Compilation Techniques for Automatic Extraction of Parallelism and Locality in Heterogeneous Architectures

José M. Andión

DOCTORAL THESIS

2015

PHD ADVISORS:

Gabriel Rodríguez
Manuel Arenaz

DEPARTMENT OF ELECTRONICS AND SYSTEMS

UNIVERSIDADE DA CORUÑA

Dr. Gabriel Rodríguez Álvarez	Dr. Manuel Arenaz Silva
Profesor Contratado Doctor	Profesor Titular de Universidad
Dpto. de Electrónica y Sistemas	Dpto. de Electrónica y Sistemas
Universidade da Coruña	Universidade da Coruña

CERTIFICAN

Que la memoria titulada "*Compilation Techniques for Automatic Extraction of Parallelism and Locality in Heterogeneous Architectures*" ha sido realizada por D. José Manuel Andión Fernández bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidade da Coruña, y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática con la Mención de Doctor Internacional.

En A Coruña, a 23 de septiembre de 2015

Fdo.: Dr. Gabriel Rodríguez Álvarez	Fdo.: Dr. Manuel Arenaz Silva
Director de la Tesis Doctoral	Director de la Tesis Doctoral

Fdo.: D. José Manuel Andión Fernández
Autor de la Tesis Doctoral

A todos los que me habéis ayudado
a convertir en realidad esta tesis

Acknowledgments

Quiero empezar estas líneas dando las gracias a mis directores Gabriel y Manuel. Sin duda, vuestra contribución ha sido fundamental para haber llevado a buen puerto esta tesis. También merece especialmente mi gratitud Juan, por haberme dado la oportunidad de iniciar mi carrera científica y por la continua atención que has puesto en ella durante todos estos años. Aunque hayamos dejado de investigar juntos, también me gustaría dar las gracias a Guillermo por haber estado siempre ahí aunque tu tiempo libre fuese muy escaso. Y, por supuesto, también quiero agradecer a Ramón y a todos los demás profesores del Grupo de Arquitectura de Computadores la cálida acogida que me habéis brindado.

A mis compañeros de laboratorio, quiero reconoceros el enorme mérito de haber logrado hacer agradables todas las horas de trabajo (¡y a pesar de no tener ventanas!). Aunque en esta parte final ya no haya estado físicamente con vosotros y ahora me llaméis *casta*, sabéis que siempre seré uno de los vuestros. La lista de todos los compañeros pasados y presentes es muy larga, y a todos os doy las gracias, pero quiero hacer una mención especial a Sabela, Jorge, Toño, Moisés, Jacobo, Fabeiro, Iván, Rober, Pablo, Raquel, CH y Carlos. Asimismo, quiero dar las gracias a Bea, Pablo y Antonio, compañeros desde que entramos en la facultad y que me habéis seguido acompañando durante esta etapa.

Mon séjour à Rennes a été l'une des expériences les plus importantes de ma vie. Je veux remercier François pour m'introduire dans le monde des cartes graphiques et pour tes apportes au Chapitre 3. Je salue aussi au personnel de la disparue CAPS Entreprise, n'importe où vous êtes maintenant. Du coté de l'IRISA, je veux faire reconnaissance à André Seznec, directeur de l'équipe ALF, et à Sylvain Collange, pour réviser ma thèse.

I also want to sincerely thank Mahmut T. Kandemir for your contributions in Chapter 4 and for reporting on my thesis.

También me gustaría dar las gracias a las siguientes entidades que han financiado este trabajo: a la Universidade da Coruña, y especialmente al Departamento de Electrónica y Sistemas; a la Unión Europea, por las redes HiPEAC (FP7, refs. ICT-217068 and ICT-287759), ComplexHPC (COST Action, ref. IC0805) y NESUS (COST Action, ref. IC1305); al Ministerio de Educación, por la ayuda del Programa de Formación de Profesorado Universitario (FPU) (ref. AP2008-01012); al Ministerio de Economía y Competitividad, por los proyectos TIN2007-67537-C03-02, TIN2010-16735 y TIN2013-42148-P; y a la Xunta de Galicia por el Programa de Consolidación de Grupos de Referencia Competitiva (refs. 2006/03, 2010/06 y GRC2013/055) y por la Rede Galega de Computación en Altas Prestacións (refs. 2007/147 y 2010/53).

Por último, me gustaría finalizar esta sección mencionando a los que me habéis apoyado en las horas que la investigación me ha dejado libres. No tengo palabras para expresar mi agradecimiento a mi hermana Mónica, a mis padres María y Manuel, a mis abuelos y demás familia. Tampoco puedo olvidarme de todos los que formáis parte de mi otra gran pasión, el ajedrez. Definitivamente estáis locos por haber confiado en mí para refundar y presidir INCUDE. Sólo me queda terminar estas líneas pidiendo disculpas a todos los que habéis padecido esos momentos en los que estaba físicamente con vosotros, pero en los que mi mente estaba pensando en las *cosas raras de informático*. Muchas gracias por comprenderme y apoyarme.

José Manuel Andión Fernández

If you don't make mistakes,
you're not working on hard enough problems.
And that's a big mistake.

Frank Wilczek

Abstract

High performance computing has become a key enabler for innovation in science and industry. This fact has unleashed a continuous demand of more computing power that the silicon industry has satisfied with parallel and heterogeneous architectures, and complex memory hierarchies. As a consequence, software developers have been challenged to write new codes and rewrite the old ones to be efficient in these new systems. Unfortunately, success cases are scarce and require huge investments in human workforce. Current compilers generate peak-peformance binary code in monocore architectures. Following this victory, this thesis explores new ideas in compiler design to overcome this challenge with the automatic extraction of parallelism and locality. First, we present a new compiler intermediate representation based on diKernels named KIR, which is insensitive to syntactic variations in the source code and exposes multiple levels of parallelism. On top of the KIR, we build a source-to-source approach that generates parallel code annotated with compiler directives: OpenMP for multicores and OpenHMPP for GPUs. Finally, we model program behavior from the point of view of the memory accesses through the reconstruction of affine loops for sequential and parallel codes. The experimental evaluations throughout the thesis corroborate the effectiveness and efficiency of the proposed solutions.

Abstract in Spanish

La computación de altas prestaciones se ha convertido en un habilitador clave para la innovación en la ciencia y la industria. Este hecho ha propiciado una demanda continua de más poder computacional que la industria del silicio ha satisfecho con arquitecturas paralelas y heterogéneas, y jerarquías de memoria complejas. Como consecuencia, los desarrolladores de software han sido desafiados a escribir códigos nuevos y reescribir los antiguos para que sean eficientes en estos nuevos sistemas. Desafortunadamente, los casos de éxito son escasos y requieren inversiones enormes en fuerza de trabajo. Los compiladores actuales generan código binario con rendimiento máximo en las arquitecturas mononúcleo. Siguiendo esta victoria, esta tesis explora nuevas ideas en el diseño de compiladores para superar este reto con la extracción automática de paralelismo y localidad. En primer lugar, presentamos una nueva representación intermedia de compilador basada en diKernels denominada KIR, la cual es insensible a variaciones sintácticas en el código de fuente y expone múltiples niveles de paralelismo. Sobre la KIR, construimos una aproximación fuente-a-fuente que genera código paralelo anotado con directivas: OpenMP para multinúcleos y OpenHMPP para GPUs. Finalmente, modelamos el comportamiento del programa desde el punto de vista de los accesos de memoria a través de la reconstrucción de bucles afines para códigos secuenciales y paralelos. Las evaluaciones experimentales a lo largo de la tesis corroboran la efectividad y eficacia de las soluciones propuestas.

Abstract in Galician

A computación de altas prestacións converteuse nun habilitador clave para a innovación na ciencia e na industria. Este feito propiciou unha demanda continua de máis poder computacional que a industria do silicio satisfixo con arquitecturas paralelas e heteroxéneas, e xerarquías de memoria complexas. Como consecuencia, os desenvolvedores de software foron desafiados a escribir códigos novos e reescribir os antigos para que sexan eficientes nestes novos sistemas. Desafortunadamente, os casos de éxito son escasos e requiren investimentos enormes en forza de traballo. Os compiladores actuais xeran código binario con rendemento máximo nas arquitecturas mononúcleo. Seguindo esta vitoria, esta tese explora novas ideas no deseño de compiladores para superar este reto coa extracción automática de paralelismo e localidade. En primeiro lugar, presentamos unha nova representación intermedia de compilador baseada en diKernels denominada KIR, a cal é insensible a variacións sintácticas no código fonte e expón múltiples niveis de paralelismo. Sobre a KIR, construímos unha aproximación fonte-a-fonte que xera código paralelo anotado con directivas: OpenMP para multinúcleos e OpenHMPP para GPUs. Finalmente, modelamos o comportamento do programa desde o punto de vista dos accesos de memoria a través da reconstrución de bucles afíns para códigos secuenciais e paralelos. As avaliacións experimentais ao longo da tese corroboran a efectividade e eficacia das solucións propostas.

Preface

Nowadays computers play a key role in our society. In particular, high performance computing has demonstrated to be a fundamental tool for the evolution of science and industry. In order to provide computing power for the constantly increasing demand of more detailed simulations, the silicon industry has been forced to introduce parallel and heterogeneous architectures, and complex memory hierarchies. As a result, the software community has been challenged to write efficient codes for the new computing devices. Note that this parallel challenge includes not only creating new programs, but also modernizing the old ones. And developers are being unsuccessful in these tasks.

Current state-of-the-art compilers hide the low-level details of sequential hardware architectures to the programmer while generating peak-performance binary code. Following this success in monocore architectures, this thesis makes a step forward and explores new ideas in the field of compiler design to address the parallel challenge.

Objectives and Work Methodology

The main purpose of this thesis is the automatic extraction of parallelism and locality in heterogeneous architectures through new compiler techniques. In order to achieve this general aim, the following objectives were established:

1. Definition of a new compiler intermediate representation

    a) enabling the detection of parallelism

   b) being independent of implementation details (e.g., complex control flows, usage of pointers or arrays, etc.)
   c) modeling the input program as a whole

2. Automatic parallelization of sequential code for multicore architectures thanks to the information gathered by the new IR

   a) inserting compiler directives to allow subsequent optimizations by other compilers and/or programmers
   b) minimizing synchronization and memory overheads in the global parallelization strategy

3. Automatic parallelization of sequential code for manycore architectures thanks to the information gathered by the new IR

   a) inserting compiler directives to allow subsequent optimizations by other compilers and/or programmers
   b) paying special attention at the efficient usage of their complex memory hierarchy

4. Characterization of an application from a trace of its memory accesses to extract locality without needing source or binary code

   a) for both sequential and parallel programs
   b) supporting irregularities in the trace (e.g., presence of noise, missing points, etc.)

5. Evaluation, in terms of effectiveness and efficiency, of the solutions proposed for the previous objectives.

The methodology for carrying out this thesis has followed the classical approach in research and engineering: analysis, design, implementation and evaluation. Hence, for each main objective (1)–(4), an study of the state-of-the-art has been performed. The weaknesses of the existing approaches have been detected, and solutions for them have been proposed. Representative test cases have been chosen to validate our contributions. Finally, experiments have been conducted on multiple hardware platforms to accomplish objective (5).

Funding and Technical Means

- Working material, human and financial support primarily by the Computer Architecture Group of the Department of Electronics and Systems of the University of A Coruña, along with a PhD Fellowship funded by the Ministry of Education of Spain (FPU program, ref. AP2008-01012).

- Access to bibliographical material through the library of the University of A Coruña.

- Additional funding through the following research projects:

  - European funding through the networks "High-Performance Embedded Architectures and Compilers - HiPEAC" (FP7, refs. ICT-217068 and ICT-287759), "Open European Network for High Performance Computing on Complex Environments - ComplexHPC" (COST Action, ref. IC0805) and "Network for Sustainable Ultrascale Computing - NESUS" (COST Action, ref. IC1305)
  - State funding by the Ministry of Economy and Competitiveness of Spain through the projects "Hardware and Software Support for High Performance Computing" (ref. TIN2007-67537-C03-02), "Architectures, Systems and Tools for High Performance Computing" (ref. TIN2010-16735), and "New Challenges in High Performance Computing: from Architectures to Applications" (ref. TIN2013-42148-P).
  - Regional funding by the Galician Government *(Xunta de Galicia)* under the Consolidation Program of Competitive Research Groups (refs. 2006/03, 2010/06 and GRC2013/055) and the network "Galician Network of High Performance Computing" (refs. 2007/147 and 2010/53).

- Access to *Pluton*, the cluster of the Computer Architecture Group of the University of A Coruña, Spain; and to the *CAPS Compute Lab*, the cluster of CAPS Entreprise, Rennes, France.

- A three-month research visit to the *Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA)* and to CAPS Entreprise, at Rennes, France. This

research visit was funded by the Ministry of Education of Spain under the FPU program.

Conclusions and Main Contributions

The main contributions of this thesis can be summarized in the following items:

1. We have formally defined the KIR, a new compiler intermediate representation designed for the detection of parallelism. Built on top of diKernels, which abstract syntactical variations of the source code, the KIR presents an unified view of the application that enables the joint parallelization of multiple loops.

2. We have introduced a new partitioning algorithm of the KIR to generate parallel code for multicore processors. Our method builds a global OpenMP parallelization of the input code. We have demonstrated the effectiveness and efficiency of our proposal with a comprehensive benchmark suite that includes linear algebra and image processing routines, and applications from SPEC CPU2000. In addition, the automatic parallelization of GCC, Intel and PLUTO compilers has been evaluated.

3. We have targeted the most popular example of manycore architectures: the GPU. We have designed a new technique for efficient code generation that takes into account the most important GPU programming features with the insertion of OpenHMPP directives. Two representative case studies from compute-intensive scientific applications have been successfully parallelized.

4. We have developed a new and efficient method for reconstructing affine loop nests from the list of memory addresses accessed by one instruction at a time. Furthermore, our technique supports noise and missing points in the trace. A piecewise reconstruction has been also developed for automatically parallelized codes. The experimental evaluation has shown the accuracy and efficiency of the approach.

As a result of part of the work done in this thesis, the spin-off company Appentra Solutions S.L. is developing the OpenMP-enabling source-to-source compiler Parallware [15].

Publications

- J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. Touriño. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. *International Journal of Parallel Programming (in press)*, 2015. [12]

- G. Rodríguez, J. M. Andión, J. Touriño, and M. T. Kandemir. Reconstructing affine codes from their memory traces. *Pennsylvania State University Technical Report CSE 15-001*, University Park, PA, USA, 2015. [117]

- J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. Touriño. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. In *Proceedings of the 7th International Symposium on High-level Parallel Programming and Applications (HLPP)*, pages 217–238, Amsterdam, Netherlands, 2014. [11]

- J. M. Andión, M. Arenaz, G. Rodríguez, and J. Touriño. A parallelizing compiler for multicore systems. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 138–141, Sankt Goar, Germany, 2014. [5]

- J. M. Andión, M. Arenaz, G. Rodríguez, and J. Touriño. A novel compiler support for automatic parallelization on multicore systems. *Parallel Computing*, 39(9):442–460, 2013. [4]

- J. M. Andión, M. Arenaz, and J. Touriño. Una nueva representación intermedia para GCC basada en el entorno de compilación XARK. In *Actas de las XXI Jornadas de Paralelismo (JP)*, pages 151–158, Valencia, Spain, 2010. [9]

- J. M. Andión, M. Arenaz, and J. Touriño. Domain-independent kernel-based intermediate representation for automatic parallelization of sequential programs. In *Poster Abstracts of the 6th International Summer School on*

Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES), pages 71–74, Terrasa, Spain, 2010. [8]

- J. M. Andión, M. Arenaz, and J. Touriño. Automatic partitioning of sequential applications driven by domain-independent kernels. In *Proceedings of the 15th Workshop on Compilers for Parallel Computing (CPC)*, CDROM, Vienna, Austria, 2010. [7]

- J. M. Andión, M. Arenaz, and J. Touriño. A new intermediate representation for GCC based on the XARK compiler framework. In *Proceedings of the 2nd International Workshop on GCC Research Opportunities (GROW) (in conjunction with the International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC))*, pages 89–100, Pisa, Italy, 2010. [6]

(Pre-prints are available at `http://gac.udc.es/~jandion/`)

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

1.1 The Parallel Challenge

Computers have enabled a new industrial revolution based on the digital technology. They have transformed our society completely. Nowadays, the processing and transmission of information have become the fundamental sources of productivity and social power [109]. In particular, High Performance Computing (from now on, HPC) strongly contributes to the excellence of science and the competitiveness of industry. HPC allows to improve existing models and products, and to develop new ones earlier and more efficiently. These beneficial properties have been stated both in Europe and in America. On the one hand, the European Union affirms that *"[. . .] HPC is a crucial asset for Europe's innovation capacity and is of strategic importance to its scientific and industrial capabilities, as well as to its citizens [. . .]"* in the H2020 Work Programme [47]. On the other hand, the President of the United States has affirmed in the recent Executive Order for Creating a National Strategic Computing Initiative [49] that *"[. . .] HPC has contributed substantially to national economic prosperity and rapidly accelerated scientific discovery [. . .]"*. The U.S. Council of Competitiveness has also presented in 2014 a survey [38] between 101 HPC-using companies which concludes that *"[. . .] 83 percent feel their businesses would benefit from improvements in modeling and simulation, and 86 percent agreed with the simple statement, 'HPC is critical to the future direction of our business.' [. . .]"*. This observatory also considers HPC as *"[. . .] inextricably linked to innovation, fu-*

eling breakthroughs in science, engineering, and business [. . .]", and it has coined the statement *"To out-compute is to out-compete"*.

Due to this demonstrated competitive advantage provided by HPC, the demand of computing power keeps increasing year after year. Both academia and industry try to create models as close as possible to the reality, which involve huge quantities of information when simulations are run. More and more data is collected from experiments performed in the real world, which need to be stored and analyzed. The level of detail of computer graphics has increased up to a level which makes difficult to differentiate computer-generated images from real ones. For illustrative purposes, Figure 1.1 shows the evolution in the level of graphic detail of the videogame saga "Tomb Raider". This is a characteristic property of the domains where HPC is used: given a problem, its size can grow arbitrarily increasing the level of granularity.

Firstly, the silicon industry responded to this growing demand being able to preserve the existing sequential programming model thanks to the Moore's Law [95]. The size of transistors could be reduced, doubling their number in an integrated circuit every two years. This fact enabled higher clock speeds, bigger cache memories, and more flexible microarchitectures. Programs run faster in each new generation of processors, as can be observed from 1990 to 2005 in the "Single-Thread Performance" line of Figure 1.2.

However, as can be seen in the "Frequency" line of Figure 1.2, clock speed cannot be augmented so fast since 2000. Until that year, an increase in the clock frequency carried along a decrease in the voltage of the integrated circuit. This property, which is known as Dennard's scaling [40], has allowed to maintain the power density nearly constant. But this scaling has broken down because voltage cannot be decreased anymore if we want to preserve the reliable operation of the transistors at the same time.

Hence, the industry has decided to introduce several general purpose processors in the same chip to profit from the increased transistor count. This architectural change has started *the multicore era*. Figure 1.3a clearly illustrates this new design trend. The TOP500 list [124] gathers, twice a year, the information about the 500 most powerful computer systems. First multicore processors appeared in

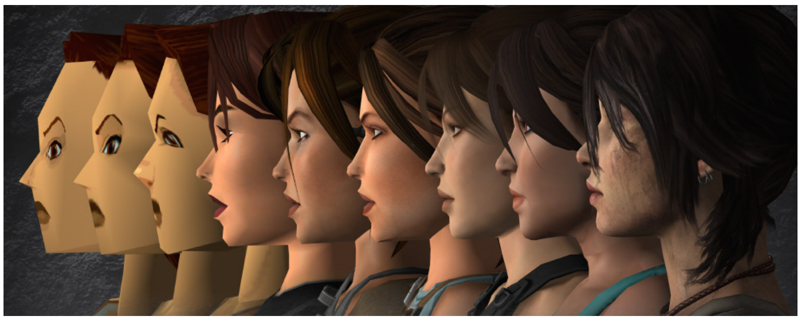


Figure 1.1 – Excerpt of "The many faces of Lara Croft: Tomb Raider" showing the evolution in the level of detail of computer graphics from 1996 to 2014 (from http://www.halloweencostumes.com/blog/p-468-tomb-raider-infographic.aspx).

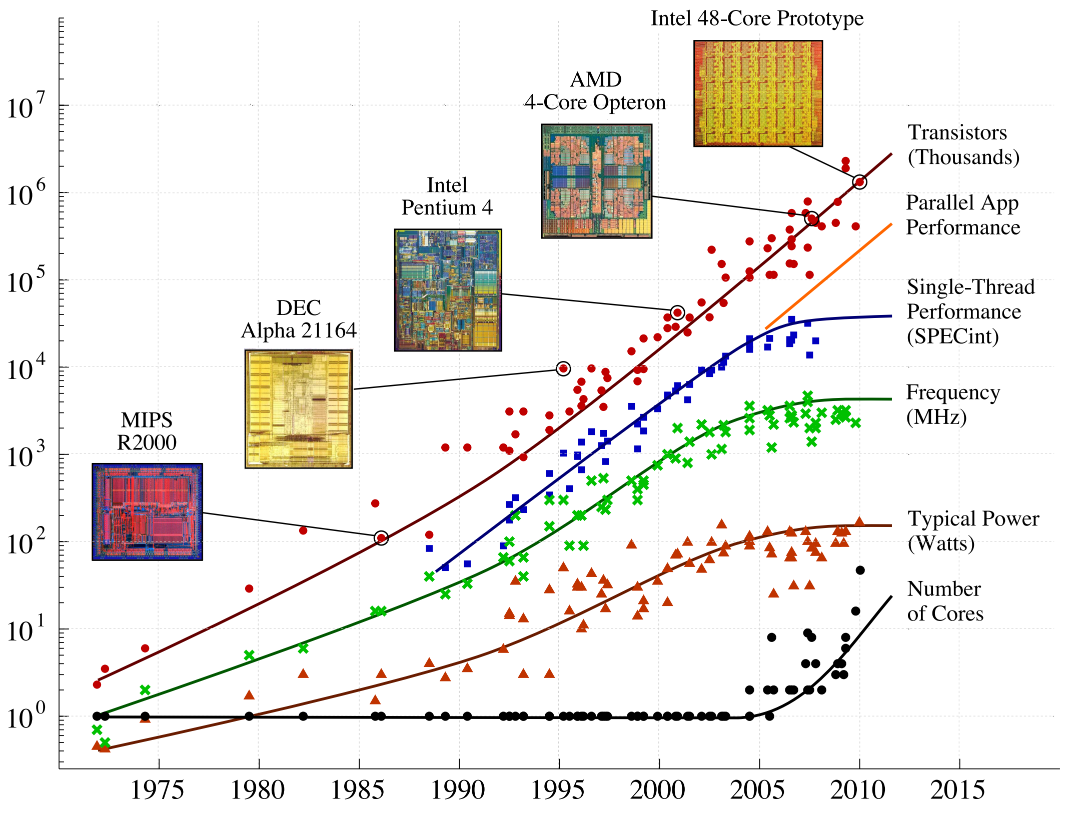
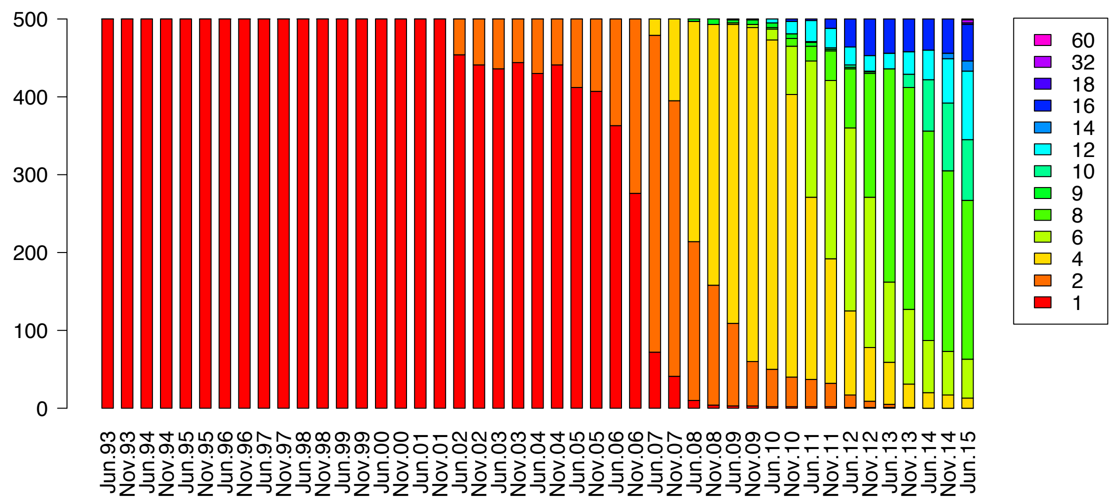
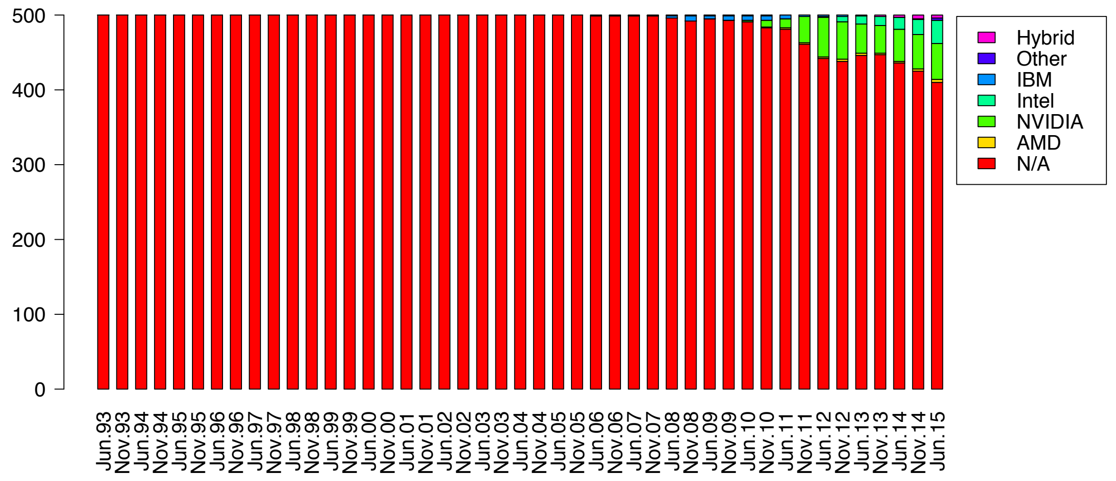


Figure 1.2 – Trends in transistors, performance, and power for general-purpose processors from 1975 to 2010 (from [22]).



(a) Development over time of the cores per socket



(b) Development over time of the accelerator families

Figure 1.3 – Systems share of the supercomputers in the TOP500 list

the November 2002 list. Since then, the cores per socket have increased up to 60 and there are not monocore processors today. This rapid evolution can be seen too in the "Number of Cores" line of Figure 1.2. As sequential programs only run on one of the cores, which are not becoming faster, the software community has been forced to develop and use parallel programming tools.

Another way to fight against heat dissipation issues is the usage of specialized processors. Commonly called *accelerators*, they are designed to carry out only specific tasks but more efficiently than general purpose processors. There exists a broad range of accelerators: manycores like Graphic Processing Units (GPUs) and the Intel Xeon Phi, Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs). Their usage is growing in the last years [107, 94] as can be viewed in Figure 1.3b, which presents the share of supercomputers with accelerators in the TOP500 list. With small beginnings in June 2006, accelerators are present in the 18 % of the supercomputing (contributing about the 34 % of the total performance). Thus, current HPC systems have heterogeneous architectures that combine different kinds of processing elements to achieve high performance with lower power consumption. Note that this design trend is also being followed in no-HPC systems (desktop, laptop, mobile and embedded devices) and even in microprocessor design to solve the problem of the dark silicon [46].

The new microprocessor designs have impacted massively in the memory hierarchy of the computing system, and memory speed is evolving at a slower rate than processor speed [141]. Several examples illustrate this idea. Firstly, current memory systems for multicore processors include three or four levels of cache to reduce latency, but data placement needs to be optimized for this purpose. Secondly, each node of a cluster commonly includes several multicore processors which have DRAM modules attached to them. All the cores of the node can access all the modules through an interconnection network, but at different latencies. These systems are known to have Non-Uniform Memory Access (NUMA) architectures. Thirdly, most accelerators have their own memory hierarchy, separated from the memory of the general purpose processors. Thus, in order to have good performance, memory transfers must be minimized between these two hierarchies.

Software developers are not trained to handle all these levels of increasing complexity [17]. Modeling reality in software is already a difficult task without being aware of the underlying computer architecture. This is one of the reasons why the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) pursues letting experts concentrate on the what-to-solve instead of the how-to-solve [45]. This difficulty in developing new software, and the high cost of rewriting it with the subsequent testing, causes that considerable amounts of legacy software is still in use. And the majority of them were written for monocore architectures. The problem with legacy parallel code is even worse. These codes were optimized for peak performance on a given system and they commonly assume specificities about the underlying hardware architecture and use non-standard extensions of programming languages, assembly code, etc. We would like the performance of a program to scale automatically in future computer systems. In fact, *"[. . .] Software scalability is the most significant limiting factor in achieving the next 10x improvements in performance, and it remains one of the most significant factors in reaching 1,000x [. . .]"* [38].

To sum up, we are in front of one of the biggest challenges in the History of the Computer Science.

1.2 Solutions for the Parallel Challenge

As introduced in the previous section, heterogeneous architectures have impacted tremendously in software development causing a new "software crisis". In the early days of computer science, programmers faced the difficulty of writing useful and efficient computer programs in the required time. The fast growth in the power of the available computers carried out a logical growth in society's ambition to apply these new machines to solve new problems [44]. Today we are facing a very similar problem in the productivity of software developers.

Next, we present several approaches to deal with the current hardware novelties. Section 1.2.1 targets parallelism, i.e., the distribution of the computations of an application to run across more than one core. Section 1.2.2 addresses locality, i.e., the efficient usage of the memory hierarchy to be able to access data as fast as possible. Although introduced separately, note that both must be taken into account simultaneously as will be seen in the rest of this thesis.

1.2.1 Extraction of Parallelism

Current computer architectures present a high degree of complexity regarding the number and organization of the processing elements. We can distinguish four main approaches that the software community has developed to exploit the capabilities of heterogeneous architectures.

The first one is the creation of libraries to write parallel programs. In general, these libraries consist of several functions which expose low-level hardware characteristics that must be explicitly called from one of the most used programming languages. The Message Passing Interface (MPI) [98] defines an interoperable set of routines to exchange data and synchronize processes in C and Fortran. Another relevant example is the Compute Unified Device Architecture (CUDA) [103], which enables the use of the GPU for general purpose programming with C, C++ and Fortran. The programmer has to detail the operation of a single thread in functions called *CUDA kernels*.

The second approach are compiler directives, which specify parallel semantics within sequential programs. In this way, the sequential and the parallel version coexist in the same file. This approach provides more readable codes and eases the interaction between application-domain experts and programmers. The developed codes are more independent from the hardware platform, and new hardware devices supported by the translator can be automatically exploited. The major example is OpenMP [105], which was designed for shared-memory parallel programming in C, C++ and Fortran, but now supports accelerators. OpenACC [131] and OpenHMPP [104] are another influential examples designed specifically for heterogeneous architectures.

The third proposal are parallel programming languages. Typically oriented to specific domains, they enable specialists to express only what needs to be done and not how to do it. Recently, there have appeared general purpose parallel programming languages focused on improving the productivity of parallel software developers. The most representative ones are the Partitioned Global Address Space (PGAS) languages [111, 39] (namely, UPC and UPC++, Titanium and Co-Array Fortran), which expose a shared memory space which is logically partitioned to enable the specification of local, and then more efficient, memory accesses (see Section 1.2.2). Instead of proposing a completely new syntax, PGAS languages are extensions of the most used ones (C and C++, Java and Fortran, respectively). In this way, they overcome the strong investments in training needed by new programming languages.

Finally, the fourth solution are parallelizing compilers that automatically rewrite existing sequential programs into a parallel counterpart [128, 58, 70, 27, 108, 115, 135, 123, 136, 79]. All the approaches mentioned in the previous paragraphs require the intervention of the programmer to write parallel code. In contrast, parallelizing compilers allow developers to write sequential applications, which is familiar to them, and keep legacy code running efficiently on new hardware architectures without the possibility of adding bugs. Code parallelization is one of the most challenging and widely studied topics in compilers since the 1970s [61]. Its difficulty resides in both the two main tasks that a parallelizing compiler has to address: (1) the detection of parallelism to determine what parts in the original source code can be executed concurrently, and (2) the generation of efficient parallel code taking into account the underlying hardware architecture. Some parallelizing compilers rely on programmer support to provide hints (e.g., code annotations) for the selection of the optimal parallel solution, or limit their work to only analyze the input source code and advise the programmer about feasible parallelizations.

Note that, in most occasions, more than one of these approaches are combined to efficiently exploit the complex architecture of computing systems. For instance, in order to increase the scalability of scientific simulations in clusters of multicores, MPI is used for message passing between nodes and OpenMP for threads inside each shared-memory node [97]. Another example of this combination is [56], which exploits the computational capacity of modern clusters that contain both GPUs and Xeon Phi accelerators with UPC++ and CUDA. The development of libraries in PGAS languages for collective communications [126, 10] and algebra routines [55] enables better programability (it is natural for developers to express these operations calling to functions) and better performance (with optimal implementations very close to the underlying hardware architecture but hiding the details to the end user). Lastly, several parallelizing compilers implement a source-to-source approach: instead of generating parallel binary code directly, they generate source code annotated with compiler directives for parallel programming. In this thesis, we have chosen the source-to-source approach (see Section 1.3).

1.2.2 Extraction of Locality

The locality of reference has been a key enabler of good performance along the whole history of computers [41]. First effects of the locality principle were observed with the introduction of virtual memory in 1959, when it was noted that the performance of the Atlas supercomputer depended on how compilers placed code and data onto memory pages. As a page fault increases the execution time of a program significantly, minimizing them is critical to system performance. In the following years, the popularization of multiprogramming caused the total colapse of contemporary systems: the computer wasted its time resolving page faults instead of running programs in the CPU. This problem was known as "thrashing".

Until 1966, the literature always considered that each program was assigned to a fixed-size memory partition. However, Denning [41] introduced a new idea, the *working set*, which was defined as *"[...] the set of memory pages used during a fixed-length sampling window in the immediate past [...]"* (without including the time needed to replace memory pages). This idea assumes that previously referenced pages are going to be referenced again in the immediate future. And, surprisingly, programs show this behavior even when it is not explicitly coded. There exist two reasons for it: temporal clustering due to loops and modules with private data; and spatial clustering due to arrays, structs, etc. Nowadays, the locality principle has been adopted in the design of both hardware and software: virtual memory, data and instruction caches in CPUs, buffers in interconnection networks, storage of recently viewed sites in web browsers, etc.

A wide range of techniques are useful to improve locality of existing applications: loop interchange, fission and fusion of loops and arrays, and tiling [3, 139]; hardware and software prefetching [68, 143]; data placement [43, 112, 138]; design of ad-hoc memory systems [34, 20]; etc. Note that these techniques have been or can be implemented in a compiler to be applied automatically. In addition, compilers have demonstrated to be successful tools for the prediction of the memory hierarchy performance [51, 13].

1.3 Thesis Approach and Structure

Compilers played a main role to overcome the first software crisis. They have been the essential piece for the actual widespread usage of high-level programming languages. The performance of compiler-generated binary code is usually very close to the peak performance of the target machine, which is extraordinarily difficult to achieve with manual tuning for large programs [61]. In addition, the powerful abstractions of high-level programming languages make developers more productive. And compilers are able to provide abstraction without performance penalty across different architectures.

In addition, significant re-engineering and refactoring of existing software is needed to enable the use of new hardware features as mentioned in Section 1.2. Due to the high cost of manual transformation, an automated approach to transform existing software would be highly beneficial. Even when parallelism and heterogeneity are taken into account from the start of a software project, writing efficient programs is a challenging task [17]. Like the European Technology Platform for High Performance Computing (ETP4HPC), we believe that compilers will be the critical piece to overcome the parallel challenge [48].

This thesis focuses on the development of compilation techniques for the automatic extraction of parallelism and locality to target heterogeneous architectures. On the one hand, we propose the automatic parallelization of sequential code in multicore CPUs and GPUs. We have chosen a source-to-source approach that generates code annotated with OpenMP/OpenHMPP directives to ease the interaction with application-domain experts and to enable the high-performance binary code generation of current and future compilers. On the other hand, we propose the modeling of program behavior from the point of view of the memory accesses. Later, existing memory optimization techniques based on the polyhedral model (see Appendix B), and any other static or dynamic optimization technique in the absence of source and/or binary code, can be applied. The rest of this manuscript is organized as follows:

- Chapter 2 introduces a new compiler intermediate representation based on the concept of domain-independent kernel (see Appendix A) named KIR, which is insensitive to syntactic variations in the source code (e.g., use of arrays, pointers and/or complex control flow), and exposes multiple levels of parallelism to the compiler. Next, the chapter presents an automatic partitioning strategy to map the parallelism exposed by the KIR to modern hardware based on multicore CPUs.

- Chapter 3 contributes a new technique to automatically rewrite sequential programs into a parallel counterpart targeting GPU-based heterogeneous systems. This approach exploits the characteristic GPU hardware architecture, in particular the memory hierarchy.

- Chapter 4 explores the automatic reconstruction of affine loops from a trace of their memory accesses for both sequential and parallel codes and supporting moderate amounts of nonlinearity.

- Finally, Chapter 5 summarizes the main contributions of the work done in this thesis and provides some insights on future research directions.

Chapter 2

A Novel Compiler Support for Multicore Systems

The widespread use of multicore processors is not a consequence of significant advances in parallel programming. On the contrary, multicore processors arise due to the complexity of building power-efficient, high-clock-rate, single-core chips as viewed in Section 1.1. Among the different proposals to overcome the parallel challenge (see Section 1.2.1), automatic parallelization of sequential applications is the ideal solution for making parallel programming as easy as writing programs for sequential computers. However, automatic parallelization remains a grand challenge due to its need for complex program analysis and the existence of unknowns during compilation. This chapter proposes a new method for converting a sequential application into a parallel counterpart that can be executed on current multicore processors. It hinges on an intermediate representation based on the concept of domain-independent kernel (from now on, *diKernels* —see Appendix A—). Such diKernel-centric view hides the complexity of the implementation details, enabling the construction of the parallel version even when the source code of the sequential application contains different syntactic variations of the computations (e.g., pointers, arrays, complex control flows). Experiments that evaluate the effectiveness and performance of our approach with respect to state-of-the-art compilers are also presented. The benchmark suite consists of synthetic codes that represent common diKernels, dense/sparse linear algebra and image processing routines, and full-scale applications from SPEC CPU2000.

The rest of the chapter is organized as follows. Section 2.1 presents our automatic parallelization approach driven by diKernels. More specifically, a new intermediate representation is formally defined, and the automatic partitioning procedure targeting multicore processors is proposed. Section 2.2 details the behavior of our approach for the case studies of the benchmark suite. Section 2.3 presents the experimental results. Section 2.4 discusses related work. Finally, Section 2.5 summarizes the main conclusions of the chapter.

2.1 Automatic Parallelization Driven by diKernels

It is clear that parallelizing compilers are a critical piece for the software community to meet the parallel challenge (see Section 1.2.1). However, despite great advances in compiler technology during the last decades [1, 3, 139], current production compilers usually fail to parallelize even simple sequential programs (as will be demonstrated in Section 2.3.2). The main reason for this failure is that they address the automatic detection of parallelism by running classical dependence analyses on standard statement-based intermediate representations (e.g., Abstract Syntax Trees —ASTs—, Data Dependence Graph —DDG—, Control Flow Graph —CFG—). Such IRs are well suited for code generation, but not for the analysis of full-scale applications because they are extremely sensitive to syntactic variations in the source code. Thus, current parallelizing compilers are driven by mathematical models that respect all the dependences present in these IRs even if they are merely implementation artifacts.

This section presents a new approach for the automatic parallelization of sequential programs based on the concept of diKernel (see Appendix A). Figure 2.1 shows the workflow of the approach, which handles the dependences of the compiler IR through diKernels (in contrast to other approaches based on a classical dependence analysis between source code statements). The first stage is the construction of a new IR built on top of diKernels, named *KIR*, that exposes multiple levels of parallelism in sequential programs. The second stage is an automatic

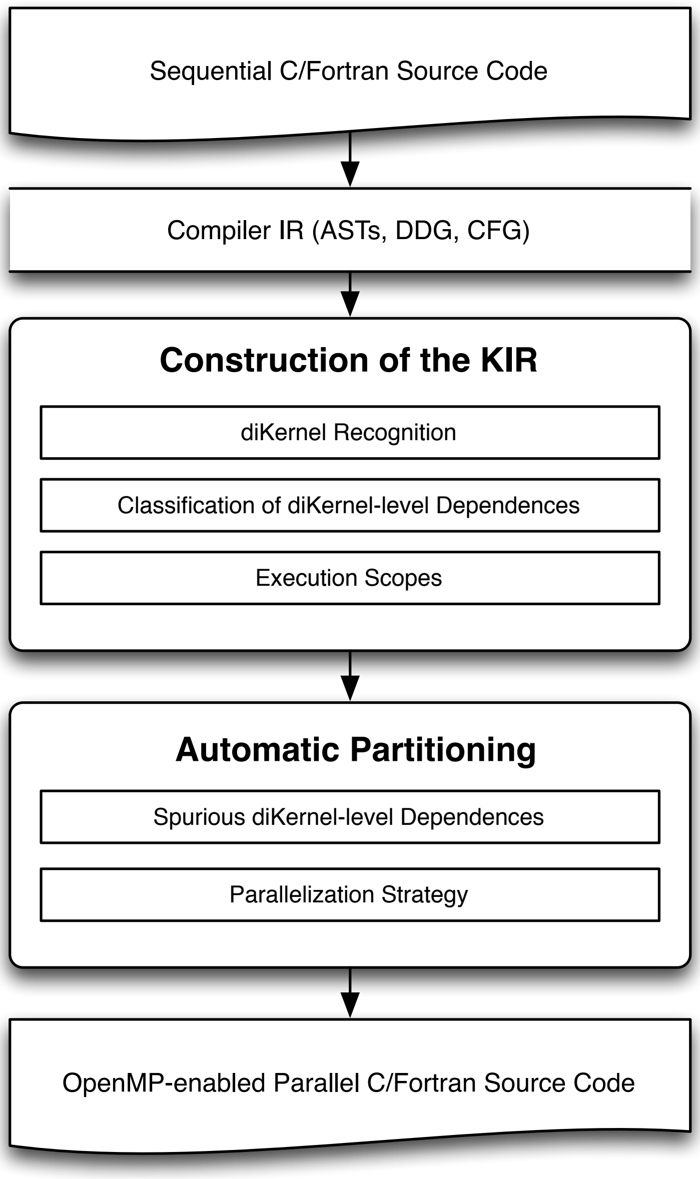

Sequential C/Fortran Source Code
Compiler IR (ASTs, DDG, CFG)
Construction of the KIR
diKernel Recognition
Classification of diKernel-level Dependences
Execution Scopes
Automatic Partitioning
Spurious diKernel-level Dependences
Parallelization Strategy
OpenMP-enabled Parallel C/Fortran Source Code

Figure 2.1 – Workflow of the automatic parallelization approach driven by domain-independent kernels.

partitioning technique that generates a parallel counterpart for a sequential application targeting multicore processors. Next, Section 2.1.1 formally defines the KIR and Section 2.1.2 sketches the KIR-driven automatic partitioning procedure.

2.1.1 KIR: A diKernel-based Intermediate Representation

Without loss of generality, assume that the source code of a program is represented by a statement-based IR that consists of a forest of ASTs, a DDG and a CFG. For illustrative purposes, Figure 2.2 shows an implementation of the dense matrix-vector multiplication. In each iteration i of the outer loop for_i, the dot product of the i^{th} row of matrix A and vector x is computed (see lines 2–5). Next, the result is stored in the i^{th} element of vector y (line 6). Figure 2.3 presents an excerpt of a typical IR where ASTs represent the source code statements. The CFG groups the ASTs into basic blocks (dashed boxes) with precedence relationships (dashed edges). Loops are represented with preheader, header and latch basic blocks (BB) that initialize the loop index, check the loop exit condition and increment the loop index (see $BB0$, $BB5$ and $BB4$, respectively, for the loop for_i). Finally, the DDG exhibits data dependences between statements (solid edges).

The construction of the KIR consists of three steps: first, the construction of the diKernels of the program and their data dependence relationships (Definitions 2.1.1–2.1.2); second, the construction of the flow dependences between diKernels (Definitions 2.1.3–2.1.5); and third, the construction of the hierarchy of execution scopes (Definitions 2.1.6–2.1.8), which reflects the computational stages of the sequential program and groups diKernels into these stages.

```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = 0; j < m; j++) {
4     t = t + A[i][j] * x[j];
5   }
6   y[i] = t;
7 }
```

Figure 2.2 – Source code of the dense matrix-vector multiplication.

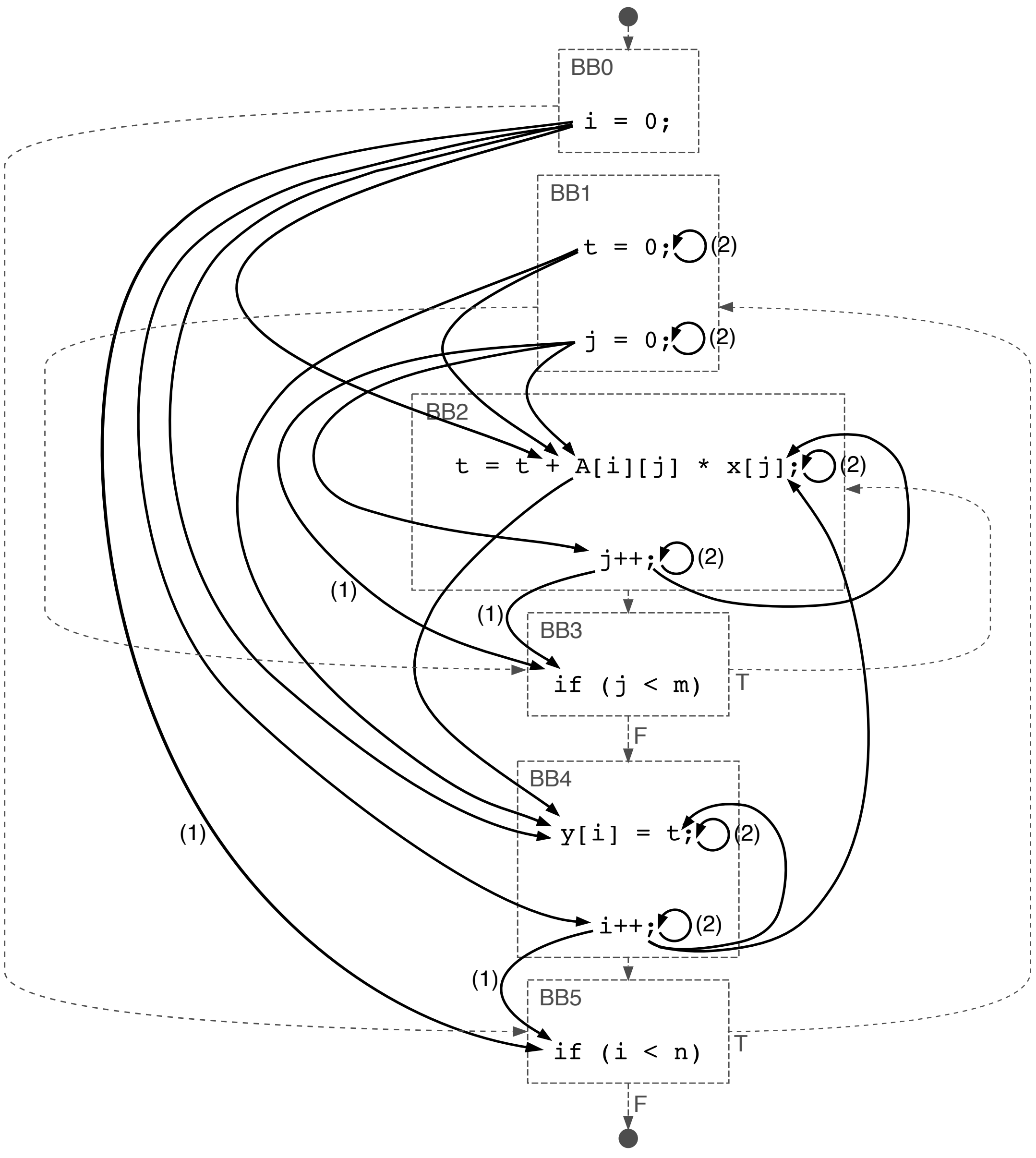


Figure 2.3 – Standard IR (ASTs, CFG and DDG) based on the statements of the program of the dense matrix-vector multiplication of Figure 2.2.

Definition 2.1.1. *Let consider the DDG of the input program. Ignore the ASTs representing flow-of-control statements (e.g., branch, return, break, jump) and their incoming edges (note that there cannot exist outgoing edges from flow-of-control statements). A **diKernel** $K = (N, E)$ is a strongly connected component of the resulting DDG where N is the set of ASTs such that each AST $x_i \in N$ is an assignment-statement, and E is the set of edges of the SCC. The term $K{<}x_1 \ldots x_n{>}$ denotes the ASTs $x_1 \ldots x_n$ that belong to N.*

Definition 2.1.2. *Let SCC_x and SCC_y be two strongly connected components of the DDG associated with diKernels $K{<}x_1 \ldots x_n{>}$ and $K{<}y_1 \ldots y_m{>}$, respectively. A **diKernel-level data dependence** is an edge $x_i \rightarrow y_j$ of the DDG such that $SCC_x \neq SCC_y$, with $x_i \in \{x_1 \ldots x_n\}$ and $y_j \in \{y_1 \ldots y_m\}$. The term $K{<}x_1 \ldots x_n{>} \rightarrow K{<}y_1 \ldots y_m{>}$ denotes that DDG edge that crosses diKernel boundaries.*

For illustrative purposes, Figure 2.4 shows the diKernel-level data dependence graph of the dense matrix-vector multiplication of Figure 2.2. According to Definition 2.1.1, the branch statements of $BB3$ (`if (j<m)`) and $BB5$ (`if (i<n)`) are ignored for the construction of the diKernels. The computation of the for_i index i (line 1, Figure 2.2) is represented by two diKernels: $K{<}i_{BB0}{>}$ for the index initialization (the term i_{BB0} denotes the statement `i=0` of the basic block $BB0$ in Figure 2.3); and $K{<}i_{BB4}{>}$ for the index update. In a similar manner, the for_j index j is represented by $K{<}j_{BB1}{>}$ and $K{<}j_{BB2}{>}$. The value of the dot product is stored in t. This temporary variable is initialized in $K{<}t_{BB1}{>}$ at the beginning of each for_i iteration (line 2, Figure 2.2) and updated in $K{<}t_{BB2}{>}$ throughout the execution of the inner loop for_j (line 4, Figure 2.2). Finally, the storage of the dot product value in the output array y is represented by the diKernel $K{<}y_{BB4}{>}$. By construction, the edges of the DDG are captured in the diKernel-level data dependence graph as follows: first, the incoming edges of branch statements are ignored (see edges with label (1) in Figure 2.3); second, the edges whose source and target statements belong to the same diKernel are subsumed in the diKernel (see edges with label (2) in Figure 2.3); and third, the edges that cross diKernel boundaries are exposed as diKernel-level data dependences in Figure 2.4 (see non-labeled forward and backward edges in Figure 2.3).

The second step in the construction of the KIR is to determine flow dependences between diKernels. The diKernel-level data dependence graph does not

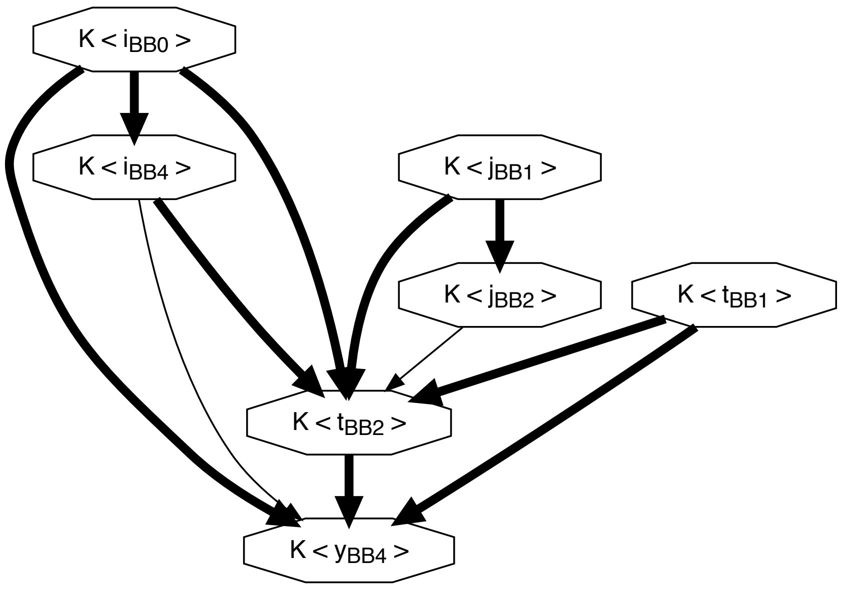


Figure 2.4 – Steps 1 and 2 of the construction of the KIR of the dense matrix-vector multiplication of Figure 2.2: diKernel-level data dependence graph (→) with diKernel-level flow dependences (➡).

completely reflect the order in which diKernels are executed. For this purpose, we define dominance relationship between statements (Definition 2.1.3) and introduce a notation for the production and use of values throughout the program (Definition 2.1.4). Next, we are able to identify flow dependences between diKernels (Definition 2.1.5) taking into account the CFG, the DDG, the Dominator Tree (DT) and the range of values produced/used in the program (both for scalar variables and ranges of non-scalar variables —e.g. arrays, structs—).

Definition 2.1.3. *Let x_i and x_j be ASTs that represent statements of a program. We say there is a **statement-level dominance relationship** in the following situations:*

- *Assume that x_i and x_j belong to the same basic block BB. If x_i precedes x_j within BB, then x_i dominates x_j.*

- *Assume that x_i and x_j belong to different basic blocks BBi and BBj. If BBi dominates BBj or BBi belongs to the body of a loop whose header BBh dominates BBj, then x_i dominates x_j.*

Definition 2.1.4. *Let x_i and y_j be ASTs representing statements of a program that define values from variables x and y, respectively.*

- *DEF(x, x_i) is the* ***range of values of variable** x **produced*** *throughout the execution of statement x_i.*
- *USE(x, y_j) is the* ***range of values of variable** x **used*** *throughout the execution of statement y_j.*

Definition 2.1.5. *Let $K{<}x_1 \ldots x_n{>} \rightarrow K{<}y_1 \ldots y_m{>}$ be a diKernel-level data dependence that connects diKernels K_x and K_y through the DDG edge $x_i \rightarrow y_j$. We say that it is a* ***diKernel-level flow dependence****, K_x ➡ K_y, if it holds that statement x_i dominates statement y_j and DEF(x, x_i) $\supseteq$ USE(x, y_j).*

For illustrative purposes, Figure 2.4 highlights the diKernel-level flow dependences of the diKernel-level data dependence graph. As can be seen, $K{<}i_{BB0}{>}$ ➡ $K{<}i_{BB4}{>}$ represents the flow between the initialization of the loop index i in the preheader of the loop ($BB0$) and its update in the latch of the loop ($BB4$). The two conditions hold as follows: first, the statement i_{BB0} dominates the statement i_{BB4} because $BB0$ dominates $BB4$; and second, i is a scalar variable, thus DEF(i, i_{BB0}) = USE(i, i_{BB4}) = $\{i\}$. The source code of the dense matrix-vector multiplication of Figure 2.2 does not contain diKernel-level flow dependences between non-scalar variables. Note that, in many programs, dependences are coded in very complex ways, for instance, through the usage of pointers. Our approach deals with pointers in the algorithm for recognition of diKernels [16], which applies *array recovery* techniques to transform pointer-based programs into a semantically equivalent array-based form (similar to [52]). Illustrative examples of ranges of values of non-scalar variables (both array-based and pointer-based) produced/used in different statements will be described later in Section 2.2.

