[i][j-1] \oplus I[i][j+1];$

def *Five points stencil with two iterations merged*:
`/* operation '·' distributive over '⊕' */`
Data: Input array $I[M][N]$
Result: Output array $O[M-2][N-2]$
for $i \leftarrow 1$ **to** $M-1$ **by** 2 **do**
for $j \leftarrow 1$ **to** $N-1$ **do**
$O[i][j] \leftarrow (\, 5 \cdot I[i][j] \,) \oplus 2 \cdot (\, I[i+1][j] \oplus I[i][j-1] \oplus$
$I[i][j+1] \oplus I[i-1][j-1] \oplus I[i+1][j+1] \oplus I[i-1][j+1] \oplus$
$I[i-1][j] \oplus I[i+1][j-1] \,) \oplus$
$I[i-2][j] \oplus I[i+2][j] \oplus I[i][j-2] \oplus I[i][j+2];$



Border Activation Only activate the stencil computation at the cell borders, skipping the computation in the center of cells while allowing information to pass through the interfaces with their neighbors. Cells having a neighbor with an alternative more precise than their will use border activation to capture a possible incoming singularity.

Stencil narrowing may rely on the fusion of multiple stencil steps to grow the size of the halo. For example, the stencil in Algorithm 6 top has a halo of distance one. It is possible to increase the size of this stencil halo by merging two stencil iterations, resulting in the bottom stencil in Algorithm 6. A larger halo offers more possibilities to build an approximate version:

Narrowing by deletion Removes one or more points of a stencil to reduce its computational footprint.

Algorithm 7: Search for the biggest stencil in a statement set

Data: Ordered statement set S
Data: For each $S_i \in S$, read access set $S_i^{\texttt{read}}$
Data: For each $S_i \in S$, write access set $S_i^{\texttt{write}}$
Result: Tuple of written and read locations of a stencil
$\texttt{Biggest}_{\texttt{read}} \leftarrow \emptyset$;
$\texttt{Biggest}_{\texttt{write}} \leftarrow \emptyset$;
$n \leftarrow \|S\|$;
while $n > 0$ **do**
  `/* Statement accesses + transitive accesses */`
  $N^{\texttt{read}} \leftarrow S_n^{\texttt{read}}$;
  repeat
    $N_{\texttt{prev}}^{\texttt{read}} \leftarrow N^{\texttt{read}}$;
    foreach $N_i \in N_{\texttt{prev}}^{\texttt{read}}$ **do**
      foreach $W_i \in \texttt{statementWrite}(S, N_i)$ **do**
        $N^{\texttt{read}} \leftarrow N^{\texttt{read}} \cup W_i^{\texttt{read}}$;
  until $N^{\texttt{read}} = N_{\texttt{prev}}^{\texttt{read}}$;
  `/* Check for stencil pattern for each arrays */`
  $G \leftarrow \texttt{groupBy}(\texttt{array}, N_{\texttt{read}})$;
  $\texttt{BestLocalMatch} \leftarrow \texttt{maxCardinal}(G)$;
  if $\|Biggest_{\texttt{read}}\| < \|\texttt{BestLocalMatch}\|$ **then**
    $Biggest_{\texttt{read}} \leftarrow \texttt{BestLocalMatch}$;
    $Biggest_{\texttt{write}} \leftarrow S_n^{\texttt{write}}$;
  $n \leftarrow n - 1$;
return $(Biggest_{\texttt{write}}, Biggest_{\texttt{read}})$
$\texttt{statementWrite}(\texttt{S}, \texttt{N})$: returns the set of statements in S which writes at the location N.
$\texttt{groupeBy}(\texttt{P}, \texttt{S})$: returns a set of set grouped by P
$\texttt{maxCardinal}(\texttt{S})$: returns the largest cardinal set in S

Narrowing weight redistribution by distance to neighbors Removes one or more points of a stencil, but redistributes the deleted stencil weight among the neighbors that are located at a maximum distance of N.

Narrowing by weight redistribution to direct neighbors Removes one or more points of a stencil, same as before with $N = 1$.

Figure 5.1 illustrates the different strategies on a von Neumann stencil with a halo of size 2. Narrowing should be effective to reduce memory contention and stress on the processor cache. For example in Figure 5.1, if the stencil is applied to a 2D array in a row major language, the deletion or redistribution of the points w_3^5 and w_3^1 results in computation savings and possible memory access savings for elements that are only accessed once. Deleting or redistributing w_1^3 and w_5^3 could be of lower importance memory-wise, because the values have a high probability of being in the cache as other elements in the line are accessed. The compiler can use different weights with respect to the Manhattan

distance to the barycenter[1] of the stencil for narrowing with redistribution. For example, if we consider that the points closer to the barycenter are twice as important than the ones that are further away, one can define the weights $[8, 4, 2, 1]$, for a redistribution up to a distance of 4. Linear or exponential functions can be used to define the weights. We provide Algorithm 8 to normalize the distribution weights among a number of points. Its principle is to find, for each distance with respect to the stencil point to be redistributed, the ratio of its weight to be distributed at that distance, such that the weight is fully redistributed.

Algorithm 8: Normalize per Distance Redistribution Multiplier

Data: List of weight attributed to each distance L_{weights}
Data: List of number of points per distance receiving the weight L_{Npoints}
$denominator = 0$;
foreach *pair* $(weightLevel, numLevel)$ ***in*** $zip(L_{\text{weights}}, L_{Npoints})$ **do**
  $denominator = denominator + weightLevel \times numLevel$;
$mulLambda = function(x) \rightarrow \frac{x}{denominator}$;
$multiplierList = map(mulLambda, L_{\text{weights}})$;
return *multiplierList*
$zip(a, b)$: returns a list of pairs by combining the elements of a and b in order
$map(f, a)$: returns a list by applying the function f to every elements of a

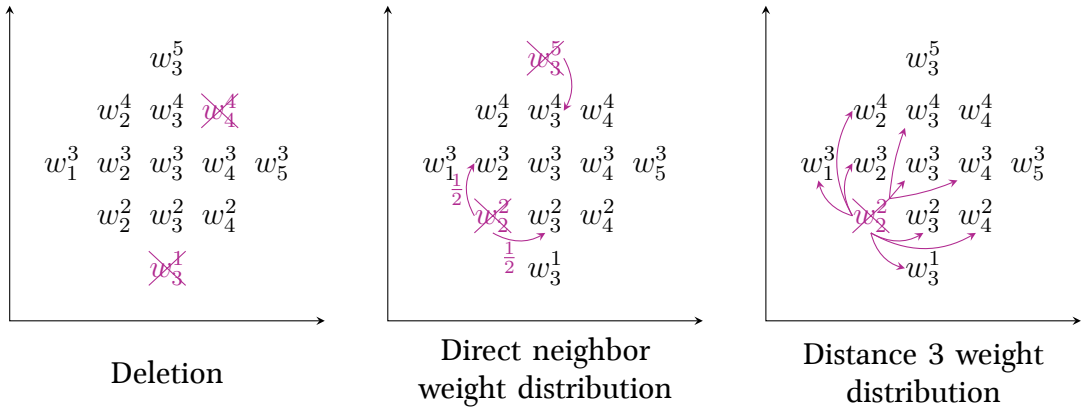


Figure 5.1 – **Stencil Narrowing Strategies** Reducing the number of computation of a stencil by reducing the number of points. The available strategies are: (1) deletion as presented on the left or (2) redistribution of stencil weights through direct neighbors (center) or up to a given distance (right).

Algorithm 9 introduces a way to generate approximate versions of a stencil. We consider that there is a correlation between the precision of the stencil and the number of points that are redistributed or deleted, because when a point of a stencil is deleted some information is inevitably lost. For example, when the stencil encodes an equation solver like Maxwell's equations (Section 2.1.1), removing a point would correspond to ignoring

[1]The location of the barycenter of a 2D stencil with n points $S = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ is $\left(\frac{\sum_{i=1}^{n} x_i}{n}, \frac{\sum_{i=1}^{n} y_i}{n}\right)$. The formula is the same in higher dimensions, handle each dimension separately.

the contribution of one side of one field (nullifying the derivative term). Differently, redistributing the point weight after merging two iterations of the kernel corresponds to an interpolation, hence, a better approximation which does not remove completely the contribution.

Algorithm 9 takes as input the stencil's point coordinates, relative to the barycenter, with their weights and the number of versions to be generated. The generated code should still be considered as a stencil. Therefore, we choose to limit the deletion and redistribution to the points further away than the direct neighbors of the barycenter. The number of versions to be generated is bounded by the number of points outside of this minimal stencil. The number of points to be removed in each version is computed. Multiple strategies can be considered to compute this list: exponential, logarithmic or a linear distribution. We opted for the simpler linear function after empirically noticing that exponential and logarithmic integer functions are not appropriate with respect to the limited amount of points available. The generation of the approximate stencil version follows. Firstly, we copy the original stencil position and weights which are going to be modified to generate a new version. Secondly, for each distance from the barycenter in descending order, we get the list of points at this distance. If not all points are to be deleted or redistributed, we select the ones that will most likely improve the application throughput, for example, evict an access that results in less cache misses. Then the weights are redistributed and the points deleted from the stencil. Finally, once the number of points to be deleted is met, the new stencil is appended to the pool of new versions.

The analysis of the data at runtime allows adaptive methods to target the expensive computations in part of the domain where significant updates are expected [63, 15]. Hence, our algorithm relies on this feature to apply the previously mentioned alternatives.

Stencil skipping by a factor of N allows the compiler to generate a version that reduces the computation intensity by a factor of N, and may affect the precision. Stencil narrowing allows to generate a wider range of approximate versions, not available with skipping only. For example, it is possible to remove P points of a stencil with the lowest absolute factors to lower the computation/precision. If the number of points removed is below what stencil skipping deletes from a level N to $N-1$, it corresponds to a fully generated intermediate approximate versions. Stencil merging combined with these techniques allows to generate enough alternatives for an adaptive purpose. In the current state of our implementation choosing a high number of alternatives leads to a high overhead. Generating only a limited number of versions gives the best results.

5.2 Ordering the Stencil Alternatives

In our nearly automatic approach for stencils, the user has to select the stencil kernel to optimize and to input the level of deviation under which it becomes acceptable to use approximation. Algorithm 9 provides the compiler with a maximum of N alternatives from the kernel. The list returned by the algorithm is ordered by construction from the lightest approximation, i.e., the smallest number of points removed or redistributed, to the most aggressive approximation.

Our approach to manage adaptiveness in this context is inspired by the grid layout in

Algorithm 9: Generate Approximate Stencil Versions

Data: The number of versions to generate: *nGen*
Data: The stencil point weight list: L_{weights}
Data: The stencil point position list: L_{pos}
`/* Keep the minimum of points to be considered a stencil */`
maxToDel $\leftarrow length(L_{\text{pos}}) - length(atDist(L_{\text{pos}}, 0)) -$
$length(atDist(L_{\text{pos}}, \textit{minDistToBarycenter}(L_{\text{pos}})))$;
nGen $\leftarrow \min(\textit{nGen}, \textit{maxToDel})$;
toDelQuot $\leftarrow \text{floor}(\frac{\textit{maxToDel}}{\textit{nGen}})$;
toDelRem $\leftarrow \textit{maxToDel} - \textit{toDelQuot} \times \textit{nGen}$;
approxVersionList $\leftarrow \emptyset$;
if *toDelQuot* $\neq 0$ **then**
  for *currGen* $\leftarrow$ *nGen* **to** 1 **do**
    delForGen $\leftarrow$ *toDelQuot* $\times$ *currGen*;
    if *toDelRem* > 0 **then**
      toDelRem $\leftarrow$ *toDelRem* $- 1$;
      delForGen $\leftarrow$ *delForGen* $+ 1$;
    (*newWeights*, *newPos*) $\leftarrow (L_{\text{weights}}, L_{\text{pos}})$;
    for *currDistance* $\leftarrow$ *maxDist*(L_{pos}) **to** *minDistToBarycenter*(L_{pos}) **do**
      pointAtDist $\leftarrow$ *atDist*(*currDistance*, *newPos*);
      while *length*(*pointAtDist*) $>$ *delForGen* **do**
        pointAtDist $\leftarrow$ *removePointTryOptimizeCache*(*pointAtDist*);
      if *Distribute the Weight* **then**
        `/* Redistribution to direct neighbors */`
        foreach *point* $\in$ *pointAtDist* **do**
          neighbors $\leftarrow$ *getNeighbors*(*point*, *newPos*, 1);
          multiplier $\leftarrow$ Algo 8 with $[1]$ and $[\textit{length}(\textit{neighbors})]$;
          foreach *neighbor* $\in$ *neighbors* **do**
            setWeight(*newWeights*, *neighbors*, *multiplier* $\times$
            getWeight$(point)$);
      Remove *pointAtDist* and the associated weights from
      (*newWeights*, *newPos*);
      delForGen $\leftarrow$ *delForGen* $-$ *length*$(pointAtDist)$;
    approxVersionList $\leftarrow$ *approxVersionList* $\cup$ (*newWeights*, *newPos*);
return *approxVersionList*

minDistToBarycenter(l): returns the minimal distance to the barycenter in l
maxDist(l): returns the maximal distance between any 2 points in l
atDist(l, a): returns the points in the list l which are at Manhattan distance of a
length(l): returns the size of the list l
setWeight(l, p, w): sets the weights of the points p in the list l to w
getNeighbors(a, l, d): returns the neighbors of a in the list l at distance d
removePointTryOptimizeCache(l): returns a list where one point has been deleted from the list l trying optimize the cache usage (e.g., delete points which are at minimum or maximum lexicographic position of outer dimensions first)

adaptive mesh refinement[2]. The grid is graded, meaning that there cannot be more than one level of refinement between two neighbors. Algorithm 10 implements this behavior for our adaptive computation grid. The first adaptive grid state of the kernel must not use approximation. Each new adaptive grid state is generated from the previous state and the precision gathered during the execution of the kernel. A cell may only use the next more aggressive approximation when it is "surrounded" by cells that are tagged with at least the same level of approximation. A cell that switches back to a precise computation will require its neighbor to use the lightest approximate version. The grid is guaranteed to be graded only if the cells at the border of an approximate region can switch to non-approximate. For example, if a cell with an approximation of level 3 is tagged with no approximation, it may take several update cycle until the grid stabilizes into a graded version. We believe that this is a good trade-off for stencil application, where the approximated and precise regions are usually propagated between neighbors. One could conceive an algorithm to do a grading on the regions marked for approximation. In that case the grid can only be processed after the end of the kernel, once all the approximation-friendly cells are known. The function in Algorithm 10 allows a parallel update of the grid, provided that the writes to the new adaptive state grid can be done atomically. Figure 5.2 shows an example of adaptive grid update using this algorithm.

Algorithm 10: Stencil Graded Grid Construction

Data: Previous Adaptive Grid State: $AGrid_{\text{prev}}$
Data: Number of Adaptive Code Versions: A_{num}
initWith$(A_{\text{num}}, AGrid_{\text{next}})$;
Function *setAdaptiveLevel*(measuredDeviation, **coordinate**)
  if measuredDeviation $\leq$ userThreshold **then**
    mostPreciseNeighbor $\leftarrow A_{\text{num}}$;
    foreach neighborCoord of **coordinate**) **do**
      mostPreciseNeighbor $\leftarrow$
       min(*mostPreciseNeighbor*, $AGrid_{\text{prev}}$(**neighborCoord**));
    $AGrid_{\text{next}}$(**coordinate**) $\leftarrow$
     min($AGrid_{\text{prev}}$(**coordinate**) $+ 1$, *mostPreciseNeighbor* $+ 1$);
  else
    $AGrid_{\text{next}}$(**coordinate**) $\leftarrow 0$;
    foreach neighborCoord of **coordinate**) **do**
      $AGrid_{\text{next}}$(**neighborCoord**) $\leftarrow \min(AGrid_{\text{next}}$(**neighborCoord**), $1)$;

[2]Our first approach was to generate subsequent deviation values from the deviation parameter provided by the user for each approximate stencil. For this we need the range of values taken by the deviation parameter before generating new ones. We could extract this range empirically, at the same time as the grid size extraction. This has the advantage that the values are known when the adaptive application is generated, and the compiler can optimize the adaptive selection. The drawback is that for a different dataset, the range may change and we may miss optimization opportunities.

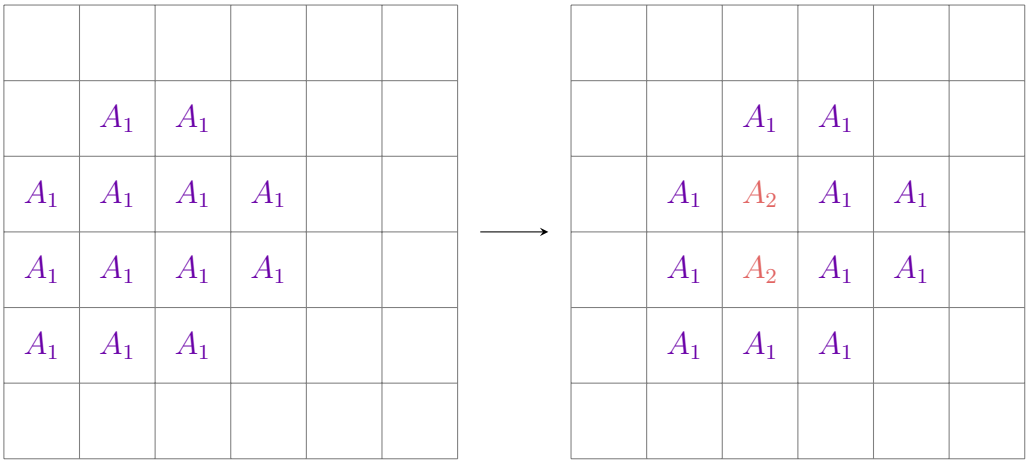


Figure 5.2 – **Stencil Adaptive Grading** The adaptive grid approximation level is updated from its configuration on the left to the configuration on the right. White cells use no approximation and cells marked with A_n use an approximation of level n. Two cells were promoted to a more aggressive approximation A_2. In the previous state they were surrounded by neighbors with an approximation level allowing the change, and they are still eligible for approximation after the kernel execution. The two rightmost cells of the left grid had more precise neighbors and did not change version yet. This is due to the incremental nature of Algorithm 10 which allows a parallel update of the next grid while the application executes the kernel.

5.3 Monitored Data Discovery

The input data of the decision function is the application's runtime data, representing the state of the problem. The compiler has to detect which data has the convenient characteristics to generate a function to select an alternative globally or on a localized portion of the application's data domain. We propose a set of characteristics extracted from our empirical study of typical applications from the field of approximate computing (more in Chapter 7.2).

The first characteristic which usually provides relevant precision information from the application data is the deviation between the updates. Mathematically, this characteristic represents the first derivative of a function [37]. Whenever this rate is measured to be low, it is a good sign that we can use a less precise update function in this area of the application's data domain. Furthermore, this effect is usually local, and in numerical simulations, singularities spread and do not appear in the middle of uniform regions, hence, missing a significant event is unlikely. This effect is intensified by the use of stencil computation, updating its state from neighbors. If the effect is localized, we consider clusters of data as a single entity and apply the alternative to each update of this cell (more in Section 5.4).

To discover this first characteristic the compiler has to look for the following patterns in the source code:

Swapping arrays They are frequently used in physics simulation and image processing where the transformation uses the data from the previous transformation as input. This characteristic is statically analyzed if the program is simple enough or

dynamically asserted by running the application on datasets and monitoring the array addresses accessed and recognize an "n cycle" pattern.

Write-only arrays They are usually the sign of a multistage update algorithm, e.g., any advection-diffusion equation which updates the physical variables in separate steps or store statistic information of the application's state. This is addressed with a static analysis of the kernel on written-only data arrays. The algorithm to find these write-only arrays from the polyhedral representation is shown in Algorithm 11.

A second characteristic, lie in the application's own statistic gathering, e.g., multimedia codec gathers information about previous frames to achieve a better compression ratio, and numerical simulations may use a physical variable or error rate as a simulation stopping condition. Statistics are usually gathered globally and have smaller size than the application's dataset. The application may gather the statistics in a dedicated function which could be marked by the user in the same fashion as the kernel to optimize. Hence, the compiler can extract correlations between the data updated by the kernel and the statistics gathered by the application. For practical reasons, we only consider statistics updates located inside the kernel to optimize. Our statistics gathering follows the following pattern:

Accumulators They can be used to store statistic information and are most likely used by the application to take algorithmic decisions. Such accumulators are of particular interest for our method as they may store the information we are interested in. Accumulator are many-to-one information gathering. The compiler may allocate memory to save this information at a finer granularity if needed.

The compiler uses Algorithm 12 to find accumulators. The accumulator should not be reused in the same kernel in another way than updating itself. This filters out many temporary variables used for the update function itself. We only support the discovery of accumulator using a scalar. We believe that runtime profiling may help discover accumulator which summarizes the information to an array, i.e. mathematically finding a surjective function $f : \mathbb{P} \to \mathbb{M}, |\mathbb{P}| > |\mathbb{M}|$.

With this information, the compiler will find the most suitable approximation level by monitoring the discovered data at runtime[3].

