UNIVERSITY OF PADOVA

MASTER DEGREE in
ICT FOR INTERNET AND MULTIMEDIA

Optimization and scalability of tiled code generation

October 7, 2019
A.A. 2018/2019

Candidate:
Gabriella BETTONTE

Supervisors:
Corinne ANCOURT
Leonardo BADIA

“The worst thing you can do to a problem is solve it completely.”

Daniel Kleitman

UNIVERSITY OF PADOVA

Abstract

Optimization and scalability of tiled code generation

by Gabriella BETTONTE

Nowadays, scientific research needs to model complex systems and process big data, which requires high computational power, usually provided by massive parallel super-computers. However, a powerful hardware alone is not enough to ensure high parallel computing performance: to make good use of parallel infrastructure capabilities, the code must be optimized as well. Hence, tools like source-to-source compilers have a primary role.

The CRI[1] research team is developing PIPS, a source-to-source compiler that performs code optimization to enforce locality for the cache memory usage and parallelism for a balanced computational load among threads.

Since programs spend most of their execution time inside loops, a critical optimization performed by PIPS takes place on loop tiling, *i.e.*, partitioning the iteration space into smaller blocks (tiles) of iterations, which fit the available cache and can be executed in parallel.

Tiled code generated by PIPS showed a lack of scalability, that is, increasing the number of threads does not lead to any advantage in terms of execution time. This thesis work aims to present the procedure we designed to investigate this scalability issue and show an invariant code optimization and parallel directive selection performed on the tiled code generated by PIPS.

As a result, we will show how the implementation of a new PIPS phase, including such optimizations, leads to a scalable tiled code and to the minimization of the parallel directive overhead. We will present how the code generated by the current version of PIPS outperforms the previous one and achieves comparable results to other state-of-the-art code optimizers in terms of speed-up.

[1] *Centre de Recherche en Informatique Mines ParisTech*

Acknowledgements

I would like to express my gratitude to Dr. Corinne Ancourt for guiding me with patience through the internship, for participating with passion to my effort, for supporting me beyond her supervisor duties and for being an inspiration to me as researcher and as human being.

My sincere thanks also goes to Dr. Leonardo Badia for supporting me during my master degree with precious advices and insights and for helping me to believe in my possibilities. Without his contribution, I would have missed the precious work experience with CRI team.

I would also like to thank all the CRI team for welcoming me with kindness and making me feel welcome. I developed an invaluable friendship with my colleagues Patryk, Bruno, Lucas, Monika, Maksim, Justyna and Maryna. Speaking, working and traveling with them enriched me with wonderful memories and fascinating new points of view that will accompany me in the future.

My colleagues from the University of Padova, Elena, Davide, Mattia, Thomas, Gabriele and Nicola, made me feel among friend since I first met them: thank to you all!

My gratitude goes also to my family for pushing me to reach my highest objectives during my academic life and, at the same time, for reassuring me during the hard moments.

Finally, I would like to thank my beloved Piercarlo for brightening my life, being my companion of adventures and inspiring me to express myself to the best.

Contents

To my mother Anna Maria

Introduction

My thesis work, developed at MINES ParisTech, aimed to study in detail some issues related to the generation of optimized parallel code by PIPS, a source-to-source compiler for Fortran and C code, developed by the members of the *Centre de Recherche en Informatique* team based in Fontainebleau.

Over the many functionalities of PIPS, the one on which I focused was the generation of efficient code, which means that when run under the same conditions, both the memory usage and execution time of such code are appreciably lower than the original code's.

While some years ago to improve code execution performances the most common approach was increase the clock frequency of the machines, nowadays this perspective, also due to the reaching of hardware improvement limits, has mostly been abandoned in favor of adding more cores to the machine and software-side optimizations for locality and parallelism.

> *"Because the clock frequency of processors fails to continue to grow (end of Dennard scaling), the only way in which the execution of programs can be accelerated is by increasing their throughput with a compiler: by increasing parallelism and improving data locality" [12].*

Those optimizations have the additional requirements of being portable and machine independent. To better study how to fulfill those requirements, the best choice is a source-to-source compiler, such as PIPS.

> *"[A source to source compiler] eases the programmability of heterogeneous architectures, to apply different optimization techniques and let the programmer have access [to code] during the optimization process" [15].*

One of the most common linear transformations meant to improve data locality and parallelism is tiling. It even allows programs dealing with huge vectors to reuse the data stored inside the cache and contextually exploits the potential parallelism of the code, greatly improving the performances of its execution both memory - and time - wise. Considering that commonly programs spend a great percentage of their execution time inside loops, applying tiling to loops offers very noticeable advantages in terms of speed up.

Loop tiling consists in partitioning the iteration space into smaller blocks (*i.e.*, tiles) of iterations, fitting the available cache memory provided by the environment while also evaluating the dependencies among variables.

While this could seem a pretty neat problem, it is important to point out that the term optimization could lead to a misconception while related to the work we will expose: being in fact an undecidable problem, it is not possible to individuate in a deterministic way an algorithm that can solve it. In other words, since we have no defined way to prove our generated code is the best possible, our goal cannot be the true optimal, but rather a good performing generated code.