The third step in the construction of the KIR is to build the hierarchy of execution scopes. Typically, loops often consume most of the execution time and thus optimizations that improve their performance may have a significant impact on the overall efficiency. The goal of the hierarchy of execution scopes is to expose the computational stages of the program to the compiler. For this purpose, execution scopes are built upon loops (Definition 2.1.6) and organized in a hierarchy of execution scopes (Definition 2.1.7). In addition, diKernels are attached to execution scopes (Definition 2.1.8) to capture the computational stage of the sequential program where they are executed. Finally, diKernels are labeled with the type of assignment, reduction or recurrence diKernel that they carry out during the computation of their execution scope (see Appendix A).

Definition 2.1.6. *Assume that a program is represented by a hierarchy of regions. An* ***execution scope*** *is a loop region* R_L *such that there exists a* perfectly *nested loop* $L, L_1, \ldots, L_n$*, being* L *the outermost loop.*

Definition 2.1.7. *The* ***hierarchy of execution scopes*** *is a tree whose set of nodes are the execution scopes of the program. The root execution scope is a special node that represents the program as a whole. The children of a node are built as follows. Let* R_L *be an execution scope,* L *its outermost loop, and* L_{parent} *the parent loop of* L*. If* L_{parent} *does not exist, then* R_L *is set as child of the root execution scope. Otherwise,* R_L *is set as child of* R_{parent}*, where* R_{parent} *is the execution scope of* L_{parent}*.*

Definition 2.1.8. *Let* $x_1 \ldots x_n$ *be the ASTs of a diKernel* $K{<}x_1 \ldots x_n{>}$*. Let* $L_1, \ldots, L_n$ *be the innermost loops that contain* $x_1, \ldots, x_n$*, respectively. We say that* $K{<}x_1 \ldots x_n{>}$ ***belongs to the execution scope*** R_L *if and only if* R_L *is the execution scope of the innermost common loop for* $L_1, \ldots, L_n$*. Nevertheless, if* x_1 *is the index of a loop* L*, and* $K{<}x_1{>}$ *is the diKernel that initializes this loop index, then* $K{<}x_1{>}$ *belongs to* R_L*.*

For illustrative purposes, Figure 2.5 shows the hierarchy of execution scopes. The dense matrix-vector multiplication of Figure 2.2 contains two loops for_i and for_j that are not perfectly nested. Thus, the execution scope of loop for_j (from now on, ES_for_j) is a child of ES_for_i, which is in turn a child of the root execution scope that represents the whole program. According to Definition 2.1.8, the diKernels $K{<}j_{BB1}{>}$ and $K{<}j_{BB2}{>}$ that capture the computation of loop index j are attached to ES_for_j (in a similar manner, $K{<}i_{BB0}{>}$ and $K{<}i_{BB4}{>}$ are attached to ES_for_i). Note that these diKernels and their incoming/outgoing diKernel-level dependences (e.g. $K{<}j_{BB1}{>} \Rightarrow K{<}j_{BB2}{>}$) are not shown in the KIR of Figure 2.5: their computation is not relevant to determine if the operations carried out in the program can be executed in parallel, and the loop indices are already represented in the notation of the execution scope and diKernel types. The remaining diKernels $K{<}t_{BB1}{>}$, $K{<}t_{BB2}{>}$ and $K{<}y_{BB4}{>}$ contain a unique assignment-statement, thus

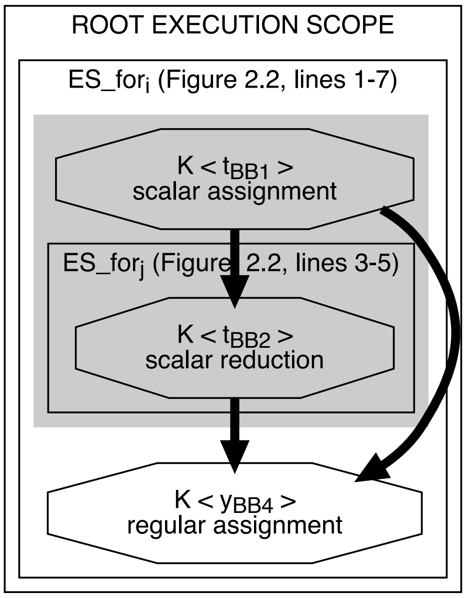


Figure 2.5 – Step 3 of the construction of the KIR of the dense matrix-vector multiplication of Figure 2.2: hierarchy of execution scopes.

they are attached to the execution scope of the innermost loop that contains each statement: ES_for_i, ES_for_j and ES_for_i, respectively.

In summary, the new diKernel-based IR captures the whole semantics of the sequential program, but it only exposes to the compiler the program features that are key to find the parallelism implicit in the sequential code.

2.1.2 Automatic Partitioning driven by the KIR

This section presents a new KIR-driven automatic partitioning technique to transform a sequential program into a parallel counterpart for multicore processors. As input, our approach takes the KIR presented in Section 2.1.1. The method consists of two steps. The first step is to filter out the diKernel-level dependences of the KIR that do not prevent the parallelization of the sequential application (from now on, *spurious* diKernel-level dependences). The second step is the construction of an efficient OpenMP-enabled parallelization strategy for the sequential program as a whole.

Regarding the first step, the principal source of *spurious* diKernel-level dependences are memory-related dependences (i.e., anti- and output dependences). Hence, it is necessary to check if there really exists a flow of information throughout the diKernel-level dependences that could prevent the parallel execution of the given code.

Definition 2.1.9. *A diKernel-level dependence is* ***spurious*** *if one of the following conditions is fullfilled:*

1. *Let $K{<}x_i{>}$ and $K{<}y_j{>}$ be diKernels connected with a diKernel-level flow dependence $K{<}x_i{>}$ ➡ $K{<}y_j{>}$. If $K{<}x_i{>}$ is* ***privatizable****, then $K{<}x_i{>}$ ➡ $K{<}y_j{>}$ is spurious.*

   A scalar variable x defined within a loop is said to be privatizable [3] with respect to that loop if and only if every path from the beginning of the loop body to a use of x within that body must pass through a definition of x before reaching that use. Thus, given a set of privatizable scalar variables $x, y, z \ldots$ in a loop L, our technique searches for connected subgraphs of the KIR contained in the execution scope associated to L that represent the computations carried out on the variables $x, y, z \ldots$ Each subgraph, including children execution scopes with only diKernels referencing these variables, is **shaded** in order to omit these parts of the KIR in the discovery of parallelism.
2. *Let $K{<}x_i{>}$ and $K{<}y_j{>}$ be diKernels connected with a diKernel-level* ***data*** *dependence $K{<}x_i{>} \rightarrow K{<}y_j{>}$. If x_i dominates y_j and $DEF(x, x_i) \cap USE(x, y_j) = \emptyset$, then $K{<}x_i{>} \rightarrow K{<}y_j{>}$ is spurious.*

3. *Consider a sequence of three execution scopes, each one with an attached diKernel $K{<}x_i{>}$, $K{<}x_j{>}$ and $K{<}y_l{>}$, respectively. Assume that the diKernels are connected with the diKernel-level* ***flow*** *dependences $K{<}x_i{>}$ ➡ $K{<}x_j{>}$, $K{<}x_j{>}$ ➡ $K{<}y_l{>}$, and $K{<}x_i{>}$ ➡ $K{<}y_l{>}$. If $DEF(x, x_i) = USE(x, x_j) = DEF(x, x_j) = USE(x, y_l)$, then $K{<}x_i{>}$ ➡ $K{<}y_l{>}$ is spurious.*

In the dense matrix-vector multiplication of Figure 2.2, the scalar variable t is privatizable because every path from the beginning of for_i to the uses at lines 4 and 6 goes through the definition of line 2. Therefore, the KIR of Figure 2.5 contains a shaded subgraph composed of $K{<}t_{BB1}{>}$, $K{<}t_{BB2}{>}$, $K{<}t_{BB1}{>} \Rightarrow K{<}t_{BB2}{>}$, and the execution scope ES_for_j. Consequently, the diKernel-level dependences $K{<}t_{BB1}{>} \Rightarrow K{<}y_{BB4}{>}$ and $K{<}t_{BB2}{>} \Rightarrow K{<}y_{BB4}{>}$ are marked as spurious according to Definition 2.1.9, case 1.

The second step is to determine the OpenMP-enabled parallelization strategy that will drive the generation of parallel source code. The key idea is to find the critical path in the KIR and execute such computations within a unique parallel region in order to minimize thread creation/destruction. Our approach is based on the existence of parallelizing transformations designed for each type of diKernel (see Appendix A for the collection of diKernels used in this thesis). Thus, the critical path is the longest path in the KIR that only contains diKernel-level flow dependences and parallelizable diKernels. The procedure is as follows: (1) scalar reduction diKernels are executed as parallel reduction operations (using the `reduction` OpenMP clause); (2) regular assignment and regular reduction diKernels are converted into forall parallel loops [2]; (3) irregular assignment and irregular reduction diKernels are transformed via an array expansion technique [50, 59]; (4) in general, recurrence diKernels cannot be transformed in parallel code, but there exist parallelizing transformations for particular cases [28]. Examples will be shown in Section 2.2. The automatic exploitation of fine-grain parallelism with SIMD instructions, in particular for non-parallelizable diKernels, is out of the scope of this thesis as it has been successfully addressed by other complementary techniques [91, 127, 122, 85].

The target architecture addressed in this chapter are multicore processors. In general, the parallelism available in parallelizable diKernels will suffice to generate a few coarse-grain threads to run on the multicore processor. As a consequence, when the KIR presents several critical paths that share computations, the non-shared parts are serialized in a unique critical path within the parallel region. Note that no synchronization between the non-shared computations is needed as they are not connected via diKernel-level dependences.

Given a parallel region of a critical path, our automatic partitioning strategy minimizes the synchronization overhead between diKernels. Thus, for each $K{<}x_i{>} \Rightarrow K{<}y_j{>}$, the algorithm checks that: (1) $K{<}x_i{>}$ and $K{<}y_j{>}$ represent conflict-free computations that can be reordered arbitrarily; and (2) given DEF(x, x_i) for $K{<}x_i{>}$ and USE(x, y_j) for $K{<}y_j{>}$, then DEF(x, x_i) = USE(x, y_j). Under these conditions, the same workload distribution is scheduled for $K{<}x_i{>}$ and $K{<}y_j{>}$ in order to guarantee that the same thread is responsible for producing the value of $K{<}x_i{>}$ that is consumed by $K{<}y_j{>}$. As a result, no barrier is needed to preserve the diKernel-level flow dependence $K{<}x_i{>} \Rightarrow K{<}y_j{>}$. The reordering to assign the same workload distribution is achieved by applying the same OpenMP scheduling clause.

Finally, the creation and destruction of OpenMP threads is minimized. If the parallel region is contained in a loop, OpenMP `parallel` directives are moved to enclose that loop. The critical path is confined between barriers, and the remaining computations in the loop are isolated into OpenMP `single` regions. This situation is very common in numerical simulations, as will be shown in Sections 2.2.4 and 2.2.5.

For illustrative purposes, Figure 2.6 shows the parallelized code of the dense matrix-vector multiplication. The critical path of the KIR (see Figure 2.5) consists of a single diKernel $K{<}y_{BB4}{>}$ attached to ES_for_i. The type of diKernel is parallelizable. Thus, the conflict-free computations of the regular assignment are converted into a forall parallel loop (see line 3 of Figure 2.6). Note that the variables covered by the shaded subgraph are privatized within the parallel region (line 1, Figure 2.6).

Overall, our approach enables the automatic parallelization of full-scale applications for multicore processors minimizing the parallel overhead. The KIR naturally reflects the structure of the source code and thus avoids the violation of the data-flow constraints specified by the programmer. The next section details more complex examples extracted from both synthetic and real codes.

2.2 Automatic Parallelization of the Benchmark Suite

In this section, the potential of our KIR-driven automatic parallelization technique is exemplified with a set of benchmarks that are representative of important problems in computational science and engineering. Section 2.2.1 presents

```
#pragma omp parallel shared(A,x,y) private(i,j,t)
{
#pragma omp for schedule(static)
  for (i = 0; i < n; i = i + 1) {
    t = 0;
    for (j = 0; j < m; j = j + 1) {
      t = (t) + ((A[i][j]) * (x[j]));
    }
    y[i] = t;
  }
}
```

Figure 2.6 – Parallelized code of the dense matrix-vector multiplication of Figure 2.2.

synthetic benchmarks that represent the main types of diKernels. Sections 2.2.2 and 2.2.3 describe important routines from dense/sparse linear algebra and image processing. Finally, Sections 2.2.4 and 2.2.5 focus on two full-scale applications from the SPEC CPU2000 benchmark suite.

2.2.1 Synthetic Benchmarks

Some simple implementations of assignment, reduction and recurrence diKernels are shown in Figure 2.7 (see Appendix A for the definition of each type of diKernel). In all cases, the KIR consists of one execution scope ES_for_i (apart from the root execution scope) that contains one diKernel (either $K{<}r_3{>}$ or $K{<}A_2{>}$). Note that the subindex refers to the line number (e.g. the term r_3 refers to the assignment-statement `r=r+i` in line 3 of Figure 2.7c). The most relevant difference between the examples is the type of diKernel (see the captions of Figure 2.7).

From the point of view of the automatic partitioning strategy, the examples of Figure 2.7 present a critical path composed of one diKernel. The parallelizing strategy hinges on the existence of parallelizing transformations specifically designed for each type of diKernel. As a result, the regular assignment of Figure 2.7a and the regular reduction of Figure 2.7f represent conflict-free loop iterations that are transformed into forall parallel loops [2]. The scalar reductions of Figures 2.7c–2.7e are executed as parallel reductions, which are usually sup-

```
for (i = 0; i < n; i++) {
  A[i] = 2;
}
```

(a) Regular assignment.

```
for (i = 0; i < n; i++) {
  A[f[i]] = 3;
}
```

(b) Irregular assignment.

```
r = 0;
for (i = 0; i < n; i++) {
  r = r + i;
}
```

(c) Scalar reduction with closed-form expression.

```
r = 0;
for (i = 0; i < n; i++) {
  r = r + A[i];
}
```

(d) Scalar reduction with array reference.

```
r = A[0];
for (i = 1; i < n; i++) {
  if (A[i] < r) r = A[i];
}
```

(e) Scalar reduction with conditional control flow.

```
for (i = 0; i < n-1; i++) {
  A[i+1] = A[i+1] + 5;
}
```

(f) Regular reduction.

```
for (i = 0; i < n; i++) {
  A[f[i]] = A[f[i]] + 3;
}
```

(g) Irregular reduction.

```
for (i = 1; i < n; i++) {
  A[i] = A[i] + A[i-1];
}
```

(h) Regular recurrence.

Figure 2.7 – Synthetic codes of representative assignment, reduction and recurrence diKernels.

ported in current parallel programming environments. The irregular assignment of Figure 2.7b and the irregular reduction of Figure 2.7g present cross-iteration dependences that are handled by parallelizing transformations based on array expansion [50, 59]. Finally, the regular recurrence of Figure 2.7h is recognized as a parallel prefix operation and an ad-hoc parallelizing transformation is applied [120].

2.2.2 Dense/Sparse Matrix-Vector Multiplication

Different versions of the matrix-vector multiplication have been proposed in the literature. In Section 2.1, the dense matrix-vector product (*DenseAMUX* from now on) was studied in detail from the point of view of our KIR-driven automatic parallelization approach. In this section, three additional sparse versions extracted from SparsKit-II [118] will be studied: *AMUX*, *AMUXMS* and *ATMUX*.

The benchmark AMUX (see Figure 2.8b) multiplies a matrix A stored in compressed sparse row (CSR) format by a vector x. The source code is very similar to DenseAMUX of Figure 2.8a, the differences being the irregular bounds of the inner loop (see $ia[i]$ and $ia[i+1]$ in line 3) and the irregular accesses of arrays A, x and ja (see line 4). Thus, both DenseAMUX and AMUX are represented by the same KIR (Figure 2.8c). Note that the compile-time unknowns introduced in AMUX by the CSR format (irregular loop bounds and irregular array accesses) are not exposed in the KIR as they do not determine the implicit parallelism available in the program (they mainly impact on locality exploitation). Consequently, the partitioning strategy behaves as described in Section 2.1.2 and succeeds in generating parallel code by privatizing t computed in $K{<}t_2{>}$ and $K{<}t_4{>}$ and generating a forall loop for the regular assignment $K{<}y_6{>}$ computed in for_i (see Figure 2.8d).

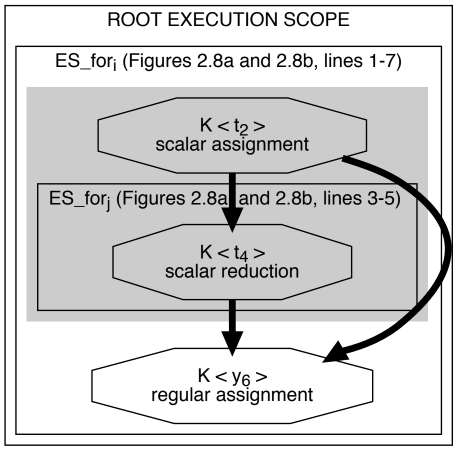
The benchmark AMUXMS (Figure 2.9a) multiplies a matrix A in modified sparse row (MSR) format by a vector x. The source code first initializes the output vector y with the product of the diagonal of matrix A (stored in the first n entries of A) and vector x. Next, the remaining operations are computed and accumulated in the result $y[j]$ (with $j \in \{0 \ldots n-1\}$), using a regular access pattern. Again, the key characteristics are the irregular loop bounds (line 5) and the irreg-

```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = 0; j < m; j++) {
4     t = t + A[i][j] * x[j];
5   }
6   y[i] = t;
7 }
```

(a) Source code of DenseAMUX (dense matrix-vector multiplication).

```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = ia[i]; j < ia[i+1]-1; j++) {
4     t = t + A[j] * x[ja[j]];
5   }
6   y[i] = t;
7 }
```

(b) Source code of routine AMUX from SparsKit-II.



(c) KIR for DenseAMUX and AMUX.

```
 1 #pragma omp parallel shared(A,ia,ja,x,y) private(i,j,t)
 2 {
 3 #pragma omp for schedule(static)
 4   for (i = 0; i < n; i++) {
 5     t = 0;
 6     for (j = ia[i]; j < (ia[i+1] - 1); j = j + 1) {
 7       t = (t) + ((A[j]) * (x[ja[j]]));
 8     }
 9     y[i] = t;
10   }
11 }
```

(d) Parallelized code of the routine AMUX.

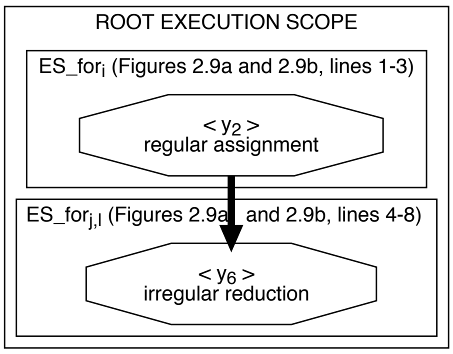
Figure 2.8 – Dense and sparse matrix-vector multiplication.

```
1 for (i = 0; i < n; i++) {
2   y[i] = A[i] * x[i];
3 }
4 for (j = 0; j < n; j++) {
5   for (l = ja[j]; l < ja[j+1]-1; l++) {
6     y[j] = y[j] + A[l] * x[ja[l]];
7   }
8 }
```

(a) Source code of routine AMUXMS from SparsKit-II.

```
1 for (i = 0; i < n; i++) {
2   y[i] = 0;
3 }
4 for (j = 0; j < n; j++) {
5   for (l = ia[j]; l < ia[j+1]-1; l++) {
6     y[ja[l]] = y[ja[l]] + x[j] * A[l];
7   }
8 }
```

(b) Source code of routine ATMUX from SparsKit-II.


ROOT EXECUTION SCOPE
ES_for$_i$ (Figures 2.9a and 2.9b, lines 1-3)
< y_2 >
regular assignment
ES_for$_{j,l}$ (Figures 2.9a and 2.9b, lines 4-8)
< y_6 >
irregular reduction

(c) KIR for AMUXMS and ATMUX.

Figure 2.9 – Variations of sparse matrix-vector multiplication.

ular accesses of arrays A, x and ja (line 6). The benchmark ATMUX (Figure 2.9b) multiplies the transpose of a matrix A in CSR format by a vector x. This routine is very similar to AMUXMS, the only differences being the initial value of y (line 2) and the use of an irregular access pattern to accumulate the results in y (see irregular access $y[ja[l]]$ in line 6).

AMUXMS and ATMUX are also represented by a unique KIR (see Figure 2.9c) with two execution scopes ES_for_i and $ES\_for_{j,l}$. Note that for_j and for_l are two perfectly nested loops and, according to Definition 2.1.6, they are represented by a unique execution scope. In both routines, ES_for_i contains a regular assignment diKernel $K{<}y_2{>}$ (note that the term $K{<}y_2{>}$ refers to the assignment-statement *y[i]=...* in line 2 of Figures 2.9a and 2.9b). The main difference is the type of reduction diKernel that appears in $ES\_for_{j,l}$. AMUXMS contains a regular reduction $K{<}y_6{>}$ that stores values in $y[j]$ (with $j \in \{0 \ldots n-1\}$) and all the irregular accesses affect read-only arrays (see line 6 in Figure 2.9a). In contrast, ATMUX contains an irregular reduction that writes values in $y[ja[l]]$ according to a irregular access pattern (see line 6 in Figure 2.9b). Until now, we have illustrated the detection of diKernel-level flow dependences with scalar variables. However, the diKernel-level dependence between $K{<}y_2{>}$ and $K{<}y_6{>}$ involves a range of values of the non-scalar variable y and, according to Definition 2.1.5, it is marked as flow due to the following reasons. First, y_2 dominates y_6 because both statements belong to two different basic blocks that belong to loops for_i and for_j such that for_i (lines 1–3) precedes for_j (lines 4–8) and, consequently, all the basic blocks of for_i dominate all the basic blocks of for_j. And second, DEF$(y, y_2) \supseteq$ USE(y, y_6) because $K{<}y_2{>}$ produces $y[0{:}n-1{:}1]$ and $K{<}y_6{>}$ uses $y[0{:}n-1{:}1]$, the triplet $F{:}L{:}S$ defining the range of values between the first position F and the last position L with a stride of S positions. As a result, in AMUXMS, DEF(y, y_2) = USE(y, y_6) = $y[0{:}n-1{:}1]$ and thus $K{<}y_2{>} \Rightarrow K{<}y_6{>}$ is a diKernel-level flow dependence. The irregular access $y[ja[l]]$ (line 6 in Figure 2.9b) represents a potential access to any element of array y and the same holds for ATMUX.

The automatic partitioning strategy of Section 2.1.2 proceeds as follows. The critical path is a unique diKernel-level flow dependence $K{<}y_2{>} \Rightarrow K{<}y_6{>}$ with two parallelizable diKernels. Consequently, our technique generates a unique parallel region that covers ES_for_i and $ES\_for_{j,l}$. In AMUXMS, both diKernels rep-

```
#pragma omp parallel shared(A,x,ja,y) private(i,j,l,t)
{
#pragma omp for schedule(static) nowait
  for (i = 0; i < n; i = i + 1) {
    y[i] = (A[i]) * (x[i]);
  }
#pragma omp for schedule(static)
  for (j = 0; j < n; j = j + 1) {
    for (l = ja[j]; l < (ja[j+1] - 1); l = l + 1) {
      y[j] = (y[j]) + ((A[l]) * (x[ja[l]]));
    }
  }
}
```

(a) Parallelized code of routine AMUXMS of Figure 2.9a.

```
#pragma omp parallel shared(A,ia,ja,x,y) private(i,j,l,y___private)
{
  if (omp_get_thread_num() == 0) {
    y___private = y;
  } else {
    y___private = (float *) malloc(n * sizeof(float));
  }
  for (i = 0; i < n; i = i + 1) {
    y___private[i] = 0;
  }
#pragma omp for schedule(static)
  for (j = 0; j < n; j = j + 1) {
    for (l = ia[j]; l < (ia[j+1] - 1); l = l + 1) {
      y___private[ja[l]] = (y___private[ja[l]]) + ((x[j]) * (A[l]));
    }
  }
#pragma omp critical
  if (omp_get_thread_num() != 0) {
    for (i = 0; i < n; i = i + 1) {
      y[i] += y___private[i];
    }
  }
  if (omp_get_thread_num() != 0) {
    free(y___private);
  }
}
```

(b) Parallelized code of routine ATMUX of Figure 2.9b.

Figure 2.10 – Parallelized codes of variations of the sparse matrix-vector multiplication.

resent conflict-free computations, their iteration spaces are equal, and DEF(y, y_2) = USE(y, y_6). As a result, for_i and $for_{j,l}$ are parallelized using the same workload distribution in order to avoid any synchronization between them (see the `nowait` clause in line 3 of Figure 2.10a). Regarding ATMUX, the parallelization of the irregular reduction $K{<}y_6{>}$ is carried out with an array expansion technique. In particular, our compiler reduces the memory consumption of this approach forcing one of the threads to use the original array as its section of the expanded array (see lines 3–7 of Figure 2.10b). Note that the parallelization requires the initialization of the sections of the expanded array y (named $y\_\_\_private$) to the value that reaches the irregular diKernel, 0 in this case. Thus, for_i is not parallelized but replicated in each thread (lines 8–10). Finally, $for_{j,l}$ is parallelized using an OpenMP worksharing loop construct (line 11) and results are consolidated in the original array y (lines 17–22).

2.2.3 Sobel Edge Filter

The *Sobel edge filter* is a well-known algorithm widely used in image processing and computer vision. It detects the edges of an image, that is, those pixels whose intensity is very different from the intensity of the neighboring pixels. Consider the implementation shown in Figure 2.11. For each pixel of the original image (see loop nest in lines 8–9), the program computes a convolution $sumX$ of the 3×3 matrix GX and the intensity of the pixel and its eight neighbors (lines 19–24). A similar convolution $sumY$ with GY is also computed (lines 25–30). Finally, the sum of the absolute values of $sumX$ and $sumY$ is truncated to the interval $[0, 255]$ (lines 34–35) and the resulting SUM is stored in the output $edgeImage\_data$ (lines 37–38). Note that SUM is set to zero for the pixels at image boundaries (see control flow at lines 13–16).

Figure 2.12 shows the KIR of the Sobel benchmark. The convolution loops are represented by two execution scopes $ES\_for_{I,J}$ and $ES\_for_{L,M}$ that contain one scalar reduction diKernel $K{<}sumX_{21}{>}$ and $K{<}sumY_{27}{>}$, respectively. The diKernels of the different execution paths of SUM are $K{<}SUM_{14}{>}$, $K{<}SUM_{16}{>}$, $K{<}SUM_{31}{>}$, $K{<}SUM_{34}{>}$ and $K{<}SUM_{35}{>}$, which represent the different values that can reach $K{<}edgeImage\_data_{37}{>}$. Note that we have selected a pointer-based

```c
void sobel(unsigned char *edgeImage_data, unsigned char *originalImage_data,
    int originalImage_rows, int originalImage_cols) {

  int                 GX[3][3], GY[3][3];
  int                 X, Y, I, J, L, M;
  long                sumX, sumY, SUM;

  for (Y = 0; Y <= (originalImage_rows - 1); Y++)  {
    for (X = 0; X <= (originalImage_cols - 1); X++)  {
      sumX = 0;
      sumY = 0;

      if (Y == 0 || Y == (originalImage_rows - 1))
        SUM = 0;
      else if (X == 0 || X == (originalImage_cols - 1))
        SUM = 0;

      else {
        for (I = -1; I <= 1; I++) {
          for (J = -1; J <= 1; J++) {
            sumX = sumX + (int)( (*(originalImage_data + X + I +
                        (Y + J) * originalImage_cols)) * GX[I+1][J+1]);
          }
        }
        for (L = -1; L <= 1; L++) {
          for (M = -1; M <= 1; M++) {
            sumY = sumY + (int)( (*(originalImage_data + X + L +
                        (Y + M) * originalImage_cols)) * GY[L+1][M+1]);
          }
        }
        SUM = abs(sumX) + abs(sumY);
      }

      if (SUM > 255) SUM = 255;
      if (SUM < 0) SUM = 0;

      *(edgeImage_data + X + Y * originalImage_cols) =
                 255 - (unsigned char)(SUM);
    }
  }
}
```

Figure 2.11 – Source code of the Sobel application.

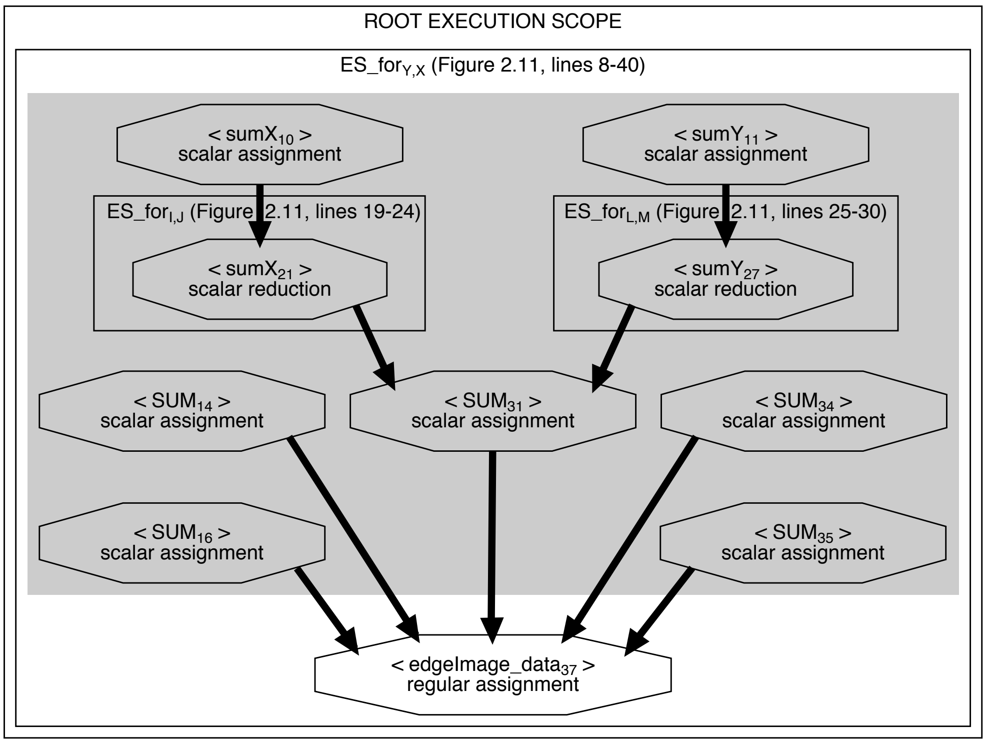

ROOT EXECUTION SCOPE
ES_for$_{Y,X}$ (Figure 2.11, lines 8-40)
< sumX$_{10}$ > scalar assignment
< sumY$_{11}$ > scalar assignment
ES_for$_{I,J}$ (Figure 2.11, lines 19-24)
ES_for$_{L,M}$ (Figure 2.11, lines 25-30)
< sumX$_{21}$ > scalar reduction
< sumY$_{27}$ > scalar reduction
< SUM$_{14}$ > scalar assignment
< SUM$_{31}$ > scalar assignment
< SUM$_{34}$ > scalar assignment
< SUM$_{16}$ > scalar assignment
< SUM$_{35}$ > scalar assignment
< edgeImage_data$_{37}$ > regular assignment

Figure 2.12 – KIR of the Sobel application of Figure 2.11.

implementation of the Sobel edge filter to demonstrate how our framework deals not only with array expressions but also with pointer-based accesses to non-scalar variables through array recovery techniques, as mentioned in Section 2.1.1. For instance, see the pointer dereference at line 37 of Figure 2.11. The pointer *edgeImage_data* remains unchanged in the body of the loop nest $for_{Y,X}$. The index of the outer loop Y ranges from 0 to *originalImage_rows-1* with step 1. The index of the inner loop X ranges from 0 to *originalImage_cols-1* with step 1, and *originalImage_cols* multiplies the index Y in the dereference expression. Thus, the loop nest is traversing the *edgeImage_data* memory region as an array, row-by-row, and the pointer-based expression can be rewritten as *edgeImage_data[Y][X]* for our analysis.

With respect to the parallelization strategy, SUM is a privatizable scalar variable because every use within the loop body is dominated by a definition at the beginning of the loop nest (e.g., uses of SUM at lines 34–38 are dominated by definitions at lines 14, 16, 31, 34 and 35). Scalar variables $sumX$ and $sumY$ are also privatized. By ignoring the shaded subgraph, the critical path consists of a unique diKernel $K{<}edgeImage\_data_{37}{>}$ executed in the scope of $ES\_for_{Y,X}$. As the type of diKernel is a regular assignment, the automatic partitioning strategy succeeds as described for the dense matrix-vector multiplication (see Figure 2.13).

2.2.4 SWIM from SPEC CPU2000

The SWIM application performs a weather prediction based on a numerical model of the shallow-water equations. It consists of an initialization phase and a time integration phase. In each time step, the subroutines `CALC1`, `CALC2` and `CALC3` (`CALC3Z` in the first time iteration) are called. Figure 2.14 shows part of the code of `CALC1` (remaining computations are very similar). Note that we have inlined these subroutines, and that there exist data dependences between iterations (throughout variables not shown in the excerpt) which prevent the parallel execution of different time steps. Thus, the rest of this section focuses on the automatic parallelization of one time step loop iteration (see lines 3–17 of Figure 2.14; contents of ES_for_{NCYCLE} in the KIR of Figure 2.15). Two loops in $ES\_for_{J,I}$ (lines 5–10) compute a subset of the values of matrix Z using as input matrices U, V and P cal-

```
void sobel(unsigned char *edgeImage_data, unsigned char *originalImage_data,
    int originalImage_rows, int originalImage_cols) {

  int                 GX[3][3], GY[3][3];
  int                 X, Y, I, J, L, M;
  long                sumX, sumY, SUM;

#pragma omp parallel shared(edgeImage_data,originalImage_data) \
                     private(Y,X,I,J,L,M,sumX,sumY,SUM)
{
#pragma omp for schedule(static)
  for (Y = 0; Y < originalImage_rows; Y = Y + 1)  {
    for (X = 0; X < originalImage_cols; X = X + 1)  {
      sumX = 0;
      sumY = 0;
      if (Y == 0 || Y == (originalImage_rows - 1))
        SUM = 0;
      else if (X == 0 || X == (originalImage_cols - 1))
        SUM = 0;

      else {
        for (I = -1; I < 2; I = I + 1) {
          for (J = -1; J < 2; J = J + 1) {
            sumX = sumX + (int)( (*(originalImage_data + X + I +
                         (Y + J) * originalImage_cols)) * GX[I+1][J+1]);
          }
        }
        for (L = -1; L < 2; L = L + 1) {
          for (M = -1; M < 2; M = M + 1) {
            sumY = sumY + (int)( (*(originalImage_data + X + L +
                         (Y + M) * originalImage_cols)) * GY[L+1][M+1]);
          }
        }
        SUM = abs(sumX) + abs(sumY);
      }
      if (SUM > 255) SUM = 255;
      if (SUM < 0) SUM = 0;

      *(edgeImage_data + X + Y * originalImage_cols) =
                 255 - (unsigned char)(SUM);
    }
  }
}
}
```

Figure 2.13 – Parallelized code of the Sobel application of Figure 2.11.

```
 1 PROGRAM SHALOW
 2   DO 90 NCYCLE = 1, ITMAX
 3     FSDX = 4.D0 / DX
 4     FSDY = 4.D0 / DY
 5     DO 100 J = 1, N
 6       DO 100 I = 1, M
 7         Z(I + 1, J + 1) = (FSDX * (V(I + 1, J + 1) - V(I, J + 1))
 8           - FSDY * (U(I + 1, J + 1) - U(I + 1, J)))
 9           / (P(I, J) + P(I + 1, J) + P(I + 1, J + 1) + P(I, J + 1))
10     100 CONTINUE
11     DO 110 R = 1, N
12       Z(1, R + 1) = Z(M + 1, R + 1)
13     110 CONTINUE
14     DO 115 S = 1, M
15       Z(S + 1, 1) = Z(S + 1, N + 1)
16     115 CONTINUE
17     Z(1, 1) = Z(M + 1, N + 1)
18 [...]
19 90    CONTINUE
```

Figure 2.14 – Excerpt of the source code of the SWIM application.

culated in previous time steps. Next, the loop in ES_for_R (lines 11–13), the loop in ES_for_S (lines 14–16) and an assignment-statement (line 17) copy values of matrix Z from the last row/column to the first row/column.

The KIR contains a sequence of execution scopes $ES\_for_{J,I}$, ES_for_R and ES_for_S one after another. First, a regular assignment $K{<}Z_7{>}$ represents the conflict-free computations of the first loop $for_{J,I}$ (lines 5–10). Second, the regular recurrence diKernel $K{<}Z_{12}{>}$ captures the copy of Z values to the first row of Z (in a similar manner, $K{<}Z_{15}{>}$ captures the copies to the first column of Z). And third, $K{<}Z_{17}{>}$ also copies the element $Z(M{+}1, N{+}1)$ to $Z(1,1)$. Figure 2.16 illustrates the ranges of defined/used values of Z for each diKernel. Finally, note that the KIR contains a shaded subgraph with $K{<}FSDX_3{>}$ and $K{<}FSDY_4{>}$ that capture the privatizable scalar variables $FSDX$ and $FSDY$.

The most relevant issue of this KIR is that $K{<}Z_7{>}$, $K{<}Z_{12}{>}$, $K{<}Z_{15}{>}$ and $K{<}Z_{17}{>}$ are connected with flow and data diKernel-level dependences. Thus, the automatic partitioning strategy starts by searching spurious diKernel-level dependences. First, $K{<}FSDX_3{>} \Rightarrow K{<}Z_7{>}$ and $K{<}FSDY_4{>} \Rightarrow K{<}Z_7{>}$ are spurious because $K{<}FSDX_3{>}$ and $K{<}FSDY_4{>}$ belong to the shaded subgraph (see

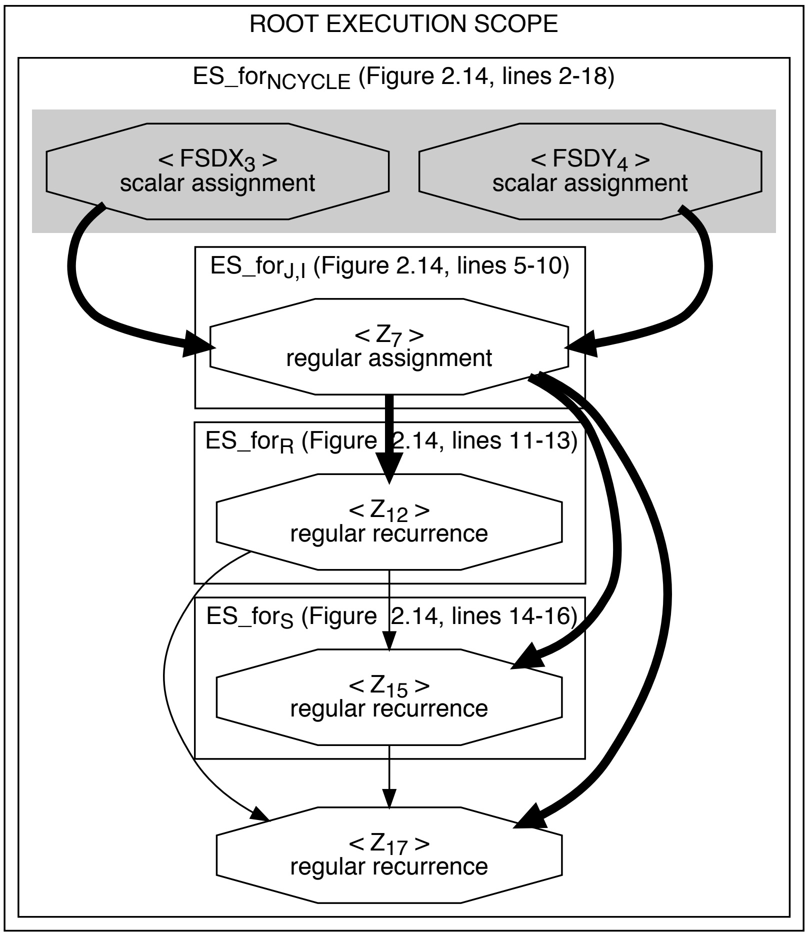
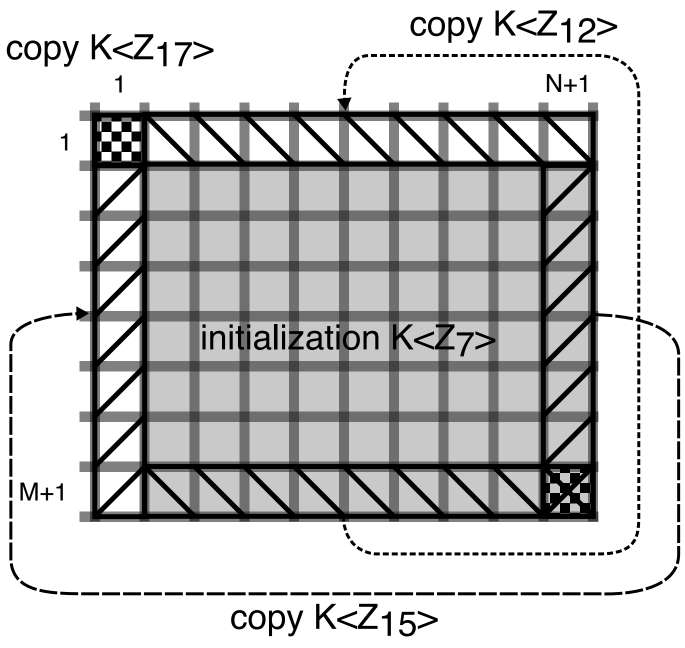

ROOT EXECUTION SCOPE
ES_for$_{NCYCLE}$ (Figure 2.14, lines 2-18)
< FSDX$_3$ > scalar assignment
< FSDY$_4$ > scalar assignment
ES_for$_{J,I}$ (Figure 2.14, lines 5-10)
< Z$_7$ > regular assignment
ES_for$_R$ (Figure 2.14, lines 11-13)
< Z$_{12}$ > regular recurrence
ES_for$_S$ (Figure 2.14, lines 14-16)
< Z$_{15}$ > regular recurrence
< Z$_{17}$ > regular recurrence

Figure 2.15 – KIR of the SWIM application of Figure 2.14.



diKernel	Defined and used data
$K{<}Z_7{>}$	DEF(Z, Z_7)=Z[2:M+1:1][2:N+1:1] USE(Z, Z_7)=Z[::][::]
$K{<}Z_{12}{>}$	DEF(Z, Z_{12})=Z[1:1:1][2:N+1:1] USE(Z, Z_{12})=Z[M+1:M+1:1][2:N+1:1]
$K{<}Z_{15}{>}$	DEF(Z, Z_{15})=Z[2:M+1:1][1:1:1] USE(Z, Z_{15})=Z[2:M+1:1][N+1:N+1:1]
$K{<}Z_{17}{>}$	DEF(Z, Z_{17})=Z[1:1:1][1:1:1] USE(Z, Z_{17})=Z[M+1:M+1:1][N+1:N+1:1]

Figure 2.16 – Data analysis of the source code (see Figure 2.14) and KIR (see Figure 2.15) of the SWIM application.

Definition 2.1.9, case 1). In addition, $K{<}Z_{12}{>} \rightarrow K{<}Z_{15}{>}$ is spurious because the statement Z_{12} dominates Z_{15} but Z_{12} modifies a set of Z entries (DEF(Z, Z_{12}) = $Z[1{:}1{:}1][2{:}N+1{:}1]$) that is not used in Z_{15} (USE(Z, Z_{15}) = $Z[2{:}M+1{:}1][N+1{:}N+1{:}1]$) and thus DEF($Z$, Z_{12}) $\cap$ USE(Z, Z_{15}) $= \emptyset$ (see Definition 2.1.9, case 2). In a similar manner, $K{<}Z_{12}{>} \rightarrow K{<}Z_{17}{>}$ and $K{<}Z_{15}{>} \rightarrow K{<}Z_{17}{>}$ are also spurious diKernel-level dependences. The rest of the KIR contains three critical paths $K{<}Z_7{>} \Rightarrow K{<}Z_{12}{>}$, $K{<}Z_7{>} \Rightarrow K{<}Z_{15}{>}$, and $K{<}Z_7{>} \Rightarrow K{<}Z_{17}{>}$ that share the computations of $K{<}Z_7{>}$. Thus, a parallel region that encloses the three critical paths is created and the computations of $K{<}Z_{12}{>}$, $K{<}Z_{15}{>}$ and $K{<}Z_{17}{>}$ are executed in sequence after a barrier to avoid diKernel-level flow dependences violation (note that the `!$OMP END DO` directive in line 15 of Figure 2.17 does not have a `NOWAIT` clause). $K{<}Z_7{>}$ is a regular assignment and is transformed into a forall loop (line 8). $K{<}Z_{12}{>}$ is a regular recurrence but, as DEF(Z, Z_{12}) $\cap$ USE(Z, Z_{12})$= \emptyset$, it can be transformed into a forall loop (line 16). The same is true for $K{<}Z_{15}{>}$ (line 21). $K{<}Z_{17}{>}$ is a diKernel with a single statement and is executed into an OpenMP `single` region (line 26). Due to their similarities, the same analysis of `CALC1` is applied to `CALC2`, `CALC3` and `CALC3Z`. Finally, the location of the OpenMP `parallel` directive is optimized: they are moved to enclose for_{NCYCLE} (see lines 3 and 31 of Figure 2.17) and a barrier (line 5) synchronizes the execution of each iteration. In this manner, creation and destruction of threads is minimized.

2.2.5 EQUAKE from SPEC CPU2000

The *EQUAKE* application simulates seismic waves in large, highly heterogeneous valleys. EQUAKE is able to recover the time history of the ground motion caused by a seismic event in any place of a valley. An unstructured mesh is used to locally resolve wavelengths with a finite element method. As a result, EQUAKE reports the displacements at both the hypocenter and epicenter of the earthquake for a predetermined number of simulation time steps. The most time-consuming part of EQUAKE is a time integration loop that computes this displacement. Similarly to SWIM, there are dependences between consecutive time iterations. Thus, the rest of this section focuses on one time step loop iteration.

```
PROGRAM SHALOW
  USE OMP_LIB
!$OMP PARALLEL PRIVATE(FSDX, FSDY)
  DO 90 NCYCLE = 1, ITMAX
!$OMP BARRIER
    FSDX = 4.D0 / DX
    FSDY = 4.D0 / DY
!$OMP DO
    DO 100 J = 1, N
      DO 100 I = 1, M
        Z(I + 1, J + 1) = (FSDX * (V(I + 1, J + 1) - V(I, J + 1))
          - FSDY * (U(I + 1, J + 1) - U(I + 1, J)))
          / (P(I, J) + P(I + 1, J) + P(I + 1, J + 1) + P(I, J + 1))
    100 CONTINUE
!$OMP END DO
!$OMP DO
    DO 110 R = 1, N
      Z(1, R + 1) = Z(M + 1, R + 1)
    110 CONTINUE
!$OMP END DO NOWAIT
!$OMP DO
    DO 115 S = 1, M
      Z(S + 1, 1) = Z(S + 1, N + 1)
    115 CONTINUE
!$OMP END DO NOWAIT
!$OMP SINGLE
    Z(1, 1) = Z(M + 1, N + 1)
!$OMP END SINGLE
[...]
90    CONTINUE
!$OMP END PARALLEL
```

Figure 2.17 – Excerpt of the parallelized code of the SWIM application of Figure 2.14.

```
for (iter = 1; iter <= timesteps; iter++) {
  for (i = 0; i < ARCHnodes; i++)
    for (j = 0; j < 3; j++)
      disp[disptplus][i][j] = 0.0;
  for (i = 0; i < ARCHnodes; i++) {
    Anext = ARCHmatrixindex[i]; Alast = ARCHmatrixindex[i+1];
    sum0 = K[Anext][0][0] * disp[dispt][i][0]
      + K[Anext][0][1] * disp[dispt][i][1]
      + K[Anext][0][2] * disp[dispt][i][2];
    sum1 = K[Anext][1][0] * ...; sum2 = K[Anext][2][0] * ...;
    Anext++;
    while (Anext < Alast) {
      col = ARCHmatrixcol[Anext];
      sum0 += K[Anext][0][0] * disp[dispt][col][0]
        + K[Anext][0][1] * disp[dispt][col][1]
        + K[Anext][0][2] * disp[dispt][col][2];
      sum1 += K[Anext][1][0]*...; sum2 += K[Anext][2][0]*...;
      disp[disptplus][col][0] +=
        K[Anext][0][0] * disp[dispt][i][0]
        + K[Anext][1][0] * disp[dispt][i][1]
        + K[Anext][2][0] * disp[dispt][i][2];
      disp[disptplus][col][1] += K[Anext][0][1] ...
      disp[disptplus][col][2] += K[Anext][0][2] ...
      Anext++;
    }
    disp[disptplus][i][0] += sum0; ...
  }
  time = iter * Exc.dt;
  for (i = 0; i < ARCHnodes; i++)
    for (j = 0; j < 3; j++)
      disp[disptplus][i][j] *= - Exc.dt * Exc.dt;
  for (i = 0; i < ARCHnodes; i++)
    for (j = 0; j < 3; j++)
      disp[disptplus][i][j] +=
        2.0 * M[i][j] * disp[dispt][i][j]
        - (M[i][j] - Exc.dt / 2.0 * C[i][j])
        * disp[disptminus][i][j] - ...
  for (i = 0; i < ARCHnodes; i++)
    for (j = 0; j < 3; j++)
      disp[disptplus][i][j] /= (M[i][j] + Exc.dt / 2.0 * C[i][j]);
  for (i = 0; i < ARCHnodes; i++)
    for (j = 0; j < 3; j++)
      vel[i][j] = 0.5 / Exc.dt * (disp[disptplus][i][j]
        - disp[disptminus][i][j]);
  i = disptminus; disptminus = dispt; dispt = disptplus; disptplus = i;
}
```

Figure 2.18 – Excerpt of the source code of the EQUAKE application.

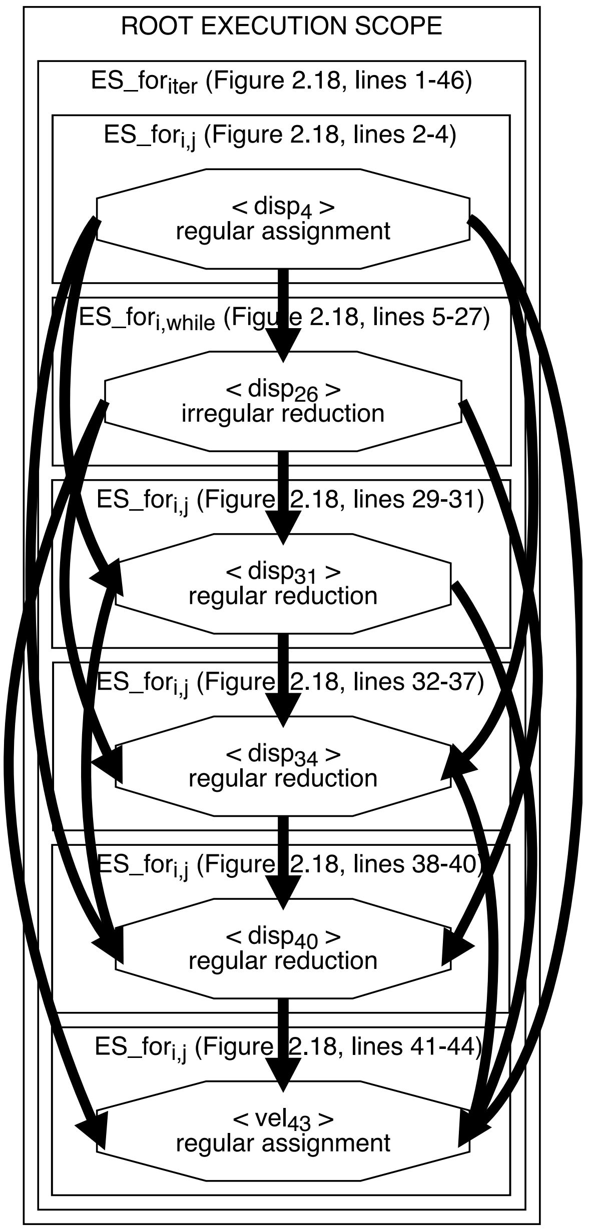

ROOT EXECUTION SCOPE
ES_for$_{iter}$ (Figure 2.18, lines 1-46)
ES_for$_{i,j}$ (Figure 2.18, lines 2-4)
< disp$_4$ > regular assignment
ES_for$_{i,while}$ (Figure 2.18, lines 5-27)
< disp$_{26}$ > irregular reduction
ES_for$_{i,j}$ (Figure 2.18, lines 29-31)
< disp$_{31}$ > regular reduction
ES_for$_{i,j}$ (Figure 2.18, lines 32-37)
< disp$_{34}$ > regular reduction
ES_for$_{i,j}$ (Figure 2.18, lines 38-40)
< disp$_{40}$ > regular reduction
ES_for$_{i,j}$ (Figure 2.18, lines 41-44)
< vel$_{43}$ > regular assignment

Figure 2.19 – KIR of the EQUAKE application of Figure 2.18.

An excerpt of the source code of EQUAKE is shown in Figure 2.18. Note that we have inlined the `smvp()` routine (see lines 5–27). For simplicity of the figure, the KIR of Figure 2.19 does not show the diKernels of privatizable scalar variables. In each time step, the sequence of diKernels $K{<}disp_4{>}$ to $K{<}disp_{40}{>}$ computes the displacement and, after it, diKernel $K{<}vel_{43}{>}$ computes the velocity. The analysis of one iteration of the time integration loop shows that the indices *disptminus*, *dispt* and *disptplus* are constant in each time iteration and they reference disjoint submatrices of the array *disp* (representing the displacement in the current and the two previous time steps). As a result, we can consider these submatrices as totally independent matrices and thus diKernels $K{<}disp_{26}{>}$ and $K{<}disp_{34}{>}$ are reductions and not recurrences.