[3] It may ask the user for advice depending on the compiler option the user selected, i.e., either interactive or automatic.

Algorithm 11: Discover Write Only Arrays from the Polyhedral Representation

Data: Statement set S of the kernel
Data: For each $S_i \in S$, read access set $S_i^{\texttt{read}}$
Data: For each $S_i \in S$, write access set $S_i^{\texttt{write}}$
`WriteOnlySet` $\leftarrow (\bigcup S_i^{\texttt{write}}) \setminus (\bigcup S_i^{\texttt{read}})$;
return `filterScalar` (`WriteOnlySet`)
`filterScalar`(`S`): returns a set without scalar accesses

5.4 Granularity Selection

To find the *granularity* we suppose that the *alternatives* and *strategies* are already available (provided in Section 5.1). Hence, we have at our disposal a function to select the precision (Section 5.3) and the approximate versions and we need to find a generic method to select the granularity. A fine grid will have a better mapping to the application's problem but will have a higher managing overhead whereas a grid that is too coarse may miss approximation opportunities. In this section we present an algorithm to select a relevant granularity based on an empirical study of a set of applications representative of the approximate computing field.

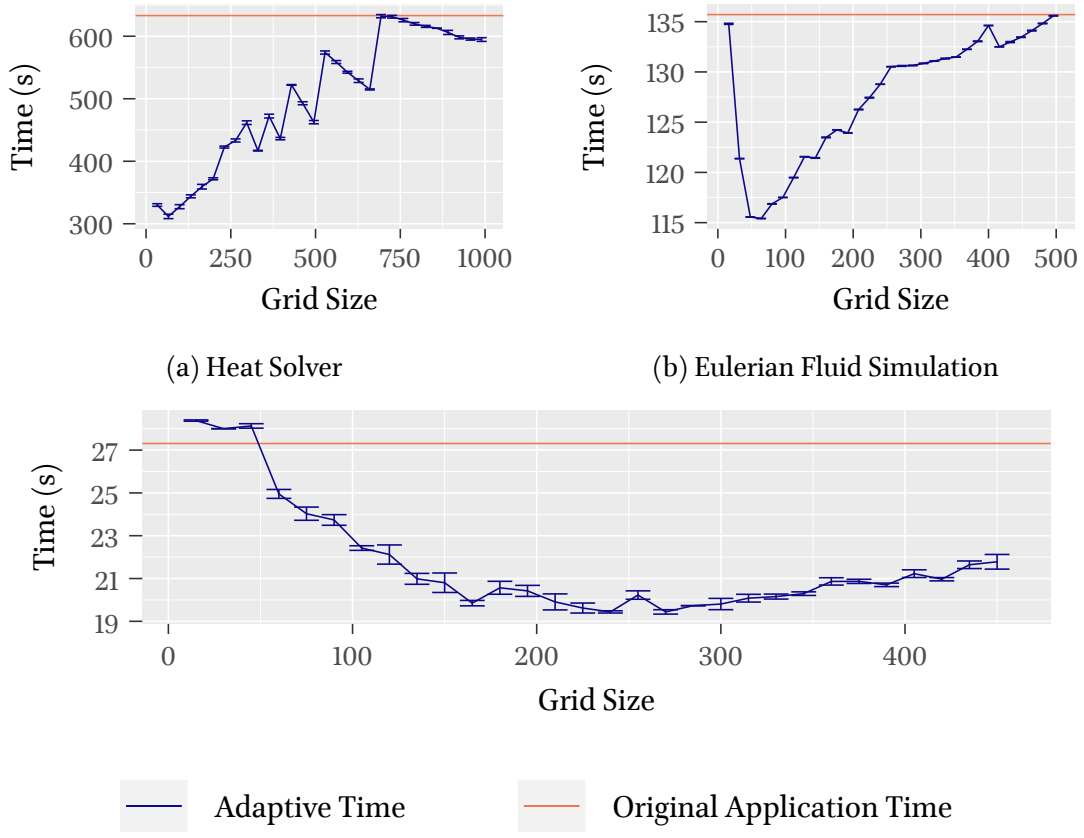
State of the art adaptive techniques use modular grids, i.e., the grid shape is updated during the execution to achieve precise computation where it is needed. A modular grid creates special cases at the interface of two cells with different precision. These cases need to be handled by the compiler and makes the automatic alternative and grid generation more complex. In this work we only consider Cartesian, static grids. We discovered that the application's behavior with respect to the grid size can be transposed between distinct input problems and data sizes. Without the maximum deviation parameter provided by the user in the annotation (Section 3.4.1) the compiler would have to search for the grid size and monitoring parameter simultaneously. However, the search space of these two parameters would be huge, also the two parameters have an influence on each other. Hence, we choose to ask the user for the maximum deviation parameter in order for our algorithm to be more robust and practical. Getting rid of this last user input parameter is left for future work.

Figure 5.3 shows the three main recognizable patterns seen while monitoring different applications. All of them show a local minimum but with a different distribution of performance. Applications that show one of these three behaviors will present the same behavior with different data sizes or problem statements (see Section 7.2). Hence, we propose to match these patterns to categorize the application among one of these three forms and compute the ratio between the maximum grid size and the value read at the local minimum. Our empirical study suggests that this ratio can be reused with different data sizes to scale the grid size accordingly. Figure 5.4 shows that such algorithm is precise enough to approach the best value for bigger datasets based on data collection on the small dataset. Hence, the compiler has an estimate of the best grid size for any dataset size and the curve indicating the slope direction.

With the current static grid back end, we cannot guarantee the approximation level of the output with respect to any input data. The guarantee we provide is that no approximated version is used when the monitored deviation is greater than the one selected by the user. The user must provide input data to the optimized application within the same granularity scaling than the one used during profiling. The user should run a new automatic profiling step for different input value scales. Ongoing work aims at removing this requirement using dynamic grids and mathematical transformations, such as the wavelet transform (Chapter 6).

Algorithm 12: Discover Potential Accumulators in the Polyhedral Representation

Data: Statement set S of the kernel
Data: For each $S_i \in S$, read access set $S_i^{\texttt{read}}$
Data: For each $S_i \in S$, write access set $S_i^{\texttt{write}}$
`PotentialAccumulatorIds` $\leftarrow \emptyset$;
foreach $S_i \in S$ **do**
  if `toScalar`$(S_i^{\texttt{write}})$ *and* `readAlso`$(S_i^{\texttt{read}}, S_i^{\texttt{write}})$ **then**
    `PotentialAccumulatorIds` $\leftarrow$ `PotentialAccumulatorIds` $\cup\, S_i^{\texttt{write}}$;
return `PotentialAccumulatorIds`
`toScalar`$(\texttt{P}^{\texttt{write}})$: returns true if it reads or writes to a single scalar
`readAlso`$(\texttt{P}^{\texttt{read}}, \texttt{P}^{\texttt{write}})$: returns true if written scalar is present in the set



(a) Heat Solver

(b) Eulerian Fluid Simulation

(c) Finite Difference Time Domain

Figure 5.3 – **Grid Size Effect on Execution Time** Running time of three optimized applications as a function of the grid size. Each application is representative of a class of programs sharing a similar behavior when increasing the grid size. In a given class, we could observe that the behavior is the same independently of the dataset size or initial conditions of the simulation.

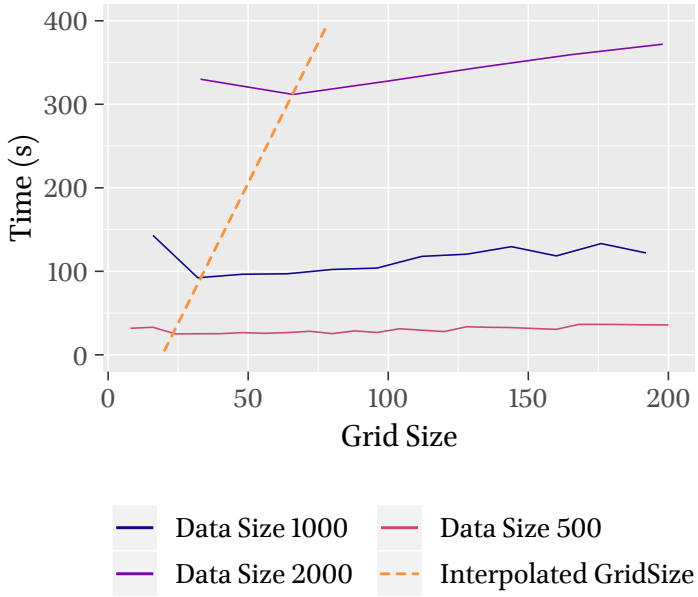


Figure 5.4 – **Finding the Granularity Empirically** Heat solver time in function of the grid size, execution with multiple dataset sizes. The grid size that performs the best can be interpolated using a linear regression.

5.5 Discussion on Output Deviation

Sections 4.3, 4.4 and 5.1, generate an adaptive version of a kernel tagged for approximation. Developers using approximate computing may want to know, to a certain extent, the effect of an approximate computing transformation on the result of a program.

For the general ACR case, how to find the information without achieving the precise computation is still an open question. It depends on the strategies, the monitoring function provided by the user and the data from where the monitoring extracts its information. However, we can provide an informative upper bound of what to expect in the following cases:

Threshold Monitoring The monitor uses a threshold T_h and the range of the application values is known to be $[Low, Big]$. Not updating or ignoring a value could generate an error of $|T_h - Low|$ or $|High - T_h|$ that can be multiplied or added depending on the operation considered.

Update Deviation A threshold T_h is placed on the derivative of the update function. If the derivative is continuous in the approximated location, one can bind the deviation of the output per update to $(1 + \alpha)T_h$ where α is the maximum second order derivative of the update function.

Domain Specific Information Some application can provide domain specific information that can be utilized for adaptive decision. For example, the k-means clustering adaptive kernel is based on a study of the convergence of the algorithm (see Section 7.1.6).

The stencil approximation belongs to the second category, where a threshold is used on the derivative of the update function. We also know which transformations were made to the kernel by Algorithm 9. Hence, our intuition is that an upper bound can be computed from the user defined deviation and the relative percentage of deleted or redistributed weight compared to the total weights. For example, if we consider that skipping the stencil entirely yields an error of E, deleting or distributing 50% of the weights should generate an error of $\frac{E}{2}$.

Unfortunately, these theories have not yet been thoroughly investigated and lack mathematical foundation. Nonetheless, we believe that our approach creates a sound basis for frameworks utilizing the specificities of applications to extract the maximum performance with automatic adaptive transformation.

Chapter 6

Automatic Data Compression

Partial differential equation (PDE) solvers are a class of applications that may require a large amount of memory for large scale simulations. The usual discrete implementation of such solvers requires huge multidimensional domain representing the physical properties of interest. The simplest but most memory-consuming data structure implementation of the domain is a multidimensional array representing a regular n-dimensional mesh. For example, Figure 6.1 shows the Kármán vortex street, a fluid simulation where 2D arrays are used to store the data. The fluid flows around a circular object, which induces a perturbation behind the object. To reduce the memory requirements, meshing techniques on a non-regular grid have been developed, but come at the price of a more complicated data management. For a maximum efficiency (in precision, time and space), they may be adaptive, allowing the non-regular grid to evolve during the execution of the simulation, refining the grid where the irregularities are located and coarsening it where the solution is smoother. Keeping track of these concepts and combining them requires a significant expertise. Moreover, implementing them is a time-consuming and error-prone development effort. We propose an abstraction allowing the developer to think about the data domain as a potentially infinite regular n-dimensional array that will be transformed by the compiler as a compressed tree data structure.

In this chapter we propose a preliminary work on an abstraction which aims at allowing the developer to think about its domain as a potentially infinite regular n-dimensional array that will be transformed by the compiler as a compressed tree data structure. The tree data structure exploits the wavelet transformation (see Section 6.1) to extract multiresolution information and to compress data. We show that this transformation allows us to analyze the data features at multiple scales, leading to further optimization opportunities like automatic adaptive transformation. We motivate our choice of wavelets because of their properties for our application domain. We study a proof of concept wavelet compression program to motivate the usability of such transformation when data are too voluminous to fit in the computer memory (Section 6.2). Lastly, we present complementary optimization to compression, e.g., using the tree structure efficiently or automatic adaptive code generation (see Section 6.3).

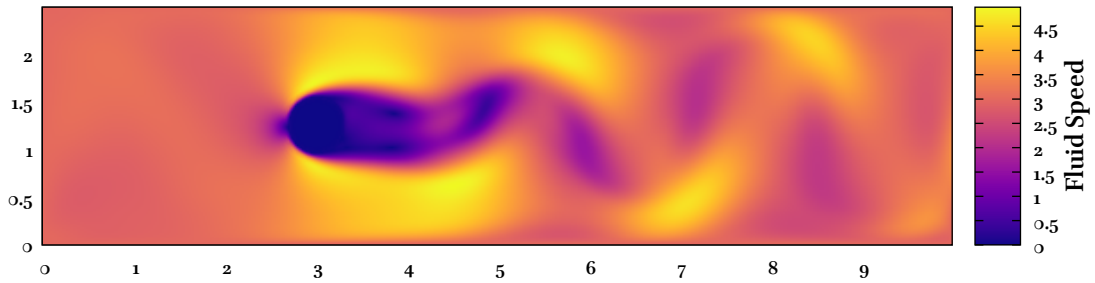


Figure 6.1 – **Kármán vortex street** The fluid flows from left to right and is disturbed by a circular object. The fluid accelerates to circle the object and an instability appears behind it. We can observe many homogeneous regions, where the fluid speed is constant. Being able to store these regions in a compressed representation saves memory for the application.

Domain of Interest

Writing application dealing with a huge amount of data is not an easy task. For example, simulating physics experiments with large domain size in a reasonable amount of time usually requires a cluster of computers. The simulation data is scattered among all the computers and the simulation is solved locally by each node. Writing such applications is complex, hence we propose an abstraction allowing developers to write easy-to-maintain codes while still being able to leverage the hardware capacity at their disposal.

Our method resembles helper libraries by providing abstraction which simplifies the developers work. In these libraries, data structures and internode communications are usually not visible to the programmer, which has only access to the higher level view exposed by the library interface. In our method we transform an application that uses multidimensional array data structures to use a more appropriate sparse representation implemented internally as a tree. Sparsity is an important characteristic because it allows bigger datasets to be processed on a single node without requiring access to slow storage areas or allocating more nodes.

Another aspect of the transformation is related to adaptive techniques, i.e., targeting precise computation where it matters. We rely on the wavelet transformation to get relevant information about the shape of the data at multiple scales (see Section 6.1). We aim at using this information to generate an adaptive grid automatically.

Our goal is to provide a convenient way to think about the data, with simple multidimensional arrays and let the compiler achieve the transformation to a sparse representation to be able to use this program in a situation where dense data arrays cannot be used, e.g., the data is too voluminous for the available memory or it is too expensive to do all the computation at this fine scale. The compiler will replace the array access by the new structure access. It may also modify the execution order of the compute intensive kernel accessing the tree data structure to extract better performance from the underlying hardware.

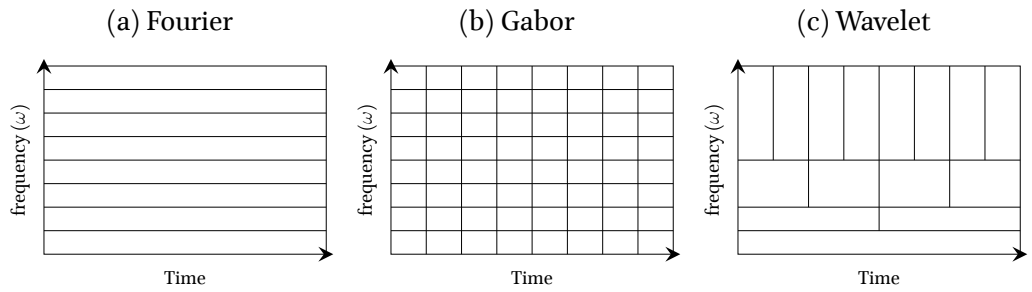


Figure 6.2 – Fourier, Gabor and Wavelet transformation time-frequency correlation

6.1 Data Compression and Multiscale Information

6.1.1 The Wavelet Basis

The wavelet transform is a routine tool for image and signal processing. Contrary to the Fourier transform [43] or Gabor transform [50], the wavelet transform provides both frequency and localization information at multiple scales. Figure 6.2 shows the frequency-time correspondence of the three transforms. The Fourier transform (Figure 6.2a) provides information about the frequency components of a signal but no time locality. Gabor introduced signal decomposition over *dictionaries* (set of functions) of time-frequency *atoms* (functions well localized in time and frequency). Figure 6.2b shows the Gabor transform of a signal where a window is translated in time and modulated in frequency, constructing the dictionary of time-frequency atoms, which are represented by rectangles in the figure. Each atom has a minimum surface area and the Heisenberg uncertainty principle links the temporal and frequency variance of the atom. This imposes limits which forbids the construction of an atom with conjoint narrow time and frequency resolution. Hence, the size of the sliding window is a trade-off between the lowest observable frequency and the time precision. More than one Gabor transform is required to conduct a multi-scale analysis of the same data.

The wavelet transform is another way of constructing dictionaries. It is constructed from a *mother wavelet* Ψ which is scaled and translated to construct the dictionary [69, 41, 56]. Figure 6.2c shows the time-frequency relation of these atoms for the wavelet transform. The wavelet transform provides a good trade-off for the analysis of signal at multiple scales while retaining good time locality. In only one transformation we obtain the multiresolution data, removing the need for redundant Gabor transforms. This multi-scale analysis property may be exploited to achieve precise computation only on the relevant parts of the function.

6.1.2 Wavelet Construction Properties

A multiresolution analysis with convenient properties is crucial to extract useful information from a signal. Our goal is to achieve a good compression ratio, i.e. a good approximation in the wavelet basis, in order to allow programs with a high memory footprint F_{mem} to be able to store its data on machines with a lower amount of physical memory $P_{mem} < F_{mem}$. Recent studies provide wavelets with various properties to apply in different scenarios [59, 40, 73, 13, 33, 79].

We choose the *coiflet* [40, 26] for its properties which are well suited for partial differential equations while providing a good compression ratio. It is defined by a scaling function Φ which carries the approximation of a signal at a given resolution, and a mother wavelet Ψ which carries the details necessary to increase the resolution of a signal approximation. The two functions have been crafted to yield the following properties:

Mother wavelet and scaling function form orthogonal basis This allows the separation of information at multiple scales. In addition, removing details does not change the total mass of the signal. It only decreases its energy. This is especially important in many physical applications.

The mother wavelet has N vanishing moments The first N terms of the Taylor's expansion of the signal do not contribute in the wavelet basis, i.e., functions that can be approximated by a polynomial up to the degree N will have vanishing wavelet coefficients. This allows for good compression of the data.

The scaling function has N vanishing moments This creates a function which is almost interpolating. In other words, the corresponding wavelet coefficients are almost local samples of the signal.

The scaling function is near symmetric Symmetry allows an easier application of the discrete wavelet transform on edge of discrete domains (e.g. images), where the data on the edge may be symmetrically replicated to avoid compression or distortion problems.

The scaling function has near linear phase Linear phase means that the wavelet transform will behave similarly at all frequency range.

The Wavelet Transform

The wavelet transform projects the signal to be analyzed on the wavelets basis. The wavelets are constructed by scaling and translating the mother wavelet and scaling function. In order to have an efficient transformation, the scaling function can be defined using a discrete filter called *conjugate mirror filter*. This definition of the scaling function is

$$\Phi(t) = \sqrt{2} \sum_{n=-p}^{p-1} h[n]\Phi(2t - n)$$

where h is a discrete filter of size $2p$. The wavelet function Ψ can also be defined in the same way and we can prove that its filter is then $g[n] = (-1)^{1-n}h[1 - n]$ for ensuring orthogonality. These filters allow for a practical implementation of the wavelet transform with a complexity of $O(n)$ for data of size n.

The fast orthogonal wavelet transform consists in consecutive application of the filters h and g defining the scaling function and the mother wavelet [88, 70]. This transformation is depicted in Figure 6.3. The g filter creates the local details, while the h filter creates a local approximation of the function free of the details. This step is repeated on the approximation generated by h to continue the analysis on a larger scale. The reverse wavelet transform works from top to bottom, recombining details and approximations to reconstruct the function.