Loop tiling optimization is fundamental for modern applications involving parallelism. One example of those are DNN-based application popular for computer vision.

> *"DNN-based applications, especially in convolutional layers and fully connected layers, are compute-intensive, as they typically apply a series of matrix computations iteratively to a massive amount of data. For this reason, loop tiling turns out to be the most significant compiler optimization" [25].*

Loop tiling optimizations are also known to improve computer performance in biology research.

> *"Current research in the field of computational biology often involves simulations on high-performance computer clusters. It is crucial that the code of such simulations is efficient and correctly reflects the model specifications.[...] These optimizations [such as loop tiling for locality and parallelism] allow simulating various biological mechanisms, in particular the simulation of millions of cells, their proliferation, movements and interactions in 3D space"[11].*

Loop tiling optimization is also exploited to achieve faster program execution in other applications related to Geo-science.

> *"High-performance computing [performed by parallel computers] is an important driver in the development of seismic exploration technology"[16].*

The necessity of optimizing the loop tiling process was pointed out by the CRI team itself, which noticed that PIPS, transforming specific types of input code such as, for example, the 3D heat equation code, was not able to generate an efficient parallel code output, meaning that run it on a larger number of threads would not produce any appreciable benefit in terms of execution time.

Initially, those scalability issues of the output code were attributed to some defect on the part of PIPS dedicated to the generation of parallel code. If confirmed, that issue would have required a deep and expensive revision of PIPS code related to loop tiling transformation to be resolved. Eventually, a deeper observation allowed us to discard that hypothesis, suggesting that a different approach should have been taken to identify the root of that behaviour.

With that objective in mind, we designed and implemented an automatized procedure (see Appendix A) which aimed to test every aspect of the optimization process that PIPS follows on C code involving loops, in order to identify where the inefficiencies were hidden.

This approach allowed us to highlight some criticality. For some identified issues, as the invariant code motion of some loop bounds and the selection of the most suitable parallel directive, we already implemented a fix (see Appendix B). For some others, such as the in-lining of the minimum and maximum functions, a possible solution has been suggested during my internship (see Appendix C) but we do not have an actual implementation yet. Finally some other issues, as the choice of the most adequate tiling matrix and version of PIPS, represent a new interesting working direction to investigate further.

Anyway, even if more work has to be done, the current version of PIPS, which implements the new PIPS phase we developed, shows very promising results on the parallel code optimization respect the previous version, for a comparison see Chapter 3. In particular in comparison with Pluto [21], another well-known source-to-source compiler that relies on a similar procedure for optimizing parallel code, we noticed comparable or better results in terms of speed-up of output code.

Chapter 1

PIPS - Parallelization Infrastructure for Parallel Systems.

PIPS is a source-to-source compiler that takes C programs as its input and re-turns C programs performing semantic analyses, applying loop transformations and generating parallel code using polyhedral methods. It is based on linear algebra techniques for analyses, *e.g.* dependence testing, as well as for code generation, *e.g.* loop interchange or tiling [20].

Over the next paragraphs we will analyze each of those PIPS aspect in order to introduce the contribution of this thesis to its optimization.

1.1 Compilers

"A compiler is a program that can read a program in one language - the source language -, and translate it into an equivalent program in another language - the target language" [1].

The input programs are generally written in high level programming languages which, while still being understandable from humans, translate the formal language into a series of precise instructions that a machine can process. A compiler proceeds further, acting as a bridge between the input program and the target machine. To achieve that goal, it must be able to understand and to extract information from the input programming language, and to build an output program understandable by the destination machine.

The compiler can therefore be defined as a set of analyses and transformations aimed to obtain from a sequence of abstract instructions, contained inside a program, a flow of smaller machine operations understandable by a processor. The picture in fig. 1.1 shows the above concept.

1.1.1 Source-to-source compilers

While a traditional compiler transforms the source code in a lower level code that can be processed by the computer (the machine-code), a source-to-source compiler takes as input a program and translates it into another program in the same language or another language, which has the same level of abstraction [13].

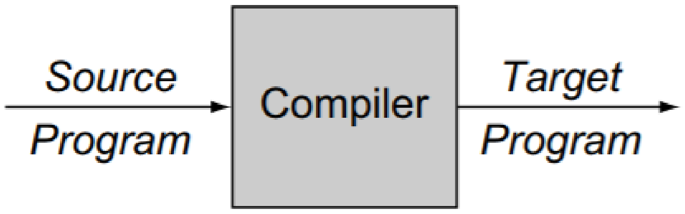


FIGURE 1.1: A compiler [7].

Figure 1.2 shows the difference between a traditional compiler and a source-to-source compiler.