The automatic partitioning procedure marks as spurious all diKernel-level flow dependences except the sequence of the six parallelizable diKernels $K{<}disp_4{>} \Rightarrow K{<}disp_{26}{>} \Rightarrow K{<}disp_{31}{>} \Rightarrow K{<}disp_{34}{>} \Rightarrow K{<}disp_{40}{>} \Rightarrow K{<}vel_{43}{>}$ because all these diKernels compute and use the whole *disp[disptplus]* matrix (see Definition 2.1.9, case 3). As a result, this is the critical path. The irregular reduction of $K{<}disp_{26}{>}$ is parallelized applying array expansion. As illustrated with ATMUX (see the last paragraph of Section 2.2.2), this technique requires the allocation and initialization of the private arrays, hence $K{<}disp_4{>}$ is replicated in each thread (see lines 3–14 of Figure 2.20). The remaining diKernels are transformed into forall loops. A barrier is inserted after the consolidation of the private arrays in the original matrix (lines 28–33). In contrast, the series of diKernels $K{<}disp_{31}{>} \Rightarrow K{<}disp_{34}{>} \Rightarrow K{<}disp_{40}{>} \Rightarrow K{<}vel_{43}{>}$ can use the same OpenMP schedule clause and be parallelized without intermediate barriers (see `nowait` clauses in lines 35, 39, 43 and 47). The creation of the parallel region is optimized enclosing the whole time integration loop as explained with SWIM (see Section 2.2.4, lines 1 and 11 in Figure 2.20). This example shows the biggest potential of our approach: the loops of the application are not analyzed in isolation, which enables the generation of more efficient parallel code with the creation of a unique parallel region.

```
#pragma omp parallel shared(disp) private(disp___disptplus___private,...)
{
  if (omp_get_thread_num() == 0) {
    disp___disptplus___private = disp[disptplus];
  } else {
    disp___disptplus___private = (double **) malloc (ARCHnodes * sizeof(double *));
    for (i = 0; i < ARCHnodes; i = i + 1)
      disp___disptplus___private[i] = (double *) malloc(3 * sizeof(double));
  }
  for (iter = 1; iter < (timesteps + 1); iter = iter + 1) {
#pragma omp barrier
    for (i = 0; i < ARCHnodes; i = i + 1)
      for (j = 0; j < 3; j = j + 1)
        disp___disptplus___private[i][j] = 0.0;
#pragma omp for schedule(static)
    for (i = 0; i < ARCHnodes; i = i + 1) {
      Anext = ARCHmatrixindex[i]; Alast = ARCHmatrixindex[i+1];
      sum0 = K[Anext][0][0] * ...
      Anext++;
      while (Anext < Alast) {
        col = ARCHmatrixcol[Anext];
        sum0 += K[Anext][0][0] * ...
        disp___disptplus___private[col][0] += K[Anext][0][0] * ...
        Anext++;
      }
      disp___disptplus___private[i][0] += sum0; ...
    }
#pragma omp critical
    if (omp_get_thread_num() != 0)
      for (i = 0; i < ARCHnodes; i = i + 1)
        for (j = 0; j < 3; j = j + 1)
          disp[disptplus][i][j] += disp___disptplus___private[i][j];
#pragma omp barrier
    time = iter * Exc.dt;
#pragma omp for schedule(static) nowait
    for (i = 0; i < ARCHnodes; i = i + 1)
      for (j = 0; j < 3; j = j + 1)
        disp[disptplus][i][j] *= - Exc.dt * Exc.dt;
#pragma omp for schedule(static) nowait
    for (i = 0; i < ARCHnodes; i = i + 1)
      for (j = 0; j < 3; j = j + 1)
        disp[disptplus][i][j] += ...
#pragma omp for schedule(static) nowait
    for (i = 0; i < ARCHnodes; i = i + 1)
      for (j = 0; j < 3; j = j + 1)
        disp[disptplus][i][j] /= ...
#pragma omp for schedule(static) nowait
    for (i = 0; i < ARCHnodes; i = i + 1)
      for (j = 0; j < 3; j = j + 1)
        vel[i][j] = ...
    i = disptminus; disptminus = dispt; dispt = disptplus; disptplus = i;
  } /* for iter */
  if (omp_get_thread_num() != 0) {
    for (i = 0; i < ARCHnodes; i = i + 1)
      free(disp___disptplus___private[i]);
    free(disp___disptplus___private);
  }
}
```

Figure 2.20 – Excerpt of the parallelized code of the EQUAKE application.

2.3 Evaluation

As shown in the previous sections, our diKernel-based automatic parallelization strategy is effective to handle full-scale applications with arrays, pointers and complex control flows. In this section, it is compared with current parallelizing compilers using the benchmark suite presented in Section 2.2. Hereafter, Section 2.3.1 describes the experimental platform, Section 2.3.2 discusses the experimental results in terms of effectiveness and Section 2.3.3 shows the performance of the generated OpenMP-enabled parallel code in terms of execution times and speedups.

2.3.1 Experimental Platform

The target multicore system consists of 2 Intel Xeon E5520 quad-core Nehalem processors at 2.26 GHz with 8 MB of cache memory per processor and 8 GB of RAM.

Three compilers have been selected to be compared with our proposal. The first one is the GNU Compiler Collection [128] (from now on, *GCC*) version 4.5.2. It supports automatic parallelization generating OpenMP code by means of the Graphite framework [133], based on a polyhedral representation. The compilation options are `-march=core2 -msse4 -O2 -floop-parallelize-all -ftree-parallelize-loops=8`. The second one is the Intel C++/Fortran Compiler [70] (from now on, *ICC*) version 11.1 for the *intel64* architecture, which also supports automatic parallelization. The compilation flags are `-O2 -xSSE4.2 -parallel`. The third one is the *PLUTO* automatic parallelization research tool [27] version 0.6.0. It uses the polyhedral model to transform C programs into OpenMP code supporting efficient tiling and fusion. The compilation flags are `--tile --parallel`. Finally, our KIR-driven automatic partitioning approach (from now on, *KIR*) generates OpenMP source code and is built on top of GCC version 4.4.0.

Table 2.1 – Effectiveness of GCC, ICC, PLUTO and KIR for the benchmark suite.

		Program Characteristics						**Compilers**			
Benchmark		diKernel	Irreg. writes	Irreg. reads	Unknown LB	Complex CF	Temp. vars	GCC	ICC	PLUTO	KIR
Synthetic	reg. assig.	regular assignment						√	√	√	√
	irreg. assig.	irregular assignment	√		√						√
	sc. reduc. 1	scalar reduction						≈	√		√
	sc. reduc. 2	scalar reduction						≈	√		√
	sc. reduc. 3	scalar reduction				√		≈	√		√
	reg. reduc.	regular reduction						√	√	√	√
	irreg. reduc.	irregular reduction	√	√	√						√
	reg. recurr.	regular recurrence									√
Algebra	DenseAMUX	regular assignment					√		√	≈	√
	AMUX	regular assignment		√	√		√				√
	AMUXMS	regular reduction		√	√						√
	ATMUX	irregular reduction	√	√	√						√
Im.	sobel1	regular assignment				√	√				√
	sobel2	regular assignment					√	√			√
Apps	SWIM	regular recurrence				√			√	*U*	√
	EQUAKE	irregular reduction	√	√	√				≈		√

2.3.2 Experimental Results: Effectiveness

Table 2.1 shows a summary of the effectiveness of GCC, ICC, PLUTO and KIR for the benchmarks described in Section 2.2. The table summarizes for each benchmark some program characteristics that impact on the effectiveness of the compilers: type of the most representative diKernel (*diKernel*), existence of irregular computations in writes (*Irreg. writes*) and reads (*Irreg. reads*), existence of compile-time unknowns in loop bounds (*Unknown LB*), complex control flows (*Complex CF*) and the use of temporary variables to store intermediate results (*Temp. vars*). The effectiveness of each compiler for automatic parallelization is measured in terms of success ($\surd$), partial success ($\approx$), failure (blank table entries), or unsupported input programming language (*U*).

The synthetic benchmarks have been designed to expose different types of diKernels of increasing complexity. The *regular assignment* benchmark (see Figure 2.7a) is successfully parallelized by all the compilers because it is a simple array-based implementation with affine accesses. The same holds for the *regular reduction* benchmark (Figure 2.7f). The introduction of an indirection array f that selects the locations to be updated in the *irregular assignment* (Figure 2.7b) causes the failure of GCC, ICC and PLUTO. GCC considers the data reference as not analyzable and ICC fails because it assumes output dependences. In contrast, KIR successfully handles diKernels with irregular computations in write operations. Note that the parallelization of the *irregular reduction* (Figure 2.7g) is also unsuccessful for GCC, ICC and PLUTO. Regarding the *regular recurrence* (Figure 2.7h), KIR detects that it is a parallel prefix sum and generates parallel code, while none of the contenders succeeds in parallelizing the benchmark. Finally, the analysis of the three scalar reductions (Figures 2.7c, 2.7d and 2.7e) provides more details about the behavior of the compilers: ICC and KIR parallelize the three implementations, GCC recognizes the scalar reductions, but no parallel code is generated, and PLUTO fails.

The linear algebra routines are variations of dense/sparse matrix-vector products. The *DenseAMUX* benchmark (see Figure 2.8a) is successfully handled by ICC: the outer loop is parallelized. PLUTO has partial success because it is very sensitive to syntactic variations in the source code. It does not parallelize

DenseAMUX due to the use of the temporary variable t to store the dot product of a matrix row and the vector (see lines 3–5 in Figure 2.8a). If the code is rewritten without t, then PLUTO parallelizes the benchmark. GCC is not able to parallelize the outer loop.

As mentioned in Section 2.2.2, the sparse *AMUX* is very similar to *DenseAMUX* from the point of view of the KIR. However, ICC fails to parallelize, indicating that the loop structure is unsupported because the loop index variable requires complex computation (that is, there exist unknown loop bounds and irregular reads). GCC and PLUTO again fail to parallelize. Regarding *AMUXMS*, it consists of two separated loops. The first loop (see lines 1–3 in Figure 2.9a) is an example of regular assignment, and the compilers succeed with this type of diKernel, as mentioned above. The second loop (see lines 4–8 in Figure 2.9a) consists of a regular reduction that cannot be parallelized by the compilers, again due to the presence of irregular reads and unknown loop bounds. Finally, GCC, ICC and PLUTO also fail in *ATMUX* (see line 6 in Figure 2.9b) which includes an irregular reduction.

The benchmark *sobel1* is an implementation of the Sobel edge filter that contains a complex control flow for processing the pixels at the image boundaries (see lines 13–16 in Figure 2.11). GCC, ICC and PLUTO cannot parallelize *sobel1*. In contrast, the *sobel2* version removes the complex control flow by processing image boundaries in two separated loops. In this case, GCC parallelizes the application by unrolling the convolution loops (see lines 19–24 and 25–30); ICC and PLUTO fail because the convolution loops have syntactically complex accesses. Rewriting the Sobel application using arrays instead of using pointers provides the same results. The Sobel benchmarks exhibit one of the main weaknesses of state-of-the-art compilers, that is, they are strongly dependent on implementation details such as access expressions, complex loop bounds or complex control flows.

SWIM is a full-scale application with regular computations only. Nevertheless, GCC is unable to parallelize any piece of code out of the initialization subroutine. Focusing on `CALC1`, ICC only parallelizes the first loop (see lines 5–10 in Figure 2.14), discarding the recurrences because they are considered not to have enough workload. SWIM is written in Fortran, thus PLUTO cannot handle it.

The last benchmark is the *EQUAKE* application. GCC only parallelizes some regular computations carried out in the auxiliary functions. In contrast, ICC is able to parallelize the computations of the regular reduction $K\textless disp_{31}\textgreater$ (see Figure 2.19). Note that both GCC and ICC execute the irregular reduction $K\textless disp_{26}\textgreater$ sequentially. PLUTO cannot handle this benchmark.

Overall, we have demonstrated that GCC, ICC and PLUTO are, in general, effective compilers in parallelizing regular computations and scalar reductions, specially in synthetic benchmarks and routines from libraries. In contrast, these approaches have shown to be ineffective with irregular computations and full-scale applications. KIR overcomes these limitations handling a comprehensive set of codes in a unified manner.

2.3.3 Experimental Results: Performance

This section presents a comparison in terms of performance. As representative example we have selected EQUAKE, a full-scale application that combines regular and irregular computations. The OpenMP-enabled parallel code generated by our KIR-driven automatic partitioning approach has been compiled with the Intel C++/Fortran Compiler (*KIR/ICC*) with the flags `-O2 -xSSE4.2 -openmp`, due to the fact that the contender is the same *ICC* compiler with the automatic parallelization support enabled.

As mentioned in Section 2.2.5, the irregular reduction $K\textless disp_{26}\textgreater$ (see Figure 2.19) is parallelized with an array expansion technique (see lines 3–33 of Fig-

Table 2.2 – Memory consumption in the parallelization with array expansion of $K\textless disp_{26}\textgreater$ of EQUAKE.

#Threads	**Pure array-expansion**		**Optimized array-expansion**		
	Elements	KB	Elements	KB	Reduction (%)
2	181014	1414	83490	652	-54 %
4	362028	2828	253872	1983	-30 %
8	724056	5657	557031	4352	-23 %

```
#pragma omp parallel shared(disp) private(disp___disptplus___private,...)
{
  if (omp_get_thread_num() == 0) {
    range = (struct it_info *)
      malloc(omp_get_num_threads() * sizeof(struct it_info));
  }
#pragma omp barrier
  id = omp_get_thread_num();
  range[id].max = 0;
#pragma omp for
  for (i = 0; i < ARCHnodes; i = i + 1) {
    Anext = ARCHmatrixindex[i]; Alast = ARCHmatrixindex[i+1]; Anext++;
    if (i > range[id].max)
      range[id].max = i;
    while (Anext < Alast) {
      col = ARCHmatrixcol[Anext];
      if (col > range[id].max)
        range[id].max = col;
      Anext++;
    }
  }
  ...
}
```

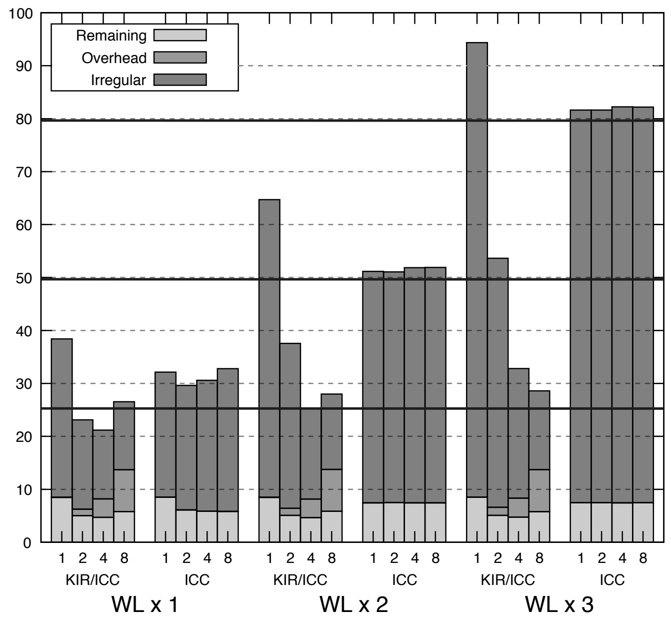
Figure 2.21 – Excerpt of the inspector code for the EQUAKE application of Figure 2.18.

ure 2.20). Its main drawback is that memory consumption may be high because it grows proportionally to the number of threads: the number of elements of the expanded array is $ARCHnodes \times 3 \times \#threads$ (with $ARCHnodes = 30169$, the number of nodes of the unstructured grid topology that represents the valley where the simulation is performed). As explained in Section 2.2.2, in order to reduce this overhead, our automatic approach fine-tunes the OpenMP parallel code forcing one of the threads to use the original array as its section of the expanded array.

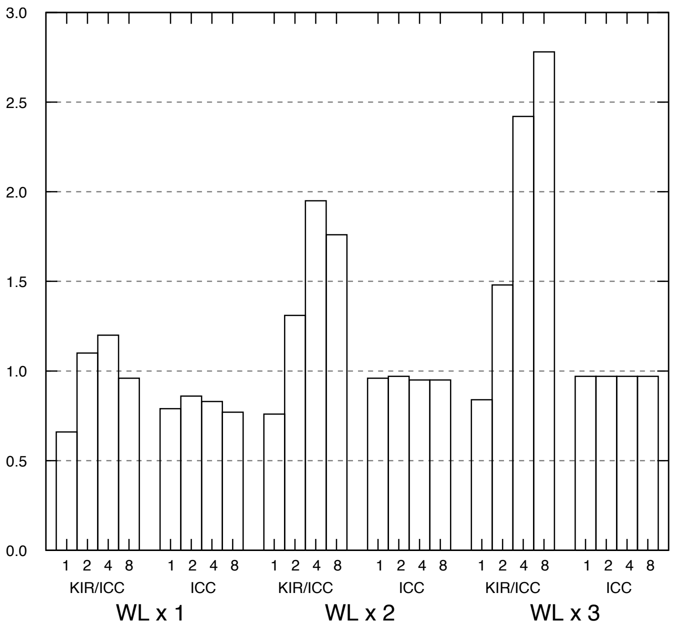
In addition, the user of our compiler can order the introduction of an inspector [59] before the time integration loop to determine the highest index of *disp* that is referenced by each thread. Figure 2.21 shows an excerpt of the inspector code, which only needs to preserve the iterators of the irregular reduction being inspected (see lines 5–27 of Figure 2.18 and lines 11–21 of Figure 2.21). In this manner, our technique allocates less memory in each section of the expanded array. Table 2.2 shows a comparison between the memory needed by a pure array-expansion implementation and the optimized code generated by our approach.

Figure 2.22 shows the execution times and speedups of EQUAKE for a number of threads ranging from 1 to 8. Note that the total execution time is decomposed showing the time of the irregular reduction $K{<}disp_{26}{>}$ (*Irregular*); the overhead of the OpenMP parallelization of KIR (*Overhead*) due to the inspector, the allocation and initialization of the auxiliary arrays, and thread synchronization; and the remaining time (*Remaining*). The label $WL \times 1$ shows the results for the workload *ref* from SPEC CPU2000. The execution time of the sequential application has been taken as baseline (see horizontal line at 24.47 seconds) to calculate the speedups. Note that the *KIR/ICC* execution time using one thread is increased due to the different optimizations applied by the Intel compiler into an OpenMP region. As can be observed, the KIR-driven approach outperforms ICC with 2, 4 and 8 threads by up to a 30 %. However, the increase in the number of threads does not have a significant impact on performance due to the low workload of the application.

Performance results are also shown for higher workloads: $WL \times 2$ and $WL \times 3$ (twice and thrice the workload $WL \times 1$). The sequential execution times are



(a) Execution times (in seconds).



(b) Speedups.

Figure 2.22 – Execution times and speedups of EQUAKE.

shown in the graph at 49.32 s and 79.49 s, respectively. Note that ICC cannot reduce the execution time of the benchmark because it does not parallelize the irregular computations, which is the most costly part of the EQUAKE application. In contrast, KIR is able to reduce the sequential execution time by up to a 64 % (for $WL \times 3$ and 8 threads), revealing that addressing irregular computations is paramount for parallelizing common applications.

2.4 Related Work

Many approaches have been explored to address the parallel challenge targeting current multicore processors. The automatic rewriting of sequential programs into a parallel counterpart is the ideal solution, but it remains an open research subject due to the complexity of dealing with full-scale applications. Several IRs have been proposed in the literature to support automatic parallelization. Typically, the IR consists of forests of ASTs+DDG+CFG and it is analyzed with statement-based dependence analysis techniques. This approach is used in modern compilers, although it is unsuccessful in handling syntactical variations in the source code. Next, we discuss alternative IRs for the automatic parallelization proposed in the literature.

The polyhedral model [54] is a mature technology that has reached production (e.g. GCC [133], LLVM (Polly) [58] and IBM XL [26]) and research compilers (e.g. PLUTO [27]). It is a mathematical framework for loop nest parallelization and optimization. Its main drawback is that its scope of application is limited only to static-control, regular loop nests (see Section 2.3.2). Benabderrahmane et al. [23] removed these limitations addressing general `while` loops and `if` conditions, although irregular data accesses were modeled conservatively (e.g., an array with a irregular access expression is considered as a single variable). A recent extension [116] is able to model irregular accesses with more precision thanks to inspector/executor techniques, preserving the regular code regions to enable composability with other polyhedral optimizations. This proposal is restricted to non-nested loops, which must be annotated by the user, with inter-iteration dependences due to reduction operations only.

Sato and Iwasaki [121] address the parallelization of complex reductions and scans. They transform the loop body into a matrix-multiplication form based on reduce and scan parallel primitives. In addition, their technique extracts max-operators from `if` statements automatically, enabling the parallelization of loops with complex control flows. However, this method does not address loop bodies with pointers or irregular accesses.

The TRACO paralleling compiler [108] generates OpenMP C code from a sequential C program. It targets affine loop nests by building the transitive closure of a relation which describes all the dependences of the loop. It extracts coarse-grain and fine-grain parallelism, supports variable privatization and the detection of parallel reductions.

The Locality-Parallelism Graph (LPG) [144] is a directed acyclic graph (DAG) designed to capture parallelism and data-locality at the same time. Nodes are *code blocks*, i.e., a subset of the iterations of a loop. It has two types of edges: one type for representing true data dependences, and another type for indicating data reuse (weighted with the amount of shared data). A heuristic algorithm tries to schedule code blocks that have data reuse between them as close as possible (in time), while respecting data dependences, to increase cache hit ratio. Authors also formulated the scheduling as an ILP problem, whose solutions were worse than the heuristic algorithm. This technique is able to consider considers all the loop nests and the access patterns of the application as a whole, but it is limited to codes with affine array accesses and affine loop bounds.

Liu et al. [90] target iteration-level parallelism as a graph optimization problem. They build the DDG for each loop, and annotate edges with weight zero to indicate an intra-iteration dependence or with the distance between inter-iteration dependences. In order to maximize the number of iterations that can be run in parallel, they apply *retiming* to the weighted DDG. This technique, also know as *the index shift method* when applied to the parallelization of nested loops [92], consists of deferring or advancing the execution steps of some statements in such a way that we can increase parallelism. A new loop is generated from the optimized graph. This technique has not been integrated into a compiler to be evaluated on a multicore processor.

Decoupled Software Pipelining (DSWP) [106] proposes to divide a loop into critical path and off-critical path threads that run concurrently but communicate in a pipelined manner. First, this technique builds the Program Dependence Graph (PDG) of a loop and searches for strongly connected components (SCCs) on it. As a result, a DAG of SCCs is generated. Next, this DAG is partitioned into threads (1) maintaining all instructions of an SCC in the same thread and (2) balancing the estimated cycles necessary to execute each thread. Edges of the DAG that cross partition boundaries represent data values and control conditions that are communicated through produce/consume operations over a queue. Using this approach, the performance improvement is limited by the number and size of the SCCs. The method has been extended by Huang et al. [66] introducing DSWP+. Instead of balancing the computational load between the pipelined threads, DSPW+ puts as much work as possible in the stages that can be subsequently parallelized with other techniques (e.g., forall, speculation, localwrite). This process has been automatized in the Nova compiler of the Parcae system by Raman et al. [115]. However, it targets only the hottest outermost loop nest and not the whole application.

Helix [30] schedules the iterations of the targeted loop in a round-robin fashion, using signals to preserve dependences across loop boundaries. The selection of the loops that must be parallelized is done by building a nesting graph for the whole program, in which each loop is represented by a node and directed edges connect the immediately enclosed ones. These nodes are labeled with the value of a heuristic based on Amdahl's law and profile information. The efficiency of the approach is based on the exploitation of the capabilities of the Intel SMT technology, but a follow-up work [29] demonstrated that it does not scale to more than four cores. The authors proposed a hardware extension to overcome this limitation.

The Paralax Infrastructure [135] proposes a combined approach for automatic extraction of parallelism in irregular codes. First, full-data structure SSA and use/def chains are used to compute the SCCs on the PDG of a loop and extract pipeline parallelism. A static performance model predicts the speedup of the parallelization and only loops with significant speedups are parallelized. Second, a light-weight programming model based on annotations helps the compiler to find thread-level parallelism. In this manner, they overcome the problem of determining the last definition of arrays. These annotations must be inserted by the programmer, although a tool based on profiling has been developed to suggest them. Our approach also addresses irregular computations, but it automatically computes the range of values produced/used throughout the execution of statements when detects diKernel-level flow dependences to overcome this problem. Moreover, OpenMP parallel code is generated instead of proposing a new programming environment that must be learnt by the user.

Canedo et al. [31] present a fully automatic parallelization approach based on a new IR called Concurrent PDG. This new IR models the whole application and has been implemented in a compiler. Nevertheless, it is only applicable to Simulink, a model-based design engineering tool that uses block diagram notation to describe mathematical models of dynamic systems and controllers. Our approach targets general-purpose programming languages.

Tournavitis and Franke [132] propose IR Profiling, a hierarchical whole program representation focused on the extraction of pipeline parallelism. From the original sequential program, an instrumented executable is generated. The application is then executed with several input files, generating a set of trace files. The new IR, based on the PDG, is built upon these traces. Finally, a heuristic-guided partitioning algorithm produces the specification of the pipeline stages and parallel code is generated accordingly. The main drawback of this approach is that the dependences to build IR Profiling depend on the employed input files. The IR will be only correct for these concrete inputs, not for general ones. As the authors expect, the user must perform the final verification of the suggested partitioning scheme. In Chapter 4, we will present a method for trace-based affine reconstruction of code.

DiscoPoP [88] is a tool for the discovering of potential parallelism in sequential programs. Instead of looking for dependences that prevent parallelization, this proposal searches for their absence. The program to be analyzed is instrumented and executed to identify Computational Units (CUs): sets of statements with no true dependences between them. Temporary intermediate variables are excluded thanks to use/def chains. Hence, DiscoPoP builds a CU graph whose nodes are the CUs and whose edges are true dependences. This CU graph is mapped onto the Program Execution Tree, which represents the executed code as a tree: its root is a special node representing the whole program, and internal nodes represent function calls and loops. The user is responsible to generate the parallel code taking into account that CUs are the smallest piece of code to be scheduled in a thread (i.e, CUs must be executed sequentially), and synchronization is needed when different paths of the CU graph merge. Recently, DiscoPoP has added a pattern-matching technique to help the programmer to create the parallel version [67], and the automatic generation of parallel code using Intel Threading Building Blocks (Intel TBB) [146].

Sambamba [123] builds the PDG of each function of the input program and then groups those nodes which share the same control condition. Next, a scheduler based on Integer Linear Programming (ILP) finds parallel candidates for these group nodes. Instead of filtering dependences due to reductions and variable privatization before scheduling, Sambamba lets the ILP technique break them. A runtime system adapts the execution by running either the sequential version or one of the parallel ones in each function call. This approach is built on top of LLVM. Authors mention that the Data Structure Analysis used in this compiler prevents some codes from being parallelized due to the flow-insensitiveness and the unification-based approach needed by this technique for the sake of scalability.

Overall, most of the techniques presented in the literature are partial approaches to automatic parallelization or they model simple loops individually. In contrast, our approach models sequential applications as a whole. In this way, KIR is able to generate a comprehensive parallelization strategy that minimizes the parallel overhead. In addition, our technique handles regular and irregular computations in a uniform manner, and addresses general-purpose languages. Finally, note that KIR is complementary to other techniques. For instance, the polyhedral model is strong in optimizing regular recurrence diKernels and may be used in conjunction with our approach.

2.5 Concluding Remarks

This chapter has presented a new effective and efficient method to parallelize sequential applications automatically. It is based on the concept of domain-independent kernel to handle syntactical variations in the source code.

The first contribution is a new compiler intermediate representation called KIR. It is built on top of diKernels, which are connected with diKernel-level dependences and are grouped in execution scopes to recognize the stages of the input sequential application.

The second contribution is a new KIR-based automatic partitioning technique that builds a global OpenMP-enabled parallelization strategy targeting current multicore processors. The potential of our approach has been illustrated using a comprehensive benchmark suite that includes synthetic codes representative of frequently used diKernels, routines from dense/sparse linear algebra and image processing, and full-scale applications.

The third contribution is a comparative evaluation with the GCC, ICC and PLUTO compilers for the automatic parallelization of the benchmark suite in terms of effectiveness. In general, GCC, ICC and PLUTO fail to parallelize regular codes with complex control flows, and irregular computations. In contrast to our KIR-based approach, the evaluated compilers analyze loops in isolation and thus fail to optimize the joint parallelization of multiple loops.

The technology developed in this chapter has been licensed to the spin-off company Appentra Solutions S.L. for the creation of Parallware [15].

Chapter 3

Locality-Aware Automatic Parallelization for GPGPU

The use of GPUs for general purpose computation (GPGPU) has increased dramatically in the past years [107, 94] mainly due to two reasons. On the one hand, the hardware industry has not been able to satisfy the rising demands of computing power while preserving the sequential programming model (as detailed in Section 1.1). On the other hand, GPUs offer a tremendous computing capacity at low cost due to the economic pressure of the video game industry. Therefore, new programming models have been developed to integrate these accelerators (GPUs, but also other manycore devices like the Intel Xeon Phi) with high-level programming languages, giving place to heterogeneous computing systems.

The main drawback of these systems is that their heterogeneity is exposed to the developer. Programming is hard, and parallel architectures make it harder because they require additional tasks to parallelize and tune for optimum performance. As commented in Section 1.2.1, developers have to deal with many low-level characteristics and limitations with libraries for GPU programming. Exploiting locality is key to achieving good performance (see Section 1.2.2), and it is more challenging in GPUs than in CPUs due to the complex GPU memory hierarchy. Compiler directives have demonstrated to combine portability and good performance in these architectures at the same time [87]. Thus, we believe that a directive-based approach is a suitable choice for the automatic parallelization of sequential applications on GPUs developed in this thesis.

The remainder of this chapter is organized as follows. Section 3.1 briefly introduces GPGPU, describes the CUDA programming model [103] and highlights the GPU hardware features that impact on performance. Section 3.2 reviews the OpenHMPP directives [104] and the additional functionalities supported by CAPS Compilers [32] that are relevant for this work. Section 3.3 gives a brief summary on chains of recurrences [18], an algebraic formalism that we use for modeling memory accesses. Section 3.4 introduces the new locality-aware optimization technique for GPUs. Section 3.5 details the operation of our approach with two representative case studies extracted from compute-intensive scientific applications: the three-dimensional discrete convolution (CONV3D), and the simple-precision general matrix multiplication (SGEMM). Section 3.6 presents the performance evaluation. Section 3.7 discusses related work and, finally, Section 3.8 summarizes the main conclusions of the chapter.

3.1 GPGPU with the CUDA Programming Model

GPUs were designed for the fast and efficient manipulation of images to be shown on displays. Certain stages of the graphics pipeline perform floating-point operations on independent data, such as transforming the positions of triangle vertices or generating pixel colors. Therefore, GPUs execute thousands of concurrent threads in an SIMD fashion requiring high-bandwidth memory access. This design goal is achieved because GPUs devote more transistors than CPUs to data processing, instead of data caching and control flow. The transition from fixed-function to programmable shaders has made these computational resources useful for general purpose programming [84].

The first GPGPU approaches (OpenGL [130], Cg [100]) forced programs to look like graphics applications that drew triangles and polygons, limiting the accessibility of GPUs. However, NVIDIA introduced in November 2006 the *Compute Unified Device Architecture (CUDA)* [103, 89], which enables the use of C as GPU programming language. The programmer must define functions, called *CUDA kernels*, which specify the operation of a single GPU thread. These light-weight parallel threads are organized into a hierarchy: a *grid* of blocks, a *block* of threads. Blocks may execute in parallel allowing easy scalability. The execution of the threads of a block can be synchronized with a barrier.

Figures 3.1 and 3.2 present the hardware implementation of the GeForce 8800, the first CUDA-enabled GPU. It consists of an array of *Streaming Multiprocessors (SMs)*, where each SM executes the threads of a block in groups of 32 called *warps*. The threads of a warp execute one common instruction at a time, although each thread has its own execution state.

Table 3.1 shows the main characteristics of the different memory types available in NVIDIA GPUs. When a CUDA kernel begins its execution, each thread gets assigned its own private subset of *registers* and a portion of *local memory*. The *shared memory* enables fast data interchange between the threads of a block. In addition, all GPU threads can access the *global memory* (the biggest one), and two more read-only memories: the *constant memory* and the *texture memory* (with special addressing modes and not targeted in this thesis). CUDA assumes that all of these memories are physically on the GPU card, separated from the memory addressed by the CPU. Thus, memory allocations and transfers must be explicitly managed by the programmer.

The *compute capability* of an NVIDIA GPU defines its core architecture (Tesla, Fermi, etc.), supported features (e.g., double-precision floating-point operations), technical specifications (e.g., the maximum dimensions of the hierarchy of threads) and architectural specifications (e.g., the number of warp schedulers).

In summary, the generation of efficient GPGPU code requires the programmer to explicitly handle the GPU hardware architecture through the following programming features exposed by CUDA:

	Location	**Access**	**Scope**
registers	SM	read & write	one GPU thread
local memory	DRAM	read & write	one GPU thread
shared memory	SM	read & write	all GPU threads in a block
global memory	DRAM	read & write	all GPU threads & CPU

Table 3.1 – Characteristics of the GPU memories considered in this thesis.

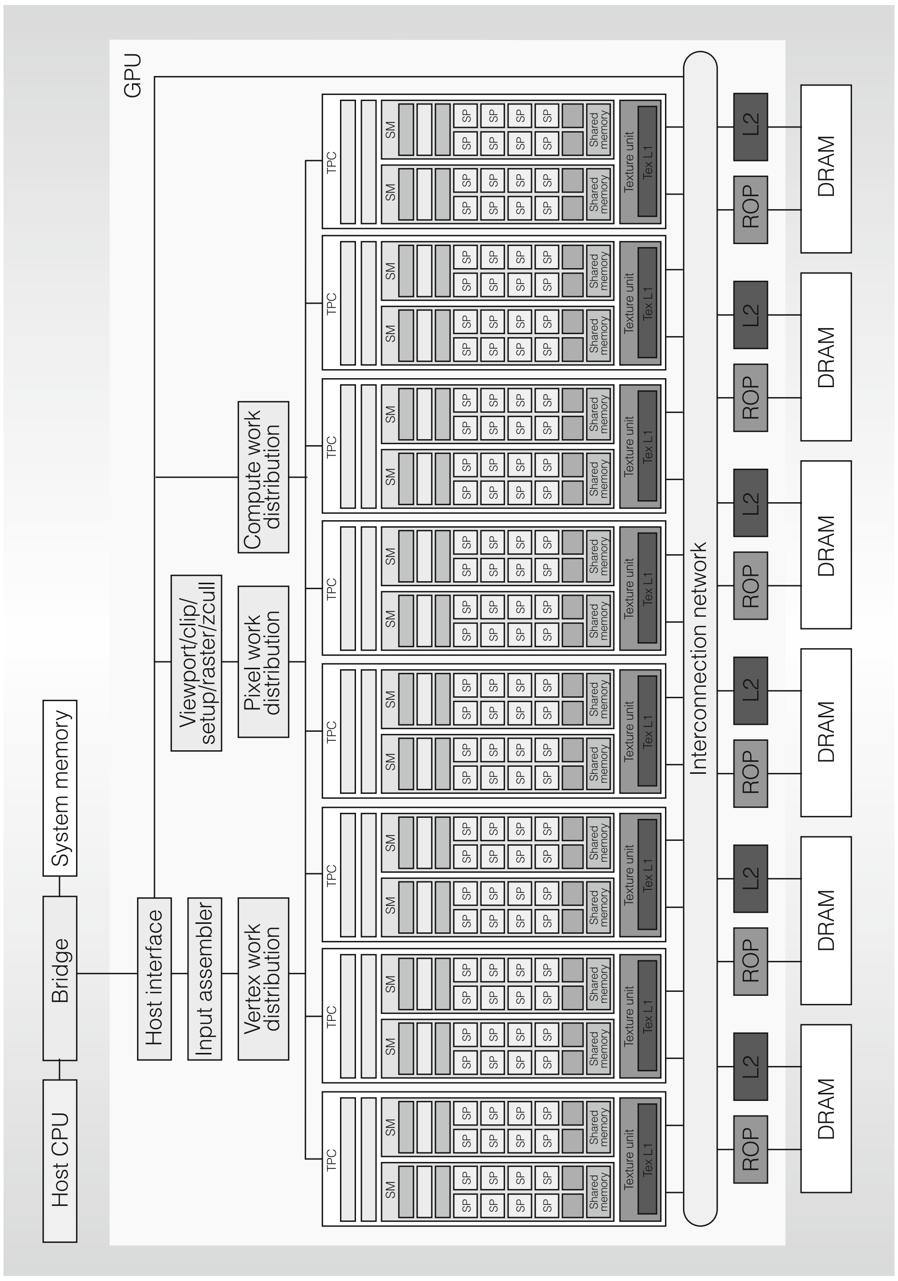


Figure 3.1 – Tesla unified graphics and computing GPU architecture. TPC: texture/processor cluster; SM: streaming multiprocessor; SP: streaming processor; Tex: texture; ROP: raster operation processor (from [89]).

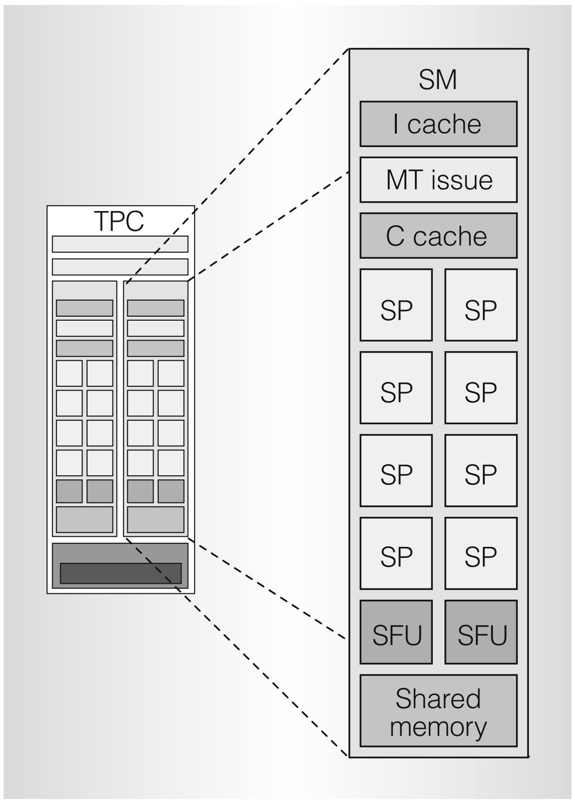


Figure 3.2 – Streaming multiprocessor (SM) of Figure 3.1. I cache: instruction cache; MT issue: multithreaded instruction fetch and issue unit; C cache: constant cache; SP: streaming processor; SFU: special function unit (from [89]).

1. *Threadification*, i.e., the policy that guides the creation of GPU threads and what code they will execute. Each thread has a unique identifier (`threadIdx.{x,y,z}` within a block, `blockIdx.{x,y,z}` within the grid) that is commonly used to access data stored in the GPU memories, in a similar way to loop indices.

2. Thread grouping, so that threads are dispatched in warps to SMs (and threads in a warp execute one common instruction at a time).

The CUDA C Best Practices Guide [102] also prioritizes some strategies to improve the performance of the GPU code:

3. Minimization of CPU-GPU data transfers.

4. Coalesced accesses to global memory, i.e., several memory accesses from different threads are handled by a unique transaction to the global memory.

5. Maximum usage of registers and shared memory to avoid redundant accesses to global memory.

6. Avoidance of thread divergence, i.e., threads within the same warp following different execution paths.

7. Sufficient occupancy, i.e., sufficient number of active threads per SM.

8. The number of threads per block must be a multiple of 32.

The most relevant programming features in points (1)–(8) have been considered in the design of our locality-aware technique to tune the performance of the automatically generated GPU parallel code. The next section describes the support provided by OpenHMPP for those programming features.

3.2 OpenHMPP Directives and CAPS Compilers

CAPS Entreprise offered a complete suite of software tools to develop high performance parallel applications targeting heterogeneous systems based on manycore accelerators. The most relevant ones are CAPS Compilers [32], which generate CUDA [103] and OpenCL [129] code from a sequential application annotated with compiler directives. Directive-based approaches (as the well-known OpenMP [105]) try to reduce the programming effort and provide more readable codes. In this way, these approaches ease the interaction between application-domain experts and programmers. The sequential and the parallel versions co-exist in the same file, offering an incremental way to migrate applications. The developed codes are independent from the hardware platform and new hardware accelerators supported by the translator are automatically exploited. In addition, reasonable performance is achieved compared to hand-written GPU codes [87]. Thus, we consider that compiler directives offer a convenient instrument for the automatic parallelization of sequential applications on GPU-based heterogeneous systems.

Among the numerous proposals of compiler directives to exploit these systems (PGI Accelerator [140], OpenMPC [86], hiCUDA [62], etc.), three standardization efforts have emerged throughout the last years: OpenHMPP [104], OpenACC [131] and, finally, OpenMP 4.0 [105]. All of them follow a similar approach regarding the interaction between the host and the accelerator: they present a Remote Procedure Call (RPC) paradigm that offloads a region of code from the CPU to be executed on the GPU. The address spaces of the host and the accelerator are considered to be disjoint, but data transfers are automatically inserted when needed. However, the programmer is allowed to explicitly manage these transfers in order to improve the performance (for instance, overlapping them with computations through asynchronous calls or specifying only portions of arrays to be copied).

Nevertheless, there exist some differences between the functionality offered by these standards. GPUs commonly have software-managed caches (e.g., the shared memory in the CUDA programming model) whose exploitation is key to achieving good performance. Only OpenACC and OpenHMPP provide a mechanism to explicitly handle this memory. Another significant difference exists when specifying parallelism. OpenHMPP exposes a set of threads where each thread executes one loop iteration. OpenACC presents three levels of parallelism: the programmer can launch a set of *gangs* executing in parallel, where each gang may support multiple *workers*, each with vector or SIMD operations. OpenMP presents a set of threads that are organized in *teams* and can run loop iterations or explicit tasks. This standard can exploit SIMD operations too.

In this work, we have selected OpenHMPP (formerly known as HMPP [25]) and the extension HMPPCG (HMPP Codelet Generator) because these sets of directives provide unique functionality to transform loop nests, which allow the fine tuning of the generated GPU code, and both their compiler and their runtime are much more mature. However, these loop transformations can be performed without directives and we will be able to use OpenACC when a complete implementation is developed. Regarding the recently approved OpenMP 4.0, the explicit management of the complete memory hierarchy by the programmer has not been considered, but our work can help to exploit locality in the implementations of the standard.

OpenHMPP supports the programming features mentioned in points (1)–(8) of Section 3.1 in the following way:

1. The `gridify` directive performs threadification on loops and thread grouping as follows. For simple loops, it generates consecutive GPU threads for consecutive loop iterations, one thread per iteration. For loop nests, it implements a 2D threadification process with the two outermost loops in the nest; consecutive GPU threads are created for consecutive iterations of the inner loop.

2. The `advancedload` and `delegatedstore` directives, with the `asynchronous` clause, allow the overlapping between CPU-GPU data transfers and computations. In addition, it is possible to specify only portions of arrays to be transferred with the triplet `start:end:stride` for each array dimension.

3. The `permute`, `distribute`, `fuse`, `unroll`, `fullunroll` and `tile` directives perform standard compiler transformations on loops. These directives are used to fine tune the performance of the generated GPU code.

4. The `gridify` directive also enables the allocation of program variables on the different GPU memories (for instance, the `shared (variableName)` clause for the shared memory).

Our technique will use these OpenHMPP mechanisms to automatically generate efficient GPU code.

3.3 Chains of Recurrences

Chains of recurrences (from now on, *chrecs*) are an algebraic formalism to represent closed-form functions which have been successfully used to expedite function evaluation at a number of points in a regular interval [18].

Definition 3.3.1. *Given a constant* $\phi \in \mathbb{Z}$*, a function* $g : \mathbb{N}^0 \to \mathbb{Z}$*, and the operator* $+$*, the* ***chrec*** $f = \{\phi, +, g\}$ *is defined as a function* $f : \mathbb{N}^0 \to \mathbb{Z}$ *such that:*

$$\{\phi, +, g\}(i) = \phi + \sum_{j=0}^{i-1} g(j)$$

Hence, the chrecs can be used for representing the iterations of a loop. For example, the loop index of for_i in Figure 3.4 takes integer values in the interval $[0, sizex - 1]$. The chrec $\{0, +, 1\}$ provides a closed-form function to compute the value of i at each for_i iteration.

The chrecs, which are given by the KIR[1], have demonstrated to be a powerful representation of the complex loops and the memory accesses that appear in full-scale real applications [13]. In the same example of Figure 3.4, the memory access pattern i in the first dimension of $input[i][j][k]$ (see line 10 of Figure 3.4) can be represented with the chrec $\{0, +, 1\}$.

The algebraic properties of chrecs provide rules for carrying out arithmetic operations with them [18]. For instance, the addition of a chrec and a constant c is given by $\{\phi, +, g\} + c = \{\phi + c, +, g\}$. This rule enables the representation of the access pattern in the first dimension of $input[i-1][j][k]$ (see line 12 of Figure 3.4) as $\{0, +, 1\} - 1 = \{-1, +, 1\}$. Hence, chrecs can be computed to completely describe the access pattern for n-dimensional arrays.

[1]Note that, for the sake of simplicity, Chapter 2 used triplet notation for the analysis of the values produced/used throughout the execution of diKernels. Chrecs represent the same and more information, and they provide a formal algebra of operations, thus chrecs are the formalism chosen for our compiler framework.

Definition 3.3.2. *Given an access $x_k[i_{k,1}][i_{k,2}]\dots[i_{k,n}]$ to an n-dimensional array x enclosed in a loop $L = L_1, L_2, \dots, L_l$, we say that **the chrec of the array access x_k**, $CHREC\_x_k$, is the result of computing the corresponding chrec for each array dimension:*

$$CHREC\_x_k = [\{\phi_{k,1}, +, g_{k,1}\}][\{\phi_{k,2}, +, g_{k,2}\}]\dots[\{\phi_{k,n}, +, g_{k,n}\}]$$

For illustrative purposes, the first two accesses to *input* (see Figure 3.4) are modeled as:

$$CHREC\_input_1 = [\{0, +, 1\}][\{0, +, 1\}][\{0, +, 1\}]$$

$$CHREC\_input_2 = [\{-1, +, 1\}][\{0, +, 1\}][\{0, +, 1\}]$$

In the context of the generation of efficient GPGPU code, it is necessary not only to consider the memory accesses required in a piece of code, but also to take into account which accesses are emitted by each thread of a warp (as will be seen in Section 3.4). Fortunately, chrecs are a powerful mechanism for this purpose.

Definition 3.3.3. *Let $CHREC\_x_k$ be the chrec of the array access x_k. The **instantiated chrec of access x_k for GPU thread** Ti, $CHREC\_x_k^{\mathrm{Ti}}$, is the result of evaluating the closed-form functions of $CHREC\_x_k$ for the particular values of the loop indices assigned to the GPU thread* Ti.

For instance, assuming that we apply the OpenHMPP `gridify` directive to for_i (i.e., consecutive GPU threads are created for consecutive loop iterations of for_i), the instantiated chrecs for $input[i][j][k]$ (see line 10 of Figure 3.4) for the GPU thread *T0* are:

$$CHREC\_input_1^{T0} = [\{0, +, 0\}][\{0, +, 1\}][\{0, +, 1\}]$$

From now on, the notation of the chrecs with the form $\{\phi, +, 0\}$ (i.e., $g = 0$) will be simplified to $\{\phi\}$. In the previous example, the chrec $\{0, +, 0\}$ will be written as $\{0\}$ representing that the GPU thread *T0* always executes $input[i][j][k]$ with $i = 0$.

$$CHREC\_input_1^{T0} = [\{0\}][\{0, +, 1\}][\{0, +, 1\}]$$

3.4 Locality-Aware Automatic Generation of Efficient GPGPU Code

As mentioned in Section 1.2.1, two complex problems have to be addressed for the successful automatic parallelization of applications: first, the detection of parallelism to determine what parts in the original source code can be executed concurrently; and second, the generation of efficient parallel code taking into account the underlying hardware architecture.

Chapter 2 presented an OpenMP-based hardware-independent approach targeting multicore processors which has demonstrated to be effective and efficient. However, for peak performance on the GPU, the generated code must exploit its characteristic hardware architecture (in particular, the complex memory hierarchy). Hereafter, we introduce a new locality-aware code generation technique that extends the previous approach considering the most impacting programming features enumerated in points (1)–(8) of Section 3.1: loop threadification (1), thread grouping (2), coalesced access to global memory (4), and maximum usage of registers and shared memory (5). The minimization of CPU-GPU data transfers (3) will be addressed with a new automatic partitioning algorithm of the KIR, which will decide what parts of the computations of full-scale applications must be executed on the CPU or on the GPU. Therefore, we assume that program data fits into the GPU memory and, in our experiments (see Section 3.6), we have measured the execution times excluding CPU-GPU data transfers. This thesis does not target the avoidance of thread divergence (6) as it has been successfully addressed by other complementary techniques [119, 76, 63, 33]. In addition, maintaining sufficient occupancy (7) or determining the best block size (8) are programming features very close to the concrete GPU hardware that executes the code and their optimization needs runtime information, thus they are out of the scope of this thesis.

Algorithm 3.1 Detection of whether an access to the GPU global memory can be coalesced

```
 1: FUNCTION isCoalescedAccess
Input: access $x_k[i_{k,1}][i_{k,2}]\ldots[i_{k,n}]$ to an $n$-dimensional array $x$ stored in row-major
    order
Input: loop nest $L = L_1, L_2, \ldots, L_l$ where $L_1$ is the threadified loop
Output: returns whether the given access $x_k$ can be coalesced after threadifying
    the loop nest $L$
 2:     $CHRECS\_x_k \leftarrow [\{\phi_{k,1}, +, g_{k,1}\}][\{\phi_{k,2}, +, g_{k,2}\}]\ldots[\{\phi_{k,n}, +, g_{k,n}\}]$
 3:     $W \leftarrow$ warp of GPU threads $\{T0, T1, T2\ldots\}$
 4:     for each thread $Ti$ in $W$ do
 5:         $CHRECS\_x_k^{Ti} \leftarrow [\{\phi_{k,1}^{Ti}, +, g_{k,1}^{Ti}\}][\{\phi_{k,2}^{Ti}, +, g_{k,2}^{Ti}\}]\ldots[\{\phi_{k,n}^{Ti}, +, g_{k,n}^{Ti}\}]$
 6:     end for
 7:     if $(\exists d \in \{1\ldots n-1\}, Tj \in W - \{T0\} : \{\phi_{k,d}^{Tj}, +, g_{k,d}^{Tj}\} \neq \{\phi_{k,d}^{T0}, +, g_{k,d}^{T0}\})$ then
 8:         return false                                  ▷ first $n-1$ chrecs differ
 9:     end if
10:     $CHRECS\_RANGE\_x_{k,n} \leftarrow \bigcup^{Ti}\{\phi_{k,n}^{Ti}, +, g_{k,n}^{Ti}\}$
11:     if $CHRECS\_RANGE\_x_{k,n}$ defines a convex set then
12:         return true               ▷ threads of the warp access consecutive locations
13:     else
14:         return $(\forall\, Tj \in W - \{T0\} : \{\phi_{k,n}^{Tj}, +, g_{k,n}^{Tj}\} = \{\phi_{k,n}^{T0}, +, g_{k,n}^{T0}\})$
                                          ▷ threads of the warp access the same location
15:     end if
16: end FUNCTION
```

3.4.1 Detection of Coalesced Accesses to the GPU Global Memory

According to the CUDA Best Practices Guide [102], coalescing is maximized (and thus memory requests are minimized) if the threads of a warp access consecutive memory locations. Algorithm 3.1 identifies coalesced accesses by taking into account loop threadification, thread grouping and chrecs. As mentioned in Section 3.3, for an access x_k to an array x in a loop nest L, the KIR provides the chrecs associated to each array dimension (see line 2 of Algorithm 3.1). Next, chrecs are instantiated to represent the memory accesses performed by each GPU thread by fixing the value of the index of L_1 that the thread executes (lines 4–6). Assuming row-major storage, consecutive memory positions are given by consecutive accesses to the last dimension of the array x. Thus, the first $n-1$ chrecs must be the same (lines 7–9). Finally, if the union of the chrecs of the last dimension defines a convex set, then the accesses are coalesced (lines 10–12). If the chrecs of the last dimension are equal, then the same memory position is accessed and only one memory transaction is needed (line 14).

For illustrative purposes, Figure 3.3a and 3.3c present two possibilities to traverse a 2D array x: row-major traversal (denoted S1) and column-major traversal (S2). Arrays are stored in row-major order in C and thus S1 accesses array x row by row, exploiting locality and minimizing data cache misses on the CPU. Assume that only the outer loop of a nest is threadified on the GPU (contrary to the OpenHMPP default policy —see Section 3.2—). Hence, each GPU thread will access consecutive memory positions: *T0* will access $x[0][0]$, $x[0][1]$, $x[0][2]\ldots$ (see Figure 3.3b). Therefore, for the iteration $j = 0$, the threads of the first warp (*T0*, *T1*, *T2*...) will access the non-consecutive memory locations $x[0][0]$, $x[1][0]$, $x[2][0]\ldots$ and these memory requests cannot be coalesced by the GPU memory controller. Algorithm 3.1 detects this non-coalesced access pattern as follows. The KIR provides (see line 2 of Algorithm 3.1):

$$CHRECS\_x_k = [\{0,+,1\}][\{0,+,1\}]$$

Next, chrecs are instantiated (lines 4–6):

$$CHRECS\_x_k^{T0} = [\{0\}][\{0,+,1\}]$$

$$CHRECS\_x_k^{T1} = [\{1\}][\{0,+,1\}]\ldots$$

They are different for the first dimension, thus the threads cannot access consecutive memory positions (lines 7–9).

In contrast, the outer loop j drives the access to the last dimension of array x in S2 (see Figure 3.3c). This code will run poorly on the CPU in the common situation when the array x is bigger than the cache memory. However, on the GPU, *T0* will access $x[0][0]$, $x[1][0]$, $x[2][0]\ldots$ (see Figure 3.3d). Hence, for the iteration $i = 0$, the threads of the first warp (*T0*, *T1*, *T2*...) will access the consecutive memory locations $x[0][0]$, $x[0][1]$, $x[0][2]\ldots$ and these memory requests can be coalesced. Algorithm 3.1 detects this coalesced access pattern as follows. The KIR

```
// only for_i is threadified
for (i = 0; i <= N; i++) {
    for (j = 0; j <= N; j++) {
        ... x[i][j] ...
    }
}
```

(a) Source code S1.

```
// only for_j is threadified
for (j = 0; j <= N; j++) {
    for (i = 0; i <= N; i++) {
        ... x[i][j] ...
    }
}
```

(c) Source code S2.

		T0 (i=0)	*T1* (i=1)	*T2* (i=2)
	j=0	$x[0][0]$	$x[1][0]$	$x[2][0]$
	j=1	$x[0][1]$	$x[1][1]$	$x[2][1]$
	j=2	$x[0][2]$	$x[1][2]$	$x[2][2]$
	...	...	...	...
chrecs	1st*dim*	$\{0\}$	$\{1\}$	$\{2\}$
	2nd*dim*	$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$

(b) Non-coalesced accesses.

		T0 (j=0)	*T1* (j=1)	*T2* (j=2)
	i=0	$x[0][0]$	$x[0][1]$	$x[0][2]$
	i=1	$x[1][0]$	$x[1][1]$	$x[1][2]$
	i=2	$x[2][0]$	$x[2][1]$	$x[2][2]$
	...	...	...	...
chrecs	1st*dim*	$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$
	2nd*dim*	$\{0\}$	$\{1\}$	$\{2\}$

(d) Coalesced accesses.

Figure 3.3 – Examples of access patterns to the GPU global memory.

provides (see line 2 of Algorithm 3.1):

$$CHRECS\_x_k = [\{0,+,1\}][\{0,+,1\}]$$

Next, chrecs are instantiated (lines 4–6):

$$CHRECS\_x_k^{T0} = [\{0,+,1\}][\{0\}]$$

$$CHRECS\_x_k^{T1} = [\{0,+,1\}][\{1\}]\ldots$$

They are the same for the first dimension, thus the threads may access consecutive memory positions (lines 7–9). The union of the last chrecs $\{0\} \cup \{1\} \cup \ldots$ defines a convex set and therefore the performed accesses are coalesced and correctly exploit the GPU global memory hierarchy (lines 10–12).

Algorithm 3.1 is invoked for all the array accesses enclosed in the loop nests of the program. If the index of the threadified loop does not drive the access to the last dimension of the array, a general strategy to try to exploit coalescing is to permute the loops of the nest (as will be seen in Section 3.5).

3.4.2 Maximization of the Usage of the GPU Registers and the Shared Memory

As mentioned in point (5) of Section 3.1, the GPU global memory is the biggest but slowest one. Both registers and shared memory are faster, but they have much less capacity. Therefore, this complex memory hierarchy should be managed with even more care than the traditional CPU memory hierarchy due to its biggest impact on performance.

Algorithm 3.2 presents a technique to detect reused data within a GPU thread. It considers all the accesses to an n-dimensional array x in a loop nest L (see line 2 of Algorithm 3.2). As mentioned in Section 3.3, the KIR provides the chrecs associated to each access in each array dimension (line 3). For each GPU thread, the chrecs are instantiated by fixing the value of the index of L_1 that the thread executes (line 5). If the intersection of the instantiated chrecs for the GPU thread is not empty, then some data are accessed several times and they can be stored in the GPU registers if they are not modified by another thread (lines 6–9). Note that the shared memory could be used for the same purpose as it has the same access time as registers.