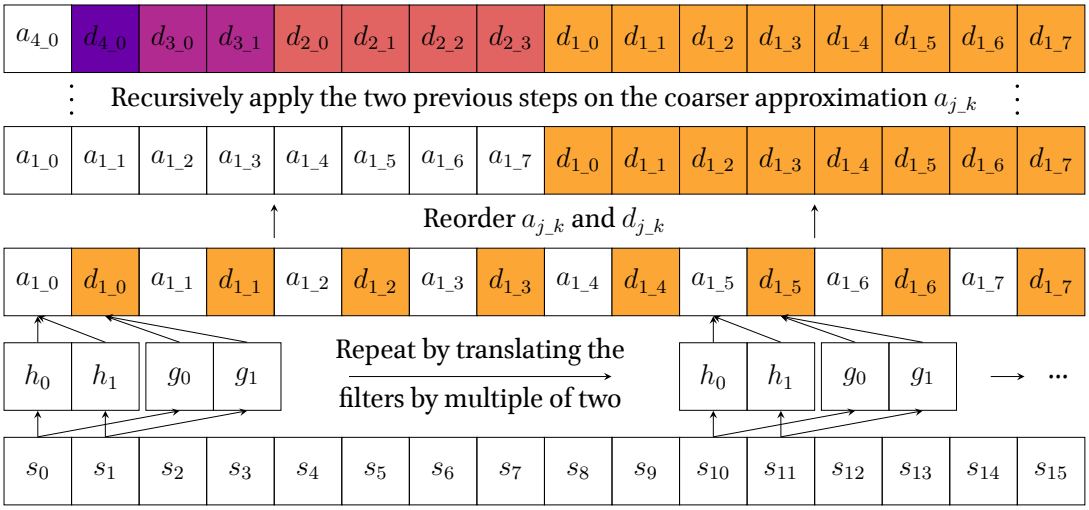


Figure 6.3 – **Discrete Wavelet Transform** The wavelet decomposition of an evenly sampled signal (s_i) with filters of size 2. The algorithm consists of an application of filters h and g which respectively compute a coarser approximation (a_{j_k}) and the details lost by applying h (d_{j_k}) at the level j and a translation of $2k$. This step is usually followed by packing the coarser approximation and details together. These two steps are applied recursively until the number of remaining coarse approximation (a_{j_k}) is lower than the size of a filter.

```
1 #define N 2048
2 #pragma sparsify<coif> foo bar
3 float foo[N], bar[N];
4 for (size_t i = 0; i < N; ++i)
5   foo[i] += bar[i];
```

The annotation at line 2 instructs the compiler to change the data representation of the arrays `foo` and `bar` to use the Coiflet. The data structure allocation, accesses and automatic compression are handled by the compiler.

Listing 6.1 – Example of code annotation to perform the data layout transformation

6.2 Using the Wavelet Transform for Data Compression

We propose a data layout transformation based on the wavelet transformation to store a dense dataset in a sparse data structure. This transformation yields a good compression ratio if the data is smooth enough. Mathematically, the transformation is limited to functions in $L^2(\mathbb{R})$, i.e., squared integrable functions $\int |f(x)|^2 dx < \infty$, hence we restrict ourselves to floating point arithmetic without infinite values.

From a programmer point of view, the approach works as follows. The developer writes the application using a simple dense data representation, i.e., an n-dimensional array, but instructs the compiler to use the sparse wavelet representation instead. Listing 6.1 sketches the implementation of such transformation. The user places an annotation inside the source code to use the capabilities of our data structure. For this example transformation, no other transformation than the data layout is applied. Taking full advantage of the new layout may require a new computation ordering. This and further optimization opportunities are discussed in Section 6.3.

6.3 Evaluation

We implemented a prototype to evaluate the potential compression ratio achievable on 1-dimensional data arrays. We implemented the coiflet wavelet transform using filters of size 6. Figure 6.4 shows the decomposition of a signal composed of a sine and exponential into a graded tree representation. We can achieve a 97% space savings of this test signal with a maximum error of 0.2% and mean error of 0.002% when comparing the approximated and the original function values. The destructive compression occurs when a detail which is really close to zero is actually set to zero to yield more compression because we don't store the details anymore. A sharp portion of the function, at each scale, has a color closer to blue and smooth ones closer to yellow. We can see from this representation that large portions of the tree can be compressed and that its shape gives us information about the function at multiple resolutions.

One observes in general much larger compression rates in higher dimension. In practice, if the compression rate is r in 1D it becomes r^D for a D-dimensional array [83]. This justifies our preliminary evaluation of the method only in 1D. We expect an increased efficiency in 2D or 3D.

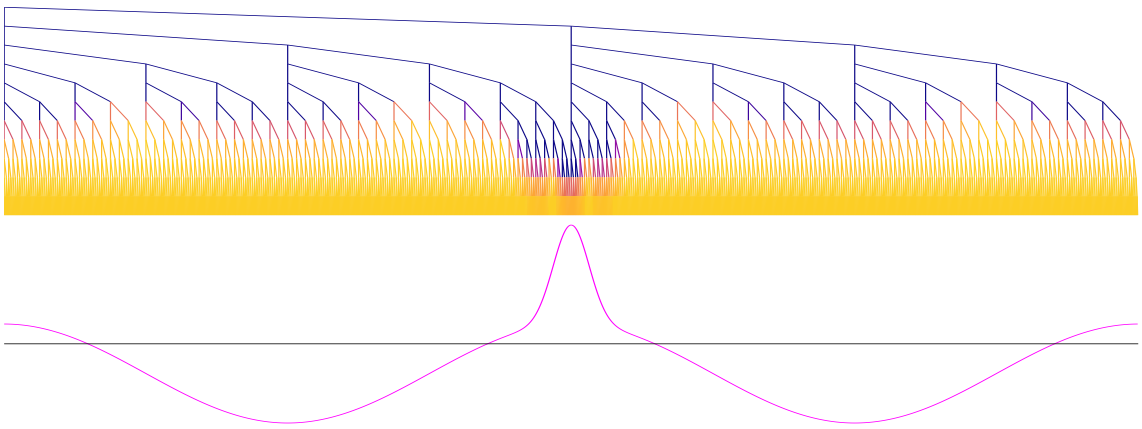


Figure 6.4 – **Wavelet Transtorm Tree Representation** Tree representation of the wavelet details from the signal present at the bottom. The edges of the tree is colored in a scale ranging from light yellow to dark blue corresponding respectively to low and high detail coefficients. Each level of the tree corresponds to a frequency level. The high frequencies are present near the leaves and the low frequencies closer to the root. The detail coefficient at a given scale has a high value if the signal is sharp or low if it is smooth.

Early Overhead Evaluation

We used our prototype (see Section 6.3) to measure a 7× slowdown between the time to access all the elements of the initial array, and the time to compress the whole array plus the time to access all its elements including decompression. This slowdown remains constant, disrespectfully of the array size. We consider it as quite encouraging since no specific optimization has been performed and considering the high potential compression ratio.

Leveraging Wavelet Sparse Information and Representation

Using a sparse representation based on the wavelet transform provides multiresolution information and a compact representation for the data. However, exploiting this representation efficiently and extracting algorithmic optimization is a real challenge. The automatic translation from dense to sparse representation (see Section 6.2) is the first step towards efficient automation of the data transformation.

We are in the process of evaluating this abstraction on a fluid dynamic simulation using the Lattice Boltzmann method [28]. This application is written in `C` using regular arrays with annotation instructing the compiler to use our special data structure (see Section 6.2). The computation is done at the finest scale (same as the dense arrays) while the data structure transparently compresses the data. Our goal is to evaluate the following optimization opportunities:

Loop ordering optimization A tree data structure is well suited to represent the wavelet transform. However the data traversal in the original loop is not likely to correspond to the new tree data layout, resulting in a bad data locality. This may be improved by applying loop tiling with appropriate tile sizes and tile ordering along with the data layout transformation.

Multiscale optimization Use the information provided by the wavelet transformation to generate a code that will do the computation at the scale where the details are high enough. An example algorithm to use the wavelet tree efficiently would be to firstly refine the tree to avoid losing details, solely achieve the computation for the tree leaves and finally compress the tree to erase the leaves with low details.

Overhead, error and compression ratio evaluation The overhead of managing the tree with the mentioned techniques has to be evaluated along with techniques to bound or predict the error coming from the algorithm used during the compression phase.

Chapter 7

Benchmarks and Evaluation

"It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong." Richard Feynman, theoretical physicist. In computer science just like in other sciences we need to empirically demonstrate or formally prove that our ideas help to improve the state of the art. In this thesis we provide application programming interfaces for adaptive computing. Hence, in this section we are going to evaluate the expressiveness of our interfaces on application codes, measure the difference in performance compared to the unoptimized program and measure the deviation induced by the approximations for both ACR (Section 7.1) and ASA (Section 7.2).

7.1 Using Adaptive Code Refinement

In this chapter, we use the ACR annotations introduced in Chapter 3 to optimize a set of applications. We rely on a prototype compiler [97] that takes as input an ACR annotated `C` source code and outputs an instrumented `C` file ready for adaptive computation as shown in Figure 3.2.

7.1.1 *LetItBench* Benchmark Set and Their ACR Annotations

In order to evaluate our adaptive technique, we searched for applications that are resilient to approximation. We investigated the possibility of using benchmarks introduced by other approximate compiler techniques, e.g., *AxBench* [128, 6], *ApproxBench* [92] or *PARSEC* [18]. Many applications of these benchmarks process the data in one pass. However, our adaptive technique needs to gather information before being able to adapt the computation. Some applications are great candidates for adaptive computing, for example *x264* from the *Parsec* benchmark suite which implements a decoder for the *H.264* video format. However, some of these applications use algorithms that are complex to analyze by a compiler, and sometimes by even humans. For the *H.264* example, the algorithm is specified by an 812-page document [118]. To help evaluation of works like ours, we propose a set of meaningful applications with an iterative kernel, that is not too complex for automatic analysis and can be analyzed by polyhedral tools. The benchmark set called *LetItBench* (Lenient to Errors, Transformations, Irregularities and Turbulence

Benchmarks) [98] is composed of standalone applications written in C, introduced below, and a benchmark runner based on CMake[1].

The benchmark runner uses a per-benchmark configuration file where the following information is defined:

- The source code location to download the benchmark project
- A set of "run" commands that define the parameters to be passed to the application to execute a benchmark
- The format for a space-separated CSV benchmark result file and the commands to gather these data after the benchmark execution
- Optional compiler or linker options

The runner provides the ability to run the benchmarks several times and gather all the run data in one location. We provide a script to print the data as bar graphs with error bars for visualization purpose. Figure 7.1 shows an example of generated plot.

7.1.2 Eulerian Fluid Simulation

We first evaluate ACR on a fluid simulation solving the Navier-Stokes equations [108]. It uses a non-conservative algorithm presented by Stam for the video-game industry [109]. Figure 3.1 shows a visual representation of the fluid density during the execution of this simulation. Stam proposes an iterative solver to reproduce smoke or flames in real time for scenes rendering. Thus, the solver must be as fast and lightweight as possible. The quality of the output should also be accurate enough as too much approximation would lower the perceived resolution. We empirically observed that less than five percent of deviation of the result is not visually perceptible. Important portions of the simulation are empty of fluid until an advanced time in the simulation. A convenient approximation strategy is to monitor the values of the density and to reduce the number of iterations of the solver in parts of the simulation where the density is close to zero.

In this simulation, only two variables are used, the fluid density ρ and the fluid velocity v. The algorithm is divided in two phases. The first phase, the density movement, updates the grid fluid density ρ to the next time step depending on the density present in neighboring cells, computing the diffusion. The second phase, the streaming, moves the density and computes the velocity in the next time step with respect to the velocity vector. Using runtime analysis of the application, we discovered that approximately 60% of the runtime is spent in the first phase and the remaining in the second phase. Inside these two phases, there is one function where most of the computation time of the application is spent. This function is sketched in Listing 7.1. It is a numerical solver using Gauss-Seidel relaxation iterative method to solve a linear equation.

Observation of the simulation data reveals that, for many scenarios, the smoke forms a funnel shape with a mushroom-like form at the top. These scenarios coincide with a source of fluid having a force applied at the base to simulate burning objects or particles

[1]We choose CMake because, as a build system, it can track the modifications of source files and issue an automatic recompilation when changes occur. Hence, we can modify the source ACR annotations and CMake will automatically recompile our code before the benchmark is executed.

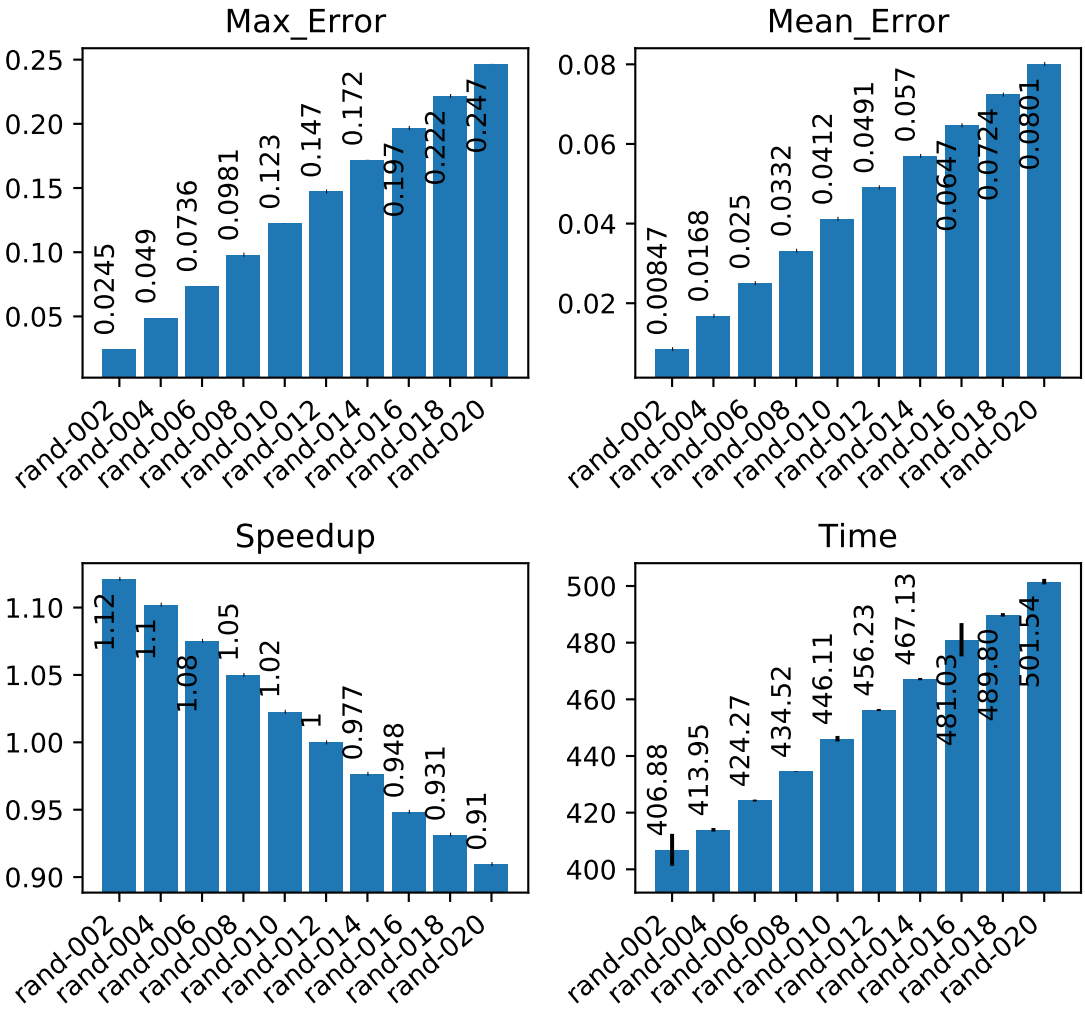


Figure 7.1 – **LetItBench Generated Plot** Plot generated from *LetItBench* space-separated CSV data. In this example, some computation of the heatsolve application (Section 7.1.3) are skipped randomly with a probability expressed in percentage and appended to the name of the benchmark *run*. Five executions of each run have been executed to generate this plot with error bars. The speedup can optionally be computed by the script, here we selected the `Time` data with the base value `rand-012`.

moving with the airflow (Figure 3.1 left). The solver is used in the diffusion and streaming steps of the algorithm. Inside the collision, the parameter 'a' of the function is set to the diffusion value of the fluid and inside the streaming step, 'a' is set as the viscosity of the fluid. Therefore, if the updated value and its neighbors are roughly the same, the tolerance factor of the iterative solver should be reached faster. Hence, the number of iterations of the solver can in this case be lowered. The initial number of iterations is set to a value that is big enough for any kind of simulation. This high value reduces the overall speed of the simulation and can be improved without a significant deviation of the result.

Following our strategy to reduce the number of iterations of the iterative solver, the first extensions to specify are the code alternatives. The lines 14 to 16 of Listing 7.1 define the alternatives that set the number of iterations of the solver to 1, 4 or 6, which is used whenever there is a really low, medium or a high density of fluid respectively. The next implementation step of the ACR extensions is to declare in which case the alternatives are used. We define three dynamic strategies lines 17 to 19 that orders the alternatives relatively to their level of approximation, fewer iterations of the solver is equivalent to a less precise solver. To select the convenient strategy at runtime, the *monitor* construct should point to the relevant problem-specific data, here the density 'x' line 11. The density is defined on a 2D plane with respect to the loop dimensions i and j. The function `density_val` is a mapping from the density values to the strategies values in the range $[0, 1, 2]$ and is used as the pre-processing for the monitoring. Inside a cell, the minimum possible strategy value is kept using the 'min' folding function. Lastly, the runtime enforces those alternatives on cells whose size is defined by the grid directive line 12. Figure 3.1 right shows a graphical representation of the dynamic monitor grid. We can observe that the regions of precise computations map to the same locations as the high density fluid on the left side representation.

An interesting aspect of this simulation is that it is not possible to generate the compile time tiled version due to the dependencies between iterations of the solver. A value at iteration k of the solver uses the neighbor values at the iteration $k-1$. Hence, it is not possible to use the data-celling transformation as it would break the dependencies between the iterations of k. Therefore, the optimized code is generated using the dynamic runtime.

These alternatives reduce the number of times that the statement code line 32 has to be executed between 70% and 30%, after 100 and 2000 time steps respectively. The observed kernel speedup relative to these two time steps are 6.19 and 2.1. The mean deviation of the result stayed below our set limit of 5%. However, a maximum deviation of 50% can be observed for values close to zero but when taken proportionally to the density values range, the change represents a fraction of $\frac{1}{10000}$, which is not significant.

The ACR extensions lead to a lowered utilization of resources without modification of the original algorithm. Specializing this kernel without ACR would require the developer to write the monitoring and the just-in-time generation of the optimized kernel manually. Such task is not trivial and may require multiple days of development on its own. On the other hand, with ACR, it takes a maximum of few minutes to compile and test various alternatives.

This first example demonstrates the simplicity of use of the ACR extension set. The sole addition of nine lines of language extensions was sufficient to add adaptiveness to

```
static inline unsigned char density_val(float a) {
  if (a < 0.1f) return 2;
  if (a < 2.f) return 1;
  return 0;
}

void lin_solve (int M, int N,
                float x[M][N],
                float x0[M][N],
                float a, float c) {

#pragma acr grid(50)
#pragma acr monitor(x[i][j], min, density_val)
#pragma acr alternative high(parameter, P = 6)
#pragma acr alternative medium(parameter, P = 4)
#pragma acr alternative low(parameter, P = 1)
#pragma acr strategy dynamic(0, high)
#pragma acr strategy dynamic(1, medium)
#pragma acr strategy dynamic(2, low)
#pragma acr checker-select(versioning,async)
  for (int k=0 ; k < P ; k++ ) {
    for (int i = 1; i < M-1; ++i)
      for (int j = 1; j < N-1; ++j)
        x[i][j] =
          ( x0[i][j] +
            a * (x[i-1][ j ] +
                 x[i+1][ j ] +
                 x[ i ][j-1] +
                 x[ i ][j+1])
          ) / c;
    set_bnd(M, N, x);
  }
}
```

Listing 7.1 – **Fluid Simulation Kernel** A Gauss-Seidel iterative solver of linear equations. Three levels of approximation are used: low, medium and high which map to modified number of iterations of the solver. The number of iterations of the solver is lowered by the runtime in the cells where the density is low or medium.

this application with the help of the compiler to generate low overhead optimized code at runtime.

7.1.3 Heat Equation

This application solves the steady state heat equation. It consists of a 2D discrete grid where each point interacts with its neighbors to spread the temperature in the material. The temperature values of the edges of the domain are set to a constant temperature and the simulation runs until the individual updates are below a user-defined threshold. Figure 7.2 shows an example of the heat distribution at the end of a simulation.

This is the main equation solved in this application:

$$\alpha \nabla^2 T = 0 \tag{7.1}$$

$$\alpha = \frac{k}{C_p \rho}$$

where:

α is the thermal diffusivity ($^{m^2}/_{s}$),

k is the thermal conductivity ($^{W}/_{m \cdot K}$),

C_p is the specific heat capacity ($^{J}/_{kg \cdot K}$),

ρ is the density ($^{kg}/_{m^3}$).

The main algorithm of this application consists in a single iterative linear solver that stops when the temperature updates during an iteration of the solver is lower than a threshold, i.e., has converged (Equation 7.1). The application implements the Jacobi, Gauss-Seidel and successive over-relaxation (SOR) iterative solver methods [53]. Each of them is available with a sequential, a vectorized (omitting Gauss-Seidel), a parallel and parallel-tiled kernels. The solver kernels have affine array accesses and a four-point stencil pattern which can easily be analyzed by a compiler. The simulation data consists in a two-dimensional temperature array which can be monitored to apply the adaptive techniques.

The optimization opportunities in this application lie in the iterative solver and the convergence of the solution. For this application we use adaptive techniques to reduce the precision in locations where the solution already reached the convergence threshold. The instructions at lines 19–21 in Listing 7.2 compute the simulation deviation between two steps and compare the maximum value to the convergence threshold as a stopping point (line 25). We add the monitor annotation at line 10 to use this already available information instead of computing it from the heat data. When the solution locally reaches stability, parameterized by `stop_criteria`, we will stop updating the solution in these zones. Hence, the monitor compares the delta to the `stop_criteria` and returns 1 if it is safe to use an approximation and 0 otherwise. We define a *zero-compute* alternative and links it to a dynamic strategy with a monitored value of 1.

The adaptive version reduces the number of executed updates by 77% and an observed speedup of 1.76 on a grid of one million elements. In this application we measure a deviation that is below the solver `stop_criteria` threshold. This is within the solver specification for a relatively good performance gain.

7.1.4 Finite-Difference Time-Domain (FDTD)

Finite-difference time-domain (FDTD) is used in computational fluid dynamics to solve Maxwell-Faraday differential electrodynamics equations [114, 129]. The software implements a solver which updates the magnetic integral of the equation as a first step and then updates the electric field at the same instant using the magnetic field.

Listing 7.3 includes the function which updates the electric field in a two-dimensional (x, y) space. The magnetic field is orthogonal to the electric field and resides in the z dimension, orthogonal to the (x,y) plane. Figure 2.3 shows a representation of the magnetic field over the 2D space of the simulation. The simulation consists in a wave starting at the left of the domain and traveling to the right. When the wave encounters the impermeable block at the middle, part of it is reflected back and the remainder continues as

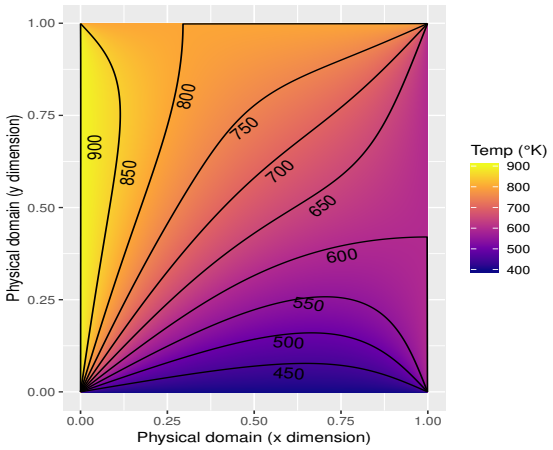


Figure 7.2 – **Heat Solver Simulation** In this simulation the border conditions are set to a constant temperature and the heat spreads by condution to fill the space.

```
static inline error_threshold(double val) {
  return val < stop_criteria;
}
unsigned solveJacobi(size_t sizeX, size_t sizeY, double stop_criteria,
    double t[sizeX][sizeY], double tNext[sizeX][sizeY]) {
  double max_delta = 0.;
  unsigned num_iter = 0;
  do {
    #pragma acr grid(30)
    #pragma acr monitor(delta, min, error_threshold)
    #pragma acr alternative low(zero-compute)
    #pragma acr strategy dynamic(1, low)
    for (size_t i = 1; i < sizeX - 1; ++i) {
      for (size_t j = 1; j < sizeY - 1; ++j) {
        tNext[i][j] = 0.25 *
          (t[i - 1][j] + t[i + 1][j] +
           t[i][j - 1] + t[i][j + 1]);
        double delta = t[i][j] - tNext[i][j];
        delta *= delta;
        max_delta = delta > max_delta ? delta : max_delta;
      }
    }
    max_delta = sqrt(max_delta);
    num_iter++;
  } while (max_delta > stop_criteria);
  return num_iter;
}
```

Listing 7.2 – **Heat conduction solver using the Jacobi method** The solver is executed repeatedly until the simulation settles, which happens whenever the maximum temperature delta is less than a given threshold, e.g., in this example the maximum difference is less than 10^{-4} degrees Kelvin.

seen on the top figure at step 700. Later in the simulation, i.e. the bottom part of the figure at step 1300, the waves on the right of the block did not reach the right wall yet while the ones on the left have already bounced back. However, it is clearly visible that many parts of the simulation retain a null magnetic field, i.e. that there is no current flowing in those parts. Therefore, for this application we consider the removal of the electric field computation where the magnetic field is null.

To nullify the computation inside a zone, the implementation uses the *zero-compute* ACR extension line 11 of Listing 7.3. The data of importance is the magnetic field Hz, which is set to be monitored at line 10. The strategy annotation line 12 adds a dynamic strategy which maps the *zero-compute* alternative to a value of 1. The pre-processing function '`hz_to_monitor`' returns one whenever the magnetic field is close enough to zero. The low alternative is active for cells filled with ones but a single zero triggers the original computation instead. The grid value is set to one fifth of the y dimension size and is drawn as overlay of the domain in Figure 2.4. The static strategy line 13 disables the computation for the portion of the simulation where impermeable object resides.

For this application the code uses dynamic code generation for maximum performance of the generated versions. The maximum observed deviation from the original output was 0.99% and a speedup of 1.35 after 5000 steps of the simulation. The ACR extensions allow reasonable speedup at the expense of very low deviation of the output. It is worth noting that after 4500 simulation steps with our initial setup, this technique does not help anymore because the magnetic field does not reach zero in a complete cell anymore. If all portions of the simulation require precise computations, adaptive methods are of no use. However, this technique can be helpful to quick start the simulation and once the precise state is met, the runtime can be disabled and the original kernel would finish the computation.

This application was optimized using ACR extensions to generate an adaptive code which is specialized at runtime. Writing adaptive and just-in-time code would be time-consuming and hard to debug without these extensions. For this application, we encountered a limitation of the adaptive methods where the runtime does not use approximate versions anymore. After that simulation point, the `ACR` runtime can be disabled to reduce the overhead.

7.1.5 Game of Life

Game of life (GOL) is a cellular automaton invented by the British mathematician John Horton Conway. The automaton is constructed using a set of rules describing the evolution of a grid of cells from one state to another. The said rules are very simple and can be used to model complex behaviors:

- A cell *becomes* alive if there are at least three alive cells in its direct surrounding at the time step before.
- A cell *stays* alive if it had two or three alive neighbors at the previous time step.
- Otherwise a cell is considered dead at the next step.

The implementation uses a 2D grid which represents the cell domain and an update function. At each time step the state of all the cells is updated using the previous step

```
unsigned char hz_to_monitor(double hzval) {
  return hzval > 0.1 && hzval < -0.1;
}

#pragma acr grid(J/5)
#pragma acr monitor(Hz[i][j], min, hz_to_monitor)
#pragma acr alternative low(zero-compute)
#pragma acr strategy dynamic(1, low)
#pragma acr strategy static([S,J] -> {[i,j] :       \
                            S+J/4 >= i >= S-J/4 and \
                            3J/4 >= j >= J/4}, low)
#pragma acr checker-select(versioning,async)
for (int i = 0; i < I-1; ++i) {
  for (int j = 0; j < J-1; ++j) {
    if (j == 0) // Borders
      Ey[i+1][0] += -alpha_Ey *
                    (Hz[i+1][0] - Hz[i][0]);
    if (i == 0 && j > 1) // Borders
      Ex[0][j] += alpha_Ex *
                  (Hz[0][j] - Hz[0][j-1]);

    Ex[i+1][j+1] += alpha_Ex *
                    (Hz[i+1][j+1] - Hz[i+1][j]);
    Ey[i+1][j+1] += -alpha_Ey *
                    (Hz[i+1][j+1] - Hz[i][j+1]);
  }
}
```

Listing 7.3 – **FDTD Kernel** Kernel updating the electric field in a 2D space of a FDTD simulation. The kernel is annotated with ACR compiler directives to disable the electric field computation whenever the magnetic field is close to null for all the values inside an ACR cell.

values. The update algorithm is shown in Listing 7.4. Advanced algorithms use the particularity that life automatons have many recurring cell patterns. They use macro-cells and a hash table to store and access the pre-computed results for these patterns [54]. We reused the macro-cell approach with our framework as it matches our proposed ACR-cells almost perfectly.

This application's optimization uses the ACR-cells as macro-cell to determine whenever the macro-cells are updated or not. The life rules state that a cell can only become alive if it has exactly three alive neighboring cells. Therefore, if an ACR-cell contains zero alive cell, the only possible cells that can rise from the dead are the ones on the borders with an alive neighboring ACR-cell. The computation of the ACR-cell is enabled if there is at least one cell alive inside of it.

The *zero-compute* and *interface-compute* alternatives are defined lines 8 and 9 of the GOL kernel Listing 7.4. These alternatives are linked to strategies in their order of approximation on lines 10 and 11. The ACR-cell grid is generated over the cell grid and represents the macro-cells. The monitoring sets an ACR-cell to use the *zero-compute* alternative whenever no cells are alive using the user defined `cell_to_precision` and the 'min' folding function line 6. The implementation uses the particularity of the 'stencil checking' to activate the border computation of neighbor cells if they are not active. Therefore, with 'stencil checking,' a non-active ACR-cell of alternative value 2 is set to the alternative value 1 if it has an active neighbor. This policy results from the definition of 'stencil' which computes the alternative value as $\min(2, 0 + 1)$, where 2 is the cell raw value, 0 is the cell value of one of its neighbors and 1 is added according to the stencil definition. The resulting program only computes the new state for active ACR-cells and for the surroundings of these, it only computes the borders with *interface-compute* if they are not active.

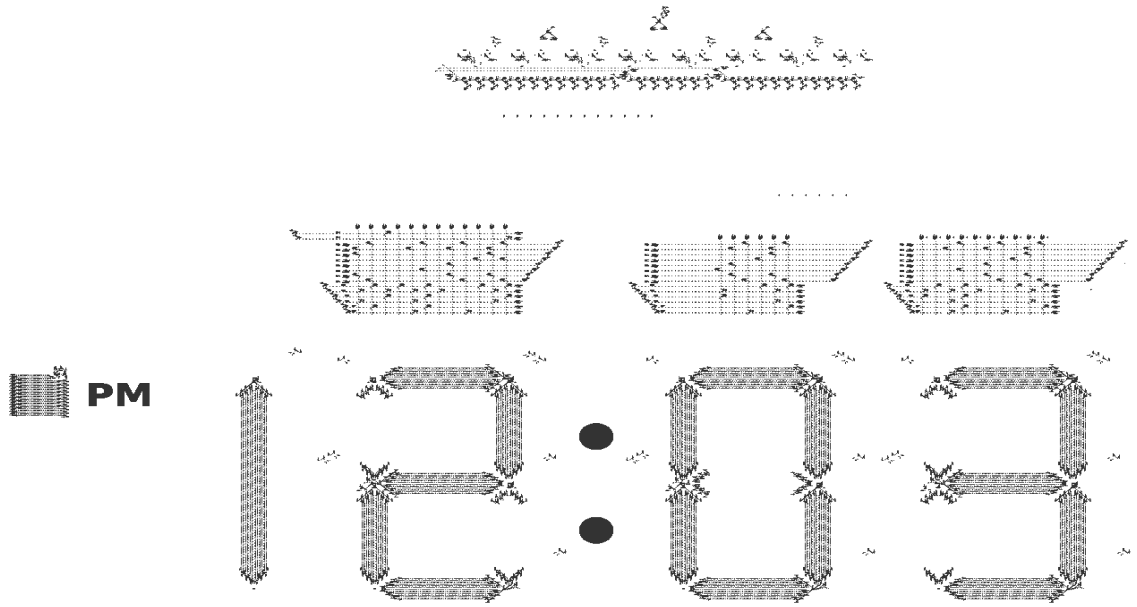


Figure 7.3 – **Game of Life Clock Automaton** This automaton simulates a 24h digital clock display. The logic at the far top is responsible for the number selection and the logic on top of the numbers triggers the digit segments.

```
unsigned char cell_to_precision(cell i) {
  return is_cell_alive(i) ? 0 : 2;
}

#pragma acr grid(macroCellSize)
#pragma acr monitor(previous_step_grid[i][j], \
                    min, cell_to_precision)
#pragma acr alternative low(zero-compute)
#pragma acr alternative medium(interface-compute, 1)
#pragma acr strategy dynamic(1, medium)
#pragma acr strategy dynamic(2, low)
#pragma acr checker-select(stencil,sync)
for (int i = 0; i < nb_row; ++i) {
  for (int j = 0; j < nb_col; ++j) {
    current_grid[i][j] =
      compute_new_value_of_cell(
        i, j, nb_row, nb_col,
        previous_step_grid);
  }
}
SWAP(current_grid, previous_step_grid)
```

Listing 7.4 – **Game of Life Kernel** The algorithm updates the state of the cell grid from one generation to the next one. The part of the cell grid where there is no living cell is not computed. The active ACR-cells are surrounded by cells that update the border of the ACR-cell and not their center because the GOL rules do not allow cell creation. This strategy both simplifies the computation that is needed and allows for a non-approximate GOL update algorithm.

The equivalent of the 'stencil checking' is not yet available with the static code generation version of our ACR implementation. Therefore, we used the dynamic version in synchronous mode. The cellular automaton state cannot be approximated, but the synchronous mode does not allow to compute a new state before the specialized code version is ready. Hence, there is no possibility for an ACR-cell state to be approximated. This method has been tested against an automaton simulating a digital clock and leads to a speedup between 1.4 and 2 after 20 and 160 generations respectively. With the help of ACR extensions we were able to quickly narrow the computations of the automaton in place where it matters, and to obtain respectable performance gains compared to the original code.

7.1.6 K-Means Clustering

K-Means is a partitioning algorithm which distributes n observations into k sets. The partitioning algorithm has the particularity to minimize the variance within clusters. Observations that are "related" or "closer" to each other have a greater chance to end up in the same cluster. The algorithm inside Listing 7.5 is actually a heuristic implementation of the NP-hard general problem. The algorithm has use cases in signal processing, cluster analysis and machine learning [72]. Most of them try to characterize noisy data using this algorithm. For example, in Figure 7.4

The implementation uses an iterative algorithm. At the start, it chooses k points from the observations as the center of the clusters. Then at each iteration, each observation

is attached to the center of the cluster that is at the lowest "distance." The "distance" is a function that takes two observations and returns lower values when the observations are closely related and bigger ones when they are less related. Once all the observations have been attributed to a cluster, the center of all the clusters are set at the barycenter of all the observations belonging to them. The algorithm loops unless the clusters stabilize, i.e., when points do not migrate between clusters, or when the ratio of unstable points is lower than a threshold (typically 5%).

Behavioral analysis of the algorithm shows that the cluster's center tends to migrate rapidly during a few iterations at the start of the algorithm, before staying approximately in the same position with fewer observation migrations [72]. Our ACR implementation can take advantage of this behavior by managing two versions of the cluster affectation. Whenever an observation has already been part of a cluster for many iterations, the probability that it will migrate to another cluster becomes very low. Unfortunately, the original algorithm has no such information available and the code needs to be slightly modified to add the "settling" time of an observation inside a cluster. The additional code is straightforward and sets the counter to zero whenever the observation switches from cluster and increments the value otherwise.

The 'complex_compute' function is the original algorithm which computes the distance between a given observation to every center and assigns the observation to the cluster which yields to the lowest distance. If the observation migrates to another cluster, it sets the convergence to false and the settle value of this observation to zero. Otherwise, the observation is not moved and its settle value is incremented. The alternative 'converged_compute' is less compute intensive than the original algorithm. It only affects the observation to its previous cluster. The alternative is used whenever the number of iterations that the observation has settled within the same cluster exceeds a threshold. The alternative is defined line 13 of Listing 7.5. It changes the function called when linked to the strategy of value one. The monitoring oversees the settle information and selects the alternative 1 whenever an observation has settled for more than seven iterations. The

Figure 7.4 – **K-Means Image Segmentation** On the left is the original picture to be segmented and on the right is the result after segmentation. The segmentation has mapped the red, green and blue 256 colors components of each pixel into one value in the range $[0-3]$. The right image is a grayscale reconstruction with the segmentation value multiplied by $\frac{255}{3}$. The zones sharing the same color in the right image had color coding that are similar in the original image.

```
1  void complex_compute(...);
2  void converged_compute(...);
3  unsigned char settleStrat(size_t a) {
4    return a > 7 ? 1 : 0;
5  }
6
7  do {
8    memset(centroids_point_num,0,k*sizeof(size_t));
9    has_converged = true; // Assume convergence
10 #pragma acr grid(1)
11 #pragma acr monitor(time_settled[pos], min, \
12                      settleStrat)
13 #pragma acr alternative low(code, \
14                              complex_compute = converged_compute)
15 #pragma acr strategy dynamic(1, low)
16   for (int pos = 0; pos < pointsss; ++pos) {
17     complex_compute(dimension, k, pos, centroids,
18         centroids_temp, data, centroids_point_num,
19         time_settled, &has_converged);
20   }
21
22   for (size_t centro = 0; centro < k; ++centro) {
23     for (size_t dim = 0; dim < dimension; ++dim)
24       centroids[centro][dim] =
25        centroids_temp[centro][dim] / total_points;
26   }
27 } while(!has_converged);
```

Listing 7.5 – **K-Means Kernel** The core algorithm where the observation points are placed into the clusters and the center of the cluster is updated. The application skips the cluster assignment complex function whenever an observation has settled for more than seven iterations within the same cluster. In that case, it is assigned to the same cluster it was at the previous iteration.

approximation is done per observation as defined line 10 with a grid of only one element.

The application was evaluated on linearized 2D images of animals and fruits of various resolutions. ACR is used with the static code generation technique. The maximum deviation of the result is 4.29%. It was observed for the biggest dataset of the benchmark which leads to a speedup of 1.54. The approximation tends to work better for larger datasets as the number of settled observations grows in correlation with the problem size. However, to use this approximation technique, we needed additional information and to construct a new structure to hold it. With the ACR extensions, we successfully leveraged this information to generate a specialized version with a low computational kernel version to achieve a valuable performance gain.

7.1.7 Overall Performance and Overhead

In this section we provide detailed benchmark results, the deviation caused by the approximation and the overhead characterization of ACR's runtime. In the first part, we focus on the application wall clock times and speedups with respect to the deviation of the results, then in the second part, we provide the methodology used to measure the overhead of the runtime. The experimental setup consists of a dodeca-core `Intel Xeon E5-2620 v3` with 16 GB of `ECC` RAM. The compilers and flags used are `GCC v7.1` (`-O3 -march=native`) and the `Tiny C Compiler v0.9.26` for fast just in time compilation of kernels. Our `ACR` extension implementation is Open Source [97].

Table 7.1 shows the averaged results obtained from runs on multiple input problems and data sizes. Our technique achieves significant speedup while keeping the deviation of the result relatively low. The proportion of cells per alternative remained equal for smaller and bigger datasets, while runs on big datasets tend to have a better response to ACR's adaptive methods. We believe that this difference comes from cache and memory latency effects. Different problem statements do not react equally to the same set of alternative parameters. An auto-tuner that explores the parameter space to better guide the developer in this task belong to future work. ACR's extensions allowed the compiler to generate the adaptive code automatically without modification to the original algorithm with little investment from the developer.

Application	Wall Time (s)		Speedup	Deviation	
	Original	Optimized		Mean	Max
Heat Solver	160.7	92.45	1.73	< 0.1%	< 0.1%
FDTD	24.53	18.67	1.31	0.29%	0.99%
GOL	170.47	93.58	1.82	None	None
Fluid Simulation	108.67	82.47	1.32	0.11%	4.64%
K-Means	15.73	11.37	1.38	1.7%	4.9%

Table 7.1 – Performance of optimized kernel versions against the original code version. The deviation of the results present both the average and the maximum value measured.

In order to measure the overhead induced by the runtime and code instrumentation, we saved the application parameters, the optimized kernels along with their utilization

timeline while the application was running. Then, programs are run a second time with the same parameters but with the ACR runtime replaced by the saved versions in the same order. This simulates the ideal scenario, i.e. where an "oracle" knows ahead of time which version to run. The monitoring and compilation time are thus eliminated.

Table 7.2 shows the overhead for the FDTD application. It is the application that exhibits the highest overhead because of the high number of generated kernels. The measured overhead increases with the complexity of the generated kernels, because the code generation and compilation take more time for complex polyhedra. Complex polyhedra are typically generated where there are local perturbations. We ran the simulations with the application's CPU affinity mask set on one or three CPU cores to identify the compilation and monitoring overheads. On one core, the application and runtime are competing for the CPU resources leading to a slowdown. On three cores, the runtime and the application can run concurrently with performance close to the optimal configuration. The overhead for three cores compared to the optimal is due to thread synchronizations and the asynchronous monitoring of the values. ACR takes advantage of modern multi-core architecture and shows low overhead on such systems.

The runtime takes advantage of multiple execution threads to overlap the computation with the monitoring and compilation of new kernels. While the compilation is sporadically called when a new version is requested, the monitoring thread must be fired at each termination of the kernel to verify the version. This generates an overhead that is visible when the runtime and the application have been pinned to the same CPU core. Otherwise the application falls back to the original kernel while the compilation is executed.