Algorithm 3.2 Usage of registers to store reused data within a GPU thread

```
 1: PROCEDURE StoreReusedDataInRegisters
Input: $n$-dimensional array $x[s_1][s_2]\dots[s_n]$
Input: loop nest $L = L_1, L_2, \dots, L_l$ where $L_1$ is the threadified loop
Output: a modified program that exploits reused data to maximize the usage of
    the GPU registers
 2:     collect accesses $x_k[i_{k,1}][i_{k,2}]\dots[i_{k,n}]$ with $k \in \{1,\dots,m\}$
 3:     $CHRECS\_x_k \leftarrow [\{\phi_{k,1},+,g_{k,1}\}][\{\phi_{k,2},+,g_{k,2}\}]\dots[\{\phi_{k,n},+,g_{k,n}\}]$
 4:     for each thread $Ti$ do
 5:         $CHRECS\_x_k^{Ti} \leftarrow [\{\phi_{k,1}^{Ti},+,g_{k,1}^{Ti}\}][\{\phi_{k,2}^{Ti},+,g_{k,2}^{Ti}\}]\dots[\{\phi_{k,n}^{Ti},+,g_{k,n}^{Ti}\}]$
 6:         $REUSED\_DATA\_x^{Ti} \leftarrow \bigcap_{k=1}^{m} CHRECS\_x_k^{Ti}$
 7:         if ($REUSED\_DATA\_x^{Ti} \neq \emptyset$) then
 8:             store reused data between the accesses made by $Ti$ in its set of
                    registers if data are private
 9:         end if
10:     end for
11: end PROCEDURE
```

However, the GPU shared memory has been specifically designed to share data between the threads of a block. Algorithm 3.3 presents a technique that takes into account all the accesses to an n-dimensional array x in a loop nest L emitted by the threads of a block (see line 2 of Algorithm 3.3). The KIR provides the chrecs associated to each access in each array dimension (line 3). For each thread of the considered block, the chrecs are instantiated by fixing the value of the index of L_1 that the thread executes (lines 5–7). If the intersection of the instantiated chrecs associated to all the accesses is not empty, then some data are accessed several times and can be stored in the shared memory (lines 8–11).

Algorithm 3.3 Usage of the GPU shared memory for data shared between the threads of a block

```
1:  PROCEDURE StoreSharedDataInSharedMemory
Input: $n$-dimensional array $x[s_1][s_2]\dots[s_n]$
Input: loop nest $L = L_1, L_2, \dots, L_l$ where $L_1$ is the threadified loop
Output: a modified program using the GPU shared memory to share data between the threads of a block
2:      collect accesses $x_k[i_{k,1}][i_{k,2}]\dots[i_{k,n}]$ with $k \in \{1,\dots,m\}$
3:      $CHRECS\_x_k \leftarrow [\{\phi_{k,1},+,g_{k,1}\}][\{\phi_{k,2},+,g_{k,2}\}]\dots[\{\phi_{k,n},+,g_{k,n}\}]$
4:      for each block $B$ do
5:          for each thread $Ti$ in $B$ do
6:              $CHRECS\_x_k^{Ti} \leftarrow [\{\phi_{k,1}^{Ti},+,g_{k,1}^{Ti}\}][\{\phi_{k,2}^{Ti},+,g_{k,2}^{Ti}\}]\dots[\{\phi_{k,n}^{Ti},+,g_{k,n}^{Ti}\}]$
7:          end for
8:          $SHDATA\_x \leftarrow \bigcap^{Ti} CHRECS\_x_k^{Ti}$ with $k \in \{1,\dots,m\}$
9:          if ($SHDATA\_x \neq \emptyset$) then
10:             store data shared between the threads of block $B$
                    in the shared memory
11:         end if
12:     end for
13: end PROCEDURE
```

Another general technique to improve performance is loop tiling. It consists of partitioning the loop iterations into blocks to ensure that data being used stay in the fastest levels of the memory hierarchy. As explained in Section 3.2, OpenHMPP implements loop threadification and thread grouping with the two outermost loops in a nest; consecutive GPU threads are created for consecutive iterations of the inner loop. Therefore, the common $m \times n$ tiling breaks coalescing because the step of L_2 is different from one and thus consecutive threads will not access consecutive memory locations. Algorithm 3.4 presents a technique for loop tiling that preserves coalescing under OpenHMPP and also considers the promotion of the enclosed scalar variables. Instead of creating a thread for each access x_k, a bigger portion of data to compute (Δ) is given to each thread. Hence, the algorithm increments the step of L_1 to $i = i + \Delta$ (see line 2 of Algorithm 3.4). Scalar variables inside L are promoted to arrays of size Δ, and their corresponding reads and writes are transformed into loops preserving dependences (lines 3–6). The optimization of the size of Δ depends on runtime information about the GPU hardware, thus it has been adjusted empirically by hand in this work.

Algorithm 3.4 Increase the computational load of a GPU thread

```
1: PROCEDURE IncreaseLoad
Input: access $x_k[i_{k,1}][i_{k,2}] \ldots [i_{k,n}]$ to an $n$-dimensional array $x$ stored in row-major order
Input: loop nest $L = L_1, L_2, \ldots, L_l$ where both $L_1, L_2$ are threadified
Input: amount of data $\Delta$ to be processed by a GPU thread
Output: a modified program after applying loop tiling under the OpenHMPP programming model
2:     increment the step of the outer loop $L_1$ to $\Delta$
3:     for each scalar variable $s$ in $L$ do
4:         promote $s$ to an array $s[\Delta]$
5:         transform reads and writes to $s$ into loops of $\Delta$ iterations
6:     end for
7: end PROCEDURE
```

The previous technique can prevent some GPU compiler optimizations: typically, these binary compilers make better optimizations if the program is coded with several instructions using scalar variables (avoiding arrays and loops). In order to solve this issue, Algorithm 3.5 applies loop unrolling and loop interchange to the output of Algorithm 3.4.

3.5 Case Studies

This section details the operation of our locality-aware automatic parallelization technique introduced in Section 3.4. We have selected two representative case studies extracted from compute-intensive scientific applications. First,

Algorithm 3.5 Use scalar variables to enable GPU compiler optimizations

```
1: PROCEDURE IncreaseLoad
Input: loop nest $L = L_1, L_2, L_3 \ldots, L_l$ that results of Algorithm 3.4 where both
    $L_1, L_2$ are threadified, the step of $L_1$ is $\Delta$, and $L_3$ is the created loop with $\Delta$
    iterations
Output: a modified program that uses more scalar variables to enable GPU com-
    piler optimizations
2:     apply loop fission to $L_3$, the loop created in line 5 of Algorithm 3.4
3:     for each loop $L'_3$ resulting from the fission of $L_3$ do
4:         interchange loops until $L'_3$ is the innermost one
5:         insert a fullunroll directive before $L'_3$
6:     end for
7: end PROCEDURE
```

Section 3.5.1 presents the study of the three-dimensional discrete convolution (CONV3D). With this case study we cover stencil codes, which are commonly found in computer simulations, image processing and finite element methods. Next, Section 3.5.2 addresses the simple-precision general matrix multiplication (SGEMM), which is one of the most important linear algebra routines commonly used in engineering, physics or economics.

3.5.1 Case Study: CONV3D

The three-dimensional discrete convolution operator can be generally written as:

$$output[i][j][k] = \sum_{n_1,n_2,n_3} coef[i][j][k] \cdot input[i-n_1][j-n_2][k-n_3]$$

with *input* being the input 3D-function data, *coef* the filter, and *output* the convoluted data. Consider the implementation shown in Figure 3.4 (from now on, denoted as variant *conv3d-cpu*). Three nested loops for_i, for_j and for_k traverse *output* (see lines 7–9). For each element $output[i][j][k]$, four elements in each sense of the three directions of the coordinate axis are taken to perform the convolution with the scalar values *coefx*, *coefy* and *coefz*, respectively. Thus, the temporary variable *tempx* (lines 10–16) stores the weighted sum of nine values of *input* along the *x*-axis, *coefx* being the weight. Similarly, temporaries *tempy* and *tempz* are along the y-axis and z-axis. Finally, these contributions are accumulated in $output[i][j][k]$ (lines 31–32).

The corresponding KIR is shown in Figure 3.5. The loops are perfectly nested, thus they are represented by a unique execution scope $ES\_for_{i,j,k}$. One diKernel is created for each temporary variable, which stores the calculations in each 3D axis: $K{<}tempx_{10}{>}$, $K{<}tempy_{17}{>}$ and $K{<}tempz_{24}{>}$. Note that the subindices refer to the line number in the source code (e.g., the term $tempx_{10}$ refers to the statement in lines 10–16 of Figure 3.4). Their contribution to the final result $K{<}output_{31}{>}$ is symbolized by diKernel-level flow dependences (➡). Scalars *tempx*, *tempy* and *tempz* are assigned new values in each $for_{i,j,k}$ iteration, thus $K{<}tempx_{10}{>}$, $K{<}tempy_{17}{>}$ and $K{<}tempz_{24}{>}$ are scalar assignments. In contrast, the value stored in $output[i][j][k]$ depends on the previous one and thus $K{<}output_{31}{>}$ is a regular reduction. The diKernels that represent loop indices are not shown because they are already represented in the notation of the execution scope and the types of the remaining diKernels. Only the regular reduction $K{<}output_{31}{>}$ determines if CONV3D is parallelizable (note that the remaining parts of the KIR are shaded because they represent privatizable temporaries). As the regular reduction diKernel represents conflict-free loop iterations, it can be converted into a forall parallel loop. On the CPU, it can be parallelized using the OpenMP `parallel for` directive (see Section 2.2.1).

Table 3.2 summarizes the GPU features addressed by our locality-aware automatic parallelization technique to generate the same optimal variant as the one written by an expert in GPU programming. The first optimized variant is *conv3d-*

GPU Features	*conv3d-cpu*	*conv3d-hmpp1*	*conv3d-hmpp2*	*conv3d-hmpp3*	*sgemm-cpu*	*sgemm-mkl*	*sgemm-hmpp1*	*sgemm-hmpp2*	*sgemm-hmpp3*	*sgemm-hmpp4*	*sgemm-cublas*
Coalescing	-	√	√	√	-	-	√	√	√	√	-
Registers	-		√	√	-	-			√	√	-
Shared Memory	-			√	-	-				√	-

Table 3.2 – GPU features exploited with each variant of CONV3D and SGEMM.

```c
int sizex, sizey, sizez, bound = 4;

void conv3d(float output[sizex][sizey][sizez],
  float input[bound+sizex+bound][4+sizey+4][4+sizez+4],
  float coefx, float coefy, float coefz) {

  for (int i = 0; i < sizex; i++) {
    for (int j = 0; j < sizey; j++) {
      for (int k = 0; k < sizez; k++) {
        float tempx = input[i][j][k] + coefx *
          (
            input[i-1][j][k] + input[i+1][j][k] +
            input[i-2][j][k] + input[i+2][j][k] +
            input[i-3][j][k] + input[i+3][j][k] +
            input[i-4][j][k] + input[i+4][j][k]
          );
        float tempy = input[i][j][k] + coefy *
          (
            input[i][j-1][k] + input[i][j+1][k] +
            input[i][j-2][k] + input[i][j+2][k] +
            input[i][j-3][k] + input[i][j+3][k] +
            input[i][j-4][k] + input[i][j+4][k]
          );
        float tempz = input[i][j][k] + coefz *
          (
            input[i][j][k-1] + input[i][j][k+1] +
            input[i][j][k-2] + input[i][j][k+2] +
            input[i][j][k-3] + input[i][j][k+3] +
            input[i][j][k-4] + input[i][j][k+4]
           );
        output[i][j][k] =
          output[i][j][k] + tempx + tempy + tempz;
      }
    }
  }
}
```

Figure 3.4 – Source code of the 3D discrete convolution operator (CONV3D).

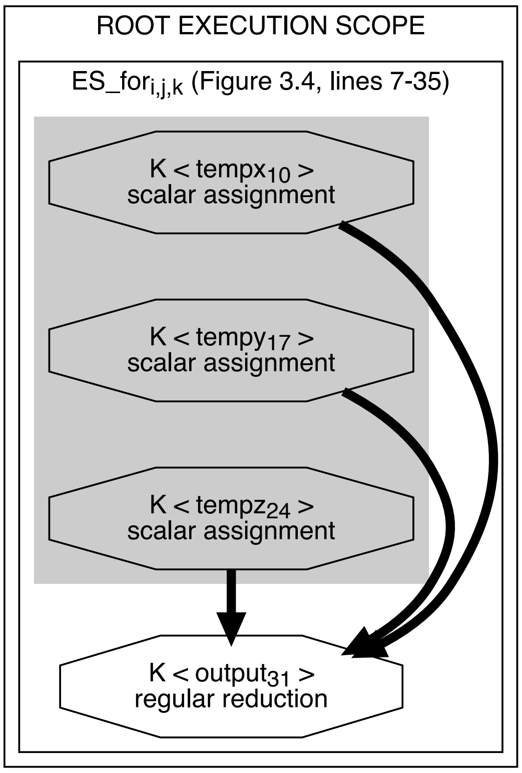

ROOT EXECUTION SCOPE
ES_for$_{i,j,k}$ (Figure 3.4, lines 7-35)
K < tempx$_{10}$ > scalar assignment
K < tempy$_{17}$ > scalar assignment
K < tempz$_{24}$ > scalar assignment
K < output$_{31}$ > regular reduction

Figure 3.5 – KIR of the 3D discrete convolution operator (CONV3D).

hmpp1, which exploits coalescing through loop interchange as follows. A basic OpenHMPP variant could be generated by simply isolating the source code between lines 7–35 of Figure 3.4. However, Algorithm 3.1 detects that this is not the correct approach due to the non-coalesced accesses. The chrecs associated to the first access to *input* (see line 10 of Figure 3.4) are:

$$CHREC\_input_1 = [\{0,+,1\}][\{0,+,1\}][\{0,+,1\}]$$

As explained in Section 3.2, the default OpenHMPP loop threadification policy creates GPU threads for the two outermost loops (for_i and for_j). Hence, the instantiated chrecs would be:

$$CHREC\_input_1^{T0} = [\{0\}][\{0\}][\{0,+,1\}]$$

$$CHREC\_input_1^{T1} = [\{0\}][\{1\}][\{0,+,1\}] \dots$$

These accesses cannot be coalesced by the memory controller (see lines 7–9 of Algorithm 3.1). However, if the loop nest is permuted to for_j, for_k, for_i, the chrecs will be:

$$CHREC\_input_1^{T0} = [\{0,+,1\}][\{0\}][\{0\}]$$

$$CHREC\_input_1^{T1} = [\{0,+,1\}][\{0\}][\{1\}] \dots$$

Thus,

$$CHREC\_RANGE\_input_{1,3} = \{0\} \cup \{1\} \cup \dots$$

defines a convex set satisfying the condition in line 11 of Algorithm 3.1.

The second optimized variant is *conv3d-hmpp2*. Note that each GPU thread along the threadified $for_{j,k}$ executes the entire innermost for_i. Hence, each thread will repeat reads to the array *input* in the x-axis in consecutive iterations of for_i (see lines 10–16 of Figure 3.4). Old values can be stored in local registers reducing the needs of memory bandwidth. Algorithm 3.2 detects this situation as follows. The chrecs for the first three accesses to array *input* are (see line 3 of Algorithm 3.2):

$$CHREC\_input_1 = [\{0,+,1\}][\{0,+,1\}][\{0,+,1\}]$$

$$CHREC\_input_2 = [\{-1,+,1\}][\{0,+,1\}][\{0,+,1\}]$$

$$CHREC\_input_3 = [\{1,+,1\}][\{0,+,1\}][\{0,+,1\}]$$

For *T0*, the instantiated chrecs are (line 5):

$$CHREC\_input_1^{T0} = [\{0,+,1\}][\{0\}][\{0\}]$$

$$CHREC\_input_2^{T0} = [\{-1,+,1\}][\{0\}][\{0\}]$$

$$CHREC\_input_3^{T0} = [\{1,+,1\}][\{0\}][\{0\}]$$

Thus,

$$\bigcap_{k=1}^{3} CHRECS\_input_k^{T0} = [\{1,+,1\}][\{0\}][\{0\}] \neq \emptyset$$

and, as *input* is only read, copies of already accessed values can be kept in registers for subsequent uses (lines 6–9). Figure 3.6 shows an excerpt of the resulting code. Each GPU thread begins its execution saving in local scalar variables (which the compiler will store in registers) the first values of the array that it will need (see lines 9–17). Hence, for the remaining iterations of for_i, only one access to $input$ will be required in order to compute *tempx* (line 27).

The variant *conv3d-hmpp3* exploits, in addition, the shared memory. Contiguous threads repeat accesses to some positions in the y, z-plane of the array $input$. Hence, those values can be stored in the shared memory and be interchanged among the threads of a block. Table 3.3 focuses on the chrecs corresponding to the first two threads, *T0* and *T1*, and the accesses performed in lines 24–30 of Figure 3.4. Algorithm 3.3 computes the intersections of all the instantiated chrecs (see lines 5–8 of Algorithm 3.3). As can be observed, the intersection is not empty and some values can be stored in the GPU shared memory (lines 9–11). Figure 3.7 shows an excerpt of the resulting code. The new variable *input___shared* will be stored in the GPU shared memory as indicates the `shared` clause of the `gridify` directive (see lines 5–6). In each for_i iteration, each GPU thread only needs to copy in the shared memory the limits of the $input$ array that will access later (see lines 18–23) because the remaining threads of the block will copy the remaining values at the same time. Hence, it is not needed to access $input$ (i.e., the GPU global memory) in the rest of the code.

```
#pragma hmpp conv3d___hmpp2 codelet
void conv3d___hmpp2(float output[sizex][sizey][sizez],
  float input[bound+sizex+bound][4+sizey+4][4+sizez+4],
  float coefx, float coefy, float coefz) {

#pragma hmppcg gridify (j, k)
  for (int j = 0; j < sizey; j++) {
    for (int k = 0; k < sizez; k++) {
      float i___minus4 = 0;
      float i___minus3 = input[-4][j][k];
      float i___minus2 = input[-3][j][k];
      float i___minus1 = input[-2][j][k];
      float i___plus0 = input[-1][j][k];
      float i___plus1 = input[0][j][k];
      float i___plus2 = input[1][j][k];
      float i___plus3 = input[2][j][k];
      float i___plus4 = input[3][j][k];
      for (int i = 0; i < sizex; i++) {
        i___minus4 = i___minus3;
        i___minus3 = i___minus2;
        i___minus2 = i___minus1;
        i___minus1 = i___plus0;
        i___plus0 = i___plus1;
        i___plus1 = i___plus2;
        i___plus2 = i___plus3;
        i___plus3 = i___plus4;
        i___plus4 = input[i+4][j][k];
        float tempx = i___plus0 + coefx *
          (
            i___minus1 + i___plus1 +
            i___minus2 + i___plus2 +
            i___minus3 + i___plus3 +
            i___minus4 + i___plus4
          );
        float tempy = ...
        float tempz = ...
        output[i][j][k] =
          output[i][j][k] + tempx + tempy + tempz;
      }
    }
  }
}
```

Figure 3.6 – Excerpt of the parallelized code of the 3D discrete convolution operator (CONV3D): variant *conv3d-hmpp2*.

```
#pragma hmpp conv3d___hmpp3 codelet
void conv3d___hmpp3(float output[sizex][sizey][sizez],
  float input[bound+sizex+bound][4+sizey+4][4+sizez+4],
  float coefx, float coefy, float coefz) {
  float input___shared[bound+8+bound][bound+32+bound];
#pragma hmppcg gridify(j,k),blocksize(32x8),shared(input___shared),unguarded
  for (int j = 0; j < sizey; j++) {
    for (int k = 0; k < sizez; k++) {
      int tx = 0;
      int ty = 0;
#pragma hmppcg set tx = RankInBlockX()
#pragma hmppcg set ty = RankInBlockY()
      int rk = tx + bound;
      int rj = ty + bound;
      float i___minus4 = ...
      for (int i = 0; i < sizex; i++) {
        i___minus4 = ...
#pragma hmppcg grid barrier
        input___shared[rj-bound][rk-bound] = input[i][j-bound][k-bound];
        input___shared[rj+bound][rk-bound] = input[i][j+bound][k-bound];
        input___shared[rj-bound][rk+bound] = input[i][j-bound][k+bound];
        input___shared[rj+bound][rk+bound] = input[i][j+bound][k+bound];
#pragma hmppcg grid barrier
        float tempx = ...
        float tempy = i___plus0 + coefy *
          (
            input___shared[rj-1][rk] + input___shared[rj+1][rk] +
            input___shared[rj-2][rk] + input___shared[rj+2][rk] +
            input___shared[rj-3][rk] + input___shared[rj+3][rk] +
            input___shared[rj-4][rk] + input___shared[rj+4][rk]
          );
        float tempz = i___plus0 + coefz *
          (
            input___shared[rj][rk-1] + input___shared[rj][rk+1] +
            input___shared[rj][rk-2] + input___shared[rj][rk+2] +
            input___shared[rj][rk-3] + input___shared[rj][rk+3] +
            input___shared[rj][rk-4] + input___shared[rj][rk+4]
           );
        output[i][j][k] =
          output[i][j][k] + tempx + tempy + tempz;
      }
    }
  }
}
```

Figure 3.7 – Excerpt of the parallelized code of the 3D discrete convolution operator (CONV3D): variant *conv3d-hmpp3*.

CHRECS	*T0*			*T1*		
	$1^{st}dim$	$2^{nd}dim$	$3^{rd}dim$	$1^{st}dim$	$2^{nd}dim$	$3^{rd}dim$
$CHRECS\_input_{19}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$	$\{0,+,1\}$	$\{0\}$	$\{1\}$
$CHRECS\_input_{20}$	$\{0,+,1\}$	$\{0\}$	$\{-1\}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$
$CHRECS\_input_{21}$	$\{0,+,1\}$	$\{0\}$	$\{1\}$	$\{0,+,1\}$	$\{0\}$	$\{2\}$
$CHRECS\_input_{22}$	$\{0,+,1\}$	$\{0\}$	$\{-2\}$	$\{0,+,1\}$	$\{0\}$	$\{-1\}$
$CHRECS\_input_{23}$	$\{0,+,1\}$	$\{0\}$	$\{2\}$	$\{0,+,1\}$	$\{0\}$	$\{3\}$
$CHRECS\_input_{24}$	$\{0,+,1\}$	$\{0\}$	$\{-3\}$	$\{0,+,1\}$	$\{0\}$	$\{-2\}$
$CHRECS\_input_{25}$	$\{0,+,1\}$	$\{0\}$	$\{3\}$	$\{0,+,1\}$	$\{0\}$	$\{4\}$
$CHRECS\_input_{26}$	$\{0,+,1\}$	$\{0\}$	$\{-4\}$	$\{0,+,1\}$	$\{0\}$	$\{-3\}$
$CHRECS\_input_{27}$	$\{0,+,1\}$	$\{0\}$	$\{4\}$	$\{0,+,1\}$	$\{0\}$	$\{5\}$

Table 3.3 – Chrecs for the accesses in lines 24–30 of Figure 3.4 (CONV3D).

3.5.2 Case Study: SGEMM

The simple-precision general matrix multiplication from the BLAS library [24] performs the matrix operation:

$$C = \alpha \cdot A \times B + \beta \cdot C$$

where A, B, C are $m \times k$, $k \times n$ and $m \times n$ matrices, respectively, and α, β are the scale factors for $A \times B$ and C. Figure 3.8 shows an implementation with two nested loops for_i and for_j that traverse the matrix C row by row (see lines 5–6). Each matrix position $C[i][j]$ is computed with the dot product between the i^{th} row of matrix A and the j^{th} column of B. The dot product is temporarily stored in the scalar variable *prod* (lines 7–10).

The KIR shown in Figure 3.9 captures the semantics of Figure 3.8 as follows. Loops for_i and for_j are perfectly nested, thus a unique execution scope $ES\_for_{i,j}$ is created. $K{<}prod_7{>}$ represents the initialization of the temporary variable *prod* at line 7. The computation of the dot product is contained in for_l. Hence, the scalar reduction $K{<}prod_9{>}$ is attached to ES_for_l. Finally, $K{<}C_{11}{>}$ is a regular reduction that updates the previous value stored in $C[i][j]$. As *prod* is a privatizable scalar variable, the parts of the KIR referring to its computations are shaded in order to be omitted in the discovery of parallelism. Thus, only $K{<}C_{11}{>}$ needs

```
 1 int m, n, k;
 2 void sgemm(float C[m][n], float alpha, float A[m][k],
 3   float B[k][n], float beta) {
 4
 5   for (int i = 0; i < m; i++) {
 6     for (int j = 0; j < n; j++) {
 7       float prod = 0;
 8       for (int l = 0; l < k; l++) {
 9         prod += A[i][l] * B[l][j];
10       }
11       C[i][j] = alpha * prod + beta * C[i][j];
12     }
13   }
14 }
```

Figure 3.8 – Source code of the simple-precision general matrix multiplication (SGEMM).

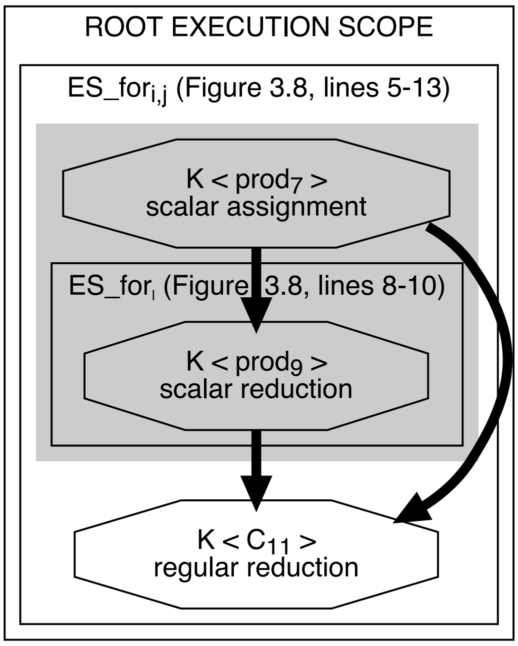

ROOT EXECUTION SCOPE
ES_for$_{i,j}$ (Figure 3.8, lines 5-13)
K < prod$_7$ > scalar assignment
ES_for$_l$ (Figure 3.8, lines 8-10)
K < prod$_9$ > scalar reduction
K < C$_{11}$ > regular reduction

Figure 3.9 – KIR of the simple-precision general matrix multiplication (SGEMM).

to be considered to decide if the source code is parallelizable. As mentioned in Section 2.2.1, a regular reduction diKernel represents conflict-free loop iterations and it is therefore parallelizable.

From the point of view of the locality, the challenge of SGEMM is to handle the tradeoff between opposite array traversals efficiently: row-major for C and A, and column-major for B. On the CPU, the general solution is to apply loop tiling: matrices are computed in small tiles to keep data in the cache memory. This approach can be also applied on the GPU using the shared memory as cache and being aware of coalescing.

The first variant of SGEMM is the sequential code shown in Figure 3.8 (*sgemm-cpu*). In addition, we have selected the `cblas_sgemm` function of the non-clustered, threaded part of the Intel MKL library [69] to build the *sgemm-mkl* variant.

The first OpenHMPP variant is *sgemm-hmpp1*. It is trivially built by offloading to the GPU the same code as *sgemm-cpu*. Table 3.4 shows the chrecs for this variant, which are analyzed by Algorithm 3.1 as follows. Regarding A, all the threads of a warp have the same chrecs and thus access the same memory position (see line 14 of Algorithm 3.1). Regarding B, coalescing is maximized because the chrecs of the first dimension are the same while the chrecs of the second one define a convex set (lines 10–12). Finally, the same situation holds for C and thus accesses are coalesced.

The second OpenHMPP variant is *sgemm-hmpp2*. Algorithm 3.4 transforms the source code of Figure 3.8 as follows. The scalar variable *prod* is promoted to an array $prod[\Delta]$, and thus a new loop for_t is created to enclose all its definitions and uses (see lines 3–6 of Algorithm 3.4). The step of the outer for_i is incremented

CHRECS	not instantiated		*T0*		*T1*	
	$1^{st}dim$	$2^{nd}dim$	$1^{st}dim$	$2^{nd}dim$	$1^{st}dim$	$2^{nd}dim$
CHRECS_A	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$
CHRECS_B	$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$	$\{1\}$
CHRECS_C	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{1\}$

Table 3.4 – Chrecs for the accesses to arrays A, B and C in SGEMM.

by Δ, and uses of the loop index i inside for_t are replaced by $i + t$. Figure 3.10 shows the resulting code.

The third OpenHMPP variant is *sgemm-hmpp3*. For the reasons mentioned in the last paragraph of Section 3.4.2, Algorithm 3.5 is applied. Our technique first performs loop fission in the new for_t giving place to for_{t1} (*prod* initialization), for_{t2} (dot product between the row of A and the column of B), and for_{t3} (computation with the old value of C). Next, `fullunroll` directives are inserted in for_{t1} and for_{t3}. In order to fully unroll for_{t2}, it is first interchanged with for_l. This way, the GPU compiler is able to store $prod[\Delta]$ in registers. The resulting code is shown in Figure 3.11.

The fourth OpenHMPP variant is *sgemm-hmpp4*. Algorithm 3.2 presented a method to store reused data in registers. In this case, as the number of registers is finite and the previous transformation in *sgemm-hmpp3* increased register pressure, we have used the shared memory to store slices of B as done with CONV3D.

Finally, the last variant is *sgemm-cublas*, the implementation provided by the NVIDIA CUBLAS library [101]. CUBLAS has been designed assuming a column-major order, thus a transformation is needed before and after calling the library.

3.6 Performance Evaluation

Two NVIDIA-based heterogeneous systems were used to carry out our experiments. The first one is `nova`, the CAPS Compute Lab, based on Tesla S1070 (compute capability 1.3 —Tesla architecture—). The GPU contains 30 multiprocessors with 8 cores each, for a total of 240 CUDA cores at 1.30 GHz. The total amount of global memory is 4 GB at 800 MHz. Each block (of up to 512 threads) can access 16 KB of shared memory and 16384 registers. The accelerator is connected to a host system consisting of 2 Intel Xeon X5560 quad-core Nehalem processors at 2.80 GHz with 8 MB of cache memory per processor and 12 GB of RAM.

The second system is `pluton`, the cluster of the Computer Architecture Group at the University of A Coruña, based on Tesla S2050 (compute capability 2.0 —Fermi architecture—). The GPU contains 14 multiprocessors with 32 cores each,

```
int m, n, k;
#define DELTA 16

#pragma hmpp sgemm___hmpp2 codelet
void sgemm___hmpp2(float C[m][n], float alpha, float A[m][k],
  float B[k][n], float beta) {

#pragma hmppcg gridify (i,j), blocksize(128x1)
  for (int i = 0; i < m; i = i + DELTA) {
    for (int j = 0; j < n; j++) {
      float prod[DELTA];
      for (int t = 0; t < DELTA; t++) {
        prod[t] = 0;
        for (int l = 0; l < k; l++) {
          prod[t] += A[i+t][l] * B[l][j];
        }
        C[i+t][j] = alpha * prod[t] + beta * C[i+t][j];
      }
    }
  }
}
```

Figure 3.10 – Excerpt of the parallelized code of the simple-precision general matrix multiplication (SGEMM): variant *sgemm-hmpp2*.

```
int m, n, k;
#define DELTA 16

#pragma hmpp sgemm___hmpp3 codelet
void sgemm___hmpp3(float C[m][n], float alpha, float A[m][k],
  float B[k][n], float beta) {

#pragma hmppcg gridify (i,j), blocksize(128x1)
  for (int i = 0; i < m; i = i + DELTA) {
    for (int j = 0; j < n; j++) {
      float prod[DELTA];
#pragma hmppcg fullunroll
      for (int t = 0; t < DELTA; t++) {
        prod[t] = 0;
      }
      for (int l = 0; l < k; l++) {
#pragma hmppcg fullunroll
        for (int t = 0; t < DELTA; t++) {
          prod[t] += A[i+t][l] * B[l][j];
        }
      }
#pragma hmppcg fullunroll
      for (int t = 0; t < DELTA; t++) {
        C[i+t][j] = alpha * prod[t] + beta * C[i+t][j];
      }
    }
  }
}
```

Figure 3.11 – Excerpt of the parallelized code of the simple-precision general matrix multiplication (SGEMM): variant *sgemm-hmpp3*.

for a total of 448 CUDA cores at 1.15 GHz. The total amount of global memory is 3 GB at 1546 MHz with ECC disabled. Each block (of up to 1024 threads) can access 48 KB of shared memory, 16 KB of L1 cache and 32768 registers. The amount of L2 cache is 768 KB. The accelerator is connected to a host system consisting of 2 Intel Xeon X5650 hexa-core Westmere processors at 2.66 GHz with 12 MB of cache memory per processor and 12 GB of RAM.

3.6.1 Performance Evaluation of CONV3D

We have run the 216 experiments corresponding to all matrix sizes for *sizex*, *sizey* and *sizez* values in 128, 256, 384, 512, 640 and 768. In each experiment, we have measured GFLOPS for all CONV3D variants. As can be viewed in Table 3.5, our experiments revealed that the obtained GFLOPS did not show a significant variation with the dimensions of the tested matrices. This is due to the fact that the limiting factor in the performance of this test case is the memory access bandwidth. For all tested sizes, even the smallest ones, more than a 50 % GPU occupancy is achieved (which is a good value for this sort of codes [137]).

Figure 3.12 depicts the performance evaluation of the CPU and the GPU-accelerated variants on our experimental platforms. The offloading of the computations on the GPU, with a loop interchange (*conv3d-hmpp1*), gets a speedup of 5.43x on `nova` and 15.27x on `pluton`. Note the big step in performance improvement between *conv3d-hmpp2* and *conv3d-hmpp3* due to the use of the shared memory: 3.35x on `nova` and 1.62x on `pluton`. The improvement is less impressive on `pluton` because of the hardware-managed cache memories present in Fermi cards that partially cover the functionality exploited by our locality-aware automatic technique.

3.6.2 Performance Evaluation of SGEMM

We have run the 6859 experiments corresponding to all matrix sizes for m, n, and k values in 128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2048, 4096, 6144 and 8192. In each experiment, we have measured GFLOPS for all SGEMM variants.

GFLOPS	nova			pluton		
	min	avg	max	min	avg	max
conv3d-cpu	1.42	2.54	2.72	-	-	-
conv3d-hmpp1	13.42	13.78	14.03	31.40	38.74	46.95
conv3d-hmpp2	18.76	19.80	20.28	51.08	67.47	78.79
conv3d-hmpp3	41.50	66.32	70.60	97.02	109.48	118.75

Table 3.5 – Minimum, average and maximum GFLOPS of CONV3D variants.

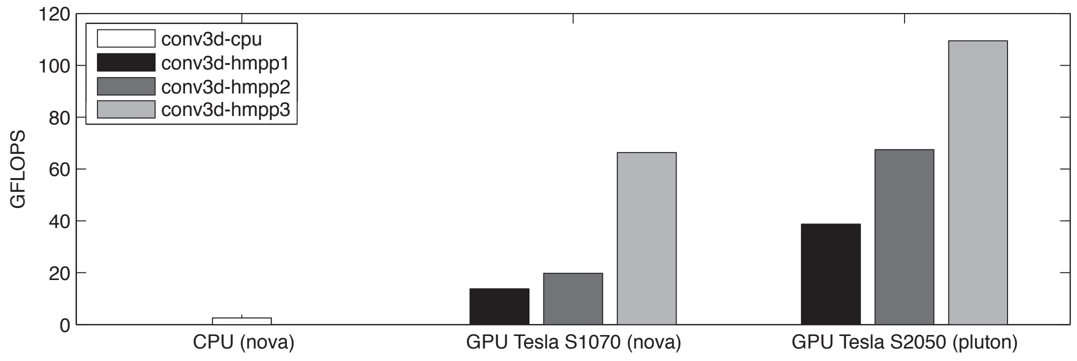


Figure 3.12 – Average GFLOPS of CONV3D variants.

Table 3.6 and Figure 3.13 present the performance evaluation of the CPU and the GPU-accelerated variants. On average, *sgemm-mkl* is better than *sgemm-hmpp1*: 5.26x on `nova` and 1.96x on `pluton`. However, we can appreciate in Figure 3.14 that for most of the combinations of $m, n, k < 2048$, *sgemm-hmpp1* is better than *sgemm-mkl* (up to 31.50x for $m = 256$, $n = 128$ and $k = 512$ on `pluton`). Hence, in contrast to CONV3D, the performance of SGEMM varies significantly for different matrix sizes (as can be also observed in the minimum, average and maximum columns of Table 3.6) and the simple use of the GPU does not always improve the best CPU variant. For the majority of the tested sizes, *sgemm-hmpp2* slightly improves *sgemm-hmpp1* on `nova` (see Figure 3.15), but not on `pluton`. This is due to the fact that accesses to $prod[\Delta]$ in *sgemm-hmpp2* read and write from the GPU memory and not from the registers. The performance improvement of *sgemm-hmpp3* with respect to *sgemm-hmpp1* is bigger (1.79x on `nova` and 2.03x on `pluton`) because the transformation allows the GPU compiler to store $prod[\Delta]$ in registers. Figure 3.16 shows where *sgemm-hmpp1* is better than *sgemm-hmpp3* on `pluton`. However, the biggest improvement factor is the usage of the shared memory, as can be observed in the *sgemm-hmpp4* results.

Figures 3.17 and 3.18 show the best variants for each tested size on both hardware platforms. The best variant on `nova` for the majority of cases is *sgemm-cublas*. However, it is only 10 % better than *sgemm-hmpp4* on average. In fact, *sgemm-hmpp4* is the best for $k < 1024$, and *sgemm-mkl* is the best for $m, n = 128$ with $512 \leq k \leq 1792$. Regarding average performance on `pluton`, *sgemm-cublas* is clearly the best, being 36 % faster than *sgemm-hmpp4*. Variant *sgemm-cublas* is only bested by *sgemm-hmpp4* for $m, n \in \{128, 256\}$, and by *sgemm-mkl* for $m, n = 128$ with $k > 1152$.

3.7 Related Work

In this chapter, we have introduced a new technique to tune the performance of automatically generated GPU parallel code that exploits locality through standard loop transformations. This technique has been successfully applied to two representative case studies, namely CONV3D and SGEMM. There exist in the literature previous works about the optimization of the execution on the GPU of these case studies that are based on templates or domain-specific languages (for instance, [35] and [145] for CONV3D, [82] for SGEMM). In contrast, we propose a general parallelization approach.

There also exist other active approaches based on the polyhedral model.

GFLOPS	`nova`			`pluton`		
	min	avg	max	min	avg	max
sgemm-cpu	0.13	0.51	1.40	-	-	-
sgemm-mkl	1.43	114.99	183.70	-	-	-
sgemm-hmpp1	8.10	21.85	27.41	15.41	58.77	79.22
sgemm-hmpp2	3.33	21.80	27.69	3.72	51.57	78.83
sgemm-hmpp3	6.93	39.19	64.81	12.04	119.56	134.74
sgemm-hmpp4	7.12	295.45	354.46	9.20	357.38	420.63
sgemm-cublas	71.30	325.91	370.12	38.78	486.41	650.16

Table 3.6 – Minimum, average and maximum GFLOPS of SGEMM variants.

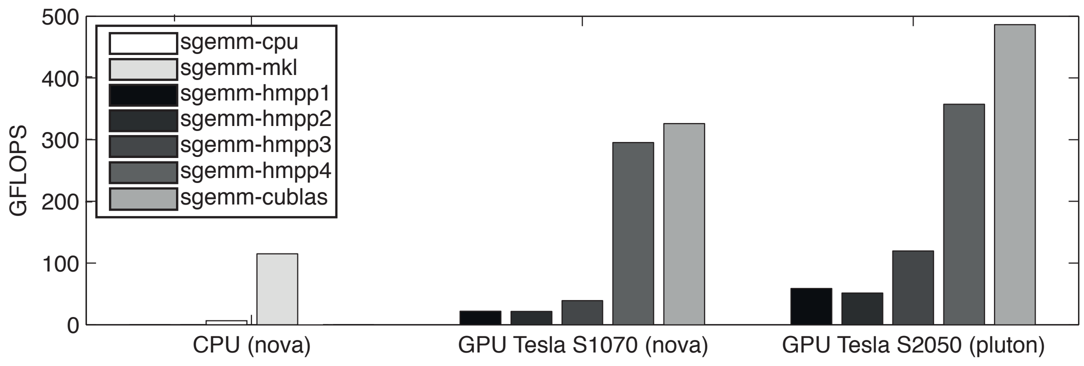


Figure 3.13 – Average GFLOPS of SGEMM variants.

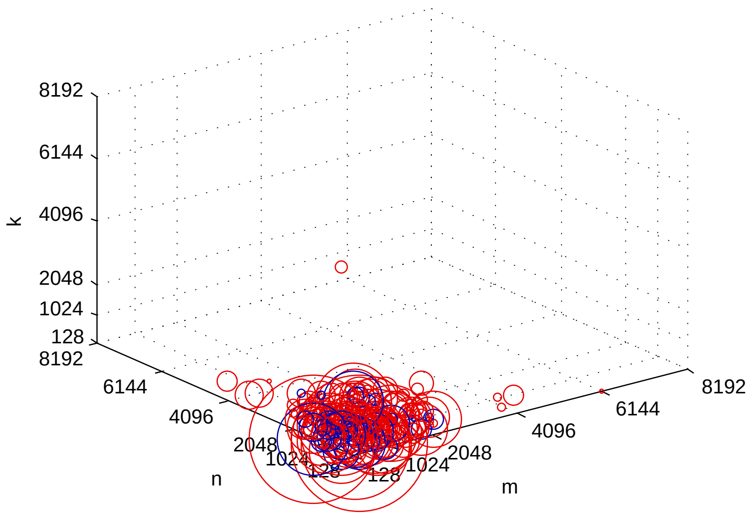


Figure 3.14 – Sizes where *sgemm-hmpp1* is better than *sgemm-mkl* on `nova` (blue) and `pluton` (red). The size of the circles is proportional to the difference in performance.

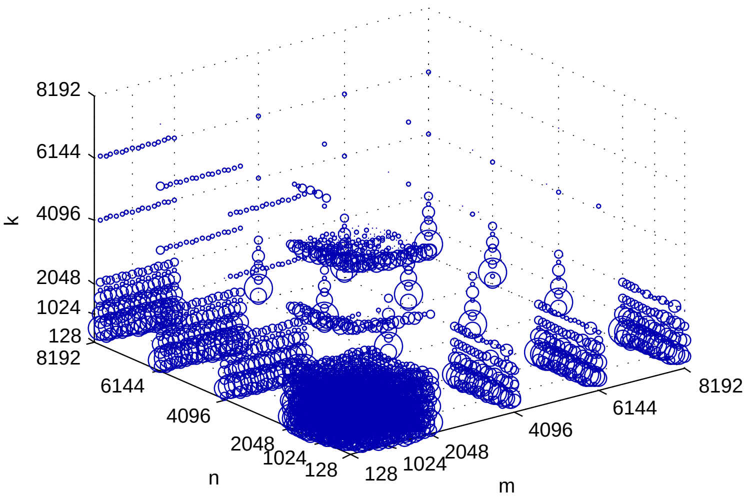


Figure 3.15 – Sizes where *sgemm-hmpp2* is better than *sgemm-hmpp1* on `nova`. The size of the circles is proportional to the difference in performance.

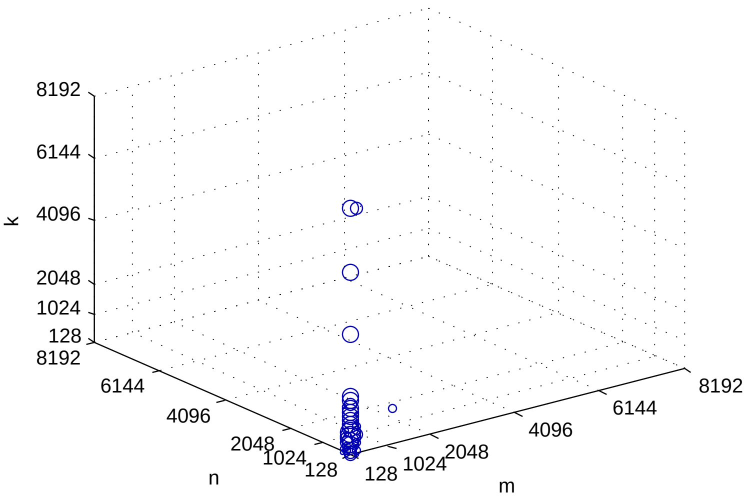


Figure 3.16 – Sizes where *sgemm-hmpp1* is better than *sgemm-hmpp3* on `pluton`. The size of the circles is proportional to the difference in performance.

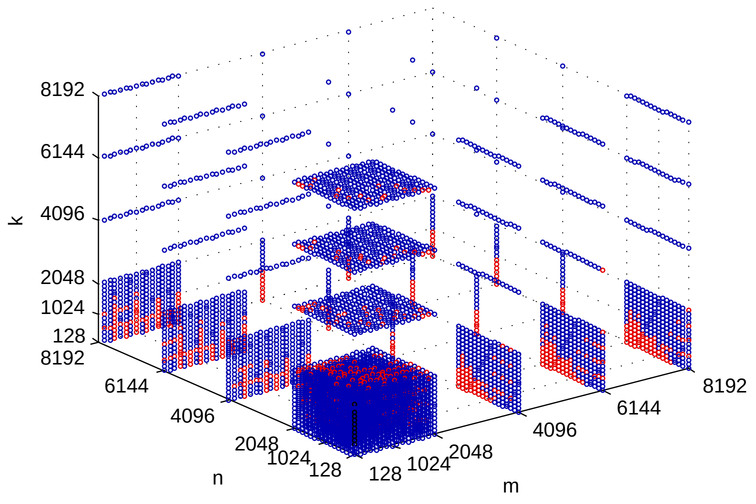


Figure 3.17 – Best variant on `nova`: *sgemm-cublas* in blue, *sgemm-hmpp4* in red, and *sgemm-mkl* in black.

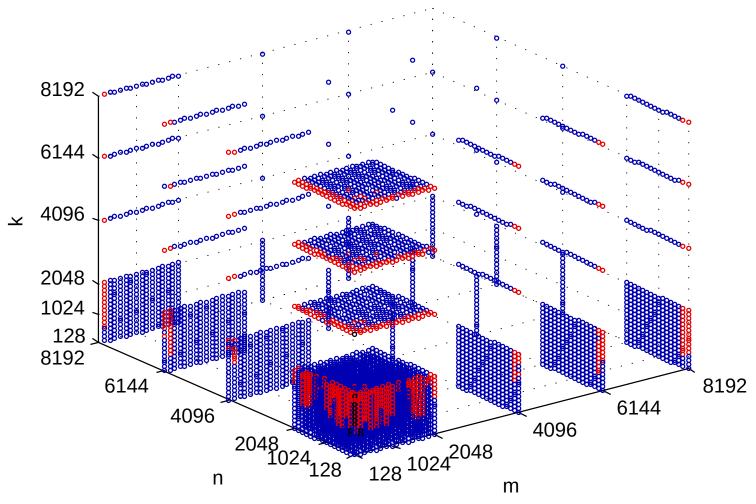


Figure 3.18 – Best variant on `pluton`: *sgemm-cublas* in blue, *sgemm-hmpp4* in red, and *sgemm-mkl* in black.

C-to-CUDA [21], based on PLUTO [27], looks for a region as large as possible and transforms memory accesses to be coalesced (using the shared memory if it is not possible). The shared memory is also employed to store the arrays that are reused in the same thread, but the reuse of data between the threads of a block is not considered.

PPCG [136] also searches for the largest possible region of code and the parallelization strategy evolved from PLUTO. It applies an elaborated policy for the use of the memory hierarchy that groups array references to copy parts of the global memory. Reused data by a single thread are placed in registers. If there are other reuses or the original accesses were not coalesced, then it places the data in the shared memory. A recent extension [74] improved the threadification of the loop nests allowing the sequential execution of some loops to benefit data reuse. Hence, the compiler now formulates and solves an optimization problem to maximize the number of running threads in the whole GPU.

Par4All [65] uses abstract interpretation for array regions, which also involves polyhedra. It treats each loop nest independently, generating a CUDA kernel for each one. Par4All does not consider the exploitation of reuse in the registers or the shared memory: all accesses are performed directly on the global memory. However, it performs powerful inter-procedural analysis on the input code.

Khan et al. [79] propose a script-based compiler framework that emits *transformation recipes* with a sequence of composable transformations written in CUDA-CHiLL (a polyhedral framework for code transformations of complex loop nests). The optimization search space is pruned to generate a reduced set of CUDA variants, which are executed in an autotuning phase to select the best-performing one. The compiler exploits the full GPU memory hierarchy, including texture and constant memories. Authors claim that this layered system facilitates savvy developers to intervene in optimization by writing their own CUDA-CHiLL scripts. Our approach, which generates code annotated with OpenHMPP standard directives, also allows this interaction.

As mentioned in Section 2.4, TRACO [108] targets affine loop nests by building the transitive closure of a relation which describes all the dependences of the loop. In addition to OpenMP, it can generate OpenACC annotated code too.

Jablin et al. [71, 72] propose a framework that automatically generates pipeline parallelizations and provides software-only shared memory. The memory allocation system ensures that addresses of equivalent allocation units on the CPU and GPU are equal, relieving the runtime library of the burden of translation and communication optimization. The compiler inserts appropriate calls into the original program. The pipeline parallelization technique exploits the fact that GPUs have abundant parallel computing resources but communication between them can be very expensive. If the loaded values were constant, each of the threads could execute the load redundantly, reducing communication overhead at the expense of computational efficiency.

Previous work has also targeted the optimization of the GPU source code in a similar way as we do with the different variants of the test cases. From a naïve kernel, Yang et al. [142] generate two kernels, one optimized for global memories and other for texture memories. The naïve variant is said to be easy to extract from the sequential CPU code: authors propose a simplified GPU hardware abstraction model which consists of an array of independent processors connected to the off-chip memory. Thus, in most cases, the naïve kernel will correspond to the computation of one element/pixel in the output matrix/image. To facilitate compiler optimizations, the user can convey the size of the input and output dimensions, and the output variable names. The proposed compiler addresses coalesced memory accesses and the use of shared memory. Authors claim that it is relatively easy to understand what/how code transformations have been applied because the compiler generates CUDA and OpenCL code in each compilation step.

In summary, most approaches partially exploit the GPU memory hierarchy and generate low-level, difficult to understand CUDA code. In contrast, our proposal based on OpenHMPP directives provides portable and understandable code, which eases the interaction between programmers and application-domain experts. Additionally, with the inclusion of auto-tuning techniques [57], OpenHMPP has demonstrated to be able to obtain even better performance than hand-coded CUDA/OpenCL codes.

3.8 Concluding Remarks

This chapter has introduced a new KIR-based locality-aware automatic parallelization technique that targets GPU-based heterogeneous systems. Our proposal is devoted to exploit locality in the complex GPU memory hierarchy in order to generate efficient code. It takes into account the most impacting GPU programming features: loop threadification, thread grouping, coalesced access to global memory, and maximum usage of registers and shared memory. We have modeled the accesses to n-dimensional arrays with chains of recurrences. This algebraic formalism allowed us to analyze the interactions between the memory accesses performed by the GPU threads in a loop nest. We have successfully applied this technique to two representative case studies extracted from compute-intensive scientific applications (namely, CONV3D, the three-dimensional convolution, and SGEMM, the simple-precision general matrix multiplication). The usage of OpenHMPP directives enabled a great understandability and portability of the generated GPU code. The performance evaluation on NVIDIA GPUs (with two different core architectures) has corroborated the effectiveness of our approach.

The technology developed in this chapter has been licensed to the spin-off Appentra Solutions S.L. for the creation of Parallware [15].

Chapter 4

Trace-based Affine Reconstruction of Code

Many static and dynamic compiler optimization techniques rely on the knowledge of the application code to work, as the ones presented in Chapters 2 and 3. Unfortunately, the source code is not always available to the compiler. In embedded systems, for example, it is common to find intellectual property (IP) cores with well defined high level functionality, but whose internals are opaque to the system designer and programmer. Even when source code is available, programmers may use complex data and control structures, including code obfuscation techniques, that mask the underlying application logic and prevent static analysis and optimization. In these cases, locality exploitation becomes key to enabling good performance (see Section 1.2.2).

This chapter presents a novel mathematical framework for *automatically reconstructing* affine loops from a trace of their memory accesses, without user intervention or access to source codes or application binaries. The proposal is based on the observation that, in affine loops, access strides must be constructed as linear combinations of loop indices. Affine codes represent an important class of problems in many computing domains, such as supercomputing, embedded systems, or multimedia applications. For the most part, these codes execute large regular loops, with static control parts that depend only on the loop index variable values through affine bounds and subscripts, accessing and operating on large arrays of data. This is the type of codes that is usually modeled and optimized using the polyhedral approach [75, 83, 54, 27] (see Appendix B). Also note that some irregular accesses in compile time are affine in run time. For instance, Gómez-Sousa et al. [60] presented an implementation of the method of moments technique which may lead to unpredictable race-conditions when parallelized. However, with knowledge about the input data, it can be assured that these race-conditions are not present. Hence, the proposal developed in this chapter can improve the performance of the automatic parallelizations presented in Chapters 2 and 3. In addition, we have developed extensions for handling imperfect and nonlinear traces to allow for extra or missing points. Furthermore, we have reconstructed automatically parallelized affine codes, which typically include moderate amounts of nonlinearity, modifying the basic algorithm to process the memory trace in a piecewise fashion.

The remainder of this chapter is organized as follows. Section 4.1 details our proposal. Section 4.2 illustrates the operation of our approach with the Cholesky decomposition, a popular linear algebra routine. Section 4.3 describes the extensions to handle imperfect and nonlinear traces, including parallel codes. Section 4.4 presents the experimental results. Section 4.5 discusses practical applications of our proposal and the related work. Finally, Section 4.6 summarizes the main conclusions of the chapter.

4.1 Trace-based Reconstruction

This section introduces our method for reconstructing affine loops from their trace of memory accesses. Section 4.1.1 formulates the problem mathematically. Section 4.1.2 gives an overview of the algorithm, which traverses the solution space until finding the minimal (in terms of number of nested loops) representation of the trace. Section 4.1.3 focuses on heuristics to accelerate the traversal of the solution space. Section 4.1.4 details the pseudo-code of the approach. And, finally, Section 4.1.5 presents a mechanism to increase the dimensionality of the minimal loop nest found by our method.

4.1.1 Mathematical Formulation

The proposed reconstruction algorithm deals with the stream of addresses generated by a single memory instruction (i.e., we focus on one reference at a time). Hence, we assume that the trace contains at least the memory address of the instruction issuing the access (or a similar way to uniquely identify the instruction), and the accessed location. This memory trace format can be generated, for instance, by Intel Pin [93]. In the general case, it is expected that a trace file will contain the entire execution of the program, including multiple loop nests and non-loop sections. Detection of loop sections in execution traces falls out of the scope of this thesis, but it has been discussed in previous work [80, 96]. Thus, a reliable mechanism to detect and extract loop sections in the trace is assumed.

Our proposal is designed to recreate the same sequence of accesses that the memory trace contains. Hence, we model the memory access to be reconstructed as:

```
DO $i_1$ = 0, $u_1(\vec{\imath})$
 DO $i_2$ = 0, $u_2(\vec{\imath})$
   ⋮
   DO $i_n$ = 0, $u_n(\vec{\imath})$
    $V[f_1(\vec{\imath})] \ldots [f_m(\vec{\imath})]$
```

where $\{u_j, 0 < j \leq n\}$ are affine functions that provide the upper bounds of loop i_j; $\{f_d(i_1, \ldots, i_n), 0 < d \leq m\}$ is the set of affine functions that converts a given point in the iteration space of the nest to a point in the data space of V; and $\vec{\imath} = \{i_1, \ldots, i_n\}^T$ is a column vector which encodes the state of each iteration variable. Note that, if the original access is not affine, it can be modeled in the worst case as one loop (with one iteration and the corresponding stride) per entry in the trace. Section 4.3 will present smarter ways to handle irregularities.