Iterations	Application time(s)			Overhead / Optimal	
	ACR 1 core	ACR 3 cores	Optimal	1 core	3 cores
500	15.47	12.47	12.20	27%	2.2%
1500	23.22	14.72	14.13	64%	4.2%
3000	36.69	19.31	18.42	99%	4.8%
5000	57.09	28.18	26.88	112%	4.8%

Table 7.2 – Overhead for the FDTD application at different steps of the simulation on one core, three cores against the optimal "oracle" version.

7.2 Using Adaptive Stencil Approximation

7.2.1 Heat Solver

This application, introduced in Section 7.1.3, is a solver for the steady state heat equation. It is used for material applications and physics simulation [51]. The main computational kernel of this application is shown in Listing 7.2.

This application is a good candidate for optimization using approximate computing techniques. By profiling the application, a developer may discover that the kernel is the biggest contention part of the program. Hence, this developer may add the following annotation in place of the ACR annotations in Listing 7.2:

`#pragma asa(stop_criteria)`. This annotation states that the following kernel can be optimized by the compiler to take advantage of adaptive techniques. In the following of this section, we explain the compiler process to extract the information necessary to generate the approximate computing version.

Heat Equation Precision Level Discovery The compiler generates then executes an instrumented version of the program on a small user-provided dataset to gather information about the array accesses at runtime. In this program, it happens that the kernel's written array switches between two addresses.[2] The compiler can automatically generate the comparison function as there are two arrays to compare, resulting in a function that is equivalent to line 18 of Listing 7.2. The developers already expressed their intention to allow approximation when deviation is below 10^{-4}. Therefore, the compiler generates the following ACR annotation to select the precision level:

```
#pragma acr monitor(tNext[i][j], min, diffT)
```

Where the `min` function selects the inside cell alternative of the most precise version and diffT is the pre-processing function that embodies the computation of the deviation between the value of arrays `tNext` and `t`. The monitor compares the deviation to 10^{-4} and returns 0 if the original code has to be executed or 1 if the approximation is considered as acceptable, allowing for one alternative along the original code.

Heat Equation Alternative Generation Algorithm 7 detects the stencil pattern. This stencil being too narrow for further optimization, the compiler modifies the kernel to merge two consecutive stencil computations to increase the width of the stencil, creating a loop where the iteration count is defined by a new variable `kernel_iter`, initialized to the value 2 which corresponds to the original kernel. A stencil-skipping alternative can be created by modifying the number of iterations of the kernel:

```
#pragma acr alternative oneIter(parameter, kernel_iter=1)
```

ACR requires an additional annotation to link the alternatives to the value obtained by the monitoring. Therefore, the following annotation links the monitoring folded value one to the early-terminated number of kernel iterations and the monitoring folded value zero is implicitly bound by ACR to the original computation:

```
#pragma acr strategy dynamic(1, oneIter)
```

Heat Equation Grid Selection The compiler tests the generated alternative on a small user-provided dataset to assess the viability of this alternative, here three. The compiler requires a metric to evaluate the deviation level of the alternative's output. The preferred metric for ACR is the relative difference or error with respect to the original application's output [116]. The compiler uses this metric to compare the result of the alternative version against the original application and requires that the difference be less than five percent in total. The application is evaluated w.r.t. multiple grid sizes to obtain the data plotted in Figure 7.5 left. Using this curve, it is possible to assess the ratio corresponding to the local minimum that will be used for other datasets, here $\frac{24}{500}$ which will be multiplied to the maximum between $sizeX$ and $sizeY$ to compute the new grid size.

Heat Equation Automatic Method Evaluation Table 7.3 shows the speedup obtained with our algorithm and the best value obtained by a search over all possible grid values. Results show both the benefits to exploit adaptive technique for this benchmark and that the automated process achieves a performance close to the best hand tuned version (see

[2] The corresponding statement is flagged using a specific annotation in our compiler prototype, but it may be discovered automatically as this is a well-known pattern in such codes.

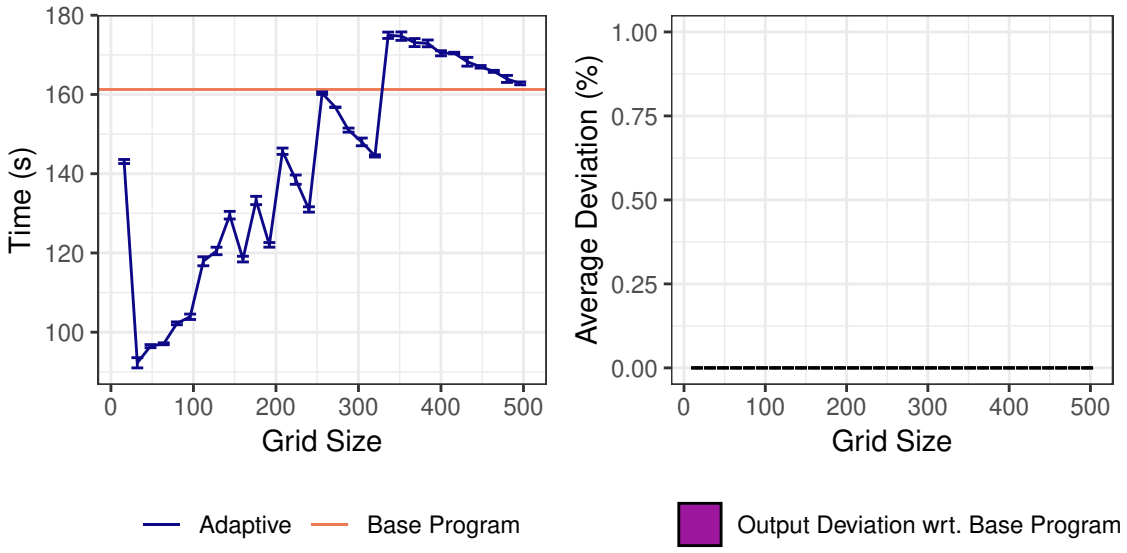


Figure 7.5 – **Grid Size Influence on Heat Solver**

Section 7.2.6). This application suffers from very low output deviation between the optimized and non-optimized application is really low as seen in Figure 7.5 right.

7.2.2 Eulerian Fluid Simulation

This application, introduced in Section 7.1.2, solves the Navier-Stokes equation using an iterative solver. This solver must be fast and the precision accurate enough to be human imperceptible as it is targeted for real-time video applications or games.

To optimize this application with ASA, the user has to replace the ACR annotation lines 12–20 in Listing 7.1 with the following annotation: `#pragma asa(1e-2)`. For this application, a deviation of 10^{-2} is selected with respect to the application parameters to ignore waves that are insignificant for the simulation.

Fluid Simulation Precision Level Discovery In this kernel, the compiler has assessed that the arrays x and $x0$ are swapped between each call to the kernel. Hence, it generates a function that will compare the two arrays to compute the update function derivative:

```
#pragma acr monitor(x[i][j], min, diffX)
```

Fluid Simulation Alternative Generation The compiler recognizes a stencil in the two inner dimensions of the kernel. This stencil repeats P times during the execution. Therefore, the compiler does not have to widen the stencil size but only has to reduce the number of stencil computation to generate an alternative:

```
#pragma acr alternative alt0(parameter, k=P)
#pragma acr alternative alt1(parameter, k=1)
#pragma acr strategy dynamic(0, alt0)
#pragma acr strategy dynamic(1, alt1)
```

Fluid Simulation Grid Selection Figure 7.6 left shows the application's behavior to the grid size. The best ratio extracted from the curve is $\frac{60}{500} \times \texttt{max}(M, N)$. The deviation of the output shown in Figure 7.6 is correlated to the gain in performance. The maximum deviation is measured in the local minima near a grid of size 45, where the optimization

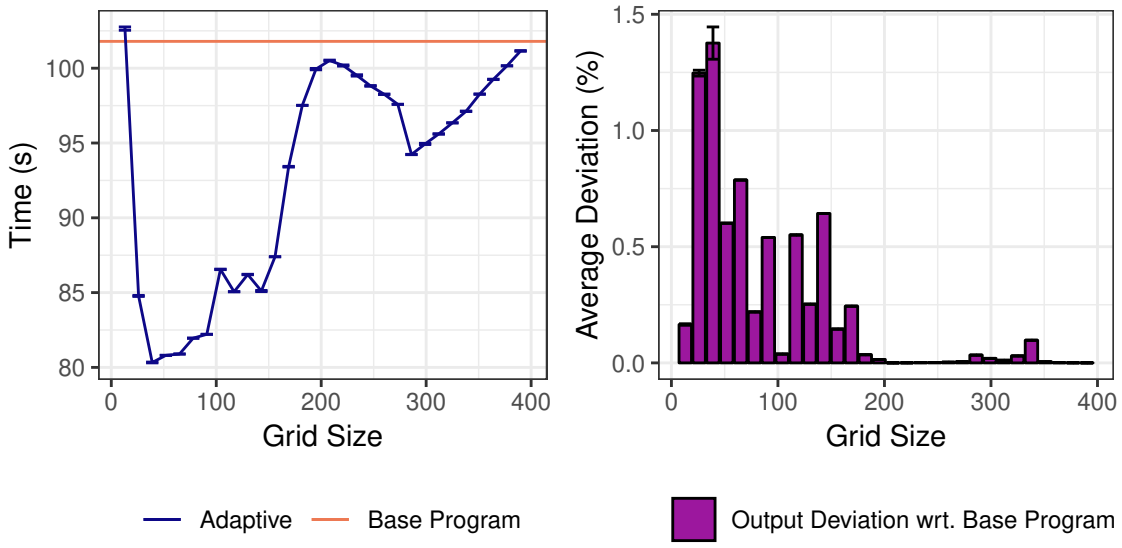


Figure 7.6 – **Grid Size Influence on Eulerian Fluid Simulation**

performs the best. We can see the presence of sweet spots, for example near a grid of size 100, where the deviation is low and the performance is respectable. Eventually, the automatically selected grid size matches the simulation accurately, creating a low deviation output.

7.2.3 Game of Life

Game of life is cellular automaton, introduced in Section 7.1.5.

This automaton has recurring patterns and empty areas that creates potential for optimization [54]. It is possible to create an adaptive version that does little computation in the empty zones. To optimize the main kernel of our benchmark implementation, we replace the annotation line 5–12 in Listing 7.4 with `#pragma asa(0)`. The value 0 means that we can use approximate computing techniques only if there is no deviation of the result.

Cellular Automaton Precision Level Discovery The compiler detects that the written array is switching between two addresses and will use it to compute the derivative:

```
#pragma acr monitor(current_grid[i][j], min, diff)
```

Cellular Automaton Alternative Generation To generate a non-approximate stencil version which allows the simulation cell states to spread while disabling the empty regions, the user decides to add the *interface-compute* alternative which activates the neighboring ACR cells in the locations where the simulation cells are active. The user can help the compiler to select an alternative by adding a back-end annotation, here the following ACR annotations:

```
#pragma acr alternative altInterface(interface-compute, 1)
#pragma acr strategy dynamic(1, altInterface)
```

The compiler can then generate the link between the monitoring and the alternative:

```
#pragma acr alternative altZero(zero-compute)
#pragma acr strategy dynamic(2, altZero)
```

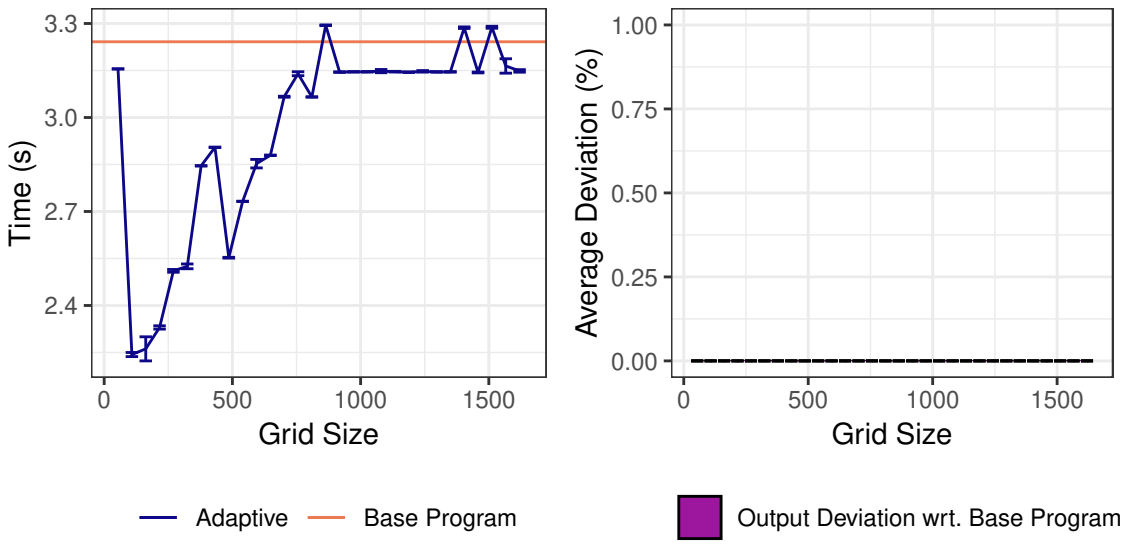


Figure 7.7 – **Grid Size Influence on Game of Life**

Cellular Automaton Grid Selection This application follows the same pattern as the heat solver Figure 7.7. The best grid ratio over data size is $\frac{275}{8000} \times \max(nb\_row, nb\_col)$. The output of the application remained unchanged after the addition of adaptive computations.

7.2.4 Finite Difference Time Domain (FDTD)

Finite-difference time-domain implements a solver for the Maxwell-Faraday equations of electrodynamics differential equations [114, 129]. The application, introduced in Section 7.1.4, presented the kernel that updates the value of the electric plane E as a function of the magnetic field H. This application's singularity resides in its two-phase algorithm, the electric field update followed by the magnetic field update.

To use ASA, the developer can replace the annotations at lines 5–12 with the following one: `#pragma asa(1e-3)`, which specifies a minimum update deviation of 10^{-3} before using approximations.

FDTD Precision Level Discovery This kernel updates two arrays at the same time, `Ex` and `Ey`. The array pointers are not modified during the execution of the program. The compiler arbitrarily chose to monitor `Ex` (in such a case, it is likely that the two updates are correlated, minimizing the impact of that choice: there is actually a correlation which can be statically analyzed in this benchmark). The ACR annotation generated is:

```
#pragma acr monitor(Ex[i][j], min, diff)
```

FDTD Alternative Generation For this application, the stencil merging technique could not help because the arrays are not swapped and the array `Hz` is not written inside the kernel. Hence, the only remaining possible alternative optimizations are stencil skipping and narrowing:

```
#pragma acr alternative alt1(zero-compute)
#pragma acr strategy dynamic(1, alt1)
```

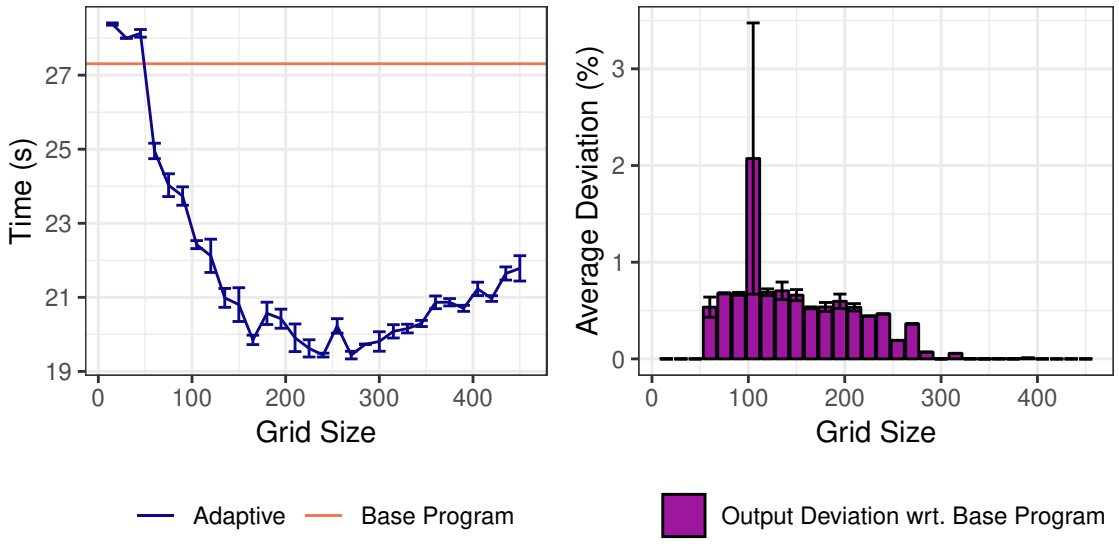


Figure 7.8 – **Grid Size Influence on FDTD**

FDTD Grid Selection Figure 7.8 left shows the application's runtime with respect to the grid size. For this application, the maximum grid size is 3000 and the best grid size is located in the plateau between 200 and 300 with $\frac{240}{3000} \times \max(I, J)$. The deviation is almost constant, with the exception of an outlier that happens to peak at 4% deviation on an execution of the program. It is the only monitored case where the adaptive runtime behaved quite differently between runs.

7.2.5 K-Means Clustering

K-means is used for data characterization, e.g., image processing, and machine learning [72]. Its purpose is to place N observations into B buckets where the observations being the closest will appear in the same bucket or centroid. The main kernel of the application has been introduced in Section 7.1.6 in Listing 7.5. Firstly, the algorithm places the observations into the closest bucket using a comparison function. Secondly, the barycenter of the bucket is updated with the observations belonging to it. The algorithm caries on until all the observations settle.

To use `ASA` the following annotation can be added in place of the `ACR` annotations lines 8–15 in Listing 7.5: `#pragma asa(0)` because we want to apply approximation on the points that do not move.

K-means Precision Level Discovery The only available array to monitor is `point_centr_map`:

```
#pragma acr monitor(point_centr_map[pos], min, diff)
```

K-means Alternative Generation Having access to the whole loop, the compiler uses the stencil merging technique to reduce the iteration count by half in the approximated regions by applying stencil skipping:

```
#pragma acr alternative oneIter(parameter, kernel_iter=1)
#pragma acr strategy dynamic(1, oneIter)
```

K-means Grid Selection We used the K-mean algorithm to characterize images of

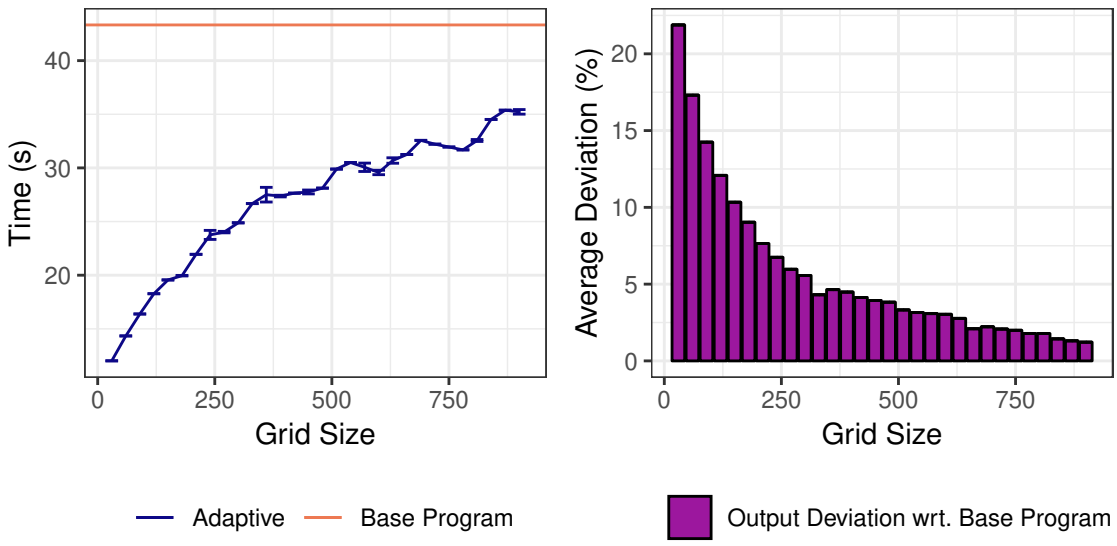


Figure 7.9 – **Grid Size Influence on K-Means**

different sizes, ranging from 640×425 to 4288×2848. The best ratio extracted for this application is $\frac{90}{272000} \times points$. The output deviation of this application follows a inverse logarithmic curve with respect to the grid size. The execution time looks like a logarithmic curve where the plateau is the base program runtime if there was no overhead. We think that this has to do with the step at which the points are not being computed. A small grid cell is more likely to use the approximation faster making the points stay in the same cluster. This creates a situation where other cells stabilize faster, creating a cascading effect towards a more approximate solution.

7.2.6 Performance Evaluation

In this section we compare the application's performance against the original code version and a hand-tuned version using the ACR annotations. Table 7.3 shows the performance and deviation of the output results of our method against user search of the parameters. We can see that, while the performance of ASA is lower than what a specialist can extract with the ACR annotations, our technique achieves sensible performance gains often close to hand-tuned versions. The deviation of the output is similar and sometimes better with the hand optimized versions and well below five percent, except one over-optimization with K-means where the automatic method reached nine percent deviation while the hand-tuned version was selected with a bigger grid size which allowed for less approximation.