From now on, the particular set of index values for the k^{th} execution of the access to V is denoted by $\vec{\imath}^k = \{i_1^k, \ldots, i_n^k\}^T$; and the complete access $V[f_1(\vec{\imath})] \ldots [f_m(\vec{\imath})]$ is abbreviated by $V(\vec{\imath})$. Note that each upper bound function $u_j(\vec{\imath})$ can only depend on scoped variables at the nesting level j, i.e., $\{i_1, \ldots, i_{j-1}\}$. This is not explicitly acknowledged to simplify notation. Iteration bounds are assumed to be inclusive, i.e., $0 \leq i_j \leq u_j(\overrightarrow{\imath})$.

Since f_j is affine, the access can be rewritten as:

$$V[f_1(\overrightarrow{\imath})] \ldots [f_m(\overrightarrow{\imath})] = V[c_0 + i_1 c_1 + \ldots + i_n c_n] \tag{4.1}$$

where V is the base address of the array, c_0 is a constant stride, and each $\{c_j, 0 < j \leq n\}$ is the *coefficient* of the loop index i_j, and it must account for the dimensionality of the original array. For instance, an access $A[2 * i][j]$ to an array $A[N][M]$ can be rewritten as $A[(2 * M) * i + j]$, where $c_i = 2M$ accounts for both the constant multiplying i in the original access (2), and the size of the fastest changing dimension (M). This is the *canonical form* into which the method proposed in this thesis reconstructs the loop.

During the execution of the loop nest, the instruction which implements the access to V will orderly issue the addresses corresponding to $V(\overrightarrow{\imath}^1)$, $V(\overrightarrow{\imath}^2)$, $V(\overrightarrow{\imath}^3)$, etc. Consider two consecutive accesses, $V(\overrightarrow{\imath}^k)$ and $V(\overrightarrow{\imath}^{k+1})$, and assume that the loop index values in $\overrightarrow{\imath}^k = \{i_1^k, \ldots, i_n^k\}$ and the upper bounds functions, $u_1(\overrightarrow{\imath}), \ldots, u_n(\overrightarrow{\imath})$ are known. The values in $\overrightarrow{\imath}^{k+1}$ can be calculated as follows:

1. An index i_j will be reset to 0 if and only if all of the following hold:

    - All inner indices are resetting.
    - Either i_j has reached its maximum iteration count, or some inner index has a negative value for its maximum iteration count when i_j increases by one:

    $$(i_j = u_j(\overrightarrow{\imath}^k)) \vee \Big(\exists l, j < l \leq n; u_l(\ldots, i_j^k + 1, \ldots) < 0\Big)$$

2. An index i_j will be increased by one if and only if all of the following hold:

    - All inner indices are resetting.
    - i_j has not reached its maximum iteration count, and all inner indices have non-negative values for their maximum iteration count when i_j increases by one:

$$(i_j < u_j(\overrightarrow{\imath}^k)) \wedge \left(\forall l, j < l \leq n; u_l(\ldots, i_j^k + 1, \ldots) \geq 0\right)$$

3. In any other case, i_j will not change.

These conditions are intuitive and a direct consequence of loop semantics and application control flow. If any internal index $(i_l, j < l \leq n)$ is not resetting, then control flow will not exit the loop at level l, and therefore it will be impossible for i_j to be modified. If all internal indices reset, then control flow will reach the post-loop section of loop at level j, increasing i_j by one unit. If $i_j^k = u_j(\overrightarrow{\imath}^k)$, then this increase will cause the index to go beyond its maximum iteration count, and control flow will exit level j. If there is an iteration $(k+1)$, then control flow must re-enter level j later, executing the pre-loop instruction and assigning $i_j = 0$. If $i_j^k < u_j(\overrightarrow{\imath}^k)$ but there is some inner level l such that its maximum iteration count takes a negative value when i_j is increased by one unit, then control flow will not enter level l, will not reach V, and no memory access may be executed until i_j resets to 0. In all other case, the next access to V will be performed in iteration $\overrightarrow{\imath}^{k+1} = \{i_1^k \ldots, i_j^k + 1, 0, \ldots, 0\}$.

Definition 4.1.1. *A set of indices built complying with these conditions will be referred to as a set of sequential indices.*

The instantaneous variation of loop index i_j between iterations k and $(k+1)$, $\delta_j^k = (i_j^{k+1} - i_j^k)$, can only take one of three possible values:

1. i_j does not change $\Rightarrow \delta_j^k = 0$
2. i_j is increased by one $\Rightarrow \delta_j^k = 1$
3. i_j is reset to 0 $\Rightarrow \delta_j^k = -i_j^k$

In the following, vector notation will be used for δ:

$$(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^k) = \begin{bmatrix} i_1^{k+1} - i_1^k \\ i_2^{k+1} - i_2^k \\ \vdots \\ i_n^{k+1} - i_n^k \end{bmatrix} = \begin{bmatrix} \delta_1^k \\ \delta_2^k \\ \vdots \\ \delta_n^k \end{bmatrix} = \overrightarrow{\delta}^k$$

Lemma 4.1.2. *The stride between two consecutive accesses* $\sigma^k = V(\overrightarrow{\imath}^{k+1}) - V(\overrightarrow{\imath}^{k})$ *is a linear combination of the coefficients of the loop indices.*

Proof. Using Equation (4.1), σ^k can be rewritten as:

$$\begin{array}{rlllll} \sigma^k = & V + (c_0 + & c_1 i_1^{k+1} & + \ldots + & c_n i_n^{k+1}) & - \\ & V + (c_0 + & c_1 i_1^{k} & + \ldots + & c_n i_n^{k}) & = \\ = & & c_1 \delta_1^k & + \ldots + & c_n \delta_n^k & = \\ = & & & & \overrightarrow{c}\,\overrightarrow{\delta}^{k} & \end{array}$$

□

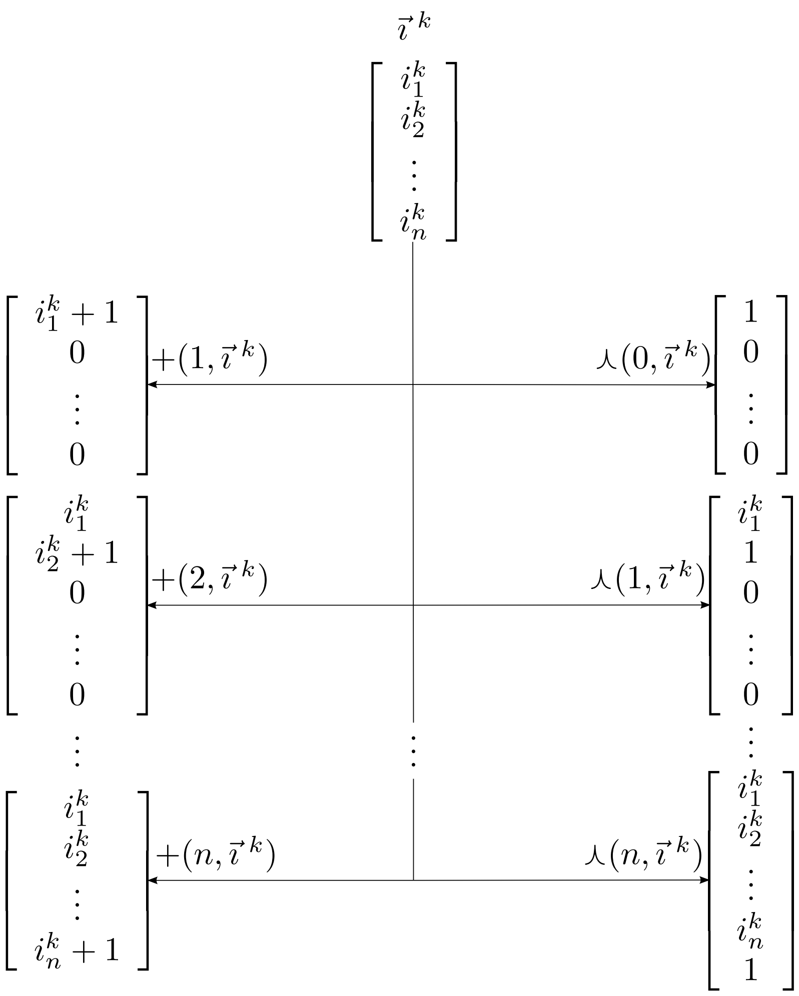
4.1.2 Reconstruction Algorithm

The proposed algorithm is essentially a guided exploration of the potential solution space, driven by the first-order differences of the addresses accessed by a given instruction (the access strides). Each node in this tree-like space represents a point in the iteration space of the loop. Its root is a trivial loop that generates the first two accesses in the trace. Children of a node in the tree are the indices that can immediately follow the parent in the iteration space. Starting from the root, an exploration engine begins incorporating one access to the reconstructed loop in each step, descending one level into the tree, until it finds a solution for the entire trace or determines that no affine loop is capable of generating the observed sequence of accesses.

Each step of the process is conceptually depicted in Figure 4.1. Starting from the k^{th} iteration vector $\overrightarrow{\imath}^{k} = \{i_1^k, \ldots, i_n^k\}$ there are $(2n+1)$ different vectors $\overrightarrow{\imath}^{k+1}$ that are considered as candidates for the $(k+1)^{th}$ iteration vector. The n alternatives on the left side are obtained using an operation $+(j, \overrightarrow{\imath})$, which increases index i_j by one and resets to zero all inner indices. The $(n+1)$ alternatives on the right are obtained by applying an operation $\curlywedge(j, \overrightarrow{\imath})$[1], which inserts a new loop at nesting level $(j+1)$.

If a solution exists, the algorithm builds the minimal nest (in terms of the

[1]Read $\curlywedge$ as *ampheck*



$\vec{\imath}^{\,k}$
$[i_1^k, i_2^k, \dots, i_n^k]$
$+(1,\vec{\imath}^{\,k})$
$\curlywedge(0,\vec{\imath}^{\,k})$
$[i_1^k+1, 0, \dots, 0]$
$[1, 0, \dots, 0]$
$+(2,\vec{\imath}^{\,k})$
$\curlywedge(1,\vec{\imath}^{\,k})$
$[i_1^k, i_2^k+1, 0, \dots, 0]$
$[i_1^k, 1, 0, \dots, 0]$
$+(n,\vec{\imath}^{\,k})$
$\curlywedge(n,\vec{\imath}^{\,k})$
$[i_1^k, i_2^k, \dots, i_n^k+1]$
$[i_1^k, i_2^k, \dots, i_n^k, 1]$

Figure 4.1 – Solution space. For each reconstructed index $\vec{\imath}^{k}$, there are $(2n+1)$ possible values for $\vec{\imath}^{k+1}$. The n alternatives on the left side are obtained using an operation $+(j, \vec{\imath})$, which increases index i_j by one and resets to zero all inner indices. The $(n+1)$ alternatives on the right are obtained by applying an operation $\curlywedge(j, \vec{\imath})$, which inserts a new loop at nesting level $(j+1)$. For instance, if $\vec{\imath}^{k} = [3,5,7]$, there are 7 alternatives for $\vec{\imath}^{k+1}$: $+(1, \vec{\imath}^{k}) = [\mathbf{4},0,0]$, $+(2, \vec{\imath}^{k}) = [3,\mathbf{6},0]$, $+(3, \vec{\imath}^{k}) = [3,5,\mathbf{8}]$, $\curlywedge(0, \vec{\imath}^{k}) = [\mathbf{1},0,0,0]$, $\curlywedge(1, \vec{\imath}^{k}) = [3,\mathbf{1},0,0]$, $\curlywedge(2, \vec{\imath}^{k}) = [3,5,\mathbf{1},0]$, and $\curlywedge(3, \vec{\imath}^{k}) = [3,5,7,\mathbf{1}]$.

number of nested loops) capable of generating the whole observed access trace. For example, a 2-level loop with indices i and j might iterate sequentially over all the elements in array $A[N][M]$ if the upper bounds are defined as $u_i = N$, $u_j = M$ and the access is $V[i * M + j]$. This can be rewritten as an equivalent 1-level loop with index i, using $u_i = N * M$ and access $V[i]$. Section 4.1.5 will detail a mechanism to increase the dimensionality of the resulting nest.

Let $\overrightarrow{a} = \{a_1, \ldots, a_N\} = \{V(\overrightarrow{\imath}^1), \ldots, V(\overrightarrow{\imath}^N)\}$ be the addresses generated by a single instruction, included in the execution trace. Since the upper bounds functions are affine, each $u_j(\overrightarrow{\imath})$ can be written as:

$$u_j(\overrightarrow{\imath}) = w_j + u_{j,1}i_1 + \ldots + u_{j,(j-1)}i_{(j-1)} \tag{4.2}$$

and therefore it is possible to build a matrix $\mathbf{U} \in \mathbb{Z}^{n \times n}$ and a column vector $\overrightarrow{w} \in \mathbb{Z}^n$ such that:

$$\mathbf{U} = \begin{bmatrix} -1 & 0 & 0 & \ldots & 0 \\ u_{2,1} & -1 & 0 & \ldots & 0 \\ u_{3,1} & u_{3,2} & -1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_{n,1} & u_{n,2} & u_{n,3} & \ldots & -1 \end{bmatrix} \text{, and } \overrightarrow{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \tag{4.3}$$

Note that $\mathbf{U}$ is a lower triangular matrix, since no index i_j can depend on an inner index; and that its main diagonal is equal to $\overrightarrow{-1} \in \mathbb{Z}^n$. Using $\mathbf{U}$ and $\overrightarrow{w}$, the condition for a given iteration tuple $\overrightarrow{\imath}$ to be valid under the loop constraints in the canonical loop form can be written as:

$$\mathbf{U}\overrightarrow{\imath} + \overrightarrow{w} \geq \overrightarrow{0}^T \tag{4.4}$$

Let us assume that the algorithm has already identified a partial solution $\mathcal{S}_n^k = \{\overrightarrow{c}, \mathbf{I}^k, \mathbf{U}, \overrightarrow{w}\}$, which reconstructs the subtrace $\{a_1, \ldots, a_k\}$ using n nested loops, whose components are defined as follows:

- Vector $\overrightarrow{c} \in \mathbb{Z}^n$ of coefficients of loop indices.
- Matrix $\mathbf{I}^k = [\overrightarrow{\imath}^1 | \ldots | \overrightarrow{\imath}^k] \in \mathbb{Z}^{n \times k}$ of reconstructed indices.

- Matrix $\mathbf{U} \in \mathbb{Z}^{n\times n}$, bounds matrix as defined in Equation (4.3).
- Vector $\overrightarrow{w} \in \mathbb{Z}^n$, bounds vector as defined in Equation (4.3).

To be a valid solution, $\mathcal{S}_n^k$ has to meet the following requirements:

1. Each consecutive pair of indices $\overrightarrow{\imath}^k$ and $\overrightarrow{\imath}^{k+1}$ must be sequential as per Definition 4.1.1.

   Note that this condition is stronger than simply requiring that the iteration indices stay inside the loop bounds, which could be written extending Equation (4.4) as:

   $$\mathbf{U}\mathbf{I}^k + \overrightarrow{w}\mathbf{1}^{1\times k} \geq \mathbf{0}^{n\times k} \tag{4.5}$$

2. The observed strides are coherent with the reconstructed ones. Using Lemma 4.1.2 this can be expressed as:

   $$\overrightarrow{c}\left(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^k\right) = \overrightarrow{c}\,\overrightarrow{\delta}^k = \sigma^k$$

Upon processing access a_{k+1}, the algorithm first calculates the observed stride:

$$\sigma^k = a_{k+1} - a_k \tag{4.6}$$

Afterwards, it builds a diophantine[2] linear equation system based on Lemma 4.1.2 to discover the potential indices $\overrightarrow{\imath}^{k+1}$ which generate an access stride that is equal to the observed one:

$$\overrightarrow{c}\left(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^k\right) = \sigma^k \Rightarrow \left(\overrightarrow{c}^T\overrightarrow{c}\right)\overrightarrow{\delta}^k = \overrightarrow{c}^T\sigma^k \tag{4.7}$$

where $\left(\overrightarrow{c}^T\overrightarrow{c}\right) \in \mathbb{Z}^{n\times n}$ is the system matrix, and $\overrightarrow{\delta}^k \in \mathbb{Z}^n$ is the solution. There are two possible situations when solving this system:

1. The system has one or more integer solutions. In this case, for each solution $\overrightarrow{\delta}^k$, the new index $\overrightarrow{\imath}^{k+1} = \overrightarrow{\imath}^k + \overrightarrow{\delta}^k$ is calculated, and $\mathbf{I}^{k+1} = \left[\mathbf{I}^k | \overrightarrow{\imath}^{k+1}\right]$. $\mathbf{U}$, $\overrightarrow{w}$, and $\overrightarrow{c}$ remain unchanged. Each of these solutions must be explored independently.

[2]The system must be diophantine, as loop indices may only have integer values.

2. The system has no solution, in which case there are three courses of action:

   2.1. Modify the boundary conditions imposed by $\mathbf{U}$ and $\overrightarrow{w}$.

   2.2. Increase the dimensionality of the solution: compute $\mathcal{S}_{n+1}^{k+1}$ modeling a loop with $(n+1)$ nesting levels.

   2.3. Discard this branch.

Section 4.1.3 describes heuristic methods to guide the search through the solution space to accelerate the traversal.

Solving the Linear Diophantine System

Although the system in Equation (4.7) has infinite solutions in the general case, only a few are valid in the context of the affine loop reconstruction, which makes it possible to develop ad-hoc solving strategies.

Lemma 4.1.3. *There are at most n valid solutions to the system in Equation (4.7). These correspond to indices:*

$$\{\overrightarrow{\imath}_l^{k+1} = +(l, \overrightarrow{\imath}^k), 0 < l \leq n\}$$

Proof. If index $\overrightarrow{\imath}^{k+1}$ must be sequential to index $\overrightarrow{\imath}^k$ as per Definition 4.1.1, then there is a single degree of freedom for $\overrightarrow{\delta}^k$: the position δ_l^k that is equal to 1.

$$\begin{bmatrix} \delta_1^k \\ \vdots \\ \delta_{l-1}^k \\ \delta_l^k \\ \delta_{l+1}^k \\ \vdots \\ \delta_n^k \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ -i_{l+1}^k \\ \vdots \\ -i_n^k \end{bmatrix} \tag{4.8}$$

Positions $\{i_j, 0 < j < l\}$ will not change between iterations k and $(k+1)$, and therefore $\delta_j^k = 0$; while positions $\{i_j, l < j \leq n\}$ will be reset to 0, and therefore $\delta_j^k = -i_j^k$. □

Taking this result into account, it is possible to find all valid solutions of the system in linear time ($O(n)$) by simply testing the n valid indices $\overrightarrow{\imath}_l^{k+1}$, calculating the predicted stride for each combination as $\hat{\sigma}_l^k = \overrightarrow{c}\,\overrightarrow{\delta}_l^k$, and accepting those solutions that generate a stride equal to the observed one ($\hat{\sigma}_l^k = \sigma^k$, obtained using Equation (4.6)). These will be particular solutions of the subtrace $\{a_1, \ldots, a_{k+1}\}$, which must be explored to construct a solution for the entire trace.

4.1.3 Exploration of the Solution Space

Branch Priority

The approach proposed in the previous section is capable of efficiently finding the relevant solutions of the linear diophantine system for each address of the trace, but can still produce a large number of potential solutions that will be discarded when processing the remaining addresses in the trace. In the general case, the time for exploring the entire solution space of a trace containing N addresses generated by n loops would be $O(n^N)$. Consequently, exploring all branches with no particular order could take a very long time. In order to guide the traversal of the solution space, consider the column vector $\overrightarrow{\gamma}^k \in \mathbb{Z}^n$ defined as:

$$\overrightarrow{\gamma}^k = \mathbf{U}\overrightarrow{\imath}^k + \overrightarrow{w} \tag{4.9}$$

Lemma 4.1.4. *Each element $\gamma_j^k \in \overrightarrow{\gamma}^k$ indicates how many more iterations of index i_j are left before it resets under bounds $\mathbf{U}$ and $\overrightarrow{w}$.*

Proof. γ_j^k is equal to the value of the upper bound of the loop in i_j, defined in Equation (4.2), minus the current value of i_j:

$$\gamma_j^k = \mathbf{U}_{(j,:)}\overrightarrow{\imath}^k + w_j = w_j + u_{j,1}i_1 + \ldots + u_{j,(j-1)}i_{(j-1)} - i_j =$$

$$= u_j(\overrightarrow{\imath}) - i_j$$

where $\mathbf{U}_{(j,:)}$ denotes the j^{th} row of matrix $\mathbf{U}$. By construction of the canonical loop form, the step of all loops is 1. Therefore, γ_j^k is equal to the number of iterations of loop i_j before $i_j > u_j(\overrightarrow{\imath})$. □

This result suggests that, assuming that $\mathbf{U}$ and $\overrightarrow{w}$ are accurate, the most plausible value for the next index is $\overrightarrow{\imath}_l^{k+1} = +(l, \overrightarrow{\imath}^k)$, where l is the position of the innermost positive element of $\overrightarrow{\gamma}^k$.

The correctness of $\overrightarrow{\imath}_l^{k+1}$ can be assessed by comparing the predicted stride $\hat{\sigma}_l^k$ with the observed one σ^k. Note that using $\overrightarrow{\gamma}^k$ as described above guarantees consistency with the boundary conditions in Equation (4.4), which further improves the efficiency of the approach by saving calculations.

In order to reduce the number and size of matrix multiplications, $\overrightarrow{\gamma}^{k+1}$ can be calculated from $\overrightarrow{\gamma}^k$ as follows:

$$\overrightarrow{\gamma}^{k+1} = \left[\gamma_0^k, \dots, \gamma_{l-1}^k, \gamma_l^k - 1, u_{l+1}(\overrightarrow{\imath}_l^{k+1}), \dots, u_n(\overrightarrow{\imath}_l^{k+1})\right]^T$$

Extracting the Loop Bounds

So far it has been assumed that the boundary conditions, $\mathbf{U}$ and $\overrightarrow{w}$, can be used to correctly predict $\overrightarrow{\imath}^{k+1}$ from $\overrightarrow{\imath}^k$. This is not true in the general case, as initially the loop bounds are unknown, as are the number of loops involved in the execution of the instruction accessing V.

As before, assume that the algorithm has already identified a partial solution $\mathcal{S}_n^k = \{\overrightarrow{c}, \mathbf{I}^k, \mathbf{U}, \overrightarrow{w}\}$. Upon processing access a_{k+1}, the algorithm will try to explore the branch which increments the index i_l corresponding to the innermost positive element of $\overrightarrow{\gamma}^k$. However, it might happen that the calculated stride for the selected branch does not match the observed stride (i.e., $\hat{\sigma}_l^k \neq \sigma^k$). A different loop index $i_{l'}$ will have to be selected as described in Section 4.1.2, but the constructed $\mathbf{I}^{k+1}$ will not be valid in the context of the extracted loop bounds, $\mathbf{U}$ and $\overrightarrow{w}$, because either $\overrightarrow{\imath}_{l'}^{k+1}$ will not be sequential to $\overrightarrow{\imath}^k$, or it will violate boundary conditions. In this scenario, it is necessary to generate new boundary conditions $\mathbf{U}'$ and $\overrightarrow{w}'$. These can be found by solving the system in Equation (4.5):

$$\mathbf{U}'\mathbf{I}^{k+1} + \overrightarrow{w}'\mathbf{1}^{1\times(k+1)} \geq \mathbf{0}^{n\times(k+1)} \tag{4.10}$$

If the system is inconsistent, then the generated iteration space is not a polytope and the solution is not valid. If the system has solutions, then it will be overdetermined in the general case. Matrix $\mathbf{U}'$ and vector $\overrightarrow{w}'$ are only partially unknown: the only rows that may vary with respect to $\mathbf{U}$ and $\overrightarrow{w}$ are those corresponding to loop indices $\{i_j, l \leq j \leq n\}$, since the outer variables cannot be affected by the inner, unscoped ones. As such, their first $(l-1)$ rows are known. Besides, in order for the indices to be sequential, it is necessary to build $\mathbf{U}'$ and $\overrightarrow{w}'$ such that loop resets as predicted by $\overrightarrow{\gamma}$ are consistent with loop resets observed in $\mathbf{I}^{k+1}$.

First, $\overrightarrow{w}'$ is calculated. The first $(l-1)$ positions are already known and are the same as those in $\overrightarrow{w}$. To calculate the remaining positions $\{w'_j, l \leq j \leq n\} \in \overrightarrow{w}'$, consider the reduced system:

$$\mathbf{U}'_{(j,:)} \overrightarrow{\imath}^z + w'_j \geq 0 \Rightarrow \sum_{r=1}^{j} u'_{j,r} i^z_r + w'_j \geq 0 \tag{4.11}$$

where $\mathbf{U}'_{(j,:)}$ is currently unknown, and $\overrightarrow{\imath}^z \in \mathbf{I}^{k+1}$ is arbitrarily selected. In order to calculate w'_j, it is possible to take advantage of the properties of the canonical loop form and choose an $\overrightarrow{\imath}^z$ such that:

$$\overrightarrow{\imath}^z = \left[0, \ldots, 0, i^z_j, \ldots, i^z_n\right]^T$$

Since every loop index must start at 0 and by the sequential construction of the columns of $\mathbf{I}^{k+1}$, such an i^z is guaranteed to exist. Replacing it in the previous equation and taking into account that the main diagonal of $\mathbf{U}'$ must be equal to $\overrightarrow{-1} \in \mathbb{Z}^n$:

$$\sum_{r=1}^{j} u'_{j,r} i^z_r + w'_j \geq 0 \Rightarrow u'_{j,j} i^z_j + w'_j \geq 0 \Rightarrow w'_j \geq i^z_j$$

Lemma 4.1.5. *In order to guarantee that the bounds conditions in Equation (4.5) hold, $\overrightarrow{\imath}^z$ must be chosen out of all the possible candidates such that i^z_j is maximum, and w'_j must be equal to i^z_j.*

Proof. If w'_j was not selected to be equal to some i^z_j, then $\mathbf{I}^{k+1}$ would not be sequential as per Definition 4.1.1 under the boundary conditions established by $\overrightarrow{w}'$. Now, assume that an $\overrightarrow{\imath}^{z'}$ is selected such that $i^{z'}_j$ is not maximum (i.e., $i^z_j > i^{z'}_j$). Then:

$$\mathbf{U}_{(j,:)} \overrightarrow{\imath}^z + w'_j = -i^z_j + i^{z'}_j < 0$$

and the constructed $\overrightarrow{w}'$ would not be consistent with some of the entries in $\mathbf{I}^{k+1}$. $\square$

Corollary 4.1.6. *In Equation (4.11), it is only necessary to calculate the value w'_l, as other elements of $\overrightarrow{w}$ will remain unchanged. Moreover, w'_l will only change if $(\forall j, 0 < j < l, i_j^{k+1} = 0)$ and, in that case, $w'_l = i_l^{k+1}$.*

$$\overrightarrow{w}' = \left[w_1, \ldots, w_{l-1}, \boldsymbol{w'_l}, w_{l+1}, \ldots, w_l\right]^T$$

Proof. If $\{w'_j, l < j \leq n\}$ can be calculated exclusively selecting vectors in the shape of $\overrightarrow{\imath}^z$ as per Lemma 4.1.5, then index $\overrightarrow{\imath}_l^{k+1} = +(l, \overrightarrow{\imath}^k)$ is not a feasible selection for $\overrightarrow{\imath}^z$ when calculating w'_j (since $i_l^{k+1} > 0$ by the definition of the $+$ operation). Therefore, w'_j will be equal to the one calculated for the previous step of the algorithm using $\mathbf{I}^k$. Using the same reasoning, if $(\exists j, 0 < j < l, i_j^{k+1} \neq 0)$, index $\overrightarrow{\imath}_l^{k+1}$ is not a feasible selection for calculating w'_l. Otherwise, and by the definition of the $+$ operation and index sequentiality, $i_l^{k+1} = i_l^z$ will be maximum. $\square$

Using this result, the calculation of $\overrightarrow{w}'$ has a complexity of $O(1)$. Once $\overrightarrow{w}'$ is computed, the unknown rows $\{\mathbf{U}'_{(j,:)}, l \leq j \leq n\}$ can be calculated by reducing the original system in Equation (4.10) to $(n - l + 1)$ equation systems of the form:

$$\mathbf{U}'_{(j,:)}\mathbf{i}^z + w'_j\mathbf{1}^{1\times n} = \mathbf{0}^{1\times n}$$

where $\mathbf{i}^z \in \mathbb{Z}^{n\times n}$ is a full rank matrix of columns extracted from $\mathbf{I}^{k+1}$. As established in Lemma 4.1.5, it is necessary to choose $\mathbf{i}^z = \{\overrightarrow{\imath}_1^z, \ldots, \overrightarrow{\imath}_n^z\}$ such that each of its columns represents an iteration where index i_j is maximum for a specific combination of indices $(i_0, \ldots, i_{j-1})$. Note that the inequality in Equation (4.10) has been changed to ensure that $\gamma_j = u_j(\overrightarrow{\imath}) = 0$ will hold for each of the selected iterations, guaranteeing index consistency.

In order to efficiently solve these systems, two optimizations can be considered. First, since $\mathbf{U}'$ must be a lower triangular matrix with known main diagonal, the previous system can be reduced to:

$$\mathbf{U}'_{(j,1:j)}\mathbf{i}^z_{(1:j,1:j-1)} + w'_j\mathbf{1}^{1\times(j-1)} = \mathbf{0}^{1\times(j-1)} \tag{4.12}$$

where $\mathbf{U}'_{(j,1:j)} \in \mathbb{Z}^{1\times j}$ denotes the first j entries of the j^{th} row of $\mathbf{U}'$, and $\mathbf{i}^z_{(1:j,1:j-1)} \in \mathbb{Z}^{j\times(j-1)}$ denotes the first $(j-1)$ entries in the first j rows of matrix $\mathbf{i}^z$. Only $(j-1)$ indexes are needed, as that is the number of unknowns in the j^{th} row of $\mathbf{U}'$. Second, note that any full rank matrix can be extracted from $\mathbf{I}^{k+1}$ to build $\mathbf{i}^z$ as long as the selected columns are iterations where index i_j is maximum. By taking advantage of the canonical loop form, this means that it is always possible to build $\mathbf{i}^z$ as a triangular matrix, and solve the system in linear time $O(j)$. By applying both optimizations, the complexity of the calculation of $\mathbf{U}'$ becomes $O(n^2)$.

Extracting the Coefficients of the Loop Indices

Once again, assume that the algorithm has found a partial solution $\mathcal{S}^k_n = \{\overrightarrow{c}, \mathbf{I}^k, \mathbf{U}, \overrightarrow{w}\}$. If no valid $\{\overrightarrow{\imath}^{k+1}_l = +(l, \overrightarrow{\imath}^k), 0 < l \leq n\}$ can be built using the methods described until now in this section, it may be caused by a loop index increasing in access $(k+1)$ which had not appeared before. This can cause σ^k to be unrepresentable either as a linear combination of the currently known coefficients in $\overrightarrow{c}$, or as a set of sequential indices $\mathbf{I}^{k+1}$. Assuming that the first k accesses have been correctly recognized, it is possible to generate a valid partial solution $\mathcal{S}^{k+1}_{n+1}$ from $\mathcal{S}^k_n$ by enlarging the dimensionality of the current solution components. There are $(n+1)$ potential solutions that need to be explored (as shown in the right half of Figure 4.1), one for each insertion position of the newly discovered index. The most common situation, particularly for large values of k, is that the newly discovered loops are outer than the previously known ones. In any case, given an insertion point $(p, 0 \leq p \leq n)$ for the new loop index i_p, the set of indices generated, $\mathbf{I}^{k+1} \in \mathbb{Z}^{(n+1)\times(k+1)}$, is as follows:

$$\mathbf{I}^{k+1} = \left[\begin{array}{c|c} \mathbf{I}^k_{(1:p,:)} & \\ \mathbf{0}^{1\times k} & \overrightarrow{\imath}^{k+1} \\ \mathbf{I}^k_{(p+1:n,:)} & \end{array}\right]$$

where a 0 in position p has been added to each index $\overrightarrow{\imath} \in \mathbf{I}^k$, and a new column $\overrightarrow{\imath}^{k+1} = \curlywedge(p, \overrightarrow{\imath}^k)$ has been added to the matrix. The coefficient c'_p associated with the new loop index can be derived from Equation (4.7):

$$\overrightarrow{c}\,(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^{k}) = \sigma^k \Rightarrow$$

$$\left[c_1, \ldots, c_p, \boldsymbol{c'_p}, c_{p+1}, \ldots, c_n\right] \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ -i^k_p \\ \vdots \\ -i^k_n \end{bmatrix} = \sigma^k \Rightarrow$$

$$c'_p = \sigma^k + \sum_{r=p+1}^{n} i^k_r c_r$$

After calculating the new $\overrightarrow{c}$, $\mathbf{U}$ and $\overrightarrow{w}$ are updated as described previously in this section to reflect any new information available. If no solution is found for the boundary conditions, then this branch is discarded. Note that there must be a practical limit to the maximum acceptable solution size, as in the general case any trace $\{a_1, \ldots, a_N\}$ can be generated using at most N affine nested loops. For this reason, the solution space should be traversed in a breadth-first fashion, to ensure that a minimal solution (in terms of number of generated nested loops) is reached.

Starting the Exploration

In the previous sections, it has been discussed how to constructively build a solution for the subtrace $\{a_1, \ldots, a_{k+1}\}$ assuming that the solution for $\{a_1, \ldots, a_k\}$ is known. The first partial solution $\mathcal{S}^2_1$ for $\{a_1, a_2\}$ is built as:

- $\overrightarrow{c} = [\sigma^1]$
- $\mathbf{U} = [-1]$
- $\mathbf{I}^2 = [\overrightarrow{\imath}^1 | \overrightarrow{\imath}^2] = [0, 1]$
- $\overrightarrow{w} = [1]$

Note that this is the only feasible solution for the subtrace $\{a_1, a_2\}$. The exploration engine can then begin to work, gradually increasing the dimensions of the partial solution, until it reaches a solution for the entire trace $\overrightarrow{a}$, or it discards the root of the solution space $\mathcal{S}_1^2$ (in which case $\overrightarrow{a}$ cannot be generated by an affine loop).

4.1.4 Algorithm

Algorithm 4.1 presents the pseudocode of the EXTRACT() function, which implements the proposed approach. The recursive solution is not practical for a real implementation, but it clearly illustrates the main ideas of our approach. The computations to calculate the new loop insertions described in Section 4.1.3 are encapsulated in the GROW() function, shown as Algorithm 4.2. The extraction starts by calling EXTRACT() with the initial $\mathcal{S}_1^2$ defined in the previous section. In the worst case, when no access can be predicted using $\overrightarrow{\gamma}$, the algorithm uses the brute force approach ($O(n^N)$). In the best case, every access is predicted by $\overrightarrow{\gamma}$ ($O(N)$).

Note that this reconstruction method does not regenerate the constant term c_0 in Equation (4.1), and assumes the base address of the access to be $V' = a_1$. This is not a problem for any practical application of the extracted loop information, as the set of accessed points is identical to that of the original, potentially non-canonical loop.

4.1.5 Reshaping the Iteration Space

The proposed approach rebuilds the minimal equivalent form of the original nest. However, it may be desirable to increase the dimensionality of the resulting nest. For this purpose, note that any given loop level can be tiled in two different levels. Consider loop level i_l, which iterates between 0 and $u_l(\overrightarrow{\imath})$, and two affine functions, $d_l(\overrightarrow{\imath})$ and $q_l(\overrightarrow{\imath})$, such that:

$$u_l(\overrightarrow{\imath}) = d_l(\overrightarrow{\imath})q_l(\overrightarrow{\imath})$$

Algorithm 4.1 Pseudocode of the proposal for trace-based affine reconstruction

```
1: FUNCTION EXTRACT
Input: a partial solution $\mathcal{S} = \{\overrightarrow{c}, \mathbf{I}, \mathbf{U}, \overrightarrow{w}\}$
Input: the execution trace $\overrightarrow{a}$
Output: a global solution or None if no solution found
2:     $k$ = #columns of I
3:     while $k < len(\overrightarrow{a}) - 1$ do
4:         $\sigma = a_{k+1} - a_k$
5:         $\overrightarrow{\gamma} = \mathbf{U}\overrightarrow{\imath}^k + \overrightarrow{w}$                ▷ Try to use $\overrightarrow{\gamma}$ (§4.1.3)
6:         $\hat{\sigma}_l = \overrightarrow{c}\,\overrightarrow{\delta}_l$
7:         if $\hat{\sigma}_l = \sigma$ then
8:             $\mathbf{I} = [\mathbf{I}| + (l, \overrightarrow{\imath}^k)]$
9:             k = k + 1
10:            continue
11:        end if
12:        for x=$n$ down to 1 do                    ▷ Brute force approach (§4.1.2)
13:            $\hat{\sigma}_x = \overrightarrow{c}\,\overrightarrow{\delta}_x$
14:            if $\hat{\sigma}_x = \sigma$ then
15:                $\mathbf{I}' = [\mathbf{I}| + (x, \overrightarrow{\imath}^k)]$
16:                $\{\mathbf{U}', \overrightarrow{w}'\}$ = update bounds                   ▷ §4.1.3
17:                if $\{\overrightarrow{c}, \mathbf{I}', \mathbf{U}', \overrightarrow{w}'\}$ is linear then
18:                    $\mathcal{S}' = \text{EXTRACT}(\{\overrightarrow{c}, \mathbf{I}', \mathbf{U}', \overrightarrow{w}'\}, \overrightarrow{a})$
19:                    if $\mathcal{S}' \neq None$ then
20:                        return $\mathcal{S}'$
21:                    end if
22:                end if
23:            end if
24:        end for
25:        for x=0 to $n$ do                          ▷ Add loop (§4.1.3)
26:            $\mathcal{S}' = \text{EXTRACT}(\text{GROW}(\mathcal{S}, x), \overrightarrow{a})$
27:            if $\mathcal{S}' \neq None$ then
28:                return $\mathcal{S}'$
29:            end if
30:        end for
31:        return None
32:    end while
33:
```

Algorithm 4.2 Pseudocode for increasing the dimensionality of a given partial solution

```
1: FUNCTION GROW
Input: the partial solution $\mathcal{S} = \{\overrightarrow{c}, \mathbf{I}, \mathbf{U}, \overrightarrow{w}\}$
Input: the insertion point $x$
Output: modified partial solution with a new loop in position $x$, or None if the
    insertion point generates a nonlinear solution
2:     $\mathbf{U} = \left[\begin{array}{c|c|c} \mathbf{U}_{(1:x,1:x)} & \mathbf{0}^{x\times 1} & \mathbf{U}_{(1:x,x+1:n)} \\ 0\dots0 & -1 & 0\dots0 \\ \mathbf{U}_{(x+1:n,1:x)} & \mathbf{0}^{(n-x)\times 1} & \mathbf{U}_{(x+1:n,x+1:n)} \end{array}\right]$     ▷ Enlarge U
3:     $\overrightarrow{w} = \left[\overrightarrow{w}_{(1:x)}|0|\overrightarrow{w}_{(x+1:n)}\right]$     ▷ Insert a new element in $\overrightarrow{w}$
4:     $\mathbf{I} = \left[\begin{array}{c|c} \mathbf{I}_{(1:x,:)} & \\ 0\dots0 & \curlywedge(x, \overrightarrow{\imath}^{k}) \\ \mathbf{I}_{(x+1:n,:)} & \end{array}\right]$     ▷ Insert new index into I
5:     update bounds U and $\overrightarrow{w}$
6:     $\overrightarrow{c} = [\overrightarrow{c}_{(1:x)}|c_x|\overrightarrow{c}_{(x+1:n)}]$
7:     if $\{\overrightarrow{c}, \mathbf{I}, \mathbf{U}, \overrightarrow{w}\}$ is not linear then
8:         return None
9:     end if
10:    return $\{\overrightarrow{c}, \mathbf{I}, \mathbf{U}, \overrightarrow{w}\}$
11:
```

Hence, nesting level l can be rewritten as:

```
DO $i_{l_1}$ = 0, $d_l(\overrightarrow{\imath})$
  DO $i_{l_2}$ = 0, $q_l(\overrightarrow{\imath})$
```

In inner levels, a variable substitution is required. Wherever i_l was used, now it must be replaced by $(i_{l_2} + i_{l_1} q_l(\overrightarrow{\imath}))$. Note that, although the transformed loop is equivalent to the original affine one, the term $i_{l_1} q_l(\overrightarrow{\imath})$ is not affine in the general case (the exception occurring when $q_l(\overrightarrow{\imath}) = q$, a single constant). In that case, the components of the minimal equivalent solution $\mathcal{S}_n^k$ can be modified to generate an equivalent affine loop as follows:

$$\overrightarrow{w}' = \left[w_1, \ldots, w_{l-1}, \boldsymbol{\frac{w_l}{q}}, \boldsymbol{q}, w_{l+1}, \ldots, w_n\right]$$

$$\overrightarrow{c}' = [c_1, \ldots, c_{l-1}, \boldsymbol{c_l q}, \boldsymbol{c_l}, c_{l+1}, \ldots, c_n]$$

$$\mathbf{U}' = \begin{bmatrix} & & & \vdots & & & \\ u_{l-1,1} & \ldots & -1 & 0 & 0 & \ldots & 0 \\ \boldsymbol{\frac{u_{l,1}}{q}} & \boldsymbol{\ldots} & \boldsymbol{\frac{u_{l,l-1}}{q}} & \boldsymbol{-1} & \boldsymbol{0} & \boldsymbol{\ldots} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{\ldots} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{-1} & \boldsymbol{\ldots} & \boldsymbol{0} \\ u_{l+1,1} & \ldots & u_{l+1,l-1} & \boldsymbol{u_{l+1,l} q} & \boldsymbol{u_{l+1,l}} & \ldots & 0 \\ & & & \vdots & & & \end{bmatrix}$$

where all components have increased their dimensionality, and now $\overrightarrow{w}', \overrightarrow{c}' \in \mathbb{Z}^{n+1}$ and $\mathbf{U}' \in \mathbb{Z}^{(n+1)\times k}$. Note that iteration space reshaping does not affect the reconstruction process, as it is performed as a post-processing step.

4.2 Case Study: CHOLESKY

In this section we present a case study of our proposal for the trace-based reconstruction of affine codes. We have selected the CHOLESKY application from the PolyBench/C 3.2 suite [114]. Figure 4.2a shows the source code of the application kernel for the "mini" problem size. For the sake of clarity, in this section we will only focus on the analysis of the access `A[i][k]` (see line 14 of Figure 4.2a). Note that this affine access to the 2D matrix A is enclosed into a 3-level loop whose inner indices depend on the outer ones.

An excerpt of the memory trace generated by `A[i][k]` is shown in Figure 4.2b. The first column uniquely identifies the instruction that emits the memory access, and the second is the address of the accessed memory location.

4.2.1 Reconstruction Process

The pseudocode that implements our proposal was presented in Section 4.1.4. As mentioned, the extraction process starts by calling EXTRACT() with the following $\mathcal{S}_1^2$:

$$\begin{cases} \overrightarrow{c} = \left[\sigma^1\right] = [a_2 - a_1] = [0] \\ \mathbf{I}^2 = \left[\overrightarrow{\imath}^1 | \overrightarrow{\imath}^2\right] = [0, 1] \\ \mathbf{U} = [-1] \\ \overrightarrow{w} = [1]^T \end{cases}$$

As commented in Section 4.1.3, this is the only feasible solution for the subtrace $\{a_1 = \texttt{0x1e2d140}, a_2 = \texttt{0x1e2d140}\}$. Next, EXTRACT() starts to process the following access in the trace:

$$a_3 = \texttt{0x1e2d140}$$

and computes the stride with the prior access (see line 4 of Algorithm 4.1):

$$\sigma^2 = a_3 - a_2 = \texttt{0x1e2d140} - \texttt{0x1e2d140} = 0$$

Then, our method tries to efficiently traverse the solution space considering:

$$\overrightarrow{\gamma}^2 = \mathbf{U}\,\overrightarrow{\imath}^2 + \overrightarrow{w} = [-1]\,[1] + [1] = [-1] + [1] = [0]$$

$\overrightarrow{\gamma}^2$ does not have positive elements, and thus cannot guide the exploration (as expected in the first iterations). As will be suggested in Section 4.4.1, a simple heuristic that solves this kind of situations is considering that, when $\overrightarrow{\gamma}$ is not yet operational, the outermost discovered loop is predicted to iterate. At this point,

```
 1  #define N 32;
 2  double p[N], A[N][N], x;
 3  int i, j, k;
 4
 5  #pragma scop
 6  for (i = 0; i < N; ++i) {
 7     x = A[i][i];
 8     for (j = 0; j <= i - 1; ++j)
 9        x = x - A[i][j] * A[i][j];
10     p[i] = 1.0 / sqrt(x);
11     for (j = i + 1; j < N; ++j) {
12        x = A[i][j];
13        for (k = 0; k <= i - 1; ++k)
14           x = x - A[j][k] * A[i][k] ;
15        A[j][i] = x * p[i];
16     }
17  }
18  #pragma endscop
```

(a) Source code.

```
0x00400cbe 0x1e2d140
0x00400cbe 0x1e2d140
0x00400cbe 0x1e2d140
...
0x00400cbe 0x1e2ef18
0x00400cbe 0x1e2ef20
0x00400cbe 0x1e2ef28
```

(b) Excerpt of the memory trace generated by the access `A[i][k]` (see line 14 of Figure 4.2a).

Figure 4.2 – The Cholesky matrix decomposition.

the engine predicts an iteration of the only loop that has been found so far:

$$\hat{\sigma}_1^2 = \vec{c}\,\vec{\delta}_1^{\,2} = [0]\,[1]^T = 0$$

which is equal to the observed stride σ^2. The matrix of reconstructed indices is updated:

$$\mathbf{I} = [\mathbf{I}| + (\mathbf{1}, \vec{\boldsymbol{\imath}}^{\,\mathbf{2}})] = \begin{bmatrix} 0 & 1 & \mathbf{2} \end{bmatrix}$$

and the loop bounds need to be recomputed. Thanks to Corollary 4.1.6:

$$\vec{w}\,' = \left[\boldsymbol{w_1'}\right]^T = \left[i_1^3\right]^T = [2]^T$$

and $\mathbf{U}$ remains unchanged. The new solution is linear and the algorithm continues processing the trace and updating $\mathbf{I}$ and $\vec{w}$ in the same way until the observed stride changes to:

$$\sigma^{30} = a_{31} - a_{30} = \mathtt{0x1e2d240} - \mathtt{0x1e2d140} \Rightarrow$$

$$\sigma^{30} = \texttt{0x100} = 256^{3}$$

Neither $\overrightarrow{\gamma}$ nor the brute force approach exploring all possible indices $\overrightarrow{\imath}_l^{k+1} = +(l, \overrightarrow{\imath}^k)$ (see Section 4.1.2) can predict a stride different from 0 because $\overrightarrow{c} = [0]$ (see Lemma 4.1.2). Therefore, the subtrace $\{a_1, \ldots, a_{31}\}$ cannot be generated with an affine access enclosed in a 1-level loop and the dimensionality of the current solution $\mathcal{S}_1^{30}$ must be increased to build $\mathcal{S}_2^{31}$. For this purpose, the function EXTRACT() calls GROW() for the two possible insertion points of the new loop (see lines 25–30 of Algorithm 4.1). As the most common situation is that newly discovered loops are outer than the previously known ones, it starts with $x = 0$. GROW() inserts a new row and column in $\mathbf{U}$:

$$\mathbf{U} = \left[\begin{array}{c|c} \mathbf{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_{(1:1,1:1)} \end{array}\right] = \left[\begin{array}{c|c} \mathbf{-1} & \mathbf{0} \\ \mathbf{0} & -1 \end{array}\right]$$

a new index into $\mathbf{I}$, updating the previous ones with a row of 0 to match the new dimensionality:

$$\mathbf{I} = \left[\begin{array}{ccc|c} \mathbf{0} & \ldots & \mathbf{0} & \mathbf{1} \\ & \mathbf{I}_{(1:30)} & & \mathbf{0} \end{array}\right] = \left[\begin{array}{ccc|c} \mathbf{0} & \ldots & \mathbf{0} & \mathbf{1} \\ 0 & \ldots & 29 & \mathbf{0} \end{array}\right]$$

and a new element in $\overrightarrow{w}$:

$$\overrightarrow{w} = \left[\mathbf{1} | \overrightarrow{w}_{(1:1)}\right]^T = [\mathbf{1}|29]^T$$

There are not enough points in the reconstructed iteration space to infer any dependence between the number of iterations of i_2 and the newly inserted i_1 (see Equation (4.12)), and therefore $\mathbf{U}$ remains unchanged. The engine checks the linearity of the calculated loop bounds as indicated in Equation (4.5):

$$\mathbf{UI} + \overrightarrow{w}\mathbf{1}^{1\times(31)} \geq \mathbf{0}^{2\times(31)}$$

[3]Note that the strides are in bytes (see Section 4.2.2).

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 & \dots & 0 & 1 \\ 0 & \dots & 29 & 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 29 \end{bmatrix} \begin{bmatrix} 1 & \dots & 1 \end{bmatrix} =$$

$$\begin{bmatrix} 0 & 0 & 0 & \dots & 0 & -1 \\ 0 & -1 & -2 & \dots & -29 & 0 \end{bmatrix} + \begin{bmatrix} 1 & \dots & 1 \\ 29 & \dots & 29 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 & 0 \\ 29 & 28 & 27 & \dots & 0 & 29 \end{bmatrix} \geq \mathbf{0}^{2\times(31)}$$

The insertion of the new loop in position $p = 0$ is accepted and the traversal of the solution space continues from $k = 31$. The next observed stride is $\sigma^{31} = 8$. Until now the engine has discovered a 2-level loop nest, hence $\overrightarrow{\gamma}$ may be operational and it is computed as:

$$\overrightarrow{\gamma}^{31} = \mathbf{U}\overrightarrow{\imath}^{31} + \overrightarrow{w} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 29 \end{bmatrix} = \begin{bmatrix} 0 \\ 29 \end{bmatrix}$$

As the innermost element of $\overrightarrow{\gamma}$ is $\gamma_2 = 29$, the next index is predicted as:

$$\overrightarrow{\imath}_2^{32} = +(2, \overrightarrow{\imath}^{31}) = +(2, \begin{bmatrix} 1 \\ 0 \end{bmatrix}) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

and the estimated stride is

$$\hat{\sigma}_2^{31} = \overrightarrow{c}\,\overrightarrow{\delta}_2^{31} = \begin{bmatrix} 256 & 0 \end{bmatrix} \begin{bmatrix} (1-1) \\ (1-0) \end{bmatrix} = 0$$

which is different from the observed stride $\sigma^{31} = 8$, and then the prediction of γ is inaccurate. The brute force search (see lines 12–24 of Algorithm 4.1) must explore the $n = 2$ possible solutions of the diophantine linear equation system of Equation (4.7) (see Lemma 4.1.3). For $n = 1$:

$$\hat{\sigma}_1^{31} = \overrightarrow{c}\,\overrightarrow{\delta}_1^{31} = \overrightarrow{c}(+(1, \overrightarrow{\imath}^{31}) - \overrightarrow{\imath}^{31}) \Rightarrow$$

$$\hat{\sigma}_1^{31} = \begin{bmatrix} 256 & 0 \end{bmatrix} \begin{bmatrix} (2-1) \\ (0-0) \end{bmatrix} = 256$$

which also is different from σ^{31}. Note that $\hat{\sigma}_2^{31}$ has been already computed and, hence, the solution must grow to $\mathcal{S}_3^{32}$ to be affine. According to lines 25–30 of Algorithm 4.1, the first insertion point that is tested is $x = 0$. Thus GROW() (see Algorithm 4.2) inserts a new row and column in $\mathbf{U}$:

$$\mathbf{U} = \left[\begin{array}{c|ccc} \mathbf{-1} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & & \mathbf{U}_{(1:2,1:2)} & \end{array}\right] = \left[\begin{array}{c|cc} \mathbf{-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & -1 & 0 \\ \mathbf{0} & 0 & -1 \end{array}\right]$$

a new index into $\mathbf{I}$, updating the previous ones with a row of 0 to match the new dimensionality:

$$\mathbf{I} = \left[\begin{array}{ccc|c} \mathbf{0} & \ldots & \mathbf{0} & \mathbf{1} \\ & \mathbf{I}_{(1:31)} & & \mathbf{0} \end{array}\right] = \left[\begin{array}{cccc|c} \mathbf{0} & \ldots & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ 0 & \ldots & 0 & 1 & \mathbf{0} \\ 0 & \ldots & 29 & 0 & \mathbf{0} \end{array}\right]$$

and a new element in $\overrightarrow{w}$:

$$\overrightarrow{w} = \left[\begin{array}{c|c} \mathbf{0} & \overrightarrow{w}_{(2:1)} \end{array}\right]^T = \left[\begin{array}{c|cc} \mathbf{0} & 1 & 29 \end{array}\right]^T$$

Then it recomputes the loop bounds:

$$\overrightarrow{w} = \left[\begin{array}{ccc} \mathbf{1} & 1 & 29 \end{array}\right]^T$$

No dependence yet can be inferred between the number of iterations of i_1 and i_2. Regarding i_3, the following system is built (see Equation (4.12)):

$$\mathbf{U}'_{3,1:3}\mathbf{i}^z_{(1:3,1:2)} + w'_j\mathbf{1}^{1\times 2} = \mathbf{0}^{1\times 2} \Rightarrow$$

$$\left[\begin{array}{ccc} u_{3,1} & u_{3,2} & -1 \end{array}\right] \left[\begin{array}{cc} 0 & 0 \\ 0 & 1 \\ 29 & 0 \end{array}\right] + \left[\begin{array}{cc} 29 & 29 \end{array}\right] = \mathbf{0}^{1\times 2} \Rightarrow$$

$$\left[\begin{array}{cc} -29 & u_{3,2} \end{array}\right] + \left[\begin{array}{cc} 29 & 29 \end{array}\right] = \mathbf{0}^{1\times 2} \Rightarrow$$

$$u_{3,2} = -29$$

Note that no value can be inferred for $u_{3,1}$, as there are not enough iterations to compute the dependence between i_3 and i_1. Finally:

$$\mathbf{U} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & \mathbf{-29} & -1 \end{bmatrix}$$

The loop coefficients are updated next:

$$c_0' = \sigma^{31} + \sum_{r=1}^{2} i_r^{31} c_r \Rightarrow$$

$$c_0' = 8 + 1 \cdot 256 + 0 \cdot 0 = 264$$

$$\vec{c} = \left[\, \mathbf{c_0'} \mid \vec{c}_{(1:2)} \, \right] = \left[\, \mathbf{264} \mid 256 \quad 0 \, \right]$$

Therefore the insertion of a new outermost loop is accepted and the exploration continues. The next observed stride is:

$$\sigma_{32} = -8$$

And $\vec{\gamma}^{32}$ predicts that the 3rd loop will iterate:

$$\vec{\gamma}^{32} = \mathbf{U}\,\vec{\imath}^{32} + \vec{w} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & -29 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 29 \end{bmatrix}$$

$$\Rightarrow \vec{\gamma}^{32} = \begin{bmatrix} 0 \\ 1 \\ 29 \end{bmatrix}$$

And therefore:

$$\vec{\imath}_3^{33} = +(3, \vec{\imath}^{32}) = +(3, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}) = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

and the corresponding predicted stride does not match the observed one:

$$\hat{\sigma}_3^{32} = \overrightarrow{c}\,\overrightarrow{\delta}_3^{32} = \begin{bmatrix} 264 & 256 & 0 \end{bmatrix} \begin{bmatrix} (1-1) \\ (0-0) \\ (1-0) \end{bmatrix} = 0$$

The brute force approach is also unsuccessful estimating the strides:

$$\begin{cases} \hat{\sigma}_1^{32} = 264 \\ \hat{\sigma}_2^{32} = 256 \end{cases}$$

and GROW() would be called to increase the dimensionality of the generated loop. As explained at the end of Section 4.1.3, there must be a practical limit to the dimensionality of the generated loop as, in the general case, any trace $\{a_1, \ldots, a_N\}$ can be reconstructed as an N-level loop. For the sake of simplicity, we assume here that the engine has been configured to explore up to 3-level loops, and that this branch is now discarded (see Section 4.2.2).