ACR uses a multithreaded infrastructure to generate and compile the adaptive versions in parallel to the program execution. This requires resources which may not be exploited due to Amdahl's law [57]. Furthermore, adaptive methods are complementary to parallelization techniques, and information from the adaptive grid may be used for a better resource scattering.

Table 7.3 – ASA and hand tuned ACR version performance and deviation of the output results. The heat solver and K-Means applications are run with increasing problem sizes. The FDTD simulation is terminated at a different time stamp, with the same parameters. The fluid simulation is executed with increasing problem sizes and different simulation setups.

Application	ASA		Hand ACR	
	Speedup	Deviation	Speedup	Deviation
Heat	1.25	$< 0.001\%$	Training set	
Heat medium	1.68	$< 0.001\%$	1.76	$< 0.001\%$
Heat big	1.81	$< 0.001\%$	2.03	$< 0.001\%$
Fluid	1.1	0%	Training set	
Fluid flame	1.17	0.8%	1.27	1.4%
Fluid vortex	1.16	0.01%	1.19	0.01%
FDTD 500	1.12	0%	Training set	
FDTD 1500	1.16	0%	1.22	0%
FDTD 3000	1.24	0.1%	1.4	0.5%
GOL	1.44	0%	Training set	
GOL big	2.09	0%	2.2	0%
K-means	2.14	3.58%	Training set	
K-means medium	1.51	0.7%	2.50	2.62%
K-means big	2.18	9.03%	1.57	4.48%

Chapter 8

Conclusion and Perspectives

High performance computing developers are usually expected to be knowledgeable about the optimization opportunities provided by the hardware in order to write efficient programs. This thesis, along with other work pushes new programming models to allow non-specialists to exploit the maximum performance from their machines. For example, a domain specific language (DSL) provides abstractions for better productivity and performance. However, each DSL has a narrow application field, in terms of supported architectures and types of problems that it helps solving. From our perspective, a programming language with extensions provides more convenience to solve a wider range of problems. This thesis explores a new language extension set to exploit approximate computing opportunities in applications that are resilient to errors.

Our language extension set is based on optional annotations to provide relevant approximation information to the compiler. With these annotations the programmer can specify the location of compute intensive portions of the application and let the compiler optimize them with approximate computing techniques. We believe that our method provides developers with a good understanding of which approximate transformations are being applied. Furthermore, the annotation location helps them identify the scope of the approximations compared to previous work on automatic approximation optimizers.

Our approximate computing framework is inspired by adaptive mesh refinement techniques (AMR) from the numerical analysis field, i.e., the amount of details of the solution to a problem is denser in the locations where more precision is required and less dense where the solution can be approximated without introducing a significant error. Our framework extracts information from the data at runtime to apply approximate computing in an "adaptive" way. Contrarily to AMR, we do not coarsen the solution in the regions of lesser interest to reduce the precision. Instead, we rely on computation relaxation and program transformations to generate approximate code versions to use in these regions. The approximate versions have a reduced computational intensity and may produce a deviation with respect to the original program. To the best of our knowledge, no other work before ours provides an interface or automatic methods to generate an adaptive program from a source code.

Our framework targets a class of applications whose output may slightly be altered without losing meaningful information. For example, image and signal processing applications do not need to generate a perfect output when the difference is barely perceptible by a human being. Our annotations allow for fast prototyping of multiple approximation

techniques, e.g., to try new ideas and see if approximate computing can be beneficial. It can also be utilized during the optimization stage of application development, after implementing and testing the program, to take advantage of approximate computing optimization techniques.

Annotations for Approximate Computing The Adaptive Code Refinement (ACR) language extension set opens up new possibilities for software developers who need to exploit approximation to trade precision for speed. The language extension set, presented in Chapter 3, provides approximation and adaptive capabilities to existing languages through specific annotations to be exploited by both a compiler and a runtime system. The annotations provide the following semantic information:

Granularity Adaptive approximation techniques target the computational resources in regions where it matters. ACR allows to define the granularity at which the approximate decision takes place. It decomposes the program solution into cells, which is our "approximation atom." For each cell, whenever an approximation is beneficial, it is applied to the contents of the cell.

Code Transformation Approximation techniques are achieved, e.g., with algorithmic tuning or removal of computations in selective conditions. Such possibilities are offered by ACR for a new level of flexibility and productivity, through alternatives defining a transformation of the source code using approximate techniques.

Data Monitoring Adaptive techniques react to changes in the solution at runtime. Hence, ACR specifies how to extract and process the information needed to take an adaptive decision.

Strategy Approximation transformations can be applied in different ways to create the approximate code versions. ACR defines which version the compiler will enforce on each cell with respect to the approximation level gathered by the monitor.

The compiler uses this information provided through annotations to generate an adaptive version of the program (see Chapter 4). ACR builds on the polyhedral representation of programs to generate monitoring and approximated code with low control overhead and limited deviation of the results. Furthermore, ACR's runtime includes several mechanisms for the user to control the precision (Section 4.4). The strategies can be enforced for the duration of the program using static strategies or adaptively depending on the dynamic state of the data. To the best of our knowledge, ACR is the first language extension set dedicated to approximation that covers such a wide range of approximation strategies.

Adding a few lines of code, ACR allows to exploit approximate computing and to achieve speedups in the range $[1.1, 2.5]$ in our representative benchmarks while maintaining a good precision. It is also important to notice that we are accelerating *sequential* codes while exploiting few cores to minimize our runtime overhead.

Adaptive Stencil With one simple annotation, our method allows for the first time developers to delegate to the compiler the time-consuming task of optimizing an appli-

cation's kernel using adaptive approximation techniques (see Section 3.4). We presented three important features that the compiler has to extract from the source code to generate a pertinent adaptive version (Chapter 5). Precision level discovery, allows the compiler to choose a precision level from the application's dynamic data. This information is critical to find where the application needs the most precision and where more or less aggressive approximation can be used. The alternative generation requires the compiler to determine what kind of computation is performed by the kernel to apply appropriate code transformations. We used pattern matching to target stencil computation and proposed to merge multiple stencil steps to unleash more optimization opportunities. The third information, the granularity, looks for the best performance/precision tradeoff. With a grid too thin, the adaptive overhead will be too high and with a grid too coarse the perturbation will not be captured by the adaptive grid.

We evaluated our method on a set of representative applications resilient to inexact computations (see Chapter 7). We showed that we can extract enough information from these kernels and their profiling to generate an adaptive optimized version automatically. We also showed that the compiler can be instructed to use a specific alternative when the user provides it. Finally, we provided experimental evidence that the generated adaptive versions perform better than the original versions while maintaining a good quality of the result.

Data Compression Writing an application with abstractions allows developers to focus their efforts on the problem rather than its implementation. In Chapter 6 we proposed a compiler-assisted data layout transformation which provides transparent compression and multi-scale information of square integrable functions exploiting wavelet transform. Provided the data is smooth enough, e.g., application in partial differential equations, image and signal processing, we allow the developers to think about their data as dense multidimensional arrays. We show that this technique may achieve high compression ratio and allow the data to be stored in main memory, even when the dense array may not fit. The wavelet transformation behind the data transformation provides a multiresolution analysis which opens a wide range of optimization opportunities.

Perspectives

Adaptive Grid The static grid used by our framework may be too restrictive for some applications. There is a balance to find between the performance gain that can be obtained from coarser cells, and the overhead due to adaptive grid management. Further investigation may determine whether an adaptive grid may use the same kind of strategies as a static grid or not.

The wavelet transform is an interesting area to explore to achieve automatic adaptive grid generation. The result of the transformation can be stored as a k-dimensional tree (2D example in Figure 6.4) where the depth may represent the level of refinement. The amount of detail may be used to select the right level of refinement.

Another area worth exploring based on k-dimensional trees would be to generate tasks from cells. A cell, here task, can split in 2^k subtasks when more precision is required or otherwise be merged back to a single task when less precision is required. The cell splitting can be forbidden after a given tree level to reduce the task runtime overhead.

Thanks to the property of k-dimensional trees, where a unique index can be assigned to each node of the tree, each task can communicate with the relevant task for merging and a scheduler can smartly schedule and place the communicating tasks.

Deviation In this work, we focused on the adaptive code generation from an application kernel. We seek to minimize the deviation induced by the approximated code versions while exploiting its potential to gain in performance. However, we did not explore ways to provide strict guarantees for the deviation of the result. Whether this might be feasible or not because of the liberty given through the alternative annotation is still an open question. Although, it might be, provided a restricted set of transformations. Some ideas may be borrowed from the wavelet transform, which provides error bounds when using the destructive data compression [69].

Adaptive Computing for PDE We believe that a compiler could generate the code that handles the interface between two grid levels for partial differential equations. Research in this direction has to look for the right balance between the amount of information to put as annotation and what information can be extracted from the kernel. This may be a kind of hybrid DSL, where the implementation is left free to the user, but the adaptive portions are generated by the compiler.

Distributed Architecture and Optimization Our approximate computing runtime implementation has shown respectable performance gain on a single node. The use of complementary code optimizations, such as polyhedral code optimizations and parallelization techniques [55, 21, 10], should be investigated. Future work should also evaluate the potential of this technique on distributed platforms. The adaptive information may be managed on each node or distributed, resulting in potential communication overhead. Communications savings may be achieved for approximated cells.

Data Representation We provide some preliminary work on data layout transformation for compression with the wavelet transform in Chapter 6. Transforming a dense data representation as a sparse one has proved to be a promising direction. We believe that there is also an opportunity in changing the way numbers are represented, for example by using fixed point arithmetic [105] or posits [58].

Bibliography

Personal Bibliography

[99] Maxime Schmitt, Cédric Bastoul, and Philippe Helluy. "A language extension set to generate adaptive versions automatically". In: *Oil & Gas Science and Technology* 73.52 (2018). DOI: 10.2516/ogst/2018049.

[100] Maxime Schmitt, Philippe Helluy, and Cédric Bastoul. "Adaptive Code Refinement: A Compiler Technique and Extensions to Generate Self-Tuning Applications". In: *IEEE 24th International Conference on High Performance Computing*. Jaipur, India, Dec. 2017, pp. 172–181. DOI: 10.1109/HiPC.2017.00028.

[101] Maxime Schmitt, Philippe Helluy, and Cédric Bastoul. "Automatic Adaptive Approximation for Stencil Computations". In: *Proceedings of the 28th International Conference on Compiler Construction*. Washington, DC, USA, 2019, pp. 170–181. DOI: 10.1145/3302516.3307348.

[102] Maxime Schmitt, Philippe Helluy, and Cédric Bastoul. "Think Unlimited and Compress Data Automatically". In: *Conférence d'informatique en Parallélisme, Architecture et Système* (*COMPAS*). Anglet, France, June 2019, pp. 1–7.

[103] Maxime Schmitt, César Sabater, and Cédric Bastoul. "Semi-Automatic Generation of Adaptive Codes". In: *7th International Workshop on Polyhedral Compilation Techniques*. Stockholm, Sweden, Jan. 2017, pp. 1–7. URL: https://hal.inria.fr/hal-01655456 (visited on July 2019).

General Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. "Selective Memoization". In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New Orleans, Louisiana, USA, 2003, pp. 14–25. DOI: 10.1145/604131.604133.

[2] Horé Alain and Djemel Ziou. "Image Quality Metrics: PSNR vs. SSIM". In: *20th International Conference on Pattern Recognition*. Istanbul, Turkey, Aug. 2010, pp. 2366–2369. DOI: 10.1109/ICPR.2010.579.

[3] Ray Andraka. "A Survey of CORDIC Algorithms for FPGA Based Computers". In: *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*. Monterey, California, USA, 1998, pp. 191–200. DOI: 10.1145/275107.275139.

[4] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Dublin, Ireland, 2009, pp. 38–49. DOI: 10.1145/1542476.1542481.

[5] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. "Language and compiler support for auto-tuning variable-accuracy algorithms". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. Chamonix, France, Apr. 2011, pp. 85–96. DOI: 10.1109/CGO.2011.5764677.

[6] *Approximate Computing Benchmark*. URL: http://www.axbench.org/ (visited on July 2019).

[7] Woongki Baek and Trishul M. Chilimbi. "Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation". In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Toronto, Ontario, Canada, 2010, pp. 198–209. DOI: 10.1145/1806596.1806620.

[8] Utpal Banerjee. *Loop transformations for restructuring compilers: the foundations*. Springer Science & Business Media, 2007.

[9] Cedric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. Antibes, Juan-les-Pins, France: IEEE Computer Society, Oct. 2004, pp. 7–16. DOI: 10.1109/PACT.2004.1342537.

[10] Cédric Bastoul. "Contributions to High-Level Program Optimization". Habilitation à diriger des recherches. Paris-Sud University, France, Dec. 2012. URL: http://icps.u-strasbg.fr/~bastoul/research/papers/Bastoul_HDR.pdf.

[11] Cédric Bastoul. "Mapping Deviation: A Technique to Adapt or to Guard Loop Transformation Intuitions for Legality". In: *Proceedings of the 25th International Conference on Compiler Construction*. Barcelona, Spain: ACM, 2016, pp. 229–239. DOI: 10.1145/2892208.2892216.

[12] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. "Putting Polyhedral Loop Transformations to Work". In: *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*. College Station, TX, USA, Oct. 2003, pp. 209–225. DOI: 10.1007/978-3-540-24644-2_14.

[13] Guy Battle. "A block spin construction of ondelettes. Part I: Lemarié functions". In: *Communications in Mathematical Physics* 110.4 (Dec. 1987), pp. 601–615. DOI: 10.1007/BF01205550.

[14] Fabrice Bellard. *Tiny C Compiler (TCC)*. URL: https://savannah.nongnu.org/projects/tinycc (visited on July 2019).

[15] M.J. Berger and P. Colella. "Local adaptive mesh refinement for shock hydrodynamics". In: *Journal of Computational Physics* 82.1 (1989), pp. 64–84. DOI: 10.1016/0021-9991(89)90035-1.

[16] Marsha J Berger and Joseph Oliger. "Adaptive mesh refinement for hyperbolic partial differential equations". In: *Journal of Computational Physics* 53.3 (1984), pp. 484–512. DOI: 10.1016/0021-9991(84)90073-1.

[17] Włodzimierz Bielecki and Marek Pałkowski. "Tiling arbitrarily nested loops by means of the transitive closure of dependence graphs". In: *International Journal of Applied Mathematics and Computer Science* 26.4 (2016), pp. 919–939. DOI: 10.1515/amcs-2016-0065.

[18] Christian Bienia. "Benchmarking Modern Multiprocessors (PARSEC)". PhD thesis. Princeton University, Jan. 2011.

[19] Christopher M Bishop. *Pattern recognition and machine learning*. Information science and statistics. Softcover published in 2016. New York, NY: Springer, 2006. URL: http://cds.cern.ch/record/998831.

[20] OpenMP Architecture Review Boards, ed. *OpenMP Specification*. Version 5.0. Nov. 2018. URL: https://www.openmp.org/specifications/ (visited on June 2019).

[21] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 101–113. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375595. URL: http://doi.acm.org/10.1145/1375581.1375595.

[22] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. "Uncertain<T>: A First-order Type for Uncertain Data". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Salt Lake City, Utah, USA: ACM, 2014, pp. 51–66. DOI: 10.1145/2541940.2541958.

[23] Surendra Byna, Jiayuan Meng, Anand Raghunathan, Srimat Chakradhar, and Srihari Cadambi. "Best-effort Semantic Document Search on GPUs". In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 86–93. DOI: 10.1145/1735688.1735705.

[24] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. "HELIX-UP: Relaxing Program Semantics to Unleash Parallelization". In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. San Francisco, California, USA, 2015, pp. 235–245. DOI: 10.1109/CGO.2015.7054203.

[25] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. Indianapolis, Indiana, USA, 2013, pp. 33–52. DOI: 10.1145/2509136.2509546.

[26] Dana Černá, Václav Finěk, and Karel Najzar. "On the exact values of coefficients of coiflets". In: *Central European Journal of Mathematics* 6.1 (Mar. 2008), pp. 159–169. DOI: 10.2478/s11533-008-0011-2.

[27] Chun Chen. "Polyhedra Scanning Revisited". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Beijing, China, 2012, pp. 499–508. DOI: 10.1145/2254064.2254123.

[28] Shiyi Chen and Gary D. Doolen. "Lattice Boltzmann method for fluid flows". In: *Annual Review of Fluid Mechanics* 30.1 (1998), pp. 329–364. DOI: 10.1146/annurev.fluid.30.1.329.

[29] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. "Analysis and Characterization of Inherent Application Resilience for Approximate Computing". In: *Proceedings of the 50th Annual Design Automation Conference*. Austin, Texas, USA, 2013, 113:1–113:9. DOI: 10.1145/2463209.2488873.

[30] Vinay K. Chippa, Debabrata Mohapatra, Anand Raghunathan, Kaushik Roy, and Srimat T. Chakradhar. "Scalable Effort Hardware Design: Exploiting Algorithmic Resilience for Energy Efficiency". In: *Proceedings of the 47th Design Automation Conference*. Anaheim, California: ACM, 2010, pp. 555–560. DOI: 10.1145/1837274.1837411.

[31] Philippe Clauss. "Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs". In: *Proceedings of the 10th International Conference on Supercomputing*. Philadelphia, Pennsylvania, USA, 1996, pp. 278–285. DOI: 10.1145/237578.237617.

[32] Philippe Clauss and Vincent Loechner. "Parametric Analysis of Polyhedral Iteration Spaces". In: *Journal of VLSI signal processing systems for signal, image and video technology* 19.2 (July 1998), pp. 179–194. DOI: 10.1023/A:1008069920230.

[33] Albert Cohen, Sidi Kaber, Siegfried Müller, and Marie Postel. "Fully Adaptive Multiresolution Finite Volume Schemes for Conservation Laws". In: *Mathematics of Computation* 72.241 (Dec. 2003), pp. 183–225. DOI: 10.1090/S0025-5718-01-01391-6.

[34] ISO/IEC JTC 1/SC 22 Technical Committee. *ISO/IEC 9899:2018 - C 2018*. Standardization Document. International Organization for Standardization / International Electrotechnical Commission, June 2018, p. 87. 520 pp. URL: https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf (visited on June 2019).

[35] Keith Cooper and Linda Torczon. *Engineering a compiler*. 2nd ed. Elsevier, Feb. 2011.

[36] R. Courant, K. Friedrichs, and H. Lewy. "Über die partiellen Differenzengleichungen der mathematischen Physik". German. In: *Mathematische Annalen* 100.1 (Dec. 1928), pp. 32–74.

[37] Richard Courant and Fritz John. *Introduction to calculus and analysis I*. Springer Science & Business Media, 2012.

[38] Harold Scott Macdonald Coxeter. *Regular polytopes*. Dover Books on Mathematics. New York, NY: Dover, 1973.

[39] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (Jan. 1998), pp. 46–55. DOI: 10.1109/99.660313.

[40] Ingrid Daubechies. "Orthonormal bases of compactly supported wavelets". In: *Communications on pure and applied mathematics* 41.7 (1988), pp. 909–996. DOI: 10.1002/cpa.3160410705.

[41] Ingrid Daubechies. *Ten Lectures on Wavelets*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992.

[42] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. "Autotuning Algorithmic Choice for Input Sensitivity". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Portland, OR, USA, 2015, pp. 379–390. DOI: 10.1145/2737924.2737969.

[43] H. Dym and H.P. McKean. *Fourier Series and Integrals*. Probability and mathematical statistics. Academic Press, 1972.

[44] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. "Architecture Support for Disciplined Approximate Programming". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. London, England, UK, 2012, pp. 301–312. DOI: 10.1145/2150976.2151008.

[45] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. "Neural Acceleration for General-Purpose Approximate Programs". In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. Vancouver, B.C., CANADA, 2012, pp. 449–460. DOI: 10.1109/MICRO.2012.48.

[46] Y. Fang, H. Li, and X. Li. "SoftPCM: Enhancing Energy Efficiency and Lifetime of Phase Change Memory in Video Applications via Approximate Write". In: *Proceedings of the IEEE 21st Asian Test Symposium*. Nov. 2012, pp. 131–136. DOI: 10.1109/ATS.2012.57.

[47] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. I. One-dimensional time". In: *International Journal of Parallel Programming* 21.5 (Oct. 1992), pp. 313–347. DOI: 10.1007/BF01407835.

[48] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time". In: *International Journal of Parallel Programming* 21.6 (Dec. 1992), pp. 389–420. DOI: 10.1007/BF01379404.

[49] Richard P Feynman, Robert B Leighton, and Matthew Sands. *The Feynman lectures on physics: mainly mechanics, radiation, and heat*. Vol. 1. Basic Books, 2011. URL: http://www.feynmanlectures.caltech.edu/ (visited on July 2019).

[50] Dennis Gabor. "Theory of communication. Part 1: The analysis of information". In: *Journal of the Institution of Electrical Engineers - Part III: Radio and Communication Engineering* 93.26 (Nov. 1946), pp. 429–441. DOI: 10.1049/ji-3-2.1946.0074.

[51] Christie John Geankoplis. *Transport processes and unit operations*. Allyn and Bacon, 1978.