At this point, the call to EXTRACT() that explored the discarded branch returns, and Algorithm 4.1 continues inside the loop in line 25, now trying to insert a new index in position $p = 1$. The new index coefficient is calculated as:

$$c'_1 = \sigma^{31} + \sum_{r=2}^{2} i_r^{31} \Rightarrow$$

$$c'_1 = 8 + 0 \cdot 0$$

$$\overrightarrow{c} = \left[\begin{array}{c|c|c} 256 & \mathbf{8} & 0 \end{array}\right]$$

The new index matrix is:

$$\mathbf{I} = \left[\begin{array}{ccc|c} & \mathbf{I}_{(1:1,:)} & & \\ \mathbf{0} & \ldots & \mathbf{0} & \overrightarrow{\imath}^{32} \\ & \mathbf{I}_{(2:2,:)} & & \end{array}\right] =$$

$$\left[\begin{array}{cccc|c} 0 & \ldots & 0 & 1 & \mathbf{1} \\ \mathbf{0} & \mathbf{\ldots} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ 0 & \ldots & 29 & 0 & \mathbf{0} \end{array}\right]$$

The new bounds vector is:

$$\overrightarrow{w}' = \left[\begin{array}{ccc} 1 & \mathbf{0} & 29 \end{array} \right]^T$$

and the new bounds matrix is:

$$\mathbf{U}' = \left[\begin{array}{ccc} -1 & 0 & 0 \\ \mathbf{1} & -1 & 0 \\ \mathbf{-29} & 0 & -1 \end{array} \right]$$

As soon as the first points are explored in this branch, the engine will find that this partial solution does not match the remainder of the trace either. It will discard the entire branch as before, and go back to try to insert the new index in position $p = 2$ (i.e., as the innermost index). The new index coefficient is:

$$c'_2 = \sigma^{31} = 8$$

$$\overrightarrow{c} = \left[\begin{array}{ccc} 256 & 0 & \mathbf{8} \end{array} \right]$$

Note that, at this point, the engine has correctly recognized the coefficients of the three levels of the original nest. It generates the new index matrix:

$$\mathbf{I} = \left[\begin{array}{ccc|c} & \mathbf{I}_{(1:2,:)} & & \overrightarrow{\imath}^{32} \\ \mathbf{0} & \dots & \mathbf{0} & \end{array} \right] =$$

$$\left[\begin{array}{cccc|c} 0 & \dots & 0 & 1 & \mathbf{1} \\ 0 & \dots & 29 & 0 & \mathbf{0} \\ \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{array} \right]$$

the new bounds vector:

$$\overrightarrow{w} = \left[\begin{array}{ccc} 1 & 29 & \mathbf{0} \end{array} \right]^T$$

and calculates the elements in the associated row of $\mathbf{U}'$. This can be done by applying Equation (4.12):

$$\mathbf{U}'_{(3,1:3)}\mathbf{i}^z_{(1:3,1:2)} + w'_3\mathbf{1}^{1\times 2} = \mathbf{0}^{1\times 2} \Rightarrow$$

$$\begin{bmatrix} u_{3,1} & u_{3,2} & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 \end{bmatrix} = \mathbf{0}^{1\times 2} \Rightarrow$$

$$\begin{bmatrix} u_{3,2} & (u_{3,1} - 1) \end{bmatrix} = \mathbf{0}^{1\times 2} \Rightarrow$$

$$\begin{cases} u_{3,1} = 1 \\ u_{3,2} = 0 \end{cases}$$

And finally, the calculated $\mathbf{U}'$ is:

$$\mathbf{U}' = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix}$$

Now, $\overrightarrow{\gamma}^{32} = [0, 256, 0]^T$ and the engine predicts $\overrightarrow{\imath}^{33} = [1, 1, 0]^T$, which generates a stride that matches the observed one. $\overrightarrow{\gamma}^{33} = [0, 255, 1]^T$ and the engine predicts $\overrightarrow{\imath}^{34} = [1, 1, 1]^T$, which also generates a stride that matches the observed one. This process continues, alternating iterations of i_2 and iterations of i_3, until the engine incorporates access a_{88} to the solution, with index $\overrightarrow{\imath}^{88} = [1, 28, 1]^T$. At this point, $\overrightarrow{\gamma}^{88} = [0, 1, 0]^T$ and the engine predicts an iteration of i_2, with $\hat{\sigma}_2^{88} = -8$. However, $\sigma^{88} = 248$. Since $\overrightarrow{\gamma}$ prediction fails, the engine defaults to the brute force mode, calculating the strides for each of the currently known indices (see Section 4.1.2). Potential strides are:

$$\begin{cases} \hat{\sigma}_1^{88} = c_1 - \sum_{r=2}^{3} c_r i_r^{88} = 256 - 0 - 8 = 248 \\ \hat{\sigma}_2^{88} = c_2 - \sum_{r=3}^{3} c_r i_r^{88} = 0 - 8 = -8 \\ \hat{\sigma}_3^{88} = c_3 = 8 \end{cases}$$

According to these calculations, the only index that can generate the observed stride is i_1. Therefore, the engine decides to explore the branch with $\overrightarrow{\imath}^{89} = [2, 0, 0]$. Calculation of the new loop bounds that would allow this index to be generated ensue. First, the bounds vector is calculated:

$$\overrightarrow{w}' = [\mathbf{2}, 29, 0]$$

and afterwards the system calculates $\mathbf{U}'$. Its first and third rows do not change. For the second, the following system is solved:

$$\mathbf{U}'_{(2,1:2)}\mathbf{i}^z_{(1:2,1)} + w'_2\mathbf{1}^{1\times1} = \mathbf{0}^{1\times1} \Rightarrow$$

$$\begin{bmatrix} u_{2,1} & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 28 \end{bmatrix} + [29] = \mathbf{0}^{1\times1} \Rightarrow$$

$$u_{2,1} = -1$$

and the calculated matrix is:

$$\mathbf{U}' = \begin{bmatrix} -1 & 0 & 0 \\ -1 & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix}$$

The engine has now collected all the information that it will need to solve the problem. From this point on, our method will keep incorporating elements in the trace to the solution, with $\overrightarrow{\gamma}$ predicting all remaining iterations, until it reaches the end of the trace having reconstructed the following terms:

$$\overrightarrow{c} = \begin{bmatrix} 256 & 0 & 8 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} -1 & 0 & 0 \\ -1 & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\overrightarrow{w} = \begin{bmatrix} 29 & 29 & 0 \end{bmatrix}$$

4.2.2 Discussion

Note that, since addresses in the trace are expressed in bytes, the coefficient of loop indexes reconstructed by the engine is also expressed in bytes: the original access $A[i][k]$ is reconstructed as $A[256 * i + 8 * k]$. These account for both the data type size (`double`, 8 bytes) and the dimensionality of the array $A[32][32]$.

The implementation in Python of our algorithm (see Section 4.4) does not perform a single step for each iteration of the innermost loop. When $\overrightarrow{\gamma}^k$ predicts Δ iterations of the innermost loop, the algorithm checks whether the following k strides in the trace are equal to the loop coefficient of the innermost loop, c_n:

$$\begin{bmatrix} \sigma^k & \dots & \sigma^{k+\Delta} \end{bmatrix} = \begin{bmatrix} c_n & \dots & c_n \end{bmatrix}$$

In this case, the Δ accesses are recognized in block. This greatly speeds the reconstruction process.

Finally, notice that in Section 4.1.3 we have stated that the solution space must be explored in a breadth-first fashion. To achieve this, before launching the reconstruction process that explores up to 3-level loops, the reconstructions exploring up to 1-level loops and 2-level loops must have failed.

4.3 Supporting Nonlinearity

The method described in Section 4.1 is guaranteed to find the minimal affine reconstruction for any trace of a linear access. However, applications in real systems require dealing with varying degrees of nonlinearity and uncertainty in the input. Instead of building one loop (with one iteration and the corresponding stride) per entry in the trace, this section covers how the exploration engine can be tuned to deal more efficiently with input noise and missing data, and how to automatically reconstruct parallelized codes.

4.3.1 Input Noise

Some trace files mainly contain references issued by a single access, but mixed with a certain amount of unrelated accesses. This may happen in case of nearly affine or unlabeled traces[4]. In this situation, the exploration of the solution space

[4]In some situations, it may not be realistic to expect accesses to be labeled according to the instruction that issued them. An example is reverse engineering accesses issued by IP cores. In these cases, the trace may be tentatively split into subtraces in which most accesses are issued by the same instruction, but some error tolerance is needed.

can be modified to discard some observations before concluding that a branch cannot lead to a solution. This feature has to be statistically guided, to avoid discarding too many points and reaching a very simplified nest version.

Hence, the algorithm has been extended to allow discarding input noise. Whenever $\overrightarrow{\gamma}^k$ predicts a $\hat{\sigma}^k$ that does not match the observed σ^k, the reconstruction engine checks whether

$$\left\{ \hat{\sigma}^k = \sum_{r=0}^{e} \sigma^{k+r}, 0 < e \leq max \right\}$$

where *max* is the maximum number of consecutive noise references to be tolerated. If this condition holds for some value of e, then it is plausible that accesses $\{a_k, \ldots, a_{k+e-1}\}$ are spurious. The engine will discard these references and resume the exploration. A backtracking point is created in case this assumption proves false.

The use of $\overrightarrow{\gamma}$ is capable of identifying errors as long as the current $\mathcal{S}_n^k$ accurately represents the trace. However, this does not happen in the initial stages of the exploration process. For this reason, whenever the exploration engine finds that a set of indices $\mathbf{I}^k$ cannot lead to a solution (as there is no feasible $\overrightarrow{\imath}^{k+1}$ to continue the exploration), it arbitrarily discards the current σ^k. In order to avoid the exploration of improbable branches, a tolerance parameter is added to indicate the percentage of accesses that may be considered spurious before the branch is definitely discarded.

4.3.2 Missing Data: Reconstructing Conditionals

In some situations, a trace file may be missing some data to make it completely representable by an affine loop. Our engine can be configured to insert "missing" observations to try to reach a linear representation. Whenever $\overrightarrow{\gamma}^k$ predicts a $\hat{\sigma}^k$ that does not match the observed σ^k, the engine may check whether inserting the access predicted by $\overrightarrow{\gamma}^k$ allows to continue exploring the branch:

$$\left\{ \sigma^k = \sum_{r=0}^{e} \hat{\sigma}^{k+r}, 0 < e \leq max \right\}$$

where max is the maximum number of missing references to be tolerated. As before, if this condition holds then it is plausible that there is a sequence of missing accesses $\{a'_k, \ldots, a'_{k+e-1}\}$ in between a_k and a_{k+1} such that $\{a'_{k+j-1} = a_k + \Sigma_{r=0}^{j} \hat{\sigma}^{k+r}, 0 < j \leq e\}$. The engine tentatively inserts these accesses and resumes the exploration. A backtracking point is created in case no solution is reached by exploring that branch. A tolerance parameter is used to avoid the exploration of improbable branches, as described in Section 4.3.1.

When allowing for missing data, the final solution for a trace $\overrightarrow{a}$ containing N points takes the form of a tuple $\mathcal{S}_n^{N'} = \{\overrightarrow{c}, \mathbf{I}^{N'}, \mathbf{U}, \overrightarrow{w}, \mathbf{i}\}$, where the iteration matrix $\mathbf{I}^{N'}$ is an over-approximation of the original iteration domain [23] that contains more indices than accesses of the original trace; and $\mathbf{i} \in \mathbb{Z}^{n \times M}$ is a matrix composed of M columns extracted from $\mathbf{I}^{N'}$ $(M = N' - N)$, which represents the extra iterations that are missing from the original trace. A class of interesting problems that may be modeled as a trace with missing points are traces generated by accesses guarded by a boolean function $g(\overrightarrow{\imath})$, which depends on loop indices and loop invariants:

```
DO $i_1$ = 0, $u_1(\overrightarrow{\imath})$
  ⋮
  DO $i_n$ = 0, $u_n(\overrightarrow{\imath})$
    IF $g(\overrightarrow{\imath})$ THEN
      $V[f_1(\overrightarrow{\imath})] \ldots [f_m(\overrightarrow{\imath})]$
```

A code that regenerates the original trace can be written using loops that iterate the constructed over-approximation of the iteration space, and adding a boolean piecewise guard function $g^p(\overrightarrow{\imath})$ such that:

$$g^p(\overrightarrow{\imath}) = \begin{cases} 0 & if\ \overrightarrow{\imath} \in \mathbf{i} \\ 1 & otherwise \end{cases}$$

If the original guard condition is an affine expression of the form $(g_0 + g_1 i_1 + \ldots + g_n i_n \neq 0)$, then it divides the n-dimensional over-approximated iteration space into two different polytopes separated by a non-accessed, $(n-1)$-dimensional rift. In this situation, it may be possible to construct an affine guard function $g^a(\overrightarrow{\imath}) = g_0^a + g_1^a i_1 + \ldots + g_n^a i_n$ to replicate the original trace. Consider the following diophantine linear equation system:

$$\left[\begin{array}{c|c} 1 & \\ \vdots & \mathbf{i}^T \\ 1 & \end{array}\right] \begin{bmatrix} g_0^a \\ \vdots \\ g_n^a \end{bmatrix} = \mathbf{0}^{M\times 1} \tag{4.13}$$

where each row in the system matrix is one of the indices in $\mathbf{i}$. Elements in its leftmost column are all ones, as their associated unknown is g_0^a (the constant term). If the system has solutions, then including $g^a(\overrightarrow{\imath})$ as guard function will remove all elements in $\mathbf{i}$ from the over-approximated polytope during the iteration. Depending on how the exploration engine reconstructed the loop, it is possible that points in the iteration space that should be iterated are removed. This happens in two situations, in which the solutions of Equation (4.13) are incorrect:

- When the dimensionality of the reconstructed loop is less than the rank of the system matrix in Equation (4.13)[5]. This might happen due to a reduction of the dimensionality of the original iteration space when the engine rebuilds the minimal equivalent form.
- When the stride c_n associated to the innermost loop is zero. In this case, it is impossible to know which exact points are missing from the iteration space.

In this case, the code should be rebuilt using the piecewise version $g^p(\overrightarrow{\imath})$.

4.3.3 Automatically Parallelized Codes

This section considers the reconstruction of traces generated by affine codes that have been automatically parallelized using PLUTO [27]. We assume that each

[5]This implies that points in $\mathbf{i}$ are not contained by any $(n-1)$-dimensional polytope, and therefore cannot be characterized as an affine restriction over the n-dimensional iteration space.

thread in the execution generates its own separate memory trace or, equivalently, that a single joint trace includes the identifier of the thread executing each access.

Codes that are parallelized by simply adding a `parallel` OpenMP pragma at the appropriate nesting level, without using tiling or modifying the scheduling, will have affine traces provided that iterations are scheduled statically. In these cases, the same reconstruction algorithm used for sequential codes in Section 4.1 can be applied. The traces of each different thread are reconstructed in exactly the same way, except for the iteration limits (vector $\overrightarrow{w}$). It is then trivial to reconstruct the original code.

However, this is not the case for codes with more complex dependences. In order to automatically parallelize these loops, PLUTO changes the original scheduling to satisfy the dependences in the program. To achieve this without additional code complexity, it introduces *nonlinear functions* as the bounds of the inner loops. As a result, the code is not representable as a single affine loop of depth similar to the original one. The mechanisms presented in Sections 4.3.1 and 4.3.2, which discard discrete references that do not conform to an affine representation as a means to reconstruct quasi-affine traces, do not work.

One possible approach to model this type of codes is to recognize the trace in a *piecewise* manner, that is, reconstructing it as a sequence of perfectly nested loops. Thus, a solution is always found by reconstructing the trace in this way (although its size is only bounded by the number of accesses in the trace). The engine now works in a greedy fashion, as shown in Algorithm 4.3. First, the entire trace is reconstructed using a sequence of consecutive loop nests of depth 1 (see line 2). Then, incrementally deeper loops are built on top of the single level ones, using a modified version of EXTRACT() that returns the largest affine trace part it can find. A set of consecutive loops may only be substituted by a deeper one that perfectly overlaps them. In this way, the algorithm tries to avoid replacing loops by marginally larger versions that do not take full advantage of the depth increase. The reconstruction ends when a maximum allowed depth is reached, or when a single loop that reconstructs the entire trace is found.

By proceeding in this greedy fashion, the solution found for a given maximum depth may not be optimal. However, the exhaustive search would make

Algorithm 4.3 Pseudocode of the piecewise reconstruction

```
1: FUNCTION PIECEWISEEXTRACT
Input: $\overrightarrow{a}$: the execution trace
Input: $max\_depth$: maximum reconstruction depth
Output: $\Omega = \{\mathcal{S}_0, \ldots, \mathcal{S}_{L-1}\}$: set of perfectly nested affine loops that form a
    piecewise reconstruction of $\overrightarrow{a}$
2:     $\Omega \leftarrow$ PIECEWISEEXTRACT($\overrightarrow{a}, depth = 1$)
3:     $curr\_depth \leftarrow 2$
4:     while $(curr\_depth \leq max\_depth) \wedge (|\Omega| > 1)$ do
5:         for $\mathcal{S}_l \in \Omega$ do
6:             $\mathcal{S}'_l \leftarrow$ EXTRACT($\mathcal{S}_l, \overrightarrow{a}, depth = curr\_depth$)
7:             if $\mathcal{S}'_l$ overlaps perfectly with $\{\mathcal{S}_l, \ldots, \mathcal{S}_{l'}\} \in \Omega$ then
8:                 $\Omega \leftarrow (\Omega - \{\mathcal{S}_l, \ldots, \mathcal{S}_{l'}\}) \cup \mathcal{S}'_l$
9:             end if
10:            $curr\_depth + +$
11:        end for
12:    end while
13:    return $\Omega$
14: end FUNCTION=0
```

the problem intractable due to the vast amount of different alternatives to be explored. Note that, when reconstructing nonlinear traces in a piecewise fashion, it is not possible in the general case to reconstruct an SPMD code that is common to all threads. Nevertheless, this technique allows to construct a piecewise affine equivalent form of codes that are not in their original form, enabling their affine analysis and optimization.

4.4 Experimental Evaluation

The proposed method has been implemented in Python and used to extract affine loops for different codes. This section analyzes the behavior of the reconstruction algorithm on completely affine codes (in order to assess the feasibility of the proposed approach) and then on codes with some nonlinearities that include noise, missing points, and automatically parallelized affine codes. Each execution was performed on an Intel Xeon E5-2660 octa-core Sandy Bridge processor at 2.22 GHz with 20 MB of cache memory and 64 GB of RAM.

4.4.1 Affine Codes

The reconstruction algorithm was run with memory traces generated by the PolyBench/C 3.2 suite [114]. It includes 30 applications from domains such as linear algebra, stencil codes, and data mining. The target was the traces generated by the parts of the code marked with `scop` pragmas, which represent the vast majority of the accesses issued by the entire program. These were split into the subtraces generated by their different instructions and stored in memory before being processed. The "standard" problem size was used, generating traces ranging from 6 million references for *jacobi-1D* (150 MB in disk) to 12.9 billion references for *3mm* (270 GB). The number of references in the kernels varies between 3 for *trmm* and 92 for *fdtd-apml*.

Figure 4.3 shows trace sizes and processing times. As can be observed, the reconstruction times largely depend on the number of loops and the access pattern. For instance, the most efficient reconstruction is achieved for *jacobi-1D*, a stencil computation which only accesses small 1-dimensional arrays. Two loops generate all traces, but the outer one iterates only once per each 10.000 iterations of the innermost one. As a result, the reconstruction process can be largely streamlined: the trace contains blocks of 10.000 elements separated by the same stride, which can be recognized in a single step using $\overrightarrow{\gamma}$ as a predictor (see Section 4.2.2). Its 6 million accesses are sequentially processed in 0.2 seconds. On the opposite end, *dynprog*, which emits 858 million references, is the one processed at the slowest rate. It features a 4-level loop nest where the largest block of single-strided accesses contains only 48 references. As such, the number of decision steps taken by the algorithm is much larger. While in the slowest case the engine is capable of processing 180.000 references per second, in the fastest one this figure goes up to 30 million references per second (167x faster).

A second set of experiments was run deactivating $\overrightarrow{\gamma}$ prediction. In this case, the engine must explore all potentially correct branches as indicated in Section 4.1.2. All subtraces were processed in parallel. The recognition was run for 48 hours, at which point the unreconstructed subtraces were considered intractable for practical purposes. Table 4.1 summarizes the results. For most codes only the smallest subtraces were recognized (accounting for less than 1% of the

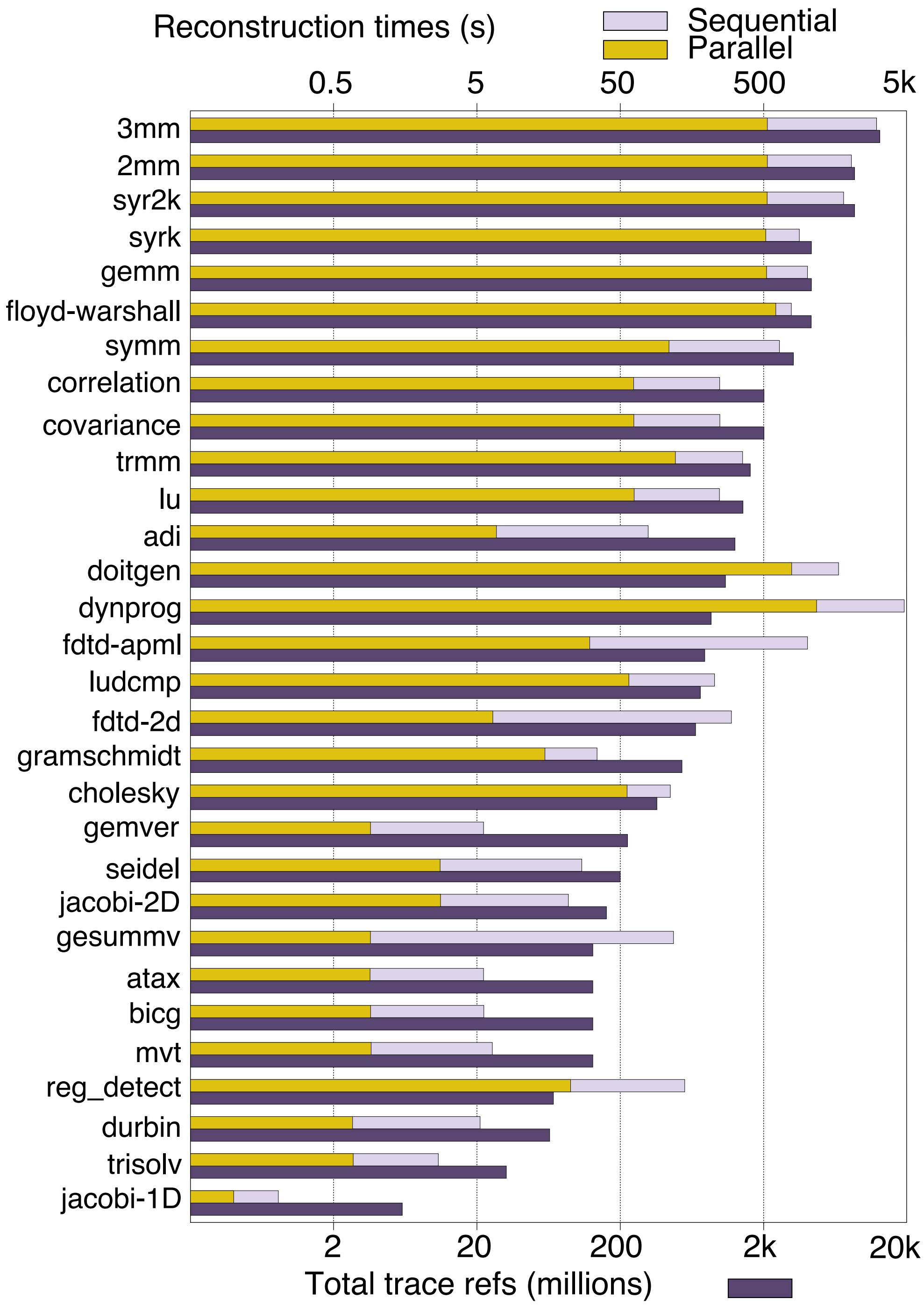


Figure 4.3 – Reconstruction times (upper axis) and trace sizes (lower axis) for the PolyBench/C benchmarks, ordered by trace size. Axes are logarithmic. Since the subtraces of a kernel are independent, they can be reconstructed in parallel achieving an average speedup of 5.6x.

total trace). *fdtd-apml*, *gemver*, *gesummv*, *atax*, *bicg*, and *mvt* contain large single-strided subtraces, which are recognized as a single block. *durbin* and *trisolv* have subtraces of 8 million references, each of which is reconstructed in 47 hours. Finally, *jacobi-1d* has subtraces of 1 million references.

The usability of the engine as an online predictor was also evaluated. Table 4.2 shows the percentage of predicted accesses. For most applications, $\overrightarrow{\gamma}$ predicted above 95% of the issued references. Exceptions are, again, *fdtd-apml*, *gemver*, *gesummv*, *atax*, *bicg*, and *mvt*. Note how their numbers are almost complementary to those in Table 4.1. The reason is that most unpredicted accesses were issued by single-strided references. These are not handled by $\overrightarrow{\gamma}$ since it cannot operate before $\overrightarrow{w}$ is calculated, and this will never happen for 1-level loops, which generate the types of traces that are tractable by the algorithm without $\overrightarrow{\gamma}$ guidance. However, these references are trivially predicted by single-stride prefetching techniques [134]. A simple way to predict this type of references is to consider that, in the absence of a $\overrightarrow{\gamma}$ prediction, the outermost discovered loop will iterate. The use of this heuristic increases the prediction rate above 99% for all the codes we tested.

Regarding memory requirements, the exploration engine needs to store, at least, $\overrightarrow{c}$, $\overrightarrow{w}$, $\mathbf{U}$, and selected indices of $\mathbf{I}$[6]. In addition to these, some memory is consumed by the backtracking points used to efficiently implement the recursion in Algorithm 4.1. The total memory requirements for the subtraces in our experimental set-up vary between 48 bytes and 60 KB.

4.4.2 Input Noise

In order to evaluate robustness against nonlinearities, random noise was injected into each subtrace with probability $p = 0.01$, 0.05, 0.10, and 0.15. The injection does not modify the originally accessed memory region, i.e., injected addresses are also accessed at some point in the original subtrace. In addition to the tolerance and maximum number of consecutive errors parameters discussed in Sec-

[6]These are used for recalculating $\overrightarrow{w}$ and $\mathbf{U}$ (see Section 4.1.3). Hence, it is not necessary to store the entire matrix $\mathbf{I}$ if memory requirements are to be optimized. The only indices needed are those $\{\overrightarrow{\imath} \in \mathbf{I}, 0 \leq j \leq n,\ u_j(\overrightarrow{\imath}) = 0\}$.

Trace	%	**Trace**	%	**Trace**	%
3mm	0.02	lu	0.11	seidel	0.00
2mm	0.04	adi	0.01	jac-2D	0.00
syr2k	0.02	doit.	0.58	gesum.	25.01
syrk	0.05	dynp.	0.00	atax	25.00
gemm	0.05	fdtd-a.	24.21	bicg	25.00
floyd	0.00	lud.	0.66	mvt	12.50
symm	0.13	fdtd-2d	0.01	reg_d.	2.07
corr.	0.67	grams.	0.58	durbin	100
covar.	0.37	chol.	0.58	trisolv	100
trmm	0.00	gemv.	21.43	jac-1D	100

Table 4.1 – Percentage of trace reconstructed after 48h without $\overrightarrow{\gamma}$ prediction.

Trace	%	**Trace**	%	**Trace**	%
3mm	99.85	lu	99.71	seidel	95.00
2mm	99.84	adi	98.00	jac-2D	95.00
syr2k	99.85	doit.	98.83	gesum.	74.95
syrk	99.83	dynp.	99.98	atax	74.96
gemm	99.83	fdtd-a.	75.62	bicg	74.96
floyd	99.88	lud.	99.99	mvt	87.46
symm	99.80	fdtd-2d	98.00	reg_d.	99.78
corr.	99.60	grams.	99.61	durbin	99.88
covar.	99.70	chol.	99.99	trisolv	99.89
trmm	99.97	gemv.	78.53	jac-1D	99.00

Table 4.2 – Percentage of trace accesses predicted by $\overrightarrow{\gamma}$.

tion 4.3.1, an additional parameter is used to abandon exploration of implausible branches faster: a *threshold* beyond which no observations are allowed to be arbitrarily discarded without $\overrightarrow{\gamma}$ guidance. These parameters were manually increased until the correct reconstruction was found.

Ten repetitions of this experiment were performed for selected PolyBench/C codes. All subtraces were eventually reconstructed. Figure 4.4 shows the reconstruction times. These increase linearly with the amount of noise in the trace. This is not as much a consequence of an increase in the number of branches explored as of the decrease in the average size of single-strided blocks. For this reason, the effect of noise injection is similar, from the complexity point of view, to an increase in the number of loops that generate the trace. This is clearly observable in *dynprog*: despite being the kernel for which the engine is least efficient, it has the lowest noise-related overhead, since single-strided blocks were already small.

The figure also shows how for some codes recognition time largely depends on the particular points affected by noise. In particular, a large overhead is caused by an access a_k being injected just before the first access with a stride that includes a yet-undiscovered loop, a_{k+1}. This causes the engine to miscalculate the associated loop coefficient, and constitutes an error that cannot be identified through $\overrightarrow{\gamma}$. In this situation, the engine must be allowed to arbitrarily discard observations up to at least position k in the trace, causing the engine to explore, and discard, many branches of the solution space before the affine pattern is recognized. This is the cause for the large variability in recognition time of *3mm* when injecting noise with $p = 0.15$. Regarding *jacobi-1D*, recognition overhead is relatively larger because of its small trace size. Hence, the overhead of arbitrarily discarding observations in the first stages of the recognition process, relatively small in other codes, becomes significant in this case.

4.4.3 Missing Data

Figure 4.4 shows extraction times for traces with missing points. For these experiments, a random affine guard function was generated for each memory reference in the trace. Ten repetitions of the experiment were performed. The number of points removed by the guard function is small (below 5%), and reconstruction

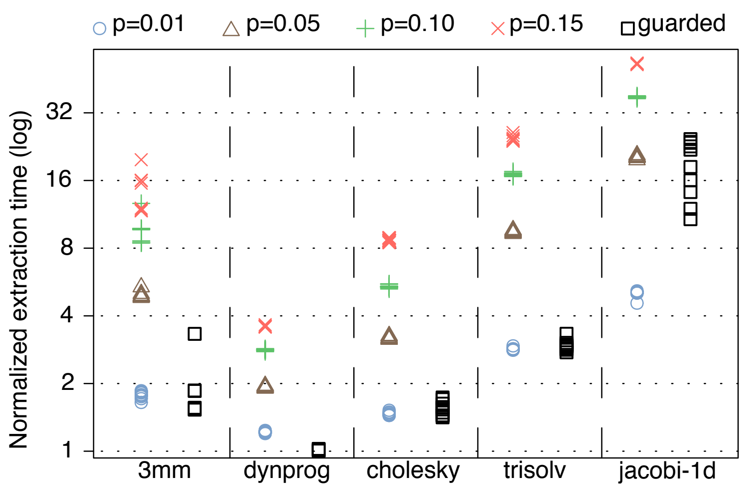


Figure 4.4 – Extraction times with injected noise, and with accesses guarded by affine functions, normalized to the times for reconstructing the unmodified trace.

times are comparable to those obtained with noise injected with $p = 0.01$ and $p = 0.05$. All guard functions were correctly regenerated under the conditions detailed in Section 4.3.2. Note that the regenerated guard functions may not match the original ones, as the iteration space is converted to the canonical form during reconstruction. They are, however, functionally equivalent, as shown in the example of Figure 4.5. Finally, reconstructions that reduce the dimensionality of the iteration space may convert an affine rift into a non-affine one, as exemplified in Figure 4.6.

4.4.4 Automatically Parallelized Codes

As mentioned in Section 4.3.3, the reconstruction of affine codes that have been automatically parallelized by PLUTO [27] with the only addition of a `parallel` OpenMP pragma at the appropriate nesting level is straightforward for the method presented in Section 4.1. Figure 4.7a presents an excerpt of the source code of *covcol* (from the PLUTO testsuite), which contains two different loops, each of them calculating one section of a correlation matrix: the first loop calculates the upper triangular portion from the data, and the second copies the lower triangular portion from the upper one. PLUTO only needs to add a `parallel` directive to parallelize it (see Figure 4.7b). Figure 4.8c shows the parallel execution

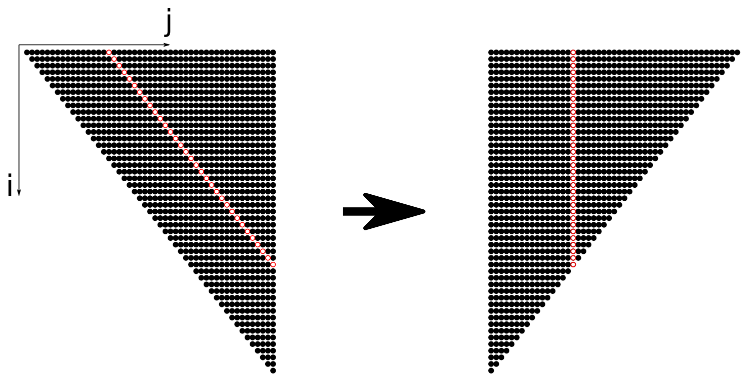


Figure 4.5 – Example of over-approximated iteration space reconstruction. Red points are removed from the iteration space by a guard function. Original loop is `DO i=0,49; DO j=i,49`; reconstructed as `DO i=0,49; DO j=0,49-i`. Original guard was $(i - j + 17 \neq 0)$, reconstructed as $(16 - j \neq 0)$.

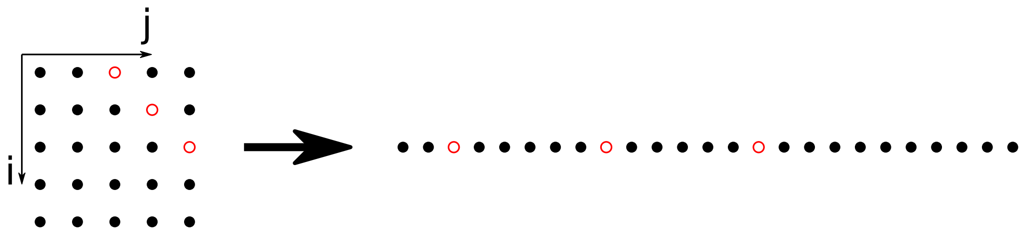


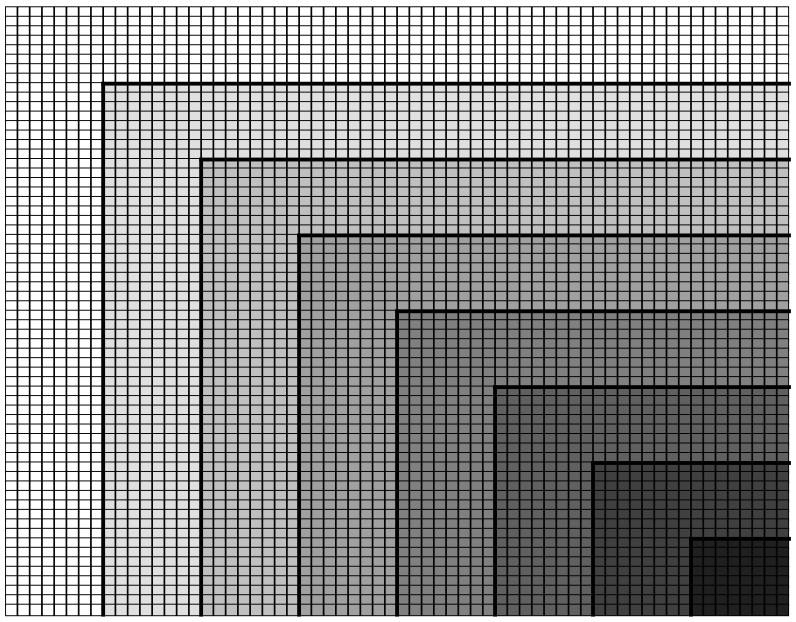
Figure 4.6 – In this case, the original loop is `DO i=0,4; DO j=0,4`; and it is reconstructed as `DO i=0,24`. Original guard was $(i - j + 2 \neq 0)$, and cannot be affinely reconstructed.

```
for (j1=1;j1<=M;j1++) {
  for (j2=j1;j2<=M;j2++) {
    for (i=1;i<=N;i++) {
      symmat[j1][j2] = ...
    }
    symmat[j2][j1] = ...
  }
}
```

(a) Original sequential code.

```
#pragma omp parallel for
  for (t2=1;t2<=M;t2++) {
    for (t3=1;t3<=N;t3++) {
      for (t4=t2;t4<=M;t4++) {
        symmat[t2][t4] = ...
      }
    }
  }
#pragma omp parallel for
  for (t2=1;t2<=M;t2++) {
    for (t3=t2;t3<=M;t3++) {
      symmat[t3][t2] = ...
    }
  }
```

(b) Automatically parallelized code.



(c) Accessed tiles of the 64 × 64 *covcol* kernel executed using 8 threads (each shade of gray represents the area calculated by a single thread).

Figure 4.7 – Original and automatically parallelized *covcol* code.

using 8 threads for an output matrix of 64×64 elements, where each shade of gray represents the area calculated by a single thread. As can be seen in the figure, different threads calculate a different number of points. However, the same reconstruction algorithm used for sequential codes (see Section 4.1) can be used. The reconstructions for the different threads are identical (except for the iteration limits —vector $\overrightarrow{w}$—) and it is then easy to reconstruct the sequential code.

Nevertheless, codes with more complex dependences are more challenging. Figure 4.8c shows the output matrix for the *seidel* kernel. The original code (see Figure 4.8a), simplified for the figure to calculate a single timestep, contains an affine two-level loop implementing a stencil computation with a 2D Moore neighborhood. PLUTO changes the original scheduling to satisfy the dependences in the parallel version program introducing *nonlinear functions* as the bounds of the inner loops (see Figure 4.8b). In order to model this type of codes, we recognize the trace in a *piecewise* manner as explained in Section 4.3.3 thanks to Algorithm 4.3.

Figures 4.9–4.12 exemplify the process, showing the evolution of the reconstructed trace pieces as the maximum allowed loop depth increases for thread #0. As the reconstruction depth is increased, neighboring pieces are fused together. Note that, in the general case, we are not able to reconstruct an unique code for all threads as shown in Figure 4.13: the different threads reconstruct a different number of loops covering different regions of the trace. However, our technique continues to enable the affine analysis and optimization of the parallelized code.

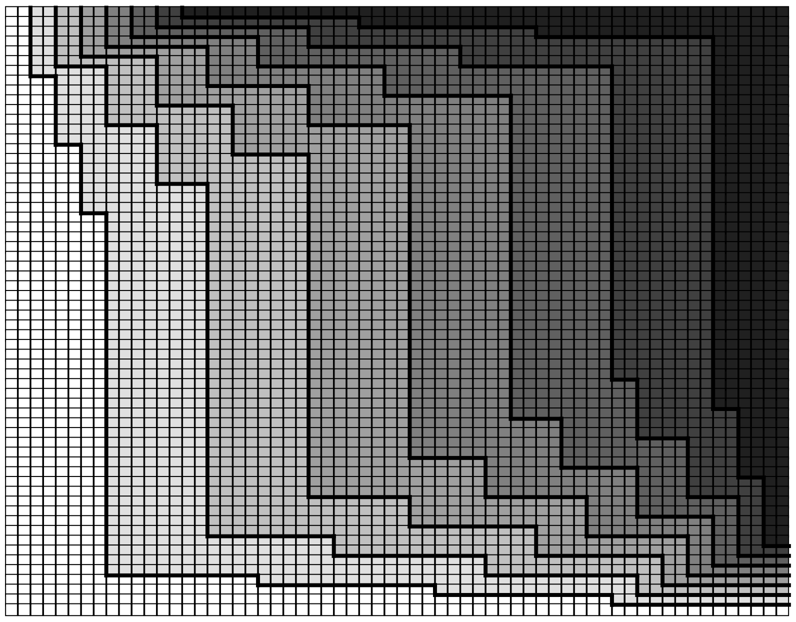
Table 4.3 assesses the fundamental characteristics of the reconstructions of the automatically parallelized PLUTO test cases that are not trivially reconstructed as a single perfectly nested loop. In order to provide an affine representation, the engine needs to insert additional loops not present in the original code that bridge the gaps in between affine sections. This bridging is usually dependent on the problem size and the number of execution threads. The most important sources of nonlinearities in these codes are tiling, and parallelizations that modify the scheduling to resolve code dependences. PLUTO inserts *nonlinear bounds* to instrument both types of transformations. In isolation, a `max` or `min` function presents a single nonlinear inflection point: the one in which the values in its left- and right-hand sides are equal. However, when these functions are used in an

```
for (i=1;i<=N-2;i++)
  for (j=1;j<=N-2;j++)
    a[i][j] = ...
```

(a) Original sequential code.

```
for (t1=3;t1<=3*N-6;t1++) {
  lbp=max(ceild(t1+1,2),t1-N+2);
  ubp=min(floord(t1+N-2,2),t1-1);
#pragma omp parallel for
  for (t2=lbp;t2<=ubp;t2++)
    a[(t1-t2)][(-t1+2*t2)] = ...
```

(b) Automatically parallelized code.



(c) Accessed tiles of the 64 × 64 *seidel* kernel executed using 8 threads (each shade of gray represents the area calculated by a single thread).

Figure 4.8 – Original and automatically parallelized *seidel* code.

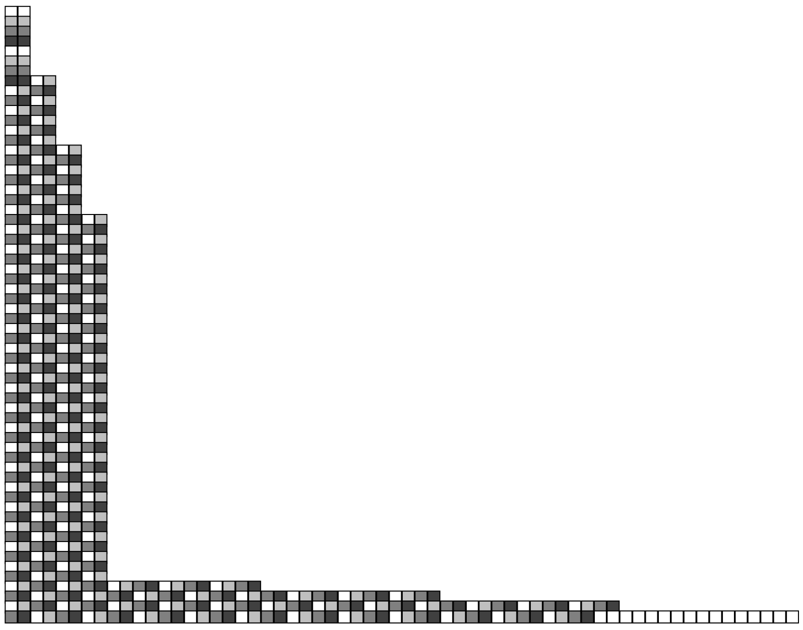


Figure 4.9 – Reconstructed trace pieces of *seidel* for the thread #0 and max_depth=1 (161 pieces)

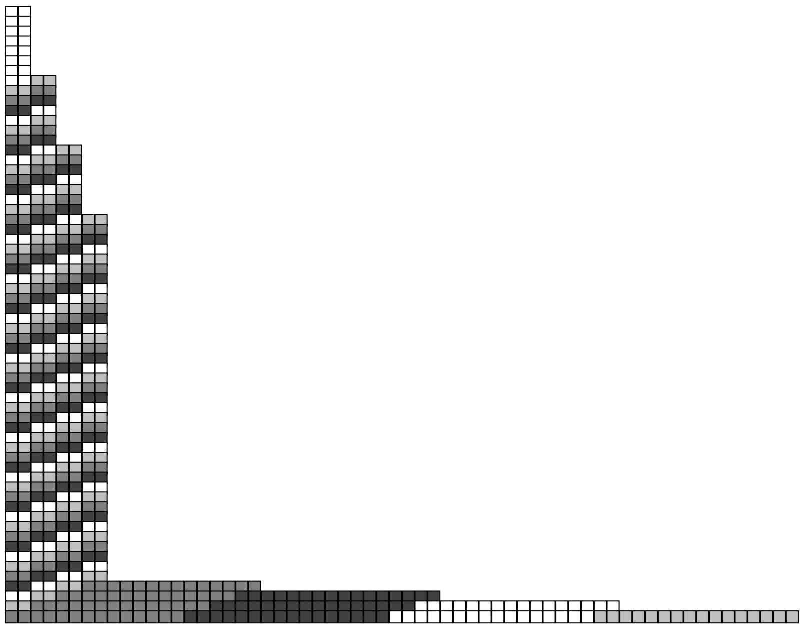


Figure 4.10 – Reconstructed trace pieces of *seidel* for the thread #0 and max_depth=2 (58 pieces)

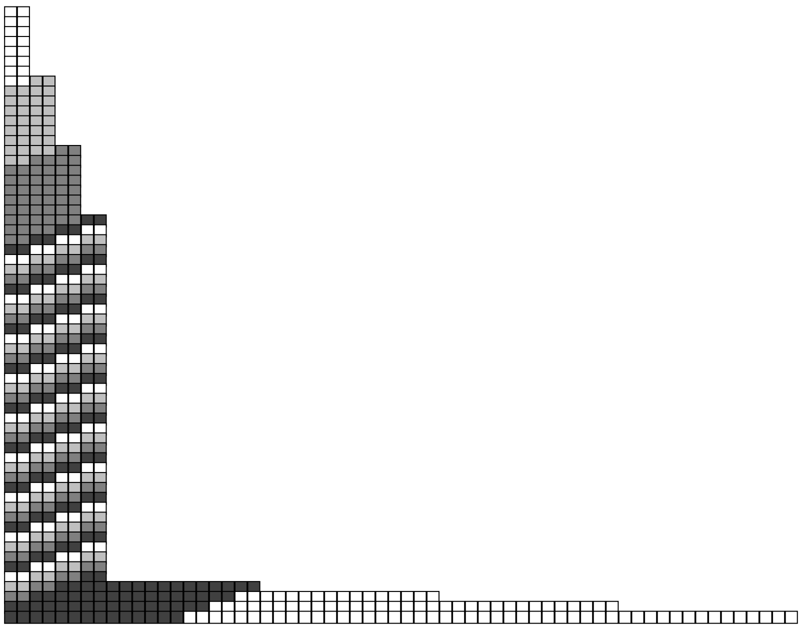


Figure 4.11 – Reconstructed trace pieces of *seidel* for the thread #0 and max_depth=3 (41 pieces)

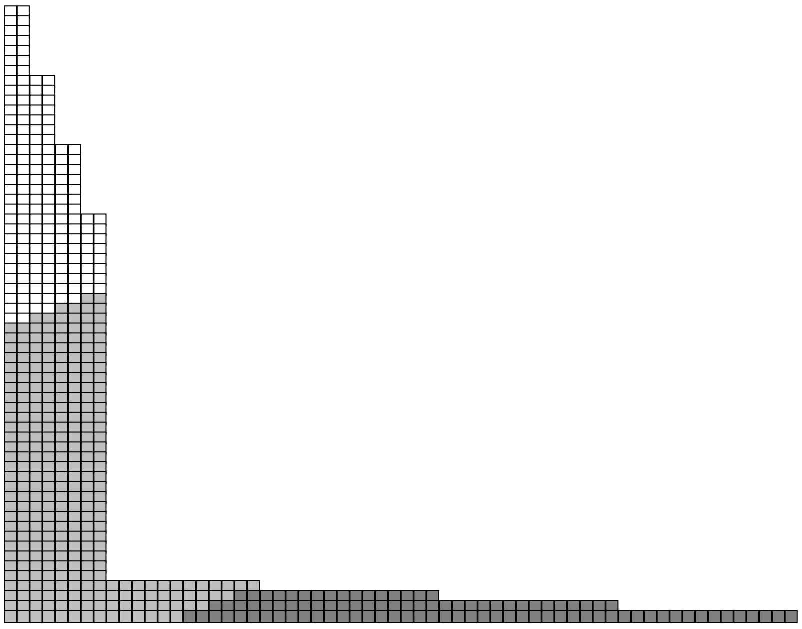


Figure 4.12 – Reconstructed trace pieces of *seidel* for the thread #0 and max_depth=4 (3 pieces). The trace is ultimately reconstructed as a single perfectly nested loop of depth 10.

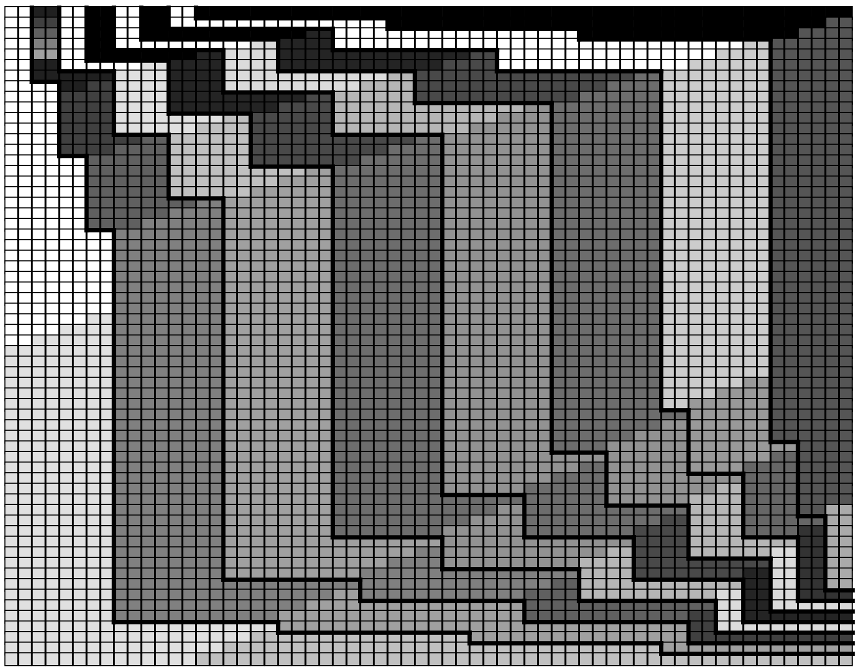


Figure 4.13 – Piecewise reconstruction of threads in the *seidel* trace of Figure 4.8c using a maximum loop depth of 4. There is not a direct correspondence of pieces among threads, which prevents the reconstruction of a single common code.

Kernel	**Refs ($\times 10^6$)**	**Pieces**	**Max. depth**	**Largest %**	**# $>$ 95%**	**Orig. depth**	**Par. level**
adi	15.99	3	21	71.20	2	4	1
covcol	402.53	2	35	99.82	1	3	1
dct	135.14	8	41	62.91	6	3	2
floyd	135.14	7167	50	0.06	5393	3	3
gemver	4.50	48	33	50.90	33	2	2
seidel	121.15	1231	47	24.60	389	6	2
ssymm	63.81	249	6	1.51	245	6	1

Table 4.3 – Reconstruction characteristics of the nontrivially reconstructed PLUTO kernels. The most complex loop for each kernel is shown. Maximum reconstruction depth is fixed to 50. “Largest %” indicates the percentage of total accesses issued by the largest reconstructed piece. “# $>$ 95%” shows the total number of pieces required to issue at least 95% of the total accesses.

internal loop (i.e., a loop that will be executed many times) the nonlinearities in them manifest more than once, potentially requiring many additional loop levels to reach an affine representation. This causes the number of affine subpieces in the trace to increase.

Figures 4.14 and 4.15 analyze how the reconstruction varies with the maximum allowed depth and with the problem size, respectively. Increasing the reconstruction depth allows to arbitrarily reduce the number of reconstructed pieces in exchange for complexity, but it has diminishing returns. Larger problems feature more nonlinearities in their trace, as internal loops with nonlinear bounds are executed more times. As shown, most reconstructed codes include a small number of large loops, making them amenable to automatic affine analysis and optimizations.

4.5 Related Work and Applications

Not many works have explored the reconstruction of loop codes from their memory access traces. This section organizes related work according to their ultimate goal, also discussing the potential applications of the exploration engine proposed in this thesis.

Clauss et al. [37] characterized program behavior using polynomial piecewise periodic and linear interpolations separated into adjacent program phases to reduce function complexity. The model can be recursively applied, interpreting coefficients of the periodic interpolation as traces in themselves. Clauss and Kenmei [36] introduced polyhedra to graphically represent the program memory behavior (including cache misses) and facilitate its understanding. Ketterlin and Clauss [78] proposed a method for trace prediction and compression based on representing memory traces as sequences of nested loops with affine bounds and subscripts. It uses a stack of terms. When a new term is pushed, it searches for a triplet of terms that can be rewritten as a loop. This approach works in a greedy way, which leads to non-minimal solutions in some cases.

One potential use of the exploration engine is cache prefetching. To improve on the one block lookahead scheme [134], Baer and Chen [19] use a prediction ta-

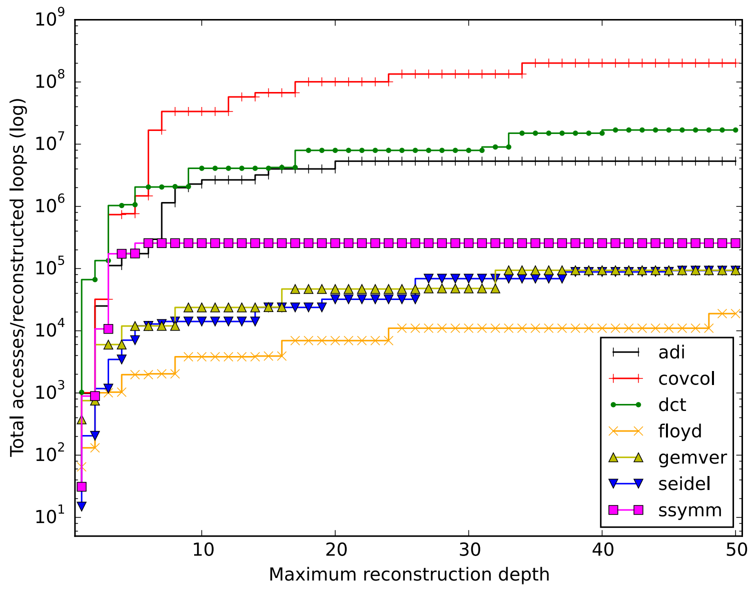


Figure 4.14 – Average number of accesses issued by each reconstructed loop with respect to the maximum allowed reconstruction depth. The y axis is logarithmic.

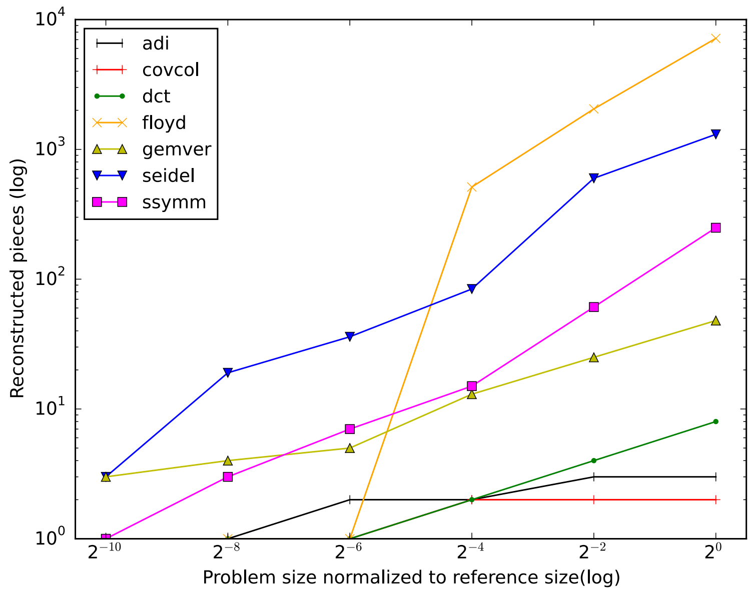


Figure 4.15 – Number of reconstructed pieces at depth=50 with respect to problem size. Both axes are logarithmic. The x axis is normalized with respect to the reference size

ble and lookahead program counter to preload regular accesses which correctly predict the stride of the innermost loop. Iacobovici et al. [68] propose a prefetcher capable of supporting up to four distinct strides. In contrast, our approach is capable of supporting an unlimited amount of strides, as well as variable trip counts. However, hardware prefetching using our loop reconstruction mechanism requires memory space to store at least the values of **U** and $\overrightarrow{w}$ for each loop, as well as $\overrightarrow{c}$ for each access instruction. The required space depends on the maximum nesting level supported. Other prefetchers integrated in the memory controller [143] could also benefit from a hardware implementation of our recognition engine to improve prediction accuracy.