[52] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. "ApproxHadoop: Bringing Approximations to MapReduce Frameworks". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. Istanbul, Turkey, Mar. 2015, pp. 383–397. DOI: 10.1145/2694344.2694351.

[53] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD, USA: Johns Hopkins University Press, 1996. ISBN: 0-8018-5414-8.

[54] R.Wm. Gosper. "Exploiting regularities in large cellular spaces". In: *Physica D: Nonlinear Phenomena* 10.1 (1984), pp. 75–80. ISSN: 0167-2789. DOI: 10.1016/0167-2789(84)90251-3.

[55] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "POLLY — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation". In: *Parallel Processing Letters* 22.04 (2012), pp. 1–28. DOI: 10.1142/S0129626412500107.

[56] Alexander Grossmann and Jean Morlet. "Decomposition of Hardy functions into square integrable wavelets of constant shape". In: *SIAM journal on mathematical analysis* 15.4 (1984), pp. 723–736. DOI: 10.1137/0515056.

[57] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Commununication of the ACM* 31.5 (May 1988), pp. 532–533. DOI: 10.1145/42411.42415.

[58] John Gustafson and Isaac Yonemoto. "Beating Floating Point at its Own Game: Posit Arithmetic". In: *Supercomputing Frontiers and Innovations* 4.2 (2017). ISSN: 2313-8734. URL: https://www.superfri.org/superfri/article/view/137.

[59] Alfred Haar. "Zur Theorie der orthogonalen Funktionensysteme". German. In: *Mathematische Annalen* 69.3 (Sept. 1910), pp. 331–371. DOI: 10.1007/BF01456326.

[60] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. "Dynamic Knobs for Responsive Power-aware Computing". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. Newport Beach, California, USA, 2011, pp. 199–212. DOI: 10.1145/1950365.1950390.

[61] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. "Towards Automatic Resource Bound Analysis for OCaml". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Paris, France, 2017, pp. 359–373. DOI: 10.1145/3009837.3009842.

[62] Hsieh Hou and H. Andrews. "Cubic splines for image interpolation and digital filtering". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26.6 (Dec. 1978), pp. 508–517. DOI: 10.1109/TASSP.1978.1163154.

[63] Weizhang Huang and Robert D Russell. *Adaptive moving mesh methods*. Vol. 174. Springer Science & Business Media, 2010.

[64] Karthik Kambatla, Giorgos Kollias, Vipin Kumar, and Ananth Grama. "Trends in big data analytics". In: *Journal of Parallel and Distributed Computing* 74.7 (2014), pp. 2561–2573.

[65] V. Kantabutra. "On hardware for computing exponential and trigonometric functions". In: *IEEE Transactions on Computers* 45.3 (Mar. 1996), pp. 328–339.

[66] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. *The Omega Calculator and Library*. User Manual. Version 1.1.0. Nov. 1996. URL: www.cs.utah.edu/~mhall/cs6963s09/lectures/omega.ps (visited on June 2019).

[67] Wayne Kelly and William Pugh. *A framework for unifying reordering transformations*. Tech. rep. University of Maryland, Oct. 1998. 24 pp. URL: https://drum.lib.umd.edu/handle/1903/607 (visited on June 2019).

[68] Shikai Li, Sunghyun Park, and Scott Mahlke. "Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation". In: *Proceedings of the 2018 International Conference on Supercomputing*. Beijing, China, 2018, pp. 341–351. DOI: 10.1145/3205289.3205317.

[69] Stéphane Mallat. *A wavelet tour of signal processing*. The Sparse Way. Elsevier, 2009.

[70] Stephane G Mallat. "A theory for multiresolution signal decomposition: the wavelet representation". In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 7 (1989), pp. 674–693. DOI: 10.1109/34.927466.

[71] Naraig Manjikian and Tarek S. Abdelrahman. "Exploiting wavefront parallelism on large-scale shared-memory multiprocessors". In: *IEEE Transactions on Parallel and Distributed Systems* 12.3 (Mar. 2001), pp. 259–271. DOI: 10.1109/71.914756.

[72] Jiayuan Meng, S. Chakradhar, and A. Raghunathan. "Best-effort parallel execution framework for Recognition and mining applications". In: *IEEE International Symposium on Parallel Distributed Processing*. Rome, Italy, May 2009, pp. 1–12. DOI: 10.1109/IPDPS.2009.5160991.

[73] Yves Meyer. "Principe d'incertitude, bases hilbertiennes et algèbres d'opérateurs". French. In: *Séminaire Bourbaki* 28 (1986), pp. 209–223. URL: http://www.numdam.org/item/SB_1985-1986__28__209_0 (visited on July 2019).

[74] Donald Michie. ""Memo" Functions and Machine Learning". In: *Nature* 218.5136 (Apr. 1968), pp. 19–22. DOI: 10.1038/218019a0.

[75] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. "Load Value Approximation". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. Cambridge, United Kingdom, 2014, pp. 127–139. DOI: 10.1109/MICRO.2014.22.

[76] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. "Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. Portland, Oregon, USA, 2014, pp. 309–328. DOI: 10.1145/2660193.2660231.

[77] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. "Probabilistically Accurate Program Transformations". In: *Static Analysis*. Ed. by Eran Yahav. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 316–333. DOI: 10.1007/978-3-642-23702-7_24.

[78] Sparsh Mittal. "A Survey of Techniques for Approximate Computing". In: *ACM Computing Surveys* 48.4 (Mar. 2016), 62:1–33. DOI: 10.1145/2893356.

[79] Siegfried Müller. *Adaptive Multiscale Schemes for Conservation Laws*. Vol. 27. Lecture Notes in Computational Science and Engineering. Springer Science & Business Media, 2012. DOI: 10.1007/978-3-642-18164-1.

[80] OpenACC Organization, ed. *OpenACC Specification*. Version 2.7. Nov. 2018. URL: https://www.openacc.org/specification (visited on June 2019).

[81] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. "Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Tucson, AZ, USA, June 2008, pp. 90–100. DOI: 10.1145/1375581.1375594.

[82] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. "Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time". In: *International Symposium on Code Generation and Optimization*. San Jose, CA, USA, Mar. 2007, pp. 144–156. DOI: 10.1109/CGO.2007.21.

[83] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes: The art of scientific computing*. 3rd ed. Cambridge university press, 2007.

[84] *Profiling Programs on Linux with perf*. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on May 2019).

[85] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. "Generation of Efficient Nested Loops from Polyhedra". In: *International Journal of Parallel Programming* 28.5 (Oct. 2000), pp. 469–498. DOI: 10.1023/A:1007554627716.

[86] A. Rahimi, L. Benini, and R. K. Gupta. "Spatial Memoization: Concurrent Instruction Reuse to Correct Timing Errors in SIMD Architectures". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 60.12 (Dec. 2013), pp. 847–851. DOI: 10.1109/TCSII.2013.2281934.

[87] Martin Rinard. "Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks". In: *Proceedings of the 20th Annual International Conference on Supercomputing*. Cairns, Queensland, Australia: ACM, 2006, pp. 324–334. DOI: 10.1145/1183401.1183447.

[88] Olivier Rioul and Pierre Duhamel. "Fast algorithms for discrete and continuous wavelet transforms". In: *IEEE Transactions on Information Theory* 38.2 (Mar. 1992), pp. 569–586. DOI: 10.1109/18.119724.

[89] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. "Precimonious: Tuning Assistant for Floating-point Precision". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Denver, Colorado, USA, Nov. 2013, 27:1–27:12. DOI: 10.1145/2503210.2503296.

[90] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. "Paraprox: Pattern-based Approximation for Data Parallel Applications". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Salt Lake City, Utah, USA, 2014, pp. 35–50. DOI: 10.1145/2541940.2541948.

[91] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. "SAGE: Self-tuning Approximation for Graphics Engines". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. Davis, California, USA, 2013, pp. 13–24. DOI: 10.1145/2540708.2540711.

[92] Adrian Sampson. *ApproxBench 2.0*. URL: http://approxbench.org/ (visited on July 2019).

[93] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. *Accept: A programmer-guided compiler framework for practical approximate computing*. Tech. rep. University of Washington, Jan. 2015, pp. 1–14. URL: https://dada.cs.washington.edu/research/tr/2015/01/UW-CSE-15-01-01.pdf (visited on June 2019).

[94] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. "EnerJ: Approximate Data Types for Safe and General Low-power Computation". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Jose, California, USA, 2011, pp. 164–174. DOI: 10.1145/1993498.1993518.

[95] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. "Approximate Storage in Solid-State Memories". In: *ACM Transactions on Computer Systems* 32.3 (Sept. 2014). DOI: 10.1145/2644808.

[96] K. Sato and Y. Yagasaki. "Adaptive MC interpolation for memory access reduction in JVT video coding". In: *Proceedings of the Seventh International Symposium on Signal Processing and Its Applications*. Vol. 1. Paris, France, July 2003, pp. 77–80. DOI: 10.1109/ISSPA.2003.1224644.

[97] Maxime Schmitt. *ACR compiler and runtime*. URL: http://gauvain.u-strasbg.fr/%7Eschmitt/acr (visited on July 2019).

[98] Maxime Schmitt. *Lenient to Errors, Transformations, Irregularities and Turbulence Benchmarks* (*LetItBench*). URL: http://gauvain.u-strasbg.fr/%7Eschmitt/LetItBench (visited on July 2019).

[99] Maxime Schmitt, Cédric Bastoul, and Philippe Helluy. "A language extension set to generate adaptive versions automatically". In: *Oil & Gas Science and Technology* 73.52 (2018). DOI: 10.2516/ogst/2018049.

[100] Maxime Schmitt, Philippe Helluy, and Cédric Bastoul. "Adaptive Code Refinement: A Compiler Technique and Extensions to Generate Self-Tuning Applications". In: *IEEE 24th International Conference on High Performance Computing*. Jaipur, India, Dec. 2017, pp. 172–181. DOI: 10.1109/HiPC.2017.00028.

[101] Maxime Schmitt, Philippe Helluy, and Cédric Bastoul. "Automatic Adaptive Approximation for Stencil Computations". In: *Proceedings of the 28th International Conference on Compiler Construction*. Washington, DC, USA, 2019, pp. 170–181. DOI: 10.1145/3302516.3307348.

[102] Maxime Schmitt, Philippe Helluy, and Cédric Bastoul. "Think Unlimited and Compress Data Automatically". In: *Conférence d'informatique en Parallélisme, Architecture et Système* (*COMPAS*). Anglet, France, June 2019, pp. 1–7.

[103] Maxime Schmitt, César Sabater, and Cédric Bastoul. "Semi-Automatic Generation of Adaptive Codes". In: *7th International Workshop on Polyhedral Compilation Techniques*. Stockholm, Sweden, Jan. 2017, pp. 1–7. URL: https://hal.inria.fr/hal-01655456 (visited on July 2019).

[104] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[105] B. Shao and P. Li. "A model for array-based approximate arithmetic computing with application to multiplier and squarer design". In: *2014 IEEE/ACM International Symposium on Low Power Electronics and Design*. Aug. 2014, pp. 9–14. DOI: 10.1145/2627369.2627617.

[106] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. "Managing performance vs. accuracy trade-offs with loop perforation". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. Szeged, Hungary, 2011, pp. 124–134. DOI: 10.1145/2025113.2025133.

[107] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. "Eon: A Language and Runtime System for Perpetual Systems". In: *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*. Sydney, Australia, 2007, pp. 161–174. DOI: 10.1145/1322263.1322279.

[108] Jos Stam. "Stable Fluids". In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128. DOI: 10.1145/311535.311548.

[109] Jos Stam. "Real-Time Fluid Dynamics for Games". In: *Proceedings of the Game Developer Conference*. Mar. 2003, p. 25. URL: https://www.cs.cmu.edu/afs/cs/academic/class/15462-s11/www/lec_slides/StamFluidforGames.pdf (visited on July 2019).

[110] Gilbert Strang. *Introduction to linear algebra*. 5th ed. Wellesley-Cambridge Press, 2016.

[111] Julius Adams Stratton. *Electromagnetic Theory*. International Series In Pure and Applied Physics. McGRAW-HILL, 1941.

[112] Arjun Suresh, Erven Rohou, and André Seznec. "Compile-time Function Memoization". In: *Proceedings of the 26th International Conference on Compiler Construction*. Austin, TX, USA: ACM, 2017, pp. 45–54. DOI: 10.1145/3033019.3033024.

[113] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. "Intercepting Functions for Memoization: A Case Study Using Transcendental Functions". In: *ACM Transactions on Architecture and Code Optimization* 12.2 (June 2015), 18:1–18:23. DOI: 10.1145/2751559.

[114] Allen Taflove and Susan C Hagness. *Computational electrodynamics: the finite-difference time-domain method*. 3rd ed. Artech House antennas and propagation library. Artech House, 2005.

[115] Joe F Thompson, Bharat K Soni, and Nigel P Weatherill. *Handbook of grid generation*. CRC press, 1998.

[116] Leo Törnqvist, Pentti Vartia, and Yrjö O. Vartia. "How Should Relative Changes be Measured?" In: *The American Statistician* 39.1 (1985), pp. 43–46. DOI: 10.1080/00031305.1985.10479385.

[117] Alan Mathison Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230.

[118] International Telecommunication Union, ed. *H.264 Specification*. Apr. 2017. URL: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-H.264-201704-S!!PDF-E&type=items (visited on July 2019).

[119] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model". In: *Third International Congress on Mathematical Software*. Kobe, Japan, Sept. 2010, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49.

[120] Sven Verdoolaege and Tobias Grosser. "Polyhedral extraction tool". In: *Second International Workshop on Polyhedral Compilation Techniques*. Paris, France, 2012, pp. 1–16.

[121] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. "Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions". In: *Algorithmica* 48.1 (May 2007), pp. 37–66. DOI: 10.1007/s00453-006-1231-0.

[122] RongGang Wang, JinTao Li, and Chao Huang. "Motion compensation memory access optimization strategies for H.264/AVC decoder". In: *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 5. Philadelphia, PA, USA, Mar. 2005, pp. 97–100. DOI: 10.1109/ICASSP.2005.1416249.

[123] John Weckert. *Computer Ethics*. 1st ed. The International Library of Essays in Public and Professional Ethics. Routledge, 2017. Chap. On the Impact of the Computer on Society. DOI: 10.4324/9781315259697.

[124] Ingo Wegener. *Complexity theory: exploring the limits of efficient algorithms*. Trans. German by Randall Pruim. Springer, 2005. DOI: 10.1007/3-540-27477-4.

[125] Michael E. Wolf and Monica S. Lam. "A Data Locality Optimizing Algorithm". In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. Toronto, Ontario, Canada, June 1991, pp. 30–44. DOI: 10.1145/113445.113449.

[126] Michael Wolfe. "More iteration space tiling". In: *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. Reno, Nevada, USA, Nov. 1989, pp. 655–664. DOI: 10.1145/76263.76337.

[127] Yi-Qing Yang. "Tests de dépendances et transformations de programme". French. PhD thesis. École nationale supérieur des mines de Paris, France, Nov. 1993. URL: http://www.cri.ensmp.fr/classement/doc/A-242.pdf.

[128] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. "AxBench: A Multiplatform Benchmark Suite for Approximate Computing". In: *IEEE Design Test* 34.2 (Apr. 2017), pp. 60–68. DOI: 10.1109/MDAT.2016.2630270.

[129] Kane Yee. "Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media". In: *IEEE Transactions on Antennas and Propagation* 14.3 (May 1966), pp. 302–307.

[130] Thomas Yeh, Petros Faloutsos, Milos Ercegovac, Sanjay Patel, and Glenn Reinman. "The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture*. Chicago, IL, USA, Dec. 2007, pp. 394–406. DOI: 10.1109/MICRO.2007.9.

Annexe A

Résumé en français

Un ordinateur moderne est une machine qui exécute une séquence d'instructions pour mener à bien un traitement. Différents formats coexistent, depuis les petits microcontrôleurs qui animent des brosses à dents pas si intelligentes que cela, jusqu'aux puissants processeurs présents au cœur d'immenses centres de données délivrant tout type de contenu sur l'Internet. Exploiter toutes les possibilités offertes par ces processeurs est un défi pour les développeurs. Pour remédier à cela, une multitude de langages, outils, techniques de compilation et bibliothèques existent pour aider les développeurs à écrire des applications efficaces. Dans cette thèse, nous proposons une nouvelle abstraction pour parvenir à de bonnes performances grâce à des techniques de calcul adaptatif, en focalisant les ressources de calcul là où l'application en a le plus besoin.

Les premiers ordinateurs étaient mécaniques et s'utilisaient, par exemple, pour des tâches de navigation, astronomie ou à fins de suivi temporel dans les calendriers. Une personne entrait les paramètres de l'ordinateur mécanique, actionnait le mécanisme pour effectuer le calcul et finissait par lire le résultat sur la machine. Les ordinateurs mécaniques, ou plus récemment électromécaniques, prenaient un temps important pour mener à bien un calcul. Au début des années 1940, les ordinateurs entièrement électroniques les ont remplacés. Ils étaient déjà capables d'effectuer 5000 additions ou soustractions par seconde dans le cas de l'ordinateur `ENIAC`. Cet ordinateur se configurait à l'aide d'interrupteurs et de câbles par des opératrices, parmi les premières programmeuses. Alan Turing a spécifié le fonctionnement des ordinateurs comme d'une machine où le programme, qui est une suite d'instructions, est stocké sur un ruban, ce qui la rend facilement programmable [117].

L'introduction dans les années 1980 d'ordinateurs compactes et rapides a ouvert d'innombrables nouvelles possibilités sociétales [123]. Les besoins en puissance de calcul et stockage ont explosé dans de nombreux domaines, tels que le calcul scientifique, les applications multimédias, les réseaux sociaux, la finance, la santé, etc. Les besoins dans ces domaines excèdent les possibilités d'approvisionnement en moyen de calcul actuel. Le ratio de l'augmentation des données générées sur la puissance de calcul ajoutée a été mesuré de l'ordre de 3 dans les années 2000 [64]. Les techniques de calcul et stockage approchés fournissent des solutions intéressantes quand les données à traiter dépassent les capacités de calcul disponibles. Ces techniques peuvent apporter des gains proportionnels au niveau d'approximation, selon la stratégie d'approximation choisie.

Dans la première section de ce résumé, nous justifions qu'une abstraction du matériel

est importante pour une productivité optimale des développeurs. Utiliser le plein potentiel du matériel est une tâche ardue, et cela le devient d'autant plus quand la quantité de données à traiter dépasse les capacités matérielles d'une seule machine. Dans la seconde section, nous justifions l'utilisation du calcul approché et nous montrons que son utilisation ajoute une couche de complexité supplémentaire dont les développeurs ont à se soucier. Nous clôturons ce chapitre par une présentation des contributions apportées dans cette thèse.

A.1 Défis de programmation

Les premiers ordinateurs numériques étaient très différents de ceux que nous connaissons aujourd'hui : le programme à exécuter prenait la forme d'encoches sur cartes perforées ou était stocké sur bande magnétique. Les sources du programme étaient probablement écrites sur papier, dans un langage de programmation de haut niveau (c.-à-d., combinant facilités de lecture/compréhension et indépendance à un matériel spécifique). Les premiers langages de haut niveau tels que Fortran, C ou Cobol, ont permis de décolérer le programme du matériel informatique. Les aspects matériels d'un processeur comprennent en particulier un nombre limité d'instructions pouvant être exécutés et un nombre limité d'emplacements mémoires, appelés registres, qui stockent les opérandes et le résultat de ces instructions. Les langages de programmation de haut niveau fournissent le plus souvent les abstractions suivantes :

Nombre de variables illimitées Les emplacements mémoires pour stocker des valeurs sont virtuellement illimités et peuvent être nommés.

Variables typées Les variables ont un type définissant l'ensemble des valeurs pouvant y être stockées ainsi que les opérations sur les données de ce type. Ceci élimine une classe entière de problèmes où par exemple un utilisateur essaiera d'additionner des pommes avec des poires.

Structures de donnée de haut niveau Un `tableau` permet de stocker plusieurs valeurs d'un même type et une `structure` permet de construire des types plus complexes à partir de types existants, par exemple un panier de pommes et poires.

Fonctions Une fonction est un nom donné à une liste d'instructions. Cette fonction peut être appelée dans le reste du programme. Elle peut prendre des paramètres en entrée et produire une valeur en sortie.

Un compilateur est un programme qui traduit un programme d'un format à un autre format, par exemple un code source écrit dans un langage de programmation vers des instructions processeur. Le compilateur alloue les ressources limitées du processeur pour produire une version équivalente du programme d'origine. La traduction d'un programme n'est pas unique et plusieurs versions sémantiquement équivalentes peuvent exister. Le compilateur va choisir une implémentation plutôt qu'une autre en se basant sur un modèle prédisant le comportement du processeur et de la mémoire pour sélectionner celle qui lui semblera la plus performante selon ses propres critères.

Les premiers ordinateurs lisaient des cartes perforées contenant le programme et utilisaient un compilateur pour générer l'exécutable qui était exécuté jusqu'à terminaison.