To reduce remote memory accesses in NUMA architectures, good data placement is essential. kMAF [43] improves data locality by dynamically analyzing page faults of running applications and migrating threads and memory pages consequently. It is engineered into the virtual memory implementation of the operating system. Piccoli et al. [112] propose a combination of static and dynamic techniques for migrating memory pages predicted to be frequently reused. A compiler infers affine expressions for array sizes and the reuse of each memory access enclosed in loops, and inserts checks to assess the profitability of potential page migrations at runtime. Our proposal can also provide the essential information for data placement in NUMA architectures, either statically after trace-based reconstruction and reconstructed code analysis, or dynamically as a software-based prediction mechanism.

Trace-based code reconstruction is also useful for automatic parallelization. Holewinski et al. [64] use dynamic data dependence graphs derived from sequential execution traces to identify vectorization opportunities. Jimborean et al. [73] proposed a dynamic mechanism for detecting data dependences using interpolated linear functions to approximate observed memory accesses to guide speculative parallelization. Similar systems can be constructed using the proposed exploration engine, capable of analyzing dependences without the need for compiler support.

Prior research investigated the problem of designing ad-hoc memory hierarchies for embedded applications. Catthoor et al. [34] proposed a compiler-based methodology to derive optimal memory regions and associated data allocation.

Angiolini et al. [14] use a trace-based method that analyzes the access histogram to determine which memory regions to allocate to scratchpad memory [20]. Our trace-based reconstruction approach can be employed to apply affine techniques for custom memory hierarchy design for applications for which affine analysis of the source code is not feasible. This is of particular interest for IP cores, commonly included in embedded devices. It can also be employed to drive scratchpad allocation managers.

4.6 Concluding Remarks

This chapter has explored the reconstruction of affine loop codes from their memory traces, considering one instruction at a time. It also deals with the reconstruction of traces generated by parallel applications, potentially containing moderate amounts of nonlinearity. Large traces are processed in a matter of minutes without user intervention or access to source or binary codes. The proposed methodology has applications such as trace compression/storage/communication, dynamic parallelization, or memory placement and memory hierarchy design. The problem has been formulated as the exploration of a tree-like solution space, in which each node represents a point in the iteration space of a loop. The mathematical relationship amongst the nodes has been established, and the system of equations that governs the trace-based reconstruction of the code has been defined. Afterwards, methods for efficient traversal of this solution space have been proposed, as well as extensions to deal with moderate nonlinearity in the trace. Experimental evaluation has shown good performance and accuracy in reconstructing affine codes, and versatility to represent quasi-affine codes in a piecewise fashion. Furthermore, it has been shown that the problem is not trivially tractable without the proposed optimizations.

Chapter 5

Conclusions and Future Research Lines

The introduction of heterogeneous computer architectures has challenged the software community in an unpreceded way. It has caused writing an efficient program to become a very difficult and error-prone task even for experienced HPC programmers. Nevertheless, both science and industry demand more and more computing power to achieve their objectives. Compilers are a fundamental tool to address this challenge and this thesis, entitled *"Compilation techniques for automatic extraction of parallelism and locality in heterogeneous architectures"*, has made several contributions in this field.

First, we have defined a new compiler intermediate representation called KIR [4, 5]. This new IR provides the program characteristics needed for the automatic parallelization of the input sequential code. It is built on top of diKernels to handle syntactical variations of the source code [8]. These diKernels are connected with diKernel-level dependences and are grouped into execution scopes in order to recognize the computational stages of the input application. A proof-of-concept has been implemented on top of GCC [6, 9].

Next, we have targeted the generation of parallel code for multicore processors with the insertion of OpenMP directives [4]. We have introduced an automatic partitioning algorithm of the KIR focused on the minimization of the overhead of thread synchronization. The ability of our proposal to be applied in several domains has been demonstrated using a comprehensive benchmark suite that includes synthetic codes representative of frequently used diKernels, routines from dense/sparse linear algebra and image processing, and simulation applications. In addition, we have carried out a comparative evaluation in terms of effectiveness with the GCC, ICC and PLUTO compilers for the automatic parallelization of the benchmark suite. In general, the contenders fail to parallelize codes that contain both regular computations with complex control flows and irregular computations, and they do not optimize the joint parallelization of multiple loops.

Due to their increasing popularity, we have also targeted GPUs as the main exponent of manycore architectures [12, 11]. Our proposal is focused in the exploitation of data locality in the complex GPU memory hierarchy. It considers the most influent GPU programming features to generate efficient code: loop threadification, thread grouping, coalesced access to global memory, and maximum usage of registers and shared memory. The chains of recurrences enabled us to model algebraically the complex interactions between the memory accesses performed by the GPU threads. For GPU code generation, we have relied on OpenHMPP directives as they provide great understandability and portability. Our technique has been successfully applied to two representative case studies extracted from compute-intensive scientific applications. The performance evaluation on NVIDIA GPUs (with two different core architectures) has corroborated its effectiveness.

Finally, we have developed a new technique for the characterization of a program from the point of view of its memory trace [117]. This technique is able to reconstruct affine loop nests considering the memory accesses made by one instruction at a time, without user intervention or access to source or binary codes. It has been formalized as the traversal of a tree-like solution space, in which each node symbolizes a point in the iteration space of a loop. In addition, we have proposed methods for the efficient traversal of this solution space, and to support moderate nonlinearity in the trace like noise, missing points and the resulting code of the PLUTO parallelizing compiler. The experimental evaluation has proved the good performance of our proposal, its preciseness in reconstructing affine codes, and its flexibility to represent quasi-affine codes in a piecewise fashion. Applications for this technique are very varied and already studied (e.g., trace compression/storage/communication, dynamic parallelization, memory placement and memory hierarchy design).

Future research directions aim at using the trace-based reconstruction to increase the information available in the construction of the KIR, and designing a new automatic partitioning algorithm of the KIR that handles the interactions between computations for heterogeneous clusters, considering both CPU-GPU interaction and inter-node communication. For this purpose, information about the hardware is needed. Some preliminary ideas about this topic were presented in [7]. We could also include auto-tuning approaches to select the best performant variant between several candidates of a parallelized diKernel. The trace-based reconstruction technique will be improved to handle a broader range of irregular computations.

The technologies developed in Chapters 2 and 3 have been licensed to the spin-off company Appentra Solutions S.L. for the creation of Parallware [15].

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2006. Cited in p. 14, 177, 191, 207

[2] E. Albert, J. D. Lukas, and G. L. Steele. Data Parallel Computers and the FORALL Statement. *Journal of Parallel and Distributed Computing*, 13(2):185–192, 1991. Cited in p. 24, 26

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001. Cited in p. 10, 14, 23, 177, 191, 207

[4] J. M. Andión, M. Arenaz, G. Rodríguez, and J. Touriño. A novel compiler support for automatic parallelization on multicore systems. *Parallel Computing*, 39(9):442–460, 2013. Cited in p. xxi, 155, 200, 215

[5] J. M. Andión, M. Arenaz, G. Rodríguez, and J. Touriño. A parallelizing compiler for multicore systems. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 138–141, Sankt Goar, Germany, 2014. Cited in p. xxi, 155, 200, 215

[6] J. M. Andión, M. Arenaz, and J. Touriño. A New Intermediate Representation for GCC based on the XARK Compiler Framework. In *Proceedings of the 2nd International Workshop on GCC Research Opportunities (GROW) (in conjunction with the International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC))*, pages 89–100, Pisa, Italy, 2010. Cited in p. xxii, 155, 200, 215

[7] J. M. Andión, M. Arenaz, and J. Touriño. Automatic Partitioning of Sequential Applications Driven by Domain-Independent Kernels. In *Proceedings of the 15th Workshop on Compilers for Parallel Computing (CPC)*, page CDROM, Vienna, Austria, 2010. Cited in p. xxii, 157

[8] J. M. Andión, M. Arenaz, and J. Touriño. Domain-Independent Kernel-Based Intermediate Representation for Automatic Parallelization of Sequential Programs. In *Proceedings of the 6th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, pages 71–74, Terrasa, Spain, 2010. Cited in p. xxii, 155

[9] J. M. Andión, M. Arenaz, and J. Touriño. Una Nueva Representación Intermedia para GCC basada en el Entorno de Compilación XARK. In *Actas de las XXI Jornadas de Paralelismo (JP)*, pages 151–158, Valencia, Spain, 2010. Cited in p. xxi, 155, 200, 215

[10] J. M. Andión, G. L. Taboada, J. Touriño, and R. Doallo. Biblioteca de Comunicaciones Colectivas para el Lenguaje de Programación Paralela UPC. In *Actas de las XX Jornadas de Paralelismo (JP)*, pages 517–522, A Coruña, Spain, 2009. Cited in p. 8

[11] J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. Touriño. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. In *Proceedings of the 7th International Symposium on High-level Parallel Programming and Applications (HLPP)*, pages 217–238, Amsterdam, Netherlands, 2014. Cited in p. xxi, 156, 200, 216

[12] J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. Touriño. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. *International Journal of Parallel Programming (in press)*, 2015. Cited in p. xxi, 156, 200, 216

[13] D. Andrade, M. Arenaz, B. B. Fraguela, J. Touriño, and R. Doallo. Automated and Accurate Cache Behavior Analysis for Codes with Irregular Access Patterns. *Concurrency and Computation: Practice and Experience*, 19(18):2407–2423, 2007. Cited in p. 10, 69, 195, 211

[14] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the 6th International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 318–326, San Jose, CA, USA, 2003. Cited in p. 153

[15] Appentra Solutions, S.L. Parallware: The OpenMP-enabling source-to-source compiler. `http://www.appentra.com/products/parallware/`. [Last visited: September 2015]. Cited in p. xxi, 60, 100, 157, 202, 217

[16] M. Arenaz, J. Touriño, and R. Doallo. XARK: An Extensible Framework for Automatic Recognition of Computational Kernels. *ACM Transactions on Programming Languages and Systems*, 30(6):32:1–32:56, 2008. Cited in p. 20, 179

[17] K. Asanovic, R. Bodík, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. A. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. A. Yelick. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52(10):56–67, 2009. Cited in p. 6, 10, 189, 205

[18] O. Bachmann, P. S. Wang, and E. V. Zima. Chains of Recurrences - a Method to Expedite the Evaluation of Closed-form Functions. In *Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 242–249, Oxford, UK, 1994. Cited in p. 62, 69, 195, 211

[19] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 4th International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 176–186, Albuquerque, NM, USA, 1991. Cited in p. 150

[20] R. Banakar, S. Steinke, L. Bo-Sik, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES)*, pages 73–78, Estes Park, CO, USA, 2002. Cited in p. 10, 153

[21] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *Proceedings of the 19th International Conference on Compiler Construction (CC)*, volume 6011 of *LNCS*, pages 244–263, Paphos, Cyprus, 2010. Cited in p. 98

[22] C. F. Batten. *Simplified vector-thread architectures for flexible and efficient data-parallel accelerators*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2010. Cited in p. 3

[23] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The Polyhedral Model Is More Widely Applicable Than You Think. In *Proceedings of the 19th International Conference on Compiler Construction (CC)*, volume 6011 of *LNCS*, pages 283–303, Paphos, Cyprus, 2010. Cited in p. 55, 133

[24] BLAS. Basic Linear Algebra Subprograms. http://www.netlib.org/blas/. [Last visited: September 2015]. Cited in p. 86

[25] F. Bodin and S. Bihan. Heterogeneous Multicore Parallel Programming for Graphics Processing Units. *Scientific Programming*, 17(4):325–336, 2009. Cited in p. 68, 195, 211

[26] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 343–352, Vienna, Austria, 2010. Cited in p. 55, 183

[27] U. Bondhugula, A. 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)*, pages 101–113, Tucson, AZ, USA, 2008. Cited in p. 8, 47, 55, 98, 102, 134, 142, 183, 197, 199, 213, 214

[28] D. Callahan. Recognizing and Parallelizing Bounded Recurrences. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 169–185, Santa Clara, CA, USA, 1992. Cited in p. 24

[29] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks. HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs. *ACM SIGARCH Computer Architecture News*, 42(3):217–228, 2014. Cited in p. 57

[30] S. Campanoni, T. Jones, G. Holloway, G.-Y. Wei, and D. Brooks. Helix: Making the Extraction of Thread-Level Parallelism Mainstream. *IEEE Micro*, 32(4):8–18, 2012. Cited in p. 57

[31] A. Canedo, T. Yoshizawa, and H. Komatsu. Automatic Parallelization of Simulink Applications. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 151–159, Toronto, Canada, 2010. Cited in p. 58

[32] CAPS Entreprise. CAPS Compilers. https://web.archive.org/web/20140712031319/http://www.caps-entreprise.com/products/caps-compilers/. [Last visited: September 2015]. Cited in p. 62, 67, 195, 210

[33] S. Carrillo, J. Siegel, and X. Li. A Control-structure Splitting Optimization for GPGPU. In *Proceedings of the 6th ACM Conference on Computing Frontiers (CF)*, pages 147–150, Ischia, Italy, 2009. Cited in p. 71

[34] F. Catthoor, S. Wuytack, G. E. de Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom memory management methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998. Cited in p. 10, 152

[35] M. Christen, O. Schenk, and H. Burkhart. Automatic Code Generation and Tuning for Stencil Kernels on Modern Shared Memory Architectures. *Computer Science - R&D*, 26(3-4):205–210, 2011. Cited in p. 94

[36] P. Clauss and B. Kenmei. Polyhedral Modeling and Analysis of Memory Access Profiles. In *Proceedings of the 2006 IEEE International Conference on Application-Specific Systems, Architecture and Processors (ASAP)*, pages 191–198, Steamboat Springs, CO, USA, 2006. Cited in p. 150

[37] P. Clauss, B. Kenmei, and J. C. Beyler. The Periodic-Linear Model of Program Behavior Capture. In *Proceedings of the 11th International Euro-Par Conference (Euro-Par)*, pages 325–335, Lisbon, Portugal, 2005. Cited in p. 150

[38] Council on Competitiveness. Solve. The Exascale Effect: the Benefits of Supercomputing Investment for U.S. Industry. http://www.compete.org/storage/images/uploads/File/PDF%20Files/

Solve_Report_Final.pdf. [Last visited: September 2015]. Cited in p. 1, 6, 187, 203

[39] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys*, 47(4):62:1–62:27, 2015. Cited in p. 8

[40] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. Cited in p. 2, 188, 204

[41] P. J. Denning. The Locality Principle. *Communications of the ACM*, 48(7):19–24, 2005. Cited in p. 9

[42] B. di Martino and G. Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *Proceedings of the 4th International Workshop on Program Comprehension (WPC)*, pages 164–174, Berlin, Germany, 1996. Cited in p. 179

[43] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß. kMAF: Automatic Kernel-level Management of Thread and Data Affinity. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 277–288, Edmonton, Canada, 2014. Cited in p. 10, 152

[44] E. W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972. Cited in p. 6

[45] M. Duranton, K. D. Bosschere, A. Cohen, J. Maebe, and H. Munk. HiPEAC Vision 2015. https://www.hipeac.net/assets/public/publications/vision/hipeac-vision-2015_Dq0boL8.pdf. [Last visited: September 2015]. Cited in p. 6

[46] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, San Jose, CA, USA, 2011. Cited in p. 5, 189, 205

[47] European Commission. Horizon 2020 Work Programme 2014-2015. http://ec.europa.eu/research/participants/portal/desktop/en/

funding/reference_docs.html. [Last visited: September 2015]. Cited in p. 1, 187, 203

[48] European Technology Platform for High Performance Computing. ETP4HPC Strategic Research Agenda: Achieving HPC Leadership in Europe. http://www.etp4hpc.eu/wp-content/uploads/2013/06/ETP4HPC_book_singlePage.pdf. [Last visited: September 2015]. Cited in p. 10

[49] Executive Office of the President. Executive Order 13702: Creating a National Strategic Computing Initiative. Federal Register Volume 80, Issue 148, pp. 46177–46180, 2015. Cited in p. 1, 187, 203

[50] P. Feautrier. Array Expansion. In *Proceedings of the 2nd International Conference on Supercomputing (ICS)*, pages 429–441, St. Malo, France, 1988. Cited in p. 24, 28, 183

[51] B. B. Fraguela, R. Doallo, J. Touriño, and E. L. Zapata. A compiler tool to predict memory hierarchy performance of scientific codes. *Parallel Computing*, 30(2):225–248, 2004. Cited in p. 10

[52] B. Franke and M. O'Boyle. Array Recovery and High-Level Transformations for DSP Applications. *ACM Transactions on Embedded Computing Systems*, 2(2):132–162, 2003. Cited in p. 20

[53] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, 1995. Cited in p. 179

[54] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006. Cited in p. 55, 102, 183, 197, 213

[55] J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, D. A. Mallón, and B. Wibecan. UPCBLAS: a library for parallel matrix computations in Unified Parallel C. *Concurrency and Computation: Practice and Experience*, 24(14):1645–1667, 2012. Cited in p. 8

[56] J. González-Domínguez, S. Ramos, J. Touriño, and B. Schmidt. Parallel Pairwise Epistasis Detection on Heterogeneous Computing Architectures. *IEEE Transactions on Parallel and Distributed Systems (in press)*, 2015. Cited in p. 8

[57] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-Tuning a High-Level Language Targeted to GPU Codes. In *Proceedings of the 1st Innovative Parallel Computing Conference (InPar)*, pages 1–10, San Jose, CA, USA, 2012. Cited in p. 99

[58] T. Grosser, A. Größlinger, and C. Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04), 2012. Cited in p. 8, 55, 183

[59] E. Gutiérrez, O. Plata, and E. L. Zapata. Data Partitioning-based Parallel Irregular Reductions. *Concurrency and Computation: Practice and Experience*, 16(2-3):155–172, 2004. Cited in p. 24, 28, 53

[60] H. Gómez-Sousa, M. Arenaz, O. Rubinos-López, and J. Martínez-Lorenzo. Novel source-to-source compiler approach for the automatic parallelization of codes based on the method of moments. In *Proceedings of the 9th European Conference on Antennas and Propagation (EuCAP)*, pages 1–6, Lisbon, Portugal, 2015. Cited in p. 102, 197, 213

[61] M. Hall, D. Padua, and K. Pingali. Compiler Research: The Next 50 Years. *Communications of the ACM*, 52(2):60–67, 2009. Cited in p. 8, 10, 190, 206

[62] T. D. Han and T. S. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):78–90, 2011. Cited in p. 67

[63] T. D. Han and T. S. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, pages 3:1–3:8, Newport Beach, CA, USA, 2011. Cited in p. 71

[64] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *Proceedings of the 33rd ACM SIGPLAN Con-*

ference on Programming Language Design and Implementation (PLDI), pages 371–382, Beijing, China, 2012. Cited in p. 152

[65] HPC Project. Par4All. http://www.par4all.org/. [Last visited: September 2015]. Cited in p. 98

[66] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 121–130, Toronto, Canada, 2010. ACM. Cited in p. 57

[67] Z. U. Huda, A. Jannesari, and F. Wolf. Using Template Matching to Infer Parallel Design Patterns. *ACM Transactions on Architecture and Code Optimization*, 11(4):64:1–64:21, 2015. Cited in p. 59

[68] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, pages 1–11, Saint Malo, France, 2004. Cited in p. 10, 152

[69] Intel Corporation. Intel Math Kernel Library. http://software.intel.com/intel-mkl/. [Last visited: September 2015]. Cited in p. 88

[70] Intel Corporation. Intel Parallel Studio. http://intel.ly/parallel-studio-xe. [Last visited: September 2015]. Cited in p. 8, 47

[71] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically Managed Data for CPU-GPU Architectures. In *Proceedings of the 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 165–174, San Jose, CA, USA, 2012. Cited in p. 99

[72] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU Communication Management and Optimization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 142–151, San Jose, CA, USA, 2011. Cited in p. 99

[73] A. Jimborean, P. Clauss, J. M. Martínez, and A. Sukumaran-Rajam. Online dynamic dependence analysis for speculative polyhedral parallelization. In *Proceedings of the 19th International Euro-Par Conference (Euro-Par)*, pages 191–202, Aachen, Germany, 2013. Cited in p. 152

[74] J. C. Juega, J. I. Gómez, C. Tenllado, and F. Catthoor. Adaptive Mapping and Parameter Selection Scheme to Improve Automatic Code Generation for GPUs. In *Proceedings of 12th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 251:251–251:261, Orlando, FL, USA, 2014. Cited in p. 98, 183

[75] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967. Cited in p. 102, 183, 197, 213

[76] R. Karrenberg and S. Hack. Improving Performance of OpenCL on CPUs. In *Proceedings of the 21st International Conference on Compiler Construction (CC)*, volume 7210 of *LNCS*, pages 1–20, Tallinn, Estonia, 2012. Cited in p. 71

[77] C. W. Keßler and C. Smith. The SPARAMAT approach to automatic comprehension of sparse matrix computations. In *Proceedings of the 7th International Workshop on Program Comprehension (WPC)*, pages 200–207, Pittsburgh, PA, USA, 1999. Cited in p. 179

[78] A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 94–103, Boston, MA, USA, 2008. Cited in p. 150

[79] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. A Script-based Autotuning Compiler System to Generate High-performance CUDA Code. *ACM Transactions on Architecture and Code Optimization*, 9(4):31:1–31:25, 2013. Cited in p. 8, 98

[80] M. Kobayashi. Dynamic characteristics of loops. *IEEE Transactions on Computers*, 33(2):125–132, 1984. Cited in p. 103, 197, 213

[81] W. Kozaczynski, J. Q. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, 1992. Cited in p. 178

[82] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 2012. Cited in p. 94

[83] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, 1974. Cited in p. 102, 183, 197, 213

[84] E. S. Larsen and D. McAllister. Fast Matrix Multiplies using Graphics Hardware. In *Proceedings of the 14th International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, page 55, Denver, CO, USA, 2001. Cited in p. 62

[85] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 145–156, Vancouver, Canada, 2000. Cited in p. 24, 193, 208

[86] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 23rd International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, New Orleans, LA, USA, 2010. Cited in p. 67

[87] S. Lee and J. S. Vetter. Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing. In *Proceedings of the 25th International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 23:1–23:11, Salt Lake City, UT, USA, 2012. Cited in p. 61, 67, 194, 210

[88] Z. Li, A. Jannesari, and F. Wolf. Discovery of potential parallelism in sequential programs. In *Proceedings of the 42nd International Conference on Parallel Processing Workshops (ICPPW), Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*, pages 1004–1013, Lyon, France, 2013. Cited in p. 58

[89] E. Lindholm, J. Nickolls, S. F. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, 2008. Cited in p. 62, 64, 65, 194, 210

[90] D. Liu, Y. Wang, Z. Shao, M. Guo, and J. Xue. Optimally Maximizing Iteration-Level Loop Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 23(3):564–572, 2012. Cited in p. 56

[91] J. Liu, Y. Zhang, O. Jang, W. Ding, and M. T. Kandemir. A compiler framework for extracting superword level parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 347–358, Beijing, China, 2012. Cited in p. 24, 193, 208

[92] L. Liu, C. Ho, and J. Sheu. On the Parallelism of Nested For-Loops Using Index Shift Method. In *Proceedings of the 1990 International Conference on Parallel Processing, Volume 2*, pages 119–123, Urbana-Champaign, IL, USA, 1990. Cited in p. 56

[93] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, Chicago, IL, USA, 2005. Cited in p. 103, 197, 213

[94] S. Mittal and J. S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys (in press)*, 2015. Cited in p. 5, 61, 189, 193, 205, 209

[95] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. Cited in p. 2, 188, 204

[96] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th International Conference on Computing Frontiers (CF)*, pages 143–152, Ischia, Italy, 2007. Cited in p. 103, 197, 213

[97] J. C. Mouriño, A. Gómez, J. M. Taboada, L. Landesa, J. M. Bértolo, F. Obelleiro, and J. L. Rodríguez. High scalability multipole method. Solv-

ing half billion of unknowns. *Computer Science - R&D*, 23(3–4):169–175, 2009. Cited in p. 8

[98] MPI Forum. MPI: A Message-Passing Interface Standard (Version 3.0). http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. [Last visited: September 2015]. Cited in p. 7

[99] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kauffman, 1997. Cited in p. 177

[100] NVIDIA Corporation. Cg Toolkit. http://developer.nvidia.com/Cg/. [Last visited: September 2015]. Cited in p. 62, 194, 210

[101] NVIDIA Corporation. CUBLAS Library. https://developer.nvidia.com/cublas/. [Last visited: September 2015]. Cited in p. 89

[102] NVIDIA Corporation. CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/. [Last visited: September 2015]. Cited in p. 66, 72

[103] NVIDIA Corporation. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. [Last visited: September 2015]. Cited in p. 7, 62, 67, 194, 195, 210

[104] OpenHMPP Consortium. OpenHMPP Concepts & Directives. http://en.wikipedia.org/wiki/OpenHMPP. [Last visited: September 2015]. Cited in p. 7, 62, 67, 190, 206

[105] OpenMP Architecture Review Board. OpenMP Application Program Interface (Version 4.0). http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf. [Last visited: September 2015]. Cited in p. 7, 67, 190, 195, 206, 210

[106] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–118, Barcelona, Spain, 2005. Cited in p. 57

[107] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008. Cited in p. 5, 61, 189, 193, 205, 209

[108] M. Palkowski and W. Bielecki. TRACO Parallelizing Compiler. In *Soft Computing in Computer and Information Science*, volume 342 of *Advances in Intelligent Systems and Computing*, pages 409–421. Springer, 2015. Cited in p. 8, 56, 98

[109] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (5th Edition)*. Morgan Kaufmann, 2014. Cited in p. 1

[110] S. Paul and A. Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994. Cited in p. 179

[111] PGAS Community. Partitioned Global Address Space. http://www.pgas.org/. [Last visited: September 2015]. Cited in p. 8

[112] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira. Compiler Support for Selective Page Migration in NUMA Architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 369–380, Edmonton, Canada, 2014. Cited in p. 10, 152

[113] W. M. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing (ICS)*, pages 444–448, Barcelona, Spain, 1995. Cited in p. 179

[114] L.-N. Pouchet. PolyBench: The polyhedral benchmark suite. http://www.cs.ucla.edu/~pouchet/software/polybench/, 2011. [Last visited: September 2015]. Cited in p. 120, 137

[115] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: a System for Flexible Parallel Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 133–144, Beijing, China, 2012. Cited in p. 8, 57

[116] M. Ravishankar, R. Dathathri, V. Elango, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Distributed Memory Code Generation for Mixed Irregular/Regular Computations. In *Proceedings of the 20th*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 65–75, San Francisco, CA, USA, 2015. Cited in p. 55

[117] G. Rodríguez, J. M. Andión, J. Touriño, and M. T. Kandemir. Reconstructing affine codes from their memory traces. Technical Report CSE 15-001, Pennsylvania State University Technical Report, University Park, PA, USA, February 2015. Cited in p. xxi, 156, 201, 216

[118] Y. Saad. SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations (Version 2). http://www-users.cs.umn.edu/~saad/software/SPARSKIT/. [Last visited: September 2015]. Cited in p. 28

[119] D. Sampaio, R. M. d. Souza, S. Collange, and F. M. Q. a. Pereira. Divergence analysis. *ACM Transactions on Programming Languages and Systems*, 35(4):13:1–13:36, 2014. Cited in p. 71

[120] E. E. Santos. Optimal and Efficient Algorithms for Summing and Prefix Summing on Parallel Machines. *Journal of Parallel and Distributed Computing*, 62(4):517–543, 2002. Cited in p. 28

[121] S. Sato and H. Iwasaki. Automatic Parallelization via Matrix Multiplication. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 470–479, San Jose, CA, USA, 2011. Cited in p. 56

[122] J. Shin, J. Chame, and M. W. Hall. Exploiting Superword-Level Locality in Multimedia Extension Architectures. *The Journal of Instruction-Level Parallelism*, 5:3:1–3:28, 2003. Cited in p. 24, 193, 208

[123] K. Streit, J. Doerfert, C. Hammacher, A. Zeller, and S. Hack. Generalized Task Parallelism. *ACM Transactions on Architecture and Code Optimization*, 12(1):8:1–8:25, 2015. Cited in p. 8, 59

[124] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. TOP500. The List. http://www.top500.org. [Last visited: September 2015]. Cited in p. 2, 188, 204

[125] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th International Conference on Supercomputing (ICS)*, pages 18–25, Philadelphia, PA, USA, 1996. Cited in p. 179

[126] C. Teijeiro, G. L. Taboada, J. Touriño, R. Doallo, J. C. Mouriño, D. A. Mallón, and B. Wibecan. Design and Implementation of an Extended Collectives Library for Unified Parallel C. *Journal of Computer Science and Technology*, 28(1):72–89, 2013. Cited in p. 8

[127] C. Tenllado, L. Piñuel, M. Prieto, F. Tirado, and F. Catthoor. Improving superword level parallelism support in modern compilers. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 303–308, Jersey City, NJ, USA, 2005. Cited in p. 24, 193, 208

[128] The Free Software Foundation, Inc. GCC, the GNU Compiler Collection. http://gcc.gnu.org/. [Last visited: September 2015]. Cited in p. 8, 47

[129] The Khronos Group Inc. The OpenCL Specification (Version 2.0). http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf. [Last visited: September 2015]. Cited in p. 67, 195, 210

[130] The Khronos Group Inc. The OpenGL Shading Language (Version 4.4). http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf. [Last visited: September 2015]. Cited in p. 62, 194, 210

[131] The OpenACC Standards Group. The OpenACC Application Programming Interface (Version 2.0a). http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf. [Last visited: September 2015]. Cited in p. 7, 67

[132] G. Tournavitis and B. Franke. Semi-automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism using Profiling Information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 377–388, Vienna, Austria, 2010. Cited in p. 58

[133] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *Proceedings of the 2nd International Workshop on GCC Research Opportunities (GROW) (in conjunction with the International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC))*, pages 4–19, Pisa, Italy, 2010. Cited in p. 47, 55, 183

[134] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32:174–199, 2000. Cited in p. 139, 150

[135] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 389–400, Vienna, Austria, 2010. Cited in p. 8, 57

[136] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4):54:1–54:23, 2013. Cited in p. 8, 98, 183

[137] V. Volkov. Better Performance at Lower Occupancy. In *Proceedings of the 2nd GPU Technology Conference (GTC)*, San Jose, CA, USA, 2010. NVIDIA. Cited in p. 92

[138] M. Wang and F. Bodin. Compiler-directed memory management for heterogeneous MPSoCs. *Journal of Systems Architecture*, 57(1):134–145, 2011. Cited in p. 10

[139] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996. Cited in p. 10, 14, 177, 191, 207

[140] M. Wolfe. Implementing the PGI Accelerator Model. In *Proceedings of the 3rd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, pages 43–50, Pittsburgh, PA, USA, 2010. Cited in p. 67

[141] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995. Cited in p. 5, 189, 205

[142] Y. Yang, P. Xiang, J. Kong, M. Mantor, and H. Zhou. A unified optimizing compiler framework for different GPGPU architectures. *ACM Transactions on Architecture and Code Optimization*, 9(2):9:1–9:33, 2012. Cited in p. 99

[143] P. Yedlapalli, J. Kotra, E. Kultursay, M. Kandemir, C. Das, and A. Sivasubramaniam. Meeting Midway: Improving CMP Performance with Memory-Side Prefetching. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 289–298, Edinburgh, Scotland, 2013. Cited in p. 10, 152

[144] T. Yemliha, M. T. Kandemir, O. Ozturk, E. Kultursay, and S. P. Muralidhara. Code scheduling for optimizing parallelism and data locality. In *Proceedings of the 16th International Euro-Par Conference (Euro-Par)*, volume 6271 of *LNCS*, pages 204–2016, Ischia, Italy, 2010. Cited in p. 56

[145] Y. Zhang and F. Mueller. Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 24(3):417–427, 2013. Cited in p. 94

[146] B. Zhao, Z. Li, A. Jannesari, F. Wolf, and W. Wu. Dependence-Based Code Transformation for Coarse-Grained Parallelism. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores (COSMIC)*, pages 1:1–1:10, 2015. Cited in p. 59

Appendix A

Background on diKernels

The importance of computers in our society and the tremendous impact of the introduction of parallel and heterogeneous architectures on software development and maintenance have been discussed in depth in Chapter 1. It is not a recent observation that compiler technology plays an important role in reducing programming effort and, therefore, cost. During the compilation process, optimizing compilers apply automatic program analysis techniques that gather information about the code. In the literature [1, 99, 139, 3], automatic program analysis is addressed from different perspectives including reaching definition analysis, dependence analysis, live analysis and inter-procedural analysis. Although current optimizing compilers combine different types of techniques, more sophisticated approaches are still needed in order to handle the complexity of real applications. Automatic kernel recognition is one of these approaches.

This appendix is structured as follows. Section A.1 introduces the different levels in which automatic kernel recognition can be carried out. Section A.2 focuses on the domain-independent level and presents the most significative kernels in that level (called *diKernels* in this thesis).

A.1 Automatic Kernel Recognition

Automatic kernel recognition is the process of discovering program constructs in the source code. In general, it can be sketched as matching a given source code against a set of program constructs. This problem has been studied for a wide variety of application areas that range from string matching and replacement in text edition, through detection of reduction variables in parallelizing compilers, up to program synthesis and modification in software engineering. Thus, kernel recognition techniques can be classified in four levels according to the information required in order to match the set of program constructs [81]: the text level, the syntactic level, the semantic level and the concept level.

At the *text level*, programs are represented as ASCII files directly. The application of recognition techniques is limited to string matching and replacement.

At the *syntactic level*, the source code is parsed in order to build an abstract syntax tree (AST) that preserves the logical information only. Thus, information to increase the readability of the code or assist parsing, such as indentation, keywords, comments, etc., is not captured. Examples of applications are variable renaming and one-to-one translation between language constructs.

At the *semantic level*, the semantic specification of the programming language in which the program is written is captured by annotating the AST with data and control flow information. Constant propagation and common subexpression elimination are standard compiler techniques that fit in this category. In many situations it is usually insufficient to understand only the syntax and the semantics of a program. For instance, in software maintenance, programmers must have gained an adequate understanding of the functionality of the code (i.e., what the code is supposed to do) before they can modify it. This information is captured at the concept level, by annotating program ASTs with both semantic information and abstract concepts.

The concept level can be further divided. On the one hand, the *domain-independent concept level* describes the functionality of the code from the point of view of the programmer. Thus, the program is represented in terms of programming concepts such as scalar reductions, irregular reductions or array recurrences. Examples of kernel recognition techniques that work at this level are [16, 53, 113, 125].

On the other hand, the *domain-specific concept level* takes into account the knowledge about problem solving that is handled by the experts in a given application domain. For instance, imposing constraints on the access patterns of the read-only arrays of a scalar reduction enables the recognition of either the inner dot product of two vectors in the linear algebra domain or the convolution of two signals represented as vectors in the signal processing domain. Some approaches proposed in the literature are [42, 77, 110].

The resulting five levels of program information are shown in Figure A.1. Note that the levels are not mutually exclusive but inclusive. Therefore, recognizing at a higher level requires recognizing at lower levels.

A.2 Collection of diKernels

As mentioned above, the diKernels do not represent domain-specific problem solvers. Instead, they characterize the computations carried out in a program from the point of view of the compiler IR. For example, a scalar reduction diKernel represents both the sum of a series of values as well as the dot product of two vectors; and a regular reduction diKernel can represent both the dense and sparse matrix-vector products. Note that diKernels can be coded in many different ways, but they exhibit the essential properties of a program from the point of view of the automatic parallelization (reduction operations in the examples above). This section describes the diKernels used in this thesis, namely assignment, reduction and recurrence. The full collection of diKernels can be consulted in [16].

Assignment diKernels

An *assignment diKernel* consists in storing a set of values in a set of memory addresses. Within a program, addresses are typically represented by scalar variables, memory pointers or indexed variables such as arrays. Thus, different types

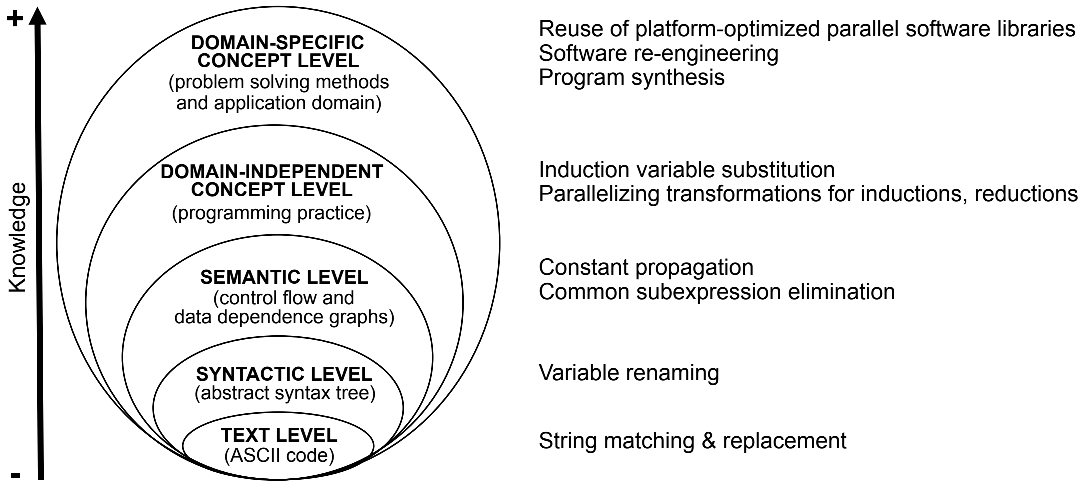

+
-
Knowledge
DOMAIN-SPECIFIC CONCEPT LEVEL
(problem solving methods and application domain)
DOMAIN-INDEPENDENT CONCEPT LEVEL
(programming practice)
SEMANTIC LEVEL
(control flow and data dependence graphs)
SYNTACTIC LEVEL
(abstract syntax tree)
TEXT LEVEL
(ASCII code)
Reuse of platform-optimized parallel software libraries
Software re-engineering
Program synthesis
Induction variable substitution
Parallelizing transformations for inductions, reductions
Constant propagation
Common subexpression elimination
Variable renaming
String matching & replacement

Figure A.1 – Five-level classification of kernel recognition techniques according to their requirements about program information. For each level, typical examples of applications are presented.

of assignment diKernels are distinguished. The simplest one is the *scalar assignment* $v = e$, which stores the value of the expression e in the memory address specified by the scalar variable v. The value e is not dependent on v, that is, neither e nor any function call within it contain occurrences of v.

A *regular assignment*, $A[i] = e(i)$ with $i \in \mathbb{N}$ taking values within the range of array A, stores the value of the expression $e(i)$ in the memory location $A[i]$ corresponding to the i^{th} entry of A. Similarly, $e(i)$ is not dependent on A. As shown in Figure A.2a, this diKernel typically represents a conflict-free loop where i is an affine expression of the loop indices.

An *irregular assignment* is represented as $A[f[i]] = e(i)$, where $i \in \mathbb{N}$, and $f[i] \in \mathbb{N}$ takes values within the range of A; f is an indirection array that introduces a compile-time unpredictable access pattern, and $e(i)$ is not dependent on A. As shown in the example of Figure A.2b, this diKernel captures the output data dependences that will appear at run time (unless f is a permutation array). Irregular assignments are usually found in computer graphics, finite element applications or sparse matrix computations.

```
for (i = 0; i < n; i++) {
  A[i] = 2;
}
```

(a) Regular assignment.

```
for (i = 0; i < n; i++) {
  A[f[i]] = 3;
}
```

(b) Irregular assignment.

```
r = 0;
for (i = 0; i < n; i++) {
  r = r + i;
}
```

(c) Scalar reduction with closed-form expression.

```
r = 0;
for (i = 0; i < n; i++) {
  r = r + A[i];
}
```

(d) Scalar reduction with array reference.

```
r = A[0];
for (i = 1; i < n; i++) {
  if (A[i] < r) r = A[i];
}
```

(e) Scalar reduction with conditional control flow.

```
for (i = 0; i < n-1; i++) {
  A[i+1] = A[i+1] + 5;
}
```

(f) Regular reduction.

```
for (i = 0; i < n; i++) {
  A[f[i]] = A[f[i]] + 3;
}
```

(g) Irregular reduction.

```
for (i = 1; i < n; i++) {
  A[i] = A[i] + A[i-1];
}
```

(h) Regular recurrence.

Figure A.2 – Synthetic codes of representative assignment, reduction and recurrence diKernels presented in Section 2.2.1.

Reduction diKernels

The *reduction diKernel* updates a memory location with a new value that depends on the current value. The most popular one is the *scalar reduction*, $v = v \oplus e(i)$, where the variable v is a scalar, $\oplus$ is an associative and commutative operator, i is an affine expression of the enclosing loop indices, and $e(i)$ is an expression that may depend on i but it must not depend on v. Scalar reductions are of widespread use, thus parallel scalar reductions are typically supported by modern programming tools. Figures A.2c–A.2e show typical examples of scalar reductions, ordered by increasing degree of complexity. In Figure A.2c there exists a closed-form expression that computes the final value of r (for illustrative purposes, $r = (n^2 + n)/2$ in this case). In Figure A.2d, $e(i)$ is an array reference $A[i]$ whose value is unknown at compile-time. Finally, Figure A.2e introduces conditional control flows to compute a minimum reduction operation.

A *regular reduction*, $A[i] = A[i] \oplus e(i)$, where $A[i]$ represents an entry of the array A, i is an affine expression of the enclosing loop indices, $\oplus$ is an associative and commutative operator, and $e(i)$ is an expression that may depend on i but it must not depend on A. In a similar manner, an *irregular reduction*, $A[f[i]] = A[f[i]] \oplus e(i)$, is characterized by the use of an indirection array f that selects the entries of the array A to be updated. Thus, this diKernel captures output and true data dependences that may appear at run time. Figures A.2f and A.2g show examples of regular and irregular reductions, respectively.

Recurrence diKernels

In contrast to reduction diKernels, a *regular recurrence*, $A[i] = A[i_1] \oplus A[i_2] \oplus \ldots \oplus e(i)$ computes a new value for $A[i]$, the indices $i, i_1, i_2 \ldots$ being affine expressions of the enclosing loop indices, and $e(i)$ not dependent on A. The distinguishing property of recurrence diKernels is that there is at least one index $i_x \in \{i_1, i_2 \ldots\}$ such that $i_x \neq i$. In a similar manner, the diKernel is an *irregular recurrence* if at least one index expression of $i, i_1, i_2 \ldots$ contains a reference to an indirection array f. Figure A.2h shows an example of regular recurrence.

Appendix B

Background on the Polyhedral Model

As discussed along this thesis, compilers need to collect the characteristics of the input code to enable the generation of high-performance code. In particular, compiler IRs need to represent large sets of dynamic executions of program instructions and be compact at the same time.

The polyhedral model [75, 83, 50, 54] has been created with this two requirements in mind. It breaks one of the key characteristics of AST-based IRs: in such representations, each statement appears only once even if it is executed many times (e.g., when it is enclosed inside a loop). On the contrary, the polyhedral model encodes each dynamic execution of an instruction as an integer point in a geometrical space. The resulting increased precision requires to deal with very large or even infinite sets. This is only feasible with a modeling based on linear algebra, which considers the solutions of systems of constraints usually expressed as affine inequalities. The part of a program that can be represented using the polyhedral model is called *Static Control Part* (from now on, *SCoP*). This model has reached production (e.g. GCC [133], LLVM (Polly) [58] and IBM XL [26]) and research compilers (e.g. PLUTO [27], PPCG [136, 74]).

The rest of this appendix gives a brief overview of the three main concepts of the polyhedral model: the iteration domains, the scattering functions and the access functions.

Iteration Domains

As mentioned before, the main difference of the polyhedral model with respect to traditional IRs is to consider each execution of a statement (*statement instances* from now on). For illustrative purposes, Figure B.1a shows the source code of a SCoP which contains two statements, namely, `S1` (see line 7) and `S2` (see line 10). Figure B.1b presents the statement instances of the execution of the code in Figure B.1a. In the polyhedral model, statements are considered as functions of the enclosing loops that may produce statement instances. For example, the notation `S2(i,j)` is used to represent the instances of the execution of the statement `S2`, which is enclosed in a loop nest with indices `i` and `j`. The ordered list of indices, `(i,j)` in this case, is called the *iteration vector*. The *iteration domain* is the set that represents all the instances of a given statement for all the possible values of its iteration domain. Hence,

$$D_{\texttt{S1}} = \{(i) \in \mathbb{Z},\, i = 0\} \subseteq \mathbb{Z}$$

$$D_{\texttt{S2}} = \{(i,j) \in \mathbb{Z}^2,\, 0 \leq i < 2 \land 0 \leq j < 4\} \subseteq \mathbb{Z}^2$$

Figure B.2 illustrates the iteration domain of `S2` ($D_{\texttt{S2}}$). Note that it is a 2-dimensional space, which is specified thanks to a set of affine constraints that depend only on the outer loop indices. It is also admitted in the polyhedral model that constraints depend on values unknown at compile time which are not modified during the execution of the whole loop (known as *parameters*). Thus, we say that the set of constraints defines a $\mathbb{Z}$*-polyhedron*.

Chapter 4 introduced a canonical form in which we reconstruct code from a trace of its memory accesses, where loop indices are forced to start in zero reducing to the half the number of constraints to store. Note that this form is equivalent. In addition, notice that matrices and vectors are used along the chapter to easily manipulate the representation.

```
double A[2][4];
double B[2][4];
double k;
int i;

#pragma scop
k = 1.25;  // S1
for (i = 0; i < 2; ++i) {
  for (j = 0; j < 4; ++j) {
    B[i][j] = k * A[i][j];  // S2
  }
}
#pragma endscop
```

(a) Source code with two statements `S1` and `S2`.

```
k = 1.25;  // S1
B[0][0] = k * A[0][0];  // S2(0,0)
B[0][1] = k * A[0][1];  // S2(0,1)
B[0][2] = k * A[0][2];  // S2(0,2)
B[0][3] = k * A[0][3];  // S2(0,3)
B[1][0] = k * A[1][0];  // S2(1,0)
B[1][1] = k * A[1][1];  // S2(1,1)
B[1][2] = k * A[1][2];  // S2(1,2)
B[1][3] = k * A[1][3];  // S2(1,3)
```

(b) List of statement instances of Figure B.1a.

Figure B.1 – Example of a SCoP.

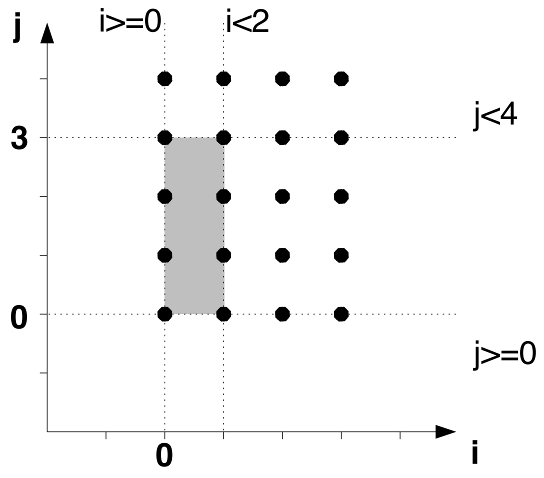


Figure B.2 – Iteration domain of the statement `S2` of Figure B.1

Scattering Functions

The iteration domain is a set, thus it does not provide any information about the ordering in which statement instances are executed. This is useful to express parallelism, but sometimes data produced by some statement instances are used by other ones and it is necessary to enforce an execution order.

We call scattering to any type of information about ordering in the polyhedral model. There are many types of ordering, such as allocation, scheduling, etc. They are all expressed in the same way, using logical stamps. In the case of scheduling, the logical stamps express at which moment a statement instance has to be executed. In the case of allocation (or placement), the logical stamp is the processor that must execute the statement instance. It is common to have one unified scattering whose dimensions have different semantics.

The scattering is normally provided as a function over the iteration vector. Thus, scattering functions are affine functions of the outer loop indices and the global parameters.

Access Functions

The polyhedral model allows to exactly model affine accesses to arrays that depend on outer loop indices and global parameters (note that scalar variables can be considered as arrays with only one memory location). Each array access depends on the statement instance, i.e., on the iteration vector. Hence, the accessed memory address can be then expressed as a linear combination of loop indices (see Equation 4.1 of Chapter 4). We have used this property for our trace-based affine reconstruction of code.

Appendix C

Summary in Spanish

Este capítulo contiene un breve resumen de los contenidos de la tesis. Tras una introducción en la que se detallan la motivación y la organización en capítulos del documento, se presenta una síntesis de las aportaciones de cada uno de ellos. Finalmente, se enumeran las principales conclusiones y líneas de investigación futuras.

Introducción

Los ordenadores han posibilitado una nueva revolución industrial basada en la tecnología digital, transformando nuestra sociedad completamente. Hoy en día, el procesamiento y transmisión de información se han convertido en las fuentes fundamentales de productividad y poder social. En particular, la Computación de Altas Prestaciones (a partir de ahora, HPC por el término en inglés *High Performance Computing*) contribuye fuertemente a la excelencia de la ciencia y a la competitividad de la industria. El HPC permite mejorar los modelos y productos existentes, así como desarrollar los nuevos más rápida y eficientemente. Estas propiedades beneficiosas del HPC han sido constatadas tanto en Europa [47] como en América [49, 38].

Debido a esta demostrada ventaja competitiva que es el HPC, la demanda de poder computacional continúa aumentando año tras año. Tanto la academia como la industria intentan crear modelos tan cercanos a la realidad como sea posible, lo que involucra enormes cantidades de información durante las simulaciones que realizan. Cada vez más datos son recolectados de los experimentos que se realizan en el mundo real, y necesitan ser almacenados y analizados. El nivel de detalle de los gráficos por computador se ha incrementado hasta un nivel que hoy en día se hace difícil distinguir las imágenes generadas por ordenador de las reales.

Al principio, la industria del silicio respondió a esta creciente demanda siendo capaz de preservar el modelo de programación secuencial existente gracias a la Ley de Moore [95]. El tamaño de los transistores se pudo reducir, doblando su número en un circuito integrado cada dos años. Este hecho permitió frecuencias de reloj más altas, memorias caché más grandes, y microarquitecturas más flexibles. Los programas se ejecutaban más rápido en cada nueva generación de procesadores.

Sin embargo, la frecuencia de reloj no pudo ser aumentada tan rápidamente desde el año 2000. Hasta dicho año, un aumento en la frecuencia de reloj llevaba aparejada una disminución del voltaje del circuito integrado. Estar propiedad, conocida como escalado de Dennard [40], permitió mantener la densidad de potencia casi constante. Pero este escalado ya no es posible porque el voltaje no puede seguir siendo reducido si queremos preservar el funcionamiento fiable de los transistores al mismo tiempo.

Por tanto, la industria ha decidido introducir varios procesadores de propósito general en el mismo chip para aprovechar el aumento en el número de transistores disponibles. Este cambio arquitectural ha dado origen, de este modo, a la *era multinúcleo*. La lista TOP500 [124] reúne, dos veces al año, la información sobre los 500 supercomputadores más potentes. Los primeros procesadores multinúcleo aparecieron en la lista de noviembre de 2011. A partir de ese momento, los núcleos por zócalo se han incrementado hasta dieciséis y ya no existen procesadores con un sólo núcleo en dicha lista. Y como los programas secuenciales sólo se ejecutan en uno de los núcleos del procesador, que no se están volviendo más rápidos, la comunidad de desarrolladores de software se ha visto forzada a desarrollar y usar herramientas de programación paralela.

Otro modo de luchar contra los problemas de disipación de calor es el uso de procesadores especializados. Comúnmente llamados aceleradores, éstos se han diseñado para llevar a cabo sólo tareas específicas pero más eficientemente que los procesadores de propósito general. Existe un amplio rango de aceleradores: tarjetas gráficas (GPUs), el Intel Xeon Phi, FPGAs y ASICs. Su uso está creciendo en los últimos años [107, 94], como se puede comprobar en el porcentaje de supercomputadores con aceleradores de la lista TOP500. Con pequeños comienzos en junio de 2006, los aceleradores representan hoy el 34 % del rendimiento total. Así, los sistemas de computación de altas prestaciones actuales tienen arquitecturas heterogéneas que combinan distintos tipos de elementos de procesamiento para alcanzar gran rendimiento con bajo consumo. Nótese que esta tendencia de diseño también se está siguiendo en los sistemas no-HPC (sobremesas, portátiles, móviles y dispositivos embebidos), e incluso en el diseño de microprocesadores para resolver el problema del *dark silicon* [46].

Los nuevos diseños de microprocesador han impactado enormemente en la jerarquía de memoria de los ordenadores, y la velocidad de la memoria está evolucionando a un menor ritmo que la velocidad del procesador [141]. Varios ejemplos ilustran esta idea. En primer lugar, los sistemas de memoria actuales para procesadores multinúcleo incluyen tres o cuatro niveles de caché para reducir la latencia de acceso, pero la colocación de los datos debe ser optimizada para conseguir este objetivo. En segundo lugar, cada nodo de un cluster normalmente incluye varios procesadores multinúcleo que tienen módulos de memoria DRAM asociados a ellos. Todos los núcleos del nodo pueden acceder a todos los módulos de memoria a través de una red de interconexión, pero a latencias diferentes. Estos sistemas se conocen como arquitecturas NUMA. En tercer lugar, la mayoría de aceleradores tienen su propia jerarquía de memoria, separada de la memoria para los procesadores de propósito general. Por lo tanto, para obtener buen rendimiento, las transferencias de memoria deben ser minimizadas entre estas dos jerarquías.

Los desarrolladores de software no están entrenados para manejar todos estos niveles de complejidad creciente [17]. Modelar la realidad en software es ya una tarea difícil sin ser consciente de la arquitectura del computador subyacente. Esta dificultad en desarrollar nuevo software, así como el alto coste de reescribirlo, con las consiguientes pruebas, provoca que cantidades considerables de software antiguo siga todavía en uso. Y la mayoría de este software fue escrito para arquitecturas mononúcleo. El problema con el código paralelo antiguo es todavía peor. Estos códigos fueron optimizados para lograr rendimiento pico en un sistema determinado, y normalmente asumen especificidades sobre la arquitectura subyacente y usan extensiones de lenguajes de programación no estándar, código ensamblador, etc. Por contra, lo deseable es que el rendimiento de un programa escale automáticamente en los ordenadores futuros.

Los compiladores han jugado un rol principal para superar la primera crisis del software. Han sido una pieza esencial para la actual predominancia de los lenguajes de programación de alto nivel. El rendimiento del código binario generado por un compilador suele ser muy cercano al rendimiento pico de la máquina, lo cuál es extremadamente difícil de conseguir manualmente para grandes programas [61]. Además, las potentes abstracciones de los lenguajes de programación de alto nivel hacen que los desarrolladores sean más productivos. Y los compiladores son capaces de proporcional esta abstracción sin penalización en el rendimiento en diferentes arquitecturas.

Asimismo, se necesita una reingeniería significativa del software existente para soportar el uso de las nuevas funcionalidades ofrecidas por el hardware. Debido al alto coste de las transformaciones manuales, una aproximación automática tendría grandes beneficios. Incluso cuando el paralelismo y la heterogeneidad se tienen en cuenta desde el principio de un proyecto software, escribir programas eficientes es una tarea difícil. Creemos que los compiladores serán una pieza crítica para superar este reto.