Ce procédé est séquentiel par nature, un seul programme à la fois est exécuté sur la machine jusqu'à ce qu'il termine. Par la suite, le matériel de stockage et les moyens d'interaction personne-machine firent tous deux un bond en avant conséquent. Les cartes perforées ont été replacées par des disquettes et les programmes ont pu être écrits sur un ordinateur muni d'un écran, d'un clavier et d'un programme permettant d'éditer du texte. La tâche du compilateur est quant à elle restée inchangée.

Tant que les limites physiques le permirent, les processeurs ont gagné en vitesse d'opération, jusqu'à ce que la chaleur produite et la quantité d'énergie consommée par les processeurs ne deviennent pas trop importantes pour continuer dans cette direction. D'autres optimisations ont été ajoutées aux processeurs pour augmenter leur performance indépendamment de leur vitesse d'horloge :

Cache Les accès à la mémoire principale (RAM) sont lents en comparaison de la vitesse à laquelle le processeur exécute ses instructions (actuellement $\approx 400\times$). Une mémoire rapide (actuellement ≈ 1 à $50\times$ plus lent que le processeur) a été ajoutée aux processeurs pour pallier la lenteur d'accès à la RAM.

Superscalaire Le processeur contient plusieurs copies des unités fonctionnelles (p. ex., additionneurs, multiplieurs, etc.). Il est ainsi possible d'exécuter plus d'une instruction en même temps.

Pipeline Une instruction peut être subdivisée en sous-tâches. Les sous-tâches de différents types peuvent être exécutées en parallèle. Par exemple, l'instruction pour confectionner un smoothie peut être décomposée en taches d'épluchage des fruits suivi par leur mixage. Faire plusieurs smoothies avec un pipeline reviendrait à commencer l'épluchage du smoothie suivant pendant que le premier arrivé se fait mixer. Si l'on considère que le processeur exécute une sous-tâche par cycle, sans pipeline le délai d'attente pour un smoothie serait de deux cycles, alors qu'avec un pipeline le premier smoothie serait attendu pendant deux cycles, mais les suivants arriveraient après un cycle chacun seulement.

Exécution dans le désordre Une instruction peut dépendre du résultat d'une autre. Attendre un résultat décale l'exécution des instructions dans le pipeline, créant des cycles d'attente sans aucune sous-tâche exécutée. L'exécution dans le désordre permet au processeur de réordonner la séquence d'instructions du programme pour réduire la probabilité d'apparition de ces attentes.

Vectorisation Un vecteur est une liste d'éléments ayant une taille donnée. Une instruction vectorielle prend un vecteur et exécute plusieurs fois la même opération sur chaque élément de ce vecteur en parallèle. Par exemple, exécuter 8 additions en une seule instruction à la place de huit instructions séparées.

Processeur multicœur Un processeur multicœur permet d'exécuter plusieurs programmes en parallèle en dupliquant les unités de calcul et de contrôle.

Les développeurs n'ont idéalement pas à se soucier du cache, du pipeline et de l'exécution dans le désordre. Ce sont des optimisations implémentées dans le processeur pour maximiser l'utilisation de ses ressources. Cependant pour tirer le meilleur parti de ces

mécanismes, les programmes doivent être traduits en les prenant en compte. Le compilateur joue ici un rôle fondamental et doit pouvoir générer un code spécifique pour une architecture matérielle donnée.

Le développeur est le plus souvent responsable de l'utilisation efficace des ressources vectorielles et des multiples cœurs de processeurs. Un compilateur peut vectoriser ou exploiter le parallélisme automatiquement s'il lui est facile d'analyser les structures de données et instructions du programme. Il est souvent complexe d'analyser un programme à partir d'un code source qui n'aurait pas été au préalable optimisé par le développeur dans cette optique. Développer un programme parallèle est une tâche complexe et sujette à des erreurs. Parmi les points particulièrement difficiles, on distingue les accès concurrents à la mémoire, des synchronisations, les interblocages, etc.

Les éléments suivants donnent un aperçu des défis de programmation parallèle : **Accès à la mémoire non uniforme (NUMA)** où plusieurs processeurs multicœurs et des accélérateurs (puissantes unités pour calcul spécialisé) sont présents dans la même machine. Chaque processeur ou accélérateur a souvent sa mémoire dédiée et les accès à la mémoire d'un autre processeur sont plus lents que ceux à sa propre mémoire. Les programmes qui ne prennent pas en charge cette spécificité seront plus lents à exécuter. **Grappe d'ordinateurs** où plusieurs machines sont mises en réseau pour partager leurs ressources. Il existe des modèles de programmation, prodiguant plusieurs niveaux d'abstraction, pour aider les programmeurs à développer des applications parallèles utilisant une grappe d'ordinateurs.

L'objectif des abstractions de programmation et des extensions est de fournir un accès plus aisé à ces ressources. Par exemple, OpenMP est un ensemble d'extensions aux langages de programmation C/C++ et Fortran qui définissent une syntaxe pour spécifier facilement des sections parallèles et des tâches [39, 20].

A.2 Calcul approché

Dans la section précédente, nous avons introduit différents défis de programmation, le rôle du compilateur dans l'environnement de programmation et l'importance des langages de programmation, qui masquent les spécificités du matériel et apportent une multitude d'outils aux développeurs pour construire un programme. Les techniques de calcul approché exploitent la capacité qu'ont certaines applications à supporter des approximations pour gagner en performance. Dans cette section, nous introduisons des techniques de calcul approché et préconisons l'utilisation d'abstractions pour fluidifier le processus de développement des applications qui les utilisent.

Le terme *approximation* dérive du latin *proximus*, qui se traduit en « proche de ». De la même manière, le calcul approché est l'étude de fonctions qui donnent un résultat proche de celui désiré, mais qui sont définies plus simplement ou plus faciles à calculer. Pour résumer, le calcul approché échange de la précision pour de la performance. Les techniques de calcul approché sont appropriées pour les applications tels que :

Signaux bruités Le stockage et le traitement de signaux bruités nécessitent une quantité importante de ressources pour en extraire les données utiles. Les capteurs analogiques sont soumis à des bruits de lecture et sont distribués avec une spécification décrivant la précision du capteur et l'environnement d'utilisation. Il n'est pas

trivial de déceler un signal quand le bruit est important. L'utilisation de techniques de calcul approché peut être compatible avec un traitement intensif de ce type de flux de donnée.

Applications résilientes aux erreurs Dans le cas d'applications dans les domaines vidéo ou audio par exemple, l'utilisation de techniques de calcul approché peut être imperceptible sur la qualité du signal perçu par l'utilisateur. Il y a une limite sur la qualité d'un signal et les techniques de calcul approché apportent davantage d'opportunités pour optimiser les applications en maintenant une qualité raisonnable du résultat.

Applications itératives Les simulations numériques ou d'intelligence artificielle utilisent souvent des méthodes itératives pour réduire la complexité du problème.

Trois principales possibilités sont envisageables pour écrire une application utilisant des techniques de calcul approché. Si le développeur est accoutumé à l'utilisation de ces techniques, il peut écrire son application en utilisant ces optimisations directement. La seconde méthode consiste à utiliser des bibliothèques spécialisées, qui implémentent déjà les approximations. La dernière solution consiste à ajouter des annotations spécialisées au code pour laisser le compilateur générer les versions utilisant les techniques de calcul approché. Dans cette thèse, nous proposons une méthode inédite se basant sur des extensions aux langages de programmation pour apporter les informations nécessaires au compilateur pour utiliser des techniques de calcul approché.

Sans indications de la part de l'utilisateur, les compilateurs doivent respecter la sémantique du programme en entrée et ne peuvent pas se permettre de changer la précision du résultat. Les extensions proposées dans cette thèse accordent explicitement l'autorisation au compilateur pour modifier la précision du code. Le compilateur peut donc modifier ou remplacer le calcul original. Nous considérons uniquement les programmes s'exécutant sur un processeur polyvalent et ne prenons pas en compte les matériels spécialisés. De tels matériels peuvent utiliser des unités arithmétiques et logiques ou du stockage mémoire approchés, ce qui permet de réduite l'empreinte énergétique, de réduire les accès à la mémoire ou d'augmenter la vitesse de calcul [25, 94, 130]. Ce matériel spécialisé est actuellement difficilement accessible en comparaison des processeurs polyvalents et notre but est de proposer des extensions qui s'adressent à une large gamme de problèmes à destination du matériel actuel.

Les techniques adaptatives sont une famille faisant partie des méthodes de calcul approché. Elles utilisent des approximations de manières « intelligentes » pour cibler les ressources de calcul aux endroits où l'application en a le plus besoin. Ces techniques tirent leur origine de l'observation que la précision requise pour un calcul peut être exprimée en fonction des données traitées. Par exemple, le conducteur d'une voiture concentrera la majorité de ses sens à l'avant de la voiture en ligne droite et sur les côtés dans les intersections. Le même principe peut être utilisé pour les programmes, où une version précise est utilisée aux endroits importants (c.-à-d., pour traiter les données importantes) et une version approchée utilisée dans le cas contraire pour gagner en rapidité de calcul.

A.3 Contributions

Dans cette thèse, nous proposons une interface de programmation pour aider les développeurs dans leur tâche d'optimisation de programme par calcul approché. Cette interface prend la forme d'extensions aux langages de programmation pour indiquer au compilateur quelles parties du programme peuvent utiliser ce type de calcul. Le compilateur se charge alors de transformer les parties du programme ciblées pour rendre l'application adaptative, allouant plus de ressources aux endroits là où une précision importante est requise et utilisant des approximations où la précision peut-être moindre. Nous avons automatisé la découverte des paramètres d'optimisation pour calcul approché que devrait fournir l'utilisateur pour les codes à stencil, qui sont souvent rencontrés dans des applications de traitement du signal, traitement d'image ou simulation numérique. Nous avons exploré des techniques de compression automatique de données pour compléter la génération de code adaptatif. Nous utilisons la transformée en ondelettes pour compresser les données et extraire des informations qui peuvent être utilisées pour trouver les zones avec des besoins en précision plus importantes.

Insertion d'informations spécifiques au calcul approché

Le développement d'une application informatique se déroule souvent en trois phases : (1) implémentation des algorithmes pour résoudre le problème, (2) test de validité du programme et (3) optimisation du code pour obtenir un résultat dans un budget raisonnable (de temps, ou d'espace, ou d'énergie, etc.). Implémenter des techniques de calcul approché durant la phase d'optimisation demande au développeur des modifications algorithmiques et structurelles importantes. Nous proposons de laisser le compilateur gérer au maximum la tâche d'optimisation. Automatiser cette tâche permet d'apporter un gain en productivité et de réduire la probabilité de création de bogues durant la phase d'optimisation. Pour atteindre cet objectif, nous proposons une extension aux langages de programmation de haut niveau pour indiquer au compilateur les modifications pouvant être appliquées au noyau de calcul existant pour générer un code adaptatif [103, 100, 99]. Les informations apportées par les extensions sont les suivantes (voir Chapitre 3) :

Codes alternatifs Une alternative est une version approchée du noyau de calcul à optimiser. L'annotation alternative permet de préciser les transformations à appliquer au code pour créer une version approchée. Le compilateur propose des transformations prédéfinies ou autorise le développeur à écrire plusieurs versions du même noyau de calcul et d'en informer le compilateur.

Supervision Pour choisir entre la version originale du noyau et les différentes alternatives, il faut extraire, traiter et interpréter les informations présentes dans les données du programme lors de son exécution. L'annotation de supervision (ou monitorage) est présente pour indiquer au compilateur dans quelle(s) structure(s) de données trouver ces informations et comment traiter ces données brutes pour en extraire une information sur le niveau d'approximation pouvant être utilisé.

Granularité Les techniques adaptatives appliquent les approximations localement, à des endroits où une version approchée est suffisante pour mener à bien le calcul,

minimisant ainsi la déviation vis-à-vis de la solution précise. La granularité spécifie ce niveau en définissant une grille cartésienne régulière sur les données supervisées. Un élément de cette grille est appelé une *cellule*.

Stratégie La stratégie permet de lier les alternatives aux informations extraites lors de l'exécution supervisée. Il est possible d'appliquer plusieurs transformations de code pour une même stratégie si leur composition est légale. La stratégie définit les versions de code qui sont générées par le compilateur à la compilation ou lors de l'exécution pour construire des versions spécialisées.

Génération de code adaptatif

La génération de code est basée sur les informations présentes dans les annotations insérées par l'utilisateur (voir Chapitre 4) ou instanciées automatiquement dans le cas de stencils (voir Chapitre 5). Nous utilisons le modèle polyédrique, qui abstrait un programme sous forme mathématique pour appliquer les transformations. Cette approche permet de s'abstraire du langage de programmation, de gagner en généralité, et de bénéficier des techniques existantes pour minimiser le surcoût de contrôle dans le code généré.

Le code du noyau de calcul ciblé par les annotations est d'abord transposé dans le modèle polyédrique. Le code est ensuite analysé pour savoir si une version adaptative peut être générée à la compilation. Si c'est le cas, les versions approchées sont générées et stockées dans l'exécutable. Le noyau de calcul est transformé par un pavage pour définir les cellules de supervision. La supervision est intégrée à chacune des cellules et un mécanisme de sélection de versions est inséré dans le noyau principal, pour choisir une version en fonction des besoins lors de l'exécution.

Si les dépendances de données n'autorisent pas la génération de code à la compilation, nous avons mis en place un environnement d'exécution (runtime) qui prend en charge la supervision des données, la génération des versions spécialisées du noyau en utilisant le modèle polyédrique et la compilation cette version à la volée avant de l'injecter dans l'application. Nous réduisons ainsi au maximum les coûts supplémentaires liés à la sélection des alternatives lors de l'exécution.

Extraction des informations pour le calcul adaptatif

Les codes à stencils font partie d'une classe de programmes où les mises à jour des données se font en utilisant les valeurs voisines selon un motif récurant. Nous avons automatisé la recherche des paramètres que les utilisateurs auraient à entrer manuellement à l'aide des extensions dans le cadre des codes à stencils [101] (voir Chapitre 5). Il reste un paramètre unique que l'utilisateur doit encore indiquer au compilateur : le seuil à partir duquel une version approchée peut être utilisée. Les paramètres sont extraits par analyse du code à stencils et sont traduits sous forme d'annotations telles qu'introduites précédemment pour générer la version adaptative du programme.

Compression des données

La génération de code adaptatif avec les annotations est basée sur la transformation de noyaux de calcul de l'application. Nous avons examiné des méthodes de compression

automatiques permettant de traiter les données en plus des calculs (voir Chapitre 6). Pour cela nous avons considéré la transformée en ondelettes pour ses propriétés intéressantes dans le cadre des calculs adaptatifs [102] :

Information multi échelle La transformée en ondelettes permet d'obtenir la décomposition du signal en fréquences localisées à plusieurs échelles. Cette information permet de déterminer les endroits où plus de précision est nécessaire et pourrait remplacer la supervision pour certaines applications.

Compression de données La transformée en ondelettes sépare le signal en une approximation et les détails permettant de reconstituer le signal d'origine. Les détails qui sont proches de la valeur zéro peuvent être nullifiés, entrainant une approximation, mais autorisant une compression des données par la même occasion.

Interpolante Les approximations générées par la transformée en ondelettes génèrent des valeurs qui sont proches de la fonction étudiée. Cela permet l'utilisation des valeurs sans avoir à décompresser les données au préalable.

Étude expérimentale

Nous avons construit une suite de programmes pour évaluer notre approche. Ces programmes couvrent les disciplines allant de la simulation numérique au traitement de données. Ils représentent le type d'applications pouvant être optimisés avec notre méthode, car ils tolèrent des approximations de leurs résultats. Les résultats obtenus montrent qu'en insérant les annotations liées à nos extensions et avec l'aide d'un compilateur supportant nos techniques, un développeur peut utiliser les approximations pour augmenter la vitesse de traitement de l'application en maintenant un niveau convenable de déviation comparé au résultat original (voir Chapter 7).

Dans cette thèse, nous proposons une interface de programmation permettant de générer une version adaptative à partir d'un code annoté par un développeur. Notre approche permet à des développeurs, experts ou non dans le domaine du calcul approché, d'utiliser des optimisations supplémentaires sans avoir à modifier les algorithmes de leur application. Les informations dédiées au calcul approché sont ajoutées sous la forme d'extensions facultatives au langage de programmation à destination du compilateur. Le compilateur se charge de générer la version adaptative du programme et un runtime spécifique se charge de sélectionner les versions à exécuter en fonction des données collectées quand le programme s'exécute. La génération de version approchée de code est complétée par la compression de données.

A.4 Perspectives

Grilles adaptatives La sélection de stratégies utilise une supervision des données avec une grille à maillage fixe. Une grille adaptative peut être avantageuse pour certaines applications où le coût supplémentaire induit par la gestion de la grille est contrebalancé

par une performance accrue de l'algorithme. De nouvelles stratégies pourraient être utilisées sur des grilles dynamiques, telles que des stratégies influant sur la grille elle-même.

La transformée en ondelettes offre une piste intéressante pour générer automatiquement une grille adaptative. Le résultat de cette transformation peut être représenté sous forme d'arbre à k-dimensions (la Figure 6.4 montre un exemple en 2D) où la profondeur d'une branche correspondrait à son niveau de raffinement. Une implémentation peut utiliser le niveau de détails pour sélectionner les parties de l'arbre à raffiner.

Utiliser des arbres k-dimensionnels permettrait de partitionner les cellules en tâches. Chaque tâche peut se diviser en 2^k sous-tâches quand une précision accrue est requise ou fusionner quand moins de précision se fait ressentir. La division en sous-tâche peut être proscrite après un niveau donné pour limiter le surcoût de gestion de tâche à granularité trop fine. L'adressage unique des nœuds d'un arbre k-dimensionnel peut permettre d'optimiser les communications inter tâches et d'effectuer un placement intelligent des tâches sur plusieurs nœuds.

Déviation du résultat Dans cette thèse nous avons concentré nos efforts sur la génération de code adaptatif depuis le noyau d'une application. Nous tentons minimiser la déviation des versions approchées tout en exploitant leurs potentiels pour gagner en performance. Il est, pour le moment, impossible de garantir la précision du résultat pré exécution. Un mécanisme similaire au calcul de la borne d'erreur liée à la compression destructive de la transformée en ondelettes [69] (c.-à-d., mettre à zéro un détail) pourrait être utilisé pour borner la déviation.

Calcul adaptatif pour EDP Nous pensons qu'un compilateur doit être capable de générer automatiquement un code interfaçant plusieurs niveaux de grille pour rendre une équation aux dérivées partielles adaptative. Il est nécessaire d'évaluer quelles informations sont à fournir au compilateur et quelles informations peuvent être extraites du noyau de calcul. Cela peut mener à une implémentation d'un langage dédié (DSL) hybride où l'implémentation du noyau reste à la charge de l'utilisateur, mais la partie adaptative serait autogénérée.

Architectures distribuées et optimisations Notre technique d'optimisation offre des performances respectables sur une machine. Il serait intéressant d'ajouter le support d'optimisations complémentaires, telles que les optimisations polyédriques et la parallélisation automatique [55, 21, 10]. De nouvelles stratégies cherchant à réduire le nombre de communications doivent être envisageables pour les applications utilisant plusieurs nœuds.

Représentation des données Nous avons ouvert une voie d'optimisation intéressante avec l'introduction de la transformation de structure de donnée dans le Chapitre 6. Nous pensons qu'il y a des opportunités similaires en utilisant une représentation alternative des nombres, tels que le calcul à virgule fixe [105] ou les posits [58].

Maxime SCHMITT

Génération automatique de codes adaptatifs

Abstract

In this thesis we introduce a new application programming interface to help developers to optimize an application with approximate computing techniques. This interface is provided as a language extension to advise the compiler about the parts of the program that may be optimized with approximate computing and what can be done about them. The code transformations of the targeted regions are entirely handled by the compiler to produce an adaptive software. The produced adaptive application allocates more computing power to the locations where more precision is required, and may use approximations where the precision is secondary. We automate the discovery of the optimization parameters for the special class of stencil programs which are common in signal/image processing and numerical simulations. Finally, we explore the possibility of compressing the application data using the wavelet transform and we use information found in this basis to locate the areas where more precision may be needed.

Résumé

Dans cette thèse nous proposons une interface de programmation pour aider les développeurs dans leur tâche d'optimisation de programme par calcul approché. Cette interface prend la forme d'extensions aux langages de programmation pour indiquer au compilateur quelles parties du programme peuvent utiliser ce type de calcul. Le compilateur se charge alors de transformer les parties du programme visées pour rendre l'application adaptative, allouant plus de ressources aux endroits où une précision importante est requise et utilisant des approximations où la précision peut être moindre. Nous avons automatisé la découverte des paramètres d'optimisation que devrait fournir l'utilisateur pour les codes à stencil, qui sont souvent rencontrés dans des applications de traitement du signal, traitement d'image ou simulation numérique. Nous avons exploré des techniques de compression automatique de données pour compléter la génération de code adaptatif. Nous utilisons la transformée en ondelettes pour compresser les données et obtenir d'autres informations qui peuvent être utilisées pour trouver les zones avec des besoins en précision plus importantes.

ÉCOLE DOCTORALE

UFR de mathématique et d'informatique
Université de Strasbourg