Esta tesis se centra en el desarrollo de técnicas de compilación para la extracción automática de paralelismo y localidad en arquitecturas heterogéneas. Por una parte, proponemos la paralelización automática de código secuencial en sistemas multinúcleo y GPUs. Hemos escogido una aproximación fuente-a-fuente que genera código paralelo anotado con directivas de compilación OpenMP [105] / OpenHMPP [104] para facilitar la interacción con los expertos del dominio de aplicación, y para permitir la generación de código binario con rendimiento pico gracias a los compiladores actuales y futuros. Por otra parte, proponemos el modelado del comportamiento del programa desde el punto de vista de los accesos a memoria. A continuación se podrían aplicar tanto las técnicas de optimización de memoria existentes basadas en el modelo poliédrico (véase el Apéndice B) como cualquier otra técnica de optimización estática o dinámica en ausencia del código fuente y/o binario. La organización de esta tesis es la siguiente:

- El Capítulo 2 introduce una nueva representación intermedia del compilador basada en el concepto de los núcleos computacionales independientes del dominio (véase el Apéndice A) denominada KIR, la cuál es insensible a las variaciones sintácticas del código fuente (p. ej., uso de arrays, punteros o flujos de control complejos), y expone varios niveles de paralelismo al compilador. A continuación, el capítulo presenta una estrategia de particionamiento automática para mapear el paralelismo expuesto por la KIR en hardware moderno basado en procesadores multinúcleo.
- El Capítulo 3 contribuye una nueva técnica para reescribir automáticamente programas secuenciales en su homólogo paralelo para sistemas heterogéneos basados en GPUs. Esta aproximación explota la característica arquitectura hardware de la tarjeta gráfica, en particular su jerarquía de memoria.
- El Capítulo 4 presenta la reconstrucción automática de bucles afines a partir de una traza de sus accesos a memoria, tanto para códigos secuenciales como paralelos. Nuestra propuesta soporta también cantidades moderadas de no linealidad.

A continuación se resumen las principales ideas de cada capítulo.

Un Nuevo Compilador para Sistemas Multinúcleo

A pesar de los grandes avances en la tecnología de los compiladores durante las últimas décadas [1, 3, 139], los compiladores de producción actuales suelen fallar al paralelizar incluso programas secuenciales sencillos. La principal razón de este fallo es que abordan la detección automática de paralelismo ejecutando análisis de dependencias clásico en representaciones intermedias basadas en sentencias. Estas representaciones intermedias son apropiadas para la generación de código, pero no para el análisis de aplicaciones completas ya que son extremadamente sensibles a las variaciones sintácticas en el código fuente. Así, los compiladores paralelizadores actuales están dirigidos por modelos matemáticos que respetan todas las dependencias presentes en estas representaciones intermedias, incluso si sólo son meros artefactos de implementación.

En esta tesis se presenta una nueva aproximación a la paralelización automática de programas secuenciales basada en el concepto de *diKernel* (véase Apéndice A). La primera etapa es la construcción de una nueva representación intermedia construida sobre los diKernels, denominada KIR, que expone varios niveles de paralelismo en los programas secuenciales. La segunda etapa es una técnica de particionamiento automática que genera una versión paralela equivalente de una aplicación secuencial para ser ejecutada en procesadores multinúcleo.

La construcción de la KIR se realiza en tres pasos: en primer lugar, la construcción de los diKernels del programa y sus relaciones de dependencias de datos; en segundo lugar, la construcción de dependencias de flujo entre diKernels; y, en tercer lugar, la construcción de la jerarquía de ámbitos de ejecución, que refleja las etapas de computación del programa secuencial y agrupa a los diKernels en dichas etapas.

La nueva técnica de particionamiento automático que transforma un programa secuencial en un homólogo paralelo para procesadores multinúcleo toma, como entrada, la KIR. El método consiste en dos pasos. El primer paso es filtrar aquellas dependencias presentes en la KIR que no evitan la paralelización de la aplicación (denominadas *dependencias a nivel de diKernel espúreas*). El segundo paso es la construcción de una estrategia de paralelización basada en OpenMP que sea eficiente para el programa secuencial en su conjunto.

La idea clave detrás de la estrategia de paralelización es encontrar el camino critico de la KIR y ejecutar las computaciones que contiene en una única región paralela para así minimizar la creación y destrucción de hilos. Nuestra aproximación se basa en la existencia de transformaciones paralelizadoras diseñadas para cada tipo de diKernel. De este modo, el camino crítico es el camino más largo de la KIR que sólo contiene dependencias de flujo y diKernels paralelizables. La explotación del paralelismo de grano fino con instrucciones SIMD, en particular para diKernels no paralelizables, queda fuera del ámbito de esta tesis ya que ha sido abordado satisfactoriamente por otras técnicas complementarias [91, 127, 122, 85].

La arquitectura abordada en esta parte de la tesis son los procesadores multinúcleo. En general, el paralelismo existente en los diKernels paralelizables será suficiente para generar unos pocos hilos de grano grueso para ser ejecutados en el procesador multinúcleo. Consecuentemente, cuando la KIR tiene varios caminos que comparten computaciones, las partes no compartidas se serializan en un único camino en la región paralela. Nótese que no es necesaria sincronización entre las partes no compartidas ya que no están conectadas por dependencias a nivel de diKernel.

Dada una región paralela de un camino crítico, nuestra estrategia de particionamiento automático minimiza la sobrecarga de sincronización entre diKernels analizando la producción y el consumo de los valores de las distintas variables. Asimismo, nuestra propuesta también minimiza la creación y destrucción de hilos. Si la región paralela está contenida dentro de un bucle, las directivas OpenMP `parallel` se mueven para envolver a dicho bucle. El camino crítico queda confinado entre barreras, y las restantes computaciones se aíslan en regiones `single`. Esta situación es muy común en las simulaciones numéricas.

En resumen, nuestra aproximación permite la paralelización automática de aplicaciones completas en arquitecturas multinúcleo minimizando la sobrecarga que conlleva. La KIR refleja naturalmente la estructura del código fuente y evita así la violación de las dependencias de flujo de datos especificadas por el programador.

Paralelización Automática para GPGPU teniendo en cuenta la Localidad

El uso de las tarjetas gráficas (a partir de ahora GPUs, por *Graphics Processing Units*) para la computación de propósito general (GPGPU) se ha incrementado dramáticamente en los últimos años [107, 94] debido, principalmente, a dos razones. Por una parte, la industria del hardware no ha sido capaz de satisfacer las demandas crecientes de potencia computacional a la vez que preservaba el modelo de programación secuencial. Por otra parte, las GPUs ofrecen un poder computacional enorme a bajo coste debido a la presión de la industria del videojuego. Esto ha llevado al desarrollo de nuevos modelos de programación para integrar estos aceleradores (GPUs, pero también otros dispositivos con arquitectura *many-core* como el Intel Xeon Phi) con lenguajes de programación de alto nivel, dando lugar a los sistemas de computación heterogéneos.

El principal inconveniente de estos sistemas es que su heterogeneidad queda expuesta al desarrollador. Como se ha mencionado, programar no es fácil, y las arquitecturas paralelas lo hacen todavía más difícil ya que requieren tareas adicionales para paralelizar y optimizar el rendimiento. De este modo, los desarrolladores tienen que tratar con muchas características de bajo nivel y limitaciones a través de librerías para la programación de GPUs. Explotar la localidad es clave para obtener buen rendimiento, y es más complicado en las GPUs que en las CPUs debido a la compleja jerarquía de memoria de las GPUs. Las directivas de compilación han demostrado lograr portabilidad y buen rendimiento en estas arquitecturas al mismo tiempo [87]. Por tanto, creemos que una aproximación basada en directivas es la elección adecuada para la paralelización automática de aplicaciones en GPUs desarrollada en esta tesis.

Las GPUs fueron diseñados para la manipulación rápida y eficiente de las imágenes que se muestran en las pantallas. Ciertas etapas del pipeline gráfico realizan operaciones en punto flotante con datos independientes, como la transformación de las posiciones de los vértices o la generación de los colores de los píxeles. Por lo tanto, las GPUs ejecutan miles de hilos concurrentes SIMD requiriendo acceso a memoria con gran ancho de banda. Este objetivo de diseño se logra porque las GPUs dedican más transistores que las CPUs al procesamiento de datos, en lugar de al almacenamiento en caché y al flujo de control. La transición de los *shaders* de función fija a los programables ha hecho que estos recursos computacionales estén disponibles para la programación de propósito general.

Los primeros enfoques GPGPU (OpenGL [130], CG [100]) obligaban a los programas a parecerse a las aplicaciones gráficas que dibujaban triángulos y polígonos, limitando así la accesibilidad de las GPU. Sin embargo, NVIDIA presentó en noviembre de 2006 la *Compute Unified Device Architecture (CUDA)* [103, 89], que permite el uso de C como lenguaje de programación para GPU. Nótese que la generación de código de GPGPU eficiente con CUDA requiere que el programador maneje explícitamente la arquitectura de hardware de la GPU.

CAPS Entreprise ofrece un conjunto completo de herramientas de software para desarrollar aplicaciones paralelas de alto rendimiento destinados a sistemas heterogéneos basados en aceleradores. La más relevante son los *CAPS Compilers* [32], que generan CUDA [103] y OpenCL [129] a partir de una aplicación secuencial anotada con directivas de compilación. Los enfoques basados en directivas (como el conocido OpenMP [105]) tratan de reducir el esfuerzo de programación y proporcionar códigos más legibles. De esta manera, estos enfoques facilitan la interacción entre los expertos del dominio de aplicación y los programadores. Las versiones secuencial y paralela coexisten en el mismo archivo, ofreciendo una manera gradual para migrar aplicaciones. Los códigos desarrollados son independientes de la plataforma hardware y los nuevos aceleradores soportadas por el traductor se explotan de forma automática. Además, se ha demostrado que es posible conseguir un rendimiento razonable en comparación con los códigos para GPU escritos a mano. Por lo tanto, consideramos que las directivas de compilación ofrecen un instrumento conveniente para la paralelización automática de aplicaciones secuenciales en sistemas heterogéneos basados en GPUs. En concreto, hemos escogido el estándar OpenHMPP (antes conocido como HMPP [25]) y su extensión HMPPCG (HMPP Codelet Generator) porque estos conjuntos de directivas proporcionan una funcionalidad única para transformar anidamientos de bucles, lo cual permite un mayor grado de optimización del código para GPU generado, y tanto su compilador como su entorno de ejecución son los más maduros.

Las cadenas de recurrencias (de ahora en adelante, *chrecs*) son un formalismo algebraico para representar funciones de forma cerrada que se han utilizado con éxito para acelerar la evaluación de una función en una serie de puntos de un intervalo regular [18]. Las chrecs, que son proporcionadas por la KIR, han demostrado ser una representación potente de los complejos bucles y accessos a memoria que aparecen en aplicaciones reales [13]. En el contexto de la generación de código GPGPU eficiente, es necesario no sólo tener en cuenta los accesos a memoria requeridos en una porción de código, sino también qué accesos se emiten por parte de cada hilo. Afortunadamente, las chrecs son un mecanismo poderoso para este propósito al permitir su *instanciación* para evaluar las funciones de forma cerrada de la chrec para los valores particulares de los índices de bucle asignados a cada hilo de la GPU.

En el Capítulo 2 de esta tesis se presenta un enfoque independiente del hardware para procesadores multinúcleo basado en OpenMP que ha demostrado ser eficaz y eficiente. Sin embargo, para obtener el máximo rendimiento en la GPU, el código generado debe explotar su característica arquitectura de hardware (en particular, la compleja jerarquía de memoria). Así, el Capítulo 3 introduce una nueva técnica de generación de código que extiende el enfoque previo teniendo en cuenta las características de programación de GPUs con mayor impacto en el rendimiento: threadificación de bucles (la política que guía la creación de hilos GPU y qué código ejecutarán), agrupamiento de hilos (ya que éstos se lanzan en *warps* que ejecutan una instrucción común a la vez), coalescencia en los accesos a memoria global (los accesos emitidos por los diferentes hilos son manejados mediante una única transacción), y uso máximo de registros y memoria compartida (las más rápidas de la jerarquía). Esta técnica, tras diversos análisis que aprovechan la potencia de la KIR y las chrecs, ha conseguido generar automáticamente código competitivo en rendimiento con la implementación realizada por programadores expertos para dos ejemplos representativos extraídos de aplicaciones científicas.

Reconstrucción Afín de Código basada en Trazas

Muchas técnicas de compilación para la optimización estática y dinámica se basan en el conocimiento del código fuente de la aplicación, como las presentadas en los capítulos 2 y 3. Por desgracia, el código fuente no siempre está disponible para el compilador. En los sistemas embebidos, por ejemplo, es común encontrar núcleos con funcionalidad de alto nivel bien definida, pero cuyos componentes internos son opacos para el diseñador del sistema completo o para el programador. Incluso cuando el código fuente está disponible, los desarrolladores de software pueden utilizar estructuras de datos y de control complejos, incluyendo técnicas de ofuscación de código, que enmascaran la lógica de la aplicación subyacente y evitan su análisis estático. En estos casos, la explotación de la localidad se convierte en clave para conseguir buen rendimiento.

El Capítulo 4 presenta un novedoso marco matemático para la reconstrucción afín de los bucles de un programa a partir de una traza de sus accesos a memoria, sin la intervención del usuario o el acceso a los códigos fuente y/o binarios de la aplicación. La propuesta se basa en la observación de que, en los bucles afines, las diferencias entre dos accesos consecutivos deben ser construidas como combinaciones lineales de los índices de bucle. Los códigos afines representan una clase importante de problemas en muchos dominios de computación, tales como la supercomputación, los sistemas embebidos, o las aplicaciones multimedia. En su mayor parte, estos códigos ejecutan grandes bucles regulares, con partes de control estático que dependen sólo de combinaciones lineales de los índices del bucle, accediendo y operando en grandes conjuntos de datos. Este es el tipo de códigos que se suele modelar y optimizar utilizando el enfoque poliédrico [75, 83, 54, 27] (véase el Apéndice B). Téngase en cuenta también que algunos accesos irregulares en tiempo de compilación son afines en tiempo de ejecución [60]. Por lo tanto, la propuesta desarrollada en este capítulo puede mejorar el rendimiento de las paralelizaciones automáticas presentadas en los Capítulos 2 y 3. Además, hemos desarrollado extensiones para el manejo de trazas no linealmente perfectas para permitir puntos adicionales o ausentes. Por otra parte, también hemos abordado la reconstrucción de códigos afines paralelizados automáticamente, que suelen incluir cantidades moderadas de no linealidad, modificando el algoritmo básico para procesar la traza de accesos a memoria por partes.

El algoritmo de reconstrucción propuesto trata con el flujo de direcciones generadas por una única instrucción de acceso a memoria. Por lo tanto, suponemos que la traza contiene por lo menos la dirección de memoria de la instrucción que ha emitido el acceso (o un modo similar que identifique unívocamente la instrucción), y la localización en memoria accedida. Este formato traza de memoria se puede generar, por ejemplo, con Intel Pin [93]. En el caso general, es de esperar que el archivo de trazas contenga toda la ejecución del programa, incluyendo múltiples bucles anidados y secciones que no sean bucles. La detección de bucles en trazas de ejecución cae fuera del alcance de esta tesis ya que se ha discutido en trabajos anteriores [80, 96]. Por lo tanto, asumimos que existe un mecanismo fiable para detectar bucles en la traza.

Nuestra propuesta está diseñada para recrear la misma secuencia de accesos que la traza de memoria proporcionada. El algoritmo que proponemos es esencialmente una exploración guiada por el espacio de potenciales soluciones, dirigida por las diferencias de primer orden entre las direcciones de memoria a las que accede una instrucción dada. Cada nodo en este espacio en forma de árbol representa un punto en el espacio de iteración del bucle. Su raíz es un bucle trivial que genera los primeros dos accesos de la traza. Los hijos de un nodo en el árbol son los índices que pueden seguir de inmediato a los padres en el espacio de iteración. A partir de la raíz, el motor de exploración comienza incorporando uno a uno los accesos a reconstruir, descendiendo un nivel en el árbol, hasta que encuentra una solución para toda la traza o determina que ningún bucle afín es capaz de generar la secuencia de accesos observada.

Gracias al desarrollo matemático presentado en el cuerpo de esta tesis, está garantizado que nuestro algoritmo encontrará la reconstrucción afín mínima para cualquier traza de acceso lineal. Sin embargo, las aplicaciones en los sistemas reales requieren tratar con diversos grados de no linealidad e incertidumbre en la traza. Una posible situación son aquellas trazas que contienen principalmente referencias emitidas por un sólo acceso, pero mezclados con una cierta cantidad de accesos no relacionados. En esta situación, la exploración del espacio de soluciones se puede modificar para descartar algunas observaciones antes de concluir que la rama explorada del árbol no puede dar lugar a una solución válida. Hemos implementado esta funcionalidad guiada estadísticamente, para evitar desechar demasiados puntos y llegar a una versión muy simplificada del bucle.

En algunas situaciones, en un archivo de traza pueden faltar algunas entradas para que sea completamente representable por un bucle afín. El motor de exploración se puede configurar para insertar observaciones “desaparecidas” para intentar llegar a una representación lineal. De este modo, nuestro método inserta tentativamente estos accesos y continúa la exploración. Se crea un punto de retroceso para el caso en que no se alcance una solución mediante la exploración de esa rama. Asimismo, se permite especificar un parámetro de tolerancia para evitar la exploración de ramas improbables.

Por último, también se ha considerado la reconstrucción de bucles a partir de las trazas generadas por códigos afines que han sido paralelizadas automáticamente por PLUTO [27]. Para ello suponemos que cada hilo genera su propia traza de accesos a memoria o, equivalentemente, que una única traza conjunta incluye el identificador del hilo que emite cada acceso. Aquellos códigos que se han paralelizado añadiendo simplemente una directiva OpenMP `parallel for` en el nivel de anidamiento adecuado tendrán trazas completamente afines siempre que las iteraciones se hayan distribuido de forma estática. En estos casos, se puede aplicar el mismo algoritmo de reconstrucción utilizado para los códigos secuenciales. La reconstrucción obtenida para cada hilo será exactamente la misma, a excepción de los límites de iteración, por lo que sería trivial reconstruir el código original.

Sin embargo, este no es el caso para los códigos con dependencias más complejas. Con el fin de paralelizar automáticamente estos bucles, PLUTO debe realizar más cambios en código secuencial para satisfacer las dependencias en la versión homóloga paralela. Para ello introduce funciones no lineales como límites de los bucles internos, haciendo que el código resultante no sea representable como un único bucle afín de profundidad similar a la original. Un posible enfoque para modelar este tipo de códigos es reconocer la traza por partes, es decir, como una secuencia de bucles. De esta manera, se garantiza encontrar siempre una reconstrucción para la traza (aunque su tamaño solamente está limitado por el número de accesos en la traza). Téngase en cuenta que, en este caso, no es siempre posible reconstruir un código SPMD común para todos los hilos. Sin embargo, la técnica presentada permite construir una forma equivalente afín a trozos de códigos que no lo están en su forma original, permitiendo su análisis afín y optimización.

Conclusiones y Líneas de Trabajo Futuro

La introducción de las arquitecturas de computador heterogéneas ha retado a la comunidad de desarrollo de software de un modo sin precedente. Esto ha causado que escribir un programa eficiente se haya convertido en una tarea muy difícil y propensa a errores, incluso para programadores expertos en HPC. Sin embargo, tanto la ciencia como la industria demandan cada vez más poder computacional para alcanzar sus objetivos. Los compiladores son una herramienta fundamental para abordar este reto y esta tesis, titulada *"Técnicas de compilación para la extracción de paralelismo y localidad en arquitecturas heterogéneas"*, hace varias contribuciones en este campo.

En primer lugar, se ha definido una nueva representación intermedia del compilador denominada KIR [4, 5]. Esta nueva representación intermedia proporciona las características del programa necesarias para la paralelización automática del código secuencial de entrada. Se construye sobre los núcleos independientes del dominio (diKernels) para soportar las variaciones sintácticas del código fuente. Estos diKernels se conectan con dependencias y se agrupan en ámbitos de ejecución para reconocer las etapas computacionales de la aplicación de entrada. Se ha implementado una prueba de concepto sobre GCC [6, 9].

A continuación se ha abordado la generación de código paralelo para procesadores multinúcleo mediante la inserción de directivas OpenMP [4]. Se ha presentado un algoritmo de particionamiento automático de la KIR centrado en minimizar la sobrecarga causada por la sincronización de los hilos. La capacidad de nuestra propuesta para ser aplicada en diferentes dominios ha sido demostrada utilizando un completo juego de programas de prueba que incluye códigos sintéticos representativos de diKernels frecuentemente usados, rutinas de álgebra lineal densa/dispersa y procesamiento de imagen, y aplicaciones de simulación. Asimismo, se ha llevado a cabo una evaluación comparativa en términos de efectividad de los compiladores GCC, ICC y PLUTO sobre su capacidad para la paralelización automática del conjunto de programas de prueba. En general, los contendientes fallan en la paralelización de códigos que contengan computaciones regulares con flujos de control complejos y computaciones irregulares, y no optimizan la paralelización conjunta de varios bucles.

Debido a su creciente popularidad, en esta tesis también se han considerado las tarjetas gráficas (GPUs) como principal exponente de las arquitecturas *manycore* [12, 11]. Nuestra propuesta se centra en explotar la localidad de datos en la compleja jerarquía de memoria de la GPU. Ha sido diseñada considerando las características de programación de GPUs más influyentes para generar código eficiente: threadificación de bucles, agrupamiento de threads, accesos coalescentes a la memoria global, y uso máximo de registros y memoria compartida. Hemos modelado algebraicamente las complejas interacciones entre los accesos a memoria de los diferentes hilos de la GPU gracias a las cadenas de recurrencias. Para la generación de código para GPU, nos hemos basado en directivas OpenHMPP ya que proporcionan gran inteligibilidad y portabilidad. Nuestra técnica ha sido aplicada satisfactoriamente a dos casos de estudio representativos de aplicaciones científicas. La evaluación del rendimiento en tarjetas gráficas de NVIDIA (con dos arquitecturas diferentes) ha corroborado su eficiencia.

Finalmente, hemos desarrollado una nueva técnica para la caracterización de un programa desde el punto de vista de su traza de accesos a memoria [117]. Esta técnica es capaz de reconstruir anidamientos de bucles afines considerando los accesos a memoria emitidos por una instrucción, sin intervención del usuario o acceso al código fuente o binario. Se ha formalizado como el recorrido de un espacio de soluciones con forma de árbol, en el cual cada nodo simboliza un punto en el espacio de iteración del bucle. Además, se han propuesto métodos para recorrer eficientemente este espacio de soluciones, y para soportar cantidades moderadas de no linealidad en la traza como ruido, puntos que faltan, o el código resultante del compilador paralelizador PLUTO. La evaluación experimental ha probado el buen rendimiento de nuestra propuesta, su precisión en la reconstrucción de códigos perfectamente afines, y su flexibilidad para representar por partes códigos casi-afines. Las aplicaciones de esta técnica son muy variadas y ya estudiadas (p. ej., compresión/almacenamiento/comunicación de trazas, paralelización dinámica, colocación de datos en memoria y diseño de la jerarquía de memoria).

Como líneas de investigación futuras podemos considerar la utilización de la reconstrucción basada en trazas para incrementar la información disponible para construir la KIR, y el diseño de un nuevo algoritmo de particionamiento automático de la KIR que considere las interacciones entre las computaciones en clusters heterogéneos, tanto la interacción CPU-GPU como la comunicación entre nodos. Para este propósito es necesario recopilar información sobre el hardware. Asimismo se podrían incluir aproximaciones basadas en *autotuning* para seleccionar la variante con mayor rendimiento entre las distintas paralelizaciones candidatas de un diKernel. La técnica de reconstrucción basada en trazas podría ser mejorada para manejar un rango más amplio de aplicaciones irregulares.

La tecnología desarrollada en los Capítulos 2 y 3 ha sido licenciada a la empresa de base tecnológica Appentra Solutions S.L. para la creación de Parallware [15].

Appendix D

Summary in Galician

Este capítulo contén un breve resumo dos contidos da tese. Tras unha introdución na que se detallan a motivación e a organización en capítulos do documento, preséntase unha síntese das achegas de cada un deles. Finalmente, enuméranse as principais conclusións e liñas de investigación futuras.

Introducción

Os computadores posibilitaron unha nova revolución industrial baseada na tecnoloxía dixital, transformando a nosa sociedade completamente. Hoxe en día, o procesamento e transmisión de información convertéronse nas fontes fundamentais de produtividade e poder social. En particular, a Computación de Altas Prestacións (a partir de agora, HPC polo termo en inglés *High Performance Computing*) contribúe fortemente á excelencia da ciencia e á competitividade da industria. O HPC permite mellorar os modelos e produtos existentes, así como desenvolver os novos máis rapida e eficientemente. Estas propiedades beneficiosas do HPC foron constatadas tanto en Europa [47] como en América [49, 38].

Debido a esta demostrada vantaxe competitiva que é o HPC, a demanda de poder computacional continúa aumentando ano tras ano. Tanto a academia como a industria tentan crear modelos tan próximos á realidade como sexa posible, o que involucra enormes cantidades de información durante as simulacións que realizan. Cada vez máis datos son colleitados dos experimentos que se realizan no mundo real, e necesitan ser almacenados e analizados. O nivel de detalle dos gráficos por computador incrementouse ata un nivel que hoxe en día se fai difícil distinguir as imaxes xeradas por computador das reais.

Ao principio, a industria do silicio respondeu a esta crecente demanda sendo capaz de preservar o modelo de programación secuencial existente grazas á Lei de Moore [95]. O tamaño dos transistores púidose reducir, dobrando o seu número nun circuíto integrado cada dous anos. Este feito permitiu frecuencias de reloxo máis altas, memorias caché máis grandes, e microarquitecturas máis flexibles. Os programas executábanse máis rápido en cada nova xeración de procesadores.

Con todo, a frecuencia de reloxo non puido ser aumentada tan rápido desde o ano 2000. Ata o devandito ano, un aumento na frecuencia de reloxo levaba aparellada unha diminución da voltaxe do circuíto integrado. Estar propiedade, coñecida como escalado de Dennard [40], permitiu manter a densidade de potencia case constante. Pero este escalado xa non é posible porque a voltaxe non pode seguir sendo reducido se queremos preservar o funcionamento fiable dos transistores ao mesmo tempo.

Polo tanto, a industria decidiu introducir varios procesadores de propósito xeral no mesmo chip para aproveitar o aumento no número de transistores dispoñibles. Este cambio arquitectural deu orixe á *era multinúcleo*. A lista TOP500 [124] reúne, dúas veces ao ano, a información sobre os 500 supercomputadores máis potentes. Os primeiros procesadores multinúcleo apareceron na lista de novembro de 2011. A partir dese momento, os núcleos por zócalo incrementáronse ata dezaseis e xa non existen procesadores cun só núcleo en dita lista. E como os programas secuenciales só se executan nun dos núcleos do procesador, que non se están volvendo máis rápidos, a comunidade de desenvolvedores de software viuse forzada a desenvolver e usar ferramentas de programación paralela.

Outro modo de loitar contra os problemas de disipación de calor é o uso de procesadores especializados. Comunmente chamados aceleradores, estes deseñáronse para levar a cabo só tarefas específicas pero máis eficientemente que os procesadores de propósito xeral. Existe un amplo rango de aceleradores: tarxetas gráficas (GPUs), o Intel Xeon Phi, FPGAs e ASICs. O seu uso está a crecer nos últimos anos [107, 94], como se pode comprobar na porcentaxe de supercomputadores con aceleradores da lista TOP500. Con pequenos comezos en xuño de 2006, os aceleradores representan hoxe o 34 % do rendemento total. Así, os sistemas de computación de altas prestacións de hoxe en día teñen arquitecturas heteroxéneas que combinan distintos tipos de elementos de procesamento para alcanzar gran rendemento con baixo consumo. Nótese que esta tendencia de deseño tamén se está a seguir nos sistemas non-HPC (sobremesas, portátiles, móbiles e dispositivos embebidos), e mesmo no deseño de microprocesadores para resolver o problema do *dark silicon* [46].

Os novos deseños de microprocesador impactaron enormemente na xerarquía de memoria dos computadores, e a velocidade da memoria está a evolucionar a un menor ritmo que a velocidade do procesador [141]. Varios exemplos ilustran esta idea. En primeiro lugar, os sistemas de memoria actuais para procesadores multinúcleo inclúen tres ou catro niveles de caché para reducir a latencia de acceso, pero a colocación dos datos debe ser optimizada para acadar este obxectivo. En segundo lugar, cada nodo dun cluster normalmente inclúe varios procesadores multinúcleo que teñen módulos de memoria DRAM asociados a eles. Todos os núcleos do nodo poden acceder a todos os módulos de memoria a través dunha rede de interconexión, pero a latencias diferentes. Estes sistemas coñécense como arquitecturas NUMA. En terceiro lugar, a maioría de aceleradores teñen a súa propia xerarquía de memoria, separada da memoria para os procesadores de propósito xeral. Polo tanto, para obter bo rendemento, as transferencias de memoria deben ser minimizadas entre estas dúas xerarquías.

Os programadores non están adestrados para manexar todos estes niveis de complexidade crecente [17]. Modelar a realidade en software é xa unha tarefa difícil sen ser consciente da arquitectura do computador subxacente. Esta dificultade en desenvolver novo software, así como o alto custo de reescribilo, coas conseguintes probas, provoca que moito software antigo siga aínda en uso. E a maioría deste software foi escrito para arquitecturas mononúcleo. O problema co código paralelo antigo é aínda peor. Estes códigos foron optimizados para rendemento pico nun sistema determinado e normalmente asumen especificidades sobre a arquitectura subxacente e usan extensións de linguaxes de programación non estándar, código ensamblador, etc. Por contra, o desexable é que o rendemento dun programa escale automaticamente nos computadores futuros.

Os compiladores xogaron un rol principal para superar a primeira crise do software. Foron unha peza esencial para a actual predominancia das linguaxes de programación de alto nivel. O rendemento do código binario xerado por un compilador adoita ser moi próximo ao rendemento pico da máquina, o cal é extremadamente difícil de conseguir manualmente para grandes programas [61]. Ademais, as potentes abstraccións das linguaxes de programación de alto nivel fan que os desenvolvedores sexan máis produtivos. E os compiladores son capaces de proporcional esta abstracción sen penalización no rendemento en diferentes arquitecturas.

Así mesmo, é preciso facer unha reenxeñería significativa do software existente para soportar o uso das novas funcionalidades ofrecidas polo hardware. Debido ao alto custo das transformacións manuais, unha aproximación automática tería grandes beneficios. Mesmo cando o paralelismo e a heteroxeneidade se teñen en conta desde o principio dun proxecto software, escribir programas eficientes é unha tarefa difícil. Cremos que os compiladores serán unha peza crítica para superar este reto.

Esta tese céntrase no desenvolvemento de técnicas de compilación para a extracción automática de paralelismo e localidade en arquitecturas heteroxéneas. Por unha banda, propoñemos a paralelización automática de código secuencial en sistemas multinúcleo e GPUs. Escollemos unha aproximación fonte-a-fonte que xera código paralelo anotado con directivas de compilación OpenMP [105] / OpenHMPP [104] para facilitar a interacción cos expertos do dominio de aplicación, e para permitir a xeración de código binario con rendemento pico grazas aos compiladores actuais e futuros. Pola outra parte, propoñemos o modelado do comportamento do programa desde o punto de vista dos accesos a memoria. A continuación poderíanse aplicar tanto as técnicas de optimización de memoria existentes baseadas no modelo poliédrico (véxase o Apéndice B) como calquera outra técnica de optimización estática ou dinámica en ausencia do código fonte e/ou binario. A organización desta tese é a seguinte:

- O Capítulo 2 introduce unha nova representación intermedia do compilador baseada no concepto dos núcleos computacionais independentes do dominio (véxase o Apéndice A) denominada KIR, a cal é insensible ás variacions sintácticas do código fonte (p. ex., uso de arrays, punteiros ou fluxos de control complexos), e expón varios niveis de paralelismo ao compilador. A continuación, o capítulo presenta unha estratexia de particionamento automática para mapear o paralelismo exposto pola KIR en hardware moderno baseado en procesadores multinúcleo.

- O Capítulo 3 contribúe unha nova técnica para reescribir automáticamente programas secuenciales no seu homólogo paralelo para sistemas heteroxéneos baseados en GPUs. Esta aproximación explota a característica arquitectura hardware da GPU, en particular a súa xerarquía de memoria.
- O Capítulo 4 presenta a reconstrucción automática de bucles afíns a partires dunha traza dos seus accesos a memoria tanto para códigos secuenciais como paralelos. A nosa proposta soporta cantidades moderadas de non linealidade.

A continuación resúmense as principais ideas de cada capítulo.

Un Novo Compilador para Sistemas Multinúcleo

A pesares dos grandes avances na tecnoloxía dos compiladores durante as últimas décadas [1, 3, 139], os compiladores de produción actuais adoitan fallar ao paralelizar mesmo programas secuenciales sinxelos. A principal razón deste fallo é que abordan a detección automática de paralelismo executando análise de dependencias clásico en representacións intermedias baseadas en sentenzas. Estas representacións intermedias son apropiadas para a xeración de código, pero non para a análise de aplicacións completas xa que son extremadamente sensibles ás variacións sintácticas no código fonte. Así, os compiladores paralelizadores actuais están dirixidos por modelos matemáticos que respectan todas as dependencias presentes nestas representacións intermedias, mesmo se só son meros artefactos de implementación.

Nesta tese preséntase unha nova aproximación á paralelización automática de programas secuenciales baseada no concepto de diKernel (véxase o Apéndice A).

A primeira etapa é a construción dunha nova representación intermedia construída sobre os diKernels, denominada KIR, que expón varios niveis de paralelismo nos programas secuenciais. A segunda etapa é unha técnica de particionamiento automática que xera unha versión paralela equivalente dunha aplicación secuencial para ser executada en procesadores multinúcleo.

A construción da KIR realízase en tres pasos: en primeiro lugar, a construción dos diKernels do programa e as súas relacións de dependencias de datos; en segundo lugar, a construción de dependencias de fluxo entre diKernels; e, en terceiro lugar, a construción da xerarquía de ámbitos de execución, que reflicte as etapas de computación do programa secuencial e agrupa aos diKernels nas devanditas etapas.

A nova técnica de particionamiento automático que transforma un programa secuencial nun homólogo paralelo para procesadores multinúcleo toma, como entrada, a KIR. O método consiste en dous pasos. O primeiro paso é filtrar aquelas dependencias presentes na KIR que non evitan a paralelización da aplicación (denominadas dependencias a nivel de diKernel espúreas). O segundo paso é a construción dunha estratexia de paralelización baseada en OpenMP que sexa eficiente para o programa secuencial no seu conxunto.

A idea chave detrás da estratexia de paralelización é atopar o camiño crítico da KIR e executar as computacións que contén nunha única rexión paralela para así minimizar a creación e destrución de fíos. A nosa aproximación baséase na existencia de transformacións paralelizadoras deseñadas para cada tipo de diKernel. Deste xeito, o camiño crítico é o camiño máis longo da KIR que só contén dependencias de fluxo e diKernels paralelizables. A explotación do paralelismo de gran fino con instrucións SIMD, en particular para diKernels non paralelizables, queda fóra do ámbito desta tese xa que foi abordado satisfactoriamente por outras técnicas complementarias [91, 127, 122, 85].

A arquitectura abordada nesta parte da tese son os procesadores multinúcleo. En xeral, o paralelismo existente nos diKernels paralelizables será suficiente para xerar uns poucos fíos de gran groso para ser executados no procesador multinúcleo. Consecuentemente, cando a KIR ten varios camiños que comparten computacións, as partes non compartidas se serializan nun único camiño na rexión paralela. Nótese que non é necesaria sincronización entre as partes non compartidas xa que non están conectadas por dependencias a nivel de diKernel.

Dada unha rexión paralela dun camiño crítico, a nosa estratexia de particionamiento automático minimiza a sobrecarga de sincronización entre diKernels analizando a produción e o consumo dos valores das distintas variables. Así mesmo, a nosa proposta tamén minimiza a creación e destrución de fíos. Se a rexión paralela está contida dentro dun bucle, as directivas OpenMP `parallel` móvense para envolver ao devandito bucle. O camiño crítico queda confinado entre barreiras, e as restantes computacións íllanse en rexións `single`. Esta situación é moi común nas simulacións numéricas.

En resumo, a nosa aproximación permite a paralelización automática de aplicacións en arquitecturas multinúcleo minimizando a sobrecarga que leva aparellada. A KIR reflicte naturalmente a estrutura do código fonte e evita así a violación das dependencias de fluxo de datos especificadas polo programador.

Paralelización Automática para GPGPU tendo en conta a Localidade

O uso das tarxetas gráficas (a partir de agora GPUs, por *Graphics Processing Units*) para a computación de propósito xeral (GPGPU) incrementouse dramaticamente nos últimos anos [107, 94] debido, principalmente, a dúas razóns. Por unha banda, a industria do hardware non foi capaz de satisfacer as demandas crecentes de potencia computacional á vez que preservaba o modelo de programación secuencial. Por outra banda, as GPUs ofrecen un poder computacional enorme a baixo custo debido á presión da industria do videoxogo. Isto levou ao desenvolvemento de novos modelos de programación para integrar estes aceleradores (GPUs, pero tamén outros dispositivos con arquitectura *manycore* como o Intel Xeon Phi) con linguaxes de programación de alto nivel, dando orixe aos sistemas de computación heteroxéneos.

O principal inconveniente destes sistemas é que a súa heteroxeneidade queda exposta ao desenvolvedor. Como se mencionou, programar non é fácil, e as arquitecturas paralelas fano aínda máis difícil xa que requiren tarefas adicionais para paralelizar e optimizar o rendemento. Deste xeito, os desenvolvedores teñen que tratar con moitas características de baixo nivel e limitacións a través de librarías para a programación de GPUs. Explotar a localidade é clave para obter bo rendemento, e é máis complicado nas GPUs que nas CPUs debido á complexa xerarquía de memoria das GPUs. As directivas de compilación demostraron acadar portabilidade e bo rendemento nestas arquitecturas ao mesmo tempo [87]. Polo tanto, cremos que unha aproximación baseada en directivas é a elección axeitada para a paralelización automática de aplicacións en GPUs desenvolvida nesta tese.

As GPUs foron deseñados para a manipulación rápida e eficiente das imaxes que se amosan nas pantallas. Certas etapas do pipeline gráfico realizan operacións en punto flotante con datos independentes, como a transformación das posicións dos vértices ou a xeración das cores dos píxeles. tese, as GPUs executan miles de fíos concorrentes SIMD requirindo acceso a memoria con gran ancho de banda. Este obxectivo de deseño lógrase porque as GPUs dedican máis transistores que as CPUs ao procesamento de datos, en lugar de ao almacenamento en caché e ao fluxo de control. A transición dos *shaders* de función fixa aos programables fixo que estos recursos computacionais estean dispoñibles para a programación de propósito xeral.

Os primeiros enfoques GPGPU (OpenGL [130], CG [100]) obrigaban aos programas a seren semellantes ás aplicacións gráficas que debuxaban triángulos e polígonos, limitando desde xeito a accesibilidade das GPUs. Sen embargo, NVIDIA presentou en novembro de 2006 a *Compute Unified Device Architecture (CUDA)* [103, 89], que permite o uso de C coma linguaxe de programación GPU. A xeración de código de GPGPU eficiente con CUDA require que o programador manexe explicitamente a arquitectura de hardware da GPU.

CAPS Entreprise ofrece un conxunto completo de ferramentas de software para desenvolver aplicacións paralelas de alto rendemento destinados a sistemas heteroxéneos baseados en aceleradores. A máis relevante son os *CAPS Compilers* [32], que xeran CUDA [103] e OpenCL [129] a partir dunha aplicación secuencial anotada con directivas de compilación. Os enfoques baseados en directivas (como o coñecido OpenMP [105]) tratan de reducir o esforzo de programación e proporcionar códigos máis lexibles. Desta maneira, estes enfoques facilitan a interacción entre os expertos do dominio de aplicación e os programadores. As versións secuencial e paralela coexisten no mesmo arquivo, ofrecendo unha maneira gradual para migrar aplicacións. Os códigos desenvolvidos son independentes da plataforma hardware e os novos aceleradores soportadas polo tradutor explótanse de forma automática. Ademais, demostrouse que é posible conseguir un rendemento razoable en comparación cos códigos para GPU escritos a man. Polo tanto, consideramos que as directivas do compilador ofrecen un instrumento conveniente para a paralelización automática de aplicacións secuenciais en sistemas heteroxéneos baseados en GPUs. En concreto, escollemos o estándar OpenHMPP (antes coñecido como HMPP [25]) e a súa extensión HMPPCG (HMPP Codelet Generator) porque estes conxuntos de directivas proporcionan unha funcionalidade única para transformar aniñamentos de bucles, o cal permite un maior grao de optimización do código para GPU xerado, e tanto o seu compilador como a súa contorna de execución son os máis maduros.

As cadeas de recurrencias (de agora en diante, *chrecs*) son un formalismo alxebraico para representar funcións de forma pechada que se utilizaron con éxito para acelerar a avaliación dunha función nunha serie de puntos dun intervalo regular [18]. As chrecs, que son proporcionadas pola KIR, demostraron ser unha representación potente dos complexos bucles e accessos a memoria que aparecen en aplicacións reais [13]. No contexto da xeración de código de GPGPU eficiente, é necesario non só ter en conta os accesos a memoria requiridos nunha porción de código, senón tamén ter en conta que accesos se emiten por parte de cada fío. Afortunadamente, as chrecs son un mecanismo poderoso para este propósito ao permitir a súa *instanciación* para avaliar as funcións de forma pechada da chrec para os valores particulares dos índices de bucle asignados ao fío da GPU.

No Capítulo 2 desta tese preséntase un enfoque independente do hardware para procesadores multinúcleo baseado en OpenMP que ten demostrado ser eficaz e eficiente. Con todo, para o máximo rendemento na GPU, o código xerado debe explotar a súa característica arquitectura hardware (en particular, a complexa xerarquía de memoria). Así, o Capítulo 3 introduce unha nova técnica de xeración de código que estende o enfoque previo tendo en conta as características de programación de GPUs con maior impacto no rendemento: threadificación de bucles (a política que guía a creación de fíos GPU e que código executarase), agrupamento de fíos (xa que estes se lanzan en *warps* que executan unha instrucción común de vez), coalescencia nos accesos a memoria global (os accesos emitidos polos diferentes fíos son manexados mediante unha única transacción), e uso máximo de rexistros e memoria compartida (as máis rápidas da xerarquía). Esta técnica, tras diversas análises que aproveitan a potencia da KIR e das chrecs, acadou a xeración automática de código competitivo en rendemento coa implementación realizada por programadores expertos para dous exemplos representativos extraídos de aplicacións científicas.

Reconstrucción Afín de Código baseada en Trazas

Moitas técnicas de compilación para a optimización estática e dinámica baséanse no coñecemento do código fonte da aplicación, como as presentadas nos capítulos 2 e 3. Por desgraza, o código fonte non sempre está dispoñible para o compilador. Nos sistemas embebidos, por exemplo, é común atopar núcleos con funcionalidade de alto nivel ben definida, pero cuxos compoñentes internos son opacos para o deseñador do sistema completo ou para o programador. Mesmo cando o código fonte está dispoñible, os desenvolvedores de software poden utilizar estruturas de datos e de control complexos, incluíndo técnicas de ofuscación de código, que enmascaran a lóxica da aplicación subxacente e evitan a súa análise estática e a súa posible optimización. Nestes casos, a explotación da localidade convértese en clave para acadar bo rendemento.

O Capítulo 4 presenta un novo marco matemático para a reconstrución afín dos bucles dun programa a partir dunha traza dos seus accesos a memoria, sen intervención do usuario ou o acceso aos códigos fonte e/ou binarios da aplicación. A proposta baséase na observación de que, nos bucles afíns, as diferenzas entre dous accesos consecutivos deben ser construídas como combinacións lineais dos índices de bucle. Os códigos afíns representan unha clase importante de problemas en moitos dominios de computación, tales como a supercomputación, os sistemas embebidos, ou as aplicacións multimedia. Na súa maior parte, estes códigos executan grandes bucles regulares, con partes de control estático que dependen só de combinacións lineais dos índices do bucle, accedendo e operando en grandes conxuntos de datos. Este é o tipo de códigos que se adoita modelar e optimizar utilizando o enfoque poliédrico [75, 83, 54, 27] (véxase o Apéndice B). Téñase en conta tamén que algúns accesos irregulares en tempo de compilación son afíns en tempo de execución [60]. Polo tanto, a proposta desenvolvida neste capítulo pode mellorar o rendemento das paralelizacións automáticas presentadas nos Capítulos 2 y 3. Ademais, desenvolvemos extensións para o manexo de trazas non linealmente perfectas permitindo puntos adicionais ou ausentes. Tamén abordamos a reconstrución de códigos afíns paralelizados automaticamente, que adoitan incluír cantidades moderadas de non linealidade, modificando o algoritmo básico para procesar por partes a traza de accesos a memoria.

O algoritmo de reconstrución proposto trata co fluxo de direccións xeradas por unha única instrución de acceso a memoria. Polo tanto, supoñemos que a traza contén polo menos a dirección de memoria da instrución que emitiu o acceso (ou un modo similar que identifique unívocamente a instrución), e a localización en memoria accedida. Este formato de traza de memoria pódese xerar, por exemplo, con Intel Pin [93]. No caso xeral, é de esperar que o arquivo de trazas conteña toda a execución do programa, incluíndo múltiples bucles aniñados e seccións que non sexan bucles. A detección de bucles en trazas de execución cae fóra do alcance desta tese xa que foi discutido en traballos anteriores [80, 96]. Polo tanto, asumimos que existe un mecanismo fiable para detectar bucles na traza.

A nosa proposta está deseñada para recrear a mesma secuencia de accesos que a traza de memoria proporcionada. O algoritmo que propoñemos é esencialmente unha exploración guiada polo espazo de potenciais solucións, dirixida polas diferenzas de primeira orde entre as direccións de memoria ás que accede unha instrución dada. Cada nodo neste espazo en forma de árbore representa un punto no espazo de iteración do bucle. A súa raíz é un bucle trivial que xera os primeiros dous accesos da traza. Os fillos dun nodo na árbore son os índices que poden seguir de inmediato aos pais no espazo de iteración. A partir da raíz, o motor de exploración comeza incorporando un a un os accesos a reconstruír, descendendo un nivel na árbore, ata que atopa unha solución para toda a traza ou determina que ningún bucle afín é capaz de xerar a secuencia de accesos observada.

Grazas ao desenvolvemento matemático presentado no corpo desta tese, está garantido que o noso algoritmo atopará a reconstrución afín mínima para calquera traza de acceso lineal. Con todo, as aplicacións nos sistemas reais requiren tratar con diversos graos de non linealidade e incerteza na traza. Unha posible situación son aquelas trazas que conteñen principalmente referencias emitidas por un só acceso, pero mesturados cunha certa cantidade de accesos non relacionados. Nesta situación, a exploración do espazo de solucións pódese modificar para descartar algunhas observacións antes de concluír que a rama explorada da árbore non pode dar lugar a unha solución válida. Implementamos esta funcionalidade guiada estatísticamente, para evitar rexeitar demasiados puntos e chegar a unha versión moi simplificada do bucle.

Nalgunhas situacións, nun arquivo de traza poden faltar algunhas entradas para que sexa completamente representable por un bucle afín. O motor de exploración pódese configurar para inserir observacións "desaparecidas" para tentar chegar a unha representación lineal. Deste xeito, o noso método insere tentativamente estes accesos e continúa a exploración. Créase un punto de retroceso para o caso en que non se alcance unha solución mediante a exploración desa rama. Así mesmo, permítese especificar un parámetro de tolerancia para evitar a exploración de ramas improbables.

Por último, tamén se considerou a reconstrución de bucles a partir das trazas xeradas por códigos afíns que foron paralelizadas automaticamente por PLUTO [27]. Para iso supoñemos que cada fío xera a súa propia traza de accesos a memoria ou, equivalentemente, que unha única traza conxunta inclúe o identificador do fío que emite cada acceso. Aqueles códigos que se paralelizaron engadindo simplemente unha directiva OpenMP `parallel for` no nivel de aniñamento axeitado terán trazas completamente afíns sempre que as iteracións se distribuíran de forma estática. Nestes casos, pódese aplicar o mesmo algoritmo de reconstrución utilizado para os códigos secuenciais. A reconstrución obtida para cada fío será exactamente a mesma, agás os límites de iteración, polo que sería trivial reconstruír o código orixinal.

Con todo, este non é o caso para os códigos con dependencias máis complexas. Co fin de paralelizar automaticamente estes bucles, PLUTO debe realizar máis cambios no código secuencial para satisfacer as dependencias na versión homóloga paralela. Para iso introduce funcións non lineais como límites dos bucles internos, facendo que o código resultante non sexa representable como un único bucle afín de profundidade similar á orixinal. Un posible enfoque para modelar este tipo de códigos é recoñecer a traza por partes, é dicir, como unha secuencia de bucles. Desta maneira, garántese atopar sempre unha reconstrución para a traza (aínda que o seu tamaño soamente estea limitado polo número de accesos na traza). Téñase en conta que, neste caso, non é sempre posible reconstruír un código SPMD común para todos os fíos. Con todo, a técnica presentada permite construír unha forma equivalente afín a anacos de códigos que non o están na súa forma orixinal, permitindo a súa análise afín e optimización.

Conclusións e Liñas de Traballo Futuro

A introdución das arquitecturas de computador heteroxéneas retou á comunidade de desenvolvedores de software dun modo sen precedente. Isto causou que escribir un programa eficiente se convertese nunha tarefa difícil e propensa a erros, mesmo para programadores expertos en HPC. Con todo, tanto a ciencia como a industria demandan cada vez máis poder computacional para alcanzar os seus obxectivos. Os compiladores son unha ferramenta fundamental para abordar este reto e esta tese, titulada *"Técnicas de compilación para a extracción de paralelismo e localidade en arquitecturas heteroxéneas"*, fai varias contribucións neste campo.

En primeiro lugar, definiuse unha nova representación intermedia do compilador denominada KIR [4, 5]. Esta nova representación intermedia proporciona as características do programa necesarias para a paralelización automática do código secuencial de entrada. Constrúese sobre os núcleos computacionais independentes do dominio (diKernels) para soportar as variacións sintácticas do código fonte. Estes diKernels conéctanse con dependencias e agrúpanse en ámbitos de execución para recoñecer as etapas computacionais da aplicación de entrada. Implementouse unha proba de concepto sobre GCC [6, 9].

A continuación abordouse a xeración de código paralelo para procesadores multinúcleo mediante a inserción de directivas OpenMP [4]. Presentouse un algoritmo de particionamiento automático da KIR centrado en minimizar a sobrecarga causada pola sincronización de fíos. A capacidade da nosa proposta para ser aplicada en diferentes dominios foi demostrada utilizando un completo xogo de programas de proba que inclúe códigos sintéticos representativos de diKernels frecuentemente usados, rutinas de álxebra lineal densa/dispersa e procesamento de imaxe, e aplicacións de simulación. Así mesmo, levouse a cabo unha avaliación comparativa en termos de efectividade dos compiladores GCC, ICC e PLUTO sobre a súa capacidade para a paralelización automática do conxunto de programas de proba. En xeral, os contendentes fallan na paralelización de códigos que conteñan computacións regulares con fluxos de control complexos e computacións irregulares, e non optimizan a paralelización conxunta de varios bucles.

Debido á súa crecente popularidade, nesta tese tamén se consideraron as tarxetas gráficas (GPUs) como principal expoñente das arquitecturas *manycore* [12, 11]. A nosa proposta céntrase en explotar a localidade de datos na complexa xerarquía de memoria da GPU. Foi deseñada considerando as características de programación de GPUs máis influíntes na xeración de código eficiente: threadificación de bucles, agrupamento de threads, accesos coalescentes á memoria global, e máximo uso de rexistros e memoria compartida. Modelamos alxebraicamente as complexas interaccións entre os accesos a memoria dos diferentes fíos da GPU grazas ás cadeas de recurrencias. Para a xeración de código, empregamos directivas OpenHMPP xa que proporcionan gran intelixibilidade e portabilidade. A nosa técnica foi aplicada satisfactoriamente a dous casos de estudo representativos de aplicacións científicas. A avaliación do rendemento en tarxetas gráficas de NVIDIA (con dúas diferentes arquitecturas) corroborou a súa eficiencia.

Finalmente, desenvolvemos unha nova técnica para a caracterización dun programa desde o punto de vista da súa traza de accesos a memoria [117]. Esta técnica é capaz de reconstruír aniñamentos de bucles afíns considerando os accesos a memoria emitidos por unha instrución, sen intervención do usuario ou acceso ao código fonte ou binario. Formalizouse como o percorrido dun espazo de solucións con forma de árbore, no cal cada nodo simboliza un punto no espazo de iteración do bucle. Ademais, propuxéronse métodos para percorrer eficientemente este espazo de solucións, e para soportar cantidades moderadas de non linealidade na traza como ruído, puntos que faltan, ou o código resultante do compilador paralelizador PLUTO. A avaliación experimental probou o bo rendemento da nosa proposta, a súa precisión na reconstrución de códigos perfectamente afíns, e a súa flexibilidade para representar por partes códigos case-afíns. As aplicacións desta técnica son moi variadas e xa estudadas (p. ex., compresión / almacenamento / comunicación de trazas, paralelización dinámica, colocación de datos en memoria e deseño da xerarquía de memoria).

Como liñas de investigación futuras podemos considerar a utilización da reconstrución baseada en trazas para incrementar a información dispoñible para construír a KIR, e o deseño dun novo algoritmo de particionamiento automático da KIR que considere as interaccións entre as computacións en clusters heteroxéneos, tanto a interacción CPU-GPU como a comunicación entre nodos. Para este propósito é necesario recompilar información sobre o hardware. Así mesmo poderíanse incluír aproximacións baseadas en *autotuning* para seleccionar a variante con maior rendemento entre as distintas paralelizaciones candidatas dun diKernel. A técnica de reconstrución baseada en trazas podería ser mellorada para manexar un rango máis amplo de aplicacións irregulares.

A tecnoloxía desenvolvida nos Capítulos 2 e 3 foi licenciada á empresa de base tecnolóxica Appentra Solutions S.L. para a creación de Parallware [15].

Appendix E

Glossary

AST Abstract Syntax Tree, a tree representation of the abstract syntactic structure of source code written in a programming language. In this thesis, we consider an AST-based IR where each AST represents a statement.

BB Basic Block, straight-line code sequence with no branches in except to the entry and no branches out except at the exit.

CFG Control Flow Graph, representation using graph notation of all paths that might be traversed through a program during its execution.

DAG Directed Acyclic Graph, collection of vertices and directed edges, each edge connecting one vertex to another, such that there is no way to start at some vertex v and follow a sequence of edges that eventually loops back to v again.

DDG Data Dependence Graph, representation using graph notation of the data-dependences between the statements of a program.

diKernel Domain-Independent Kernel (see Appendix A).

DT Dominator Tree, tree representation of the dominance relationship. We say node d of a flow graph dominates node n if every path from the entry node of the flow graph to n goes through d, In this thesis, we assume that our compiler framework provides a DT of BBs built from the CFG.

GPU Graphics Processing Unit (see Section 3.1).

Hierarchy of regions Representation of a program based on regions, which are portions of the flow graph that have only one point of entry.

KIR diKernel-based IR (see Section 2.1.1).

ILP Integer Linear Programming, mathematical optimization or feasibility program in which some or all of the variables are restricted to be integers and the objective function and the constraints are linear.

IR Intermediate Representation of a compiler.

irregular (array) access Access to a variable with an indirection array that introduces a compile-time unpredictable access pattern. Also found in the literature as irregular reads/writes, subscripted subscripts, etc.

SCC Let $G = (N, E)$ be a directed graph. A **strongly connected region (SCR)** is a set of nodes S such that there is a path $S_1 \xrightarrow{*} S_2$ for any two nodes $S_1, S_2 \in S$. By this definition, every node by itself is a trivial SCR. The **strongly connected components (SCCs)** are the maximal SCRs (SCRs that are not proper subsets of any other SCR).