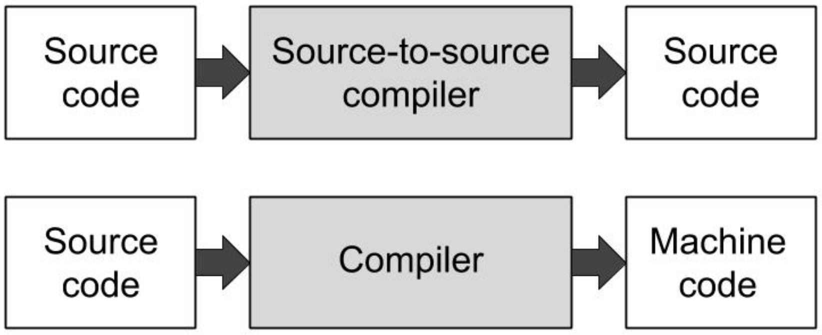


FIGURE 1.2: Comparison between traditional and source-to-source compilers [13].

Source-to-source compilers are mostly used in the field of research about compilers, while in industry classical compilers are the rule.
The reason to prefer source-to-source compilers in research is they offer some advantages over the traditional ones. First of all, they return human readable code, so it is easier to understand which transformations the compiler applied to the code. Then, researchers have the possibility to insert directives for parallel code, such as OpenMP directives which we will discuss later.

1.1.2 Principles of compilation theory

While accomplishing its tasks, a compiler has to respect some primary principles of compilation theory [7]:

- The output must be correct: the compiler must preserve the semantics of the input program.
- The compiler must improve the input program. This improvement must be noticeable. Generally a compiler improves the input code by

making it directly executable on the target machine, but some compilers perform other improvements in addition to that.

1.1.3 Compiler structure overview

The compilation process is divided in two fundamental steps: front-end and back-end.
The front-end phase aims to understand the original code and to perform some analyses in order to construct the intermediate representation (IR).
An intermediate representation is an abstract representation of the original program internal to the compiler. There are many types of IRs, and choosing the right one is critical for the efficiency of the compiler, but all share some common properties, such as being independent from the source and target languages, being easy to produce and to be understood by the back-end. The back-end phase, in fact, will map the intermediate representation to the target machine.
Producing an IR allows to easily re-target the compiler changing the back-end phase in order to return a target code in another language or for another processor, while keeping the same language as input (see Fig. 1.3).

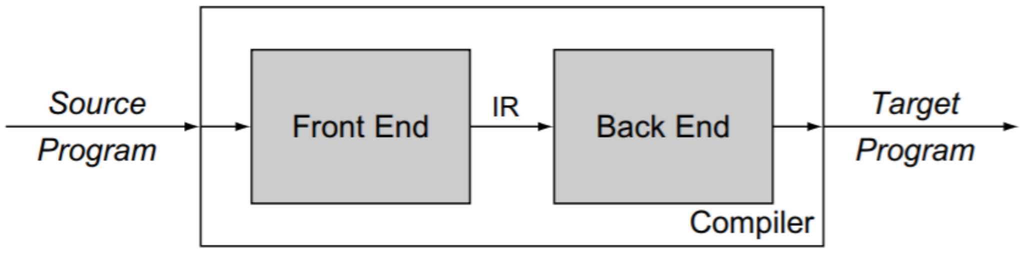


FIGURE 1.3: A two-phase compiler.[7]

Multi-pass compilers take advantage of generating an intermediate representation by performing structural optimizations on the code inside the process of compilation, such as for example dead-code elimination, constant propagation, code motion and reachability analysis. In that case, the optimization phase takes as input the IR generated by the front-end phase and returns another version of the program in the IR form that will be the input for the back-end phase [7]. Fig. 1.4 shows the three phase of a multi-pass compiler.

1.1.4 Phases of a compiler

Each of those three main phases of the compiler can be broken into smaller steps, see Fig. 1.5.

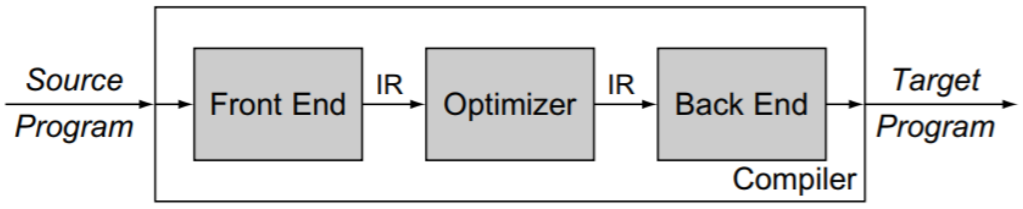


FIGURE 1.4: A three-phase multi-pass compiler.[7]

Front-end phase

This phase basically performs lexical, syntactic and semantic analysis on the input code. The lexical analysis consists in scanning the program and breaking each instruction into tokens. The syntactic analysis involves grouping the tokens of the original code into grammatical phrases that the compiler will use to create the IR. Those phrases are defined by a grammar: a finite set of rules necessary to decide if a sequence of characters is valid or illegal [7]. The grammatical phrase of a compiler usually is a parse tree (see fig. 1.6).
The semantic analysis is meant to check semantic errors that could be in the original code and to keep the information about the type of variables. In that phase the compiler forces the programmer to be consistent with the constraints of the input language, such as for example to only perform operations between variables of the same type and having integer indexes for arrays [1].

Optimization phase

The optimizer analyzes the IR to rewrite the original program taking into account the efficiency requirements. Those requirements could be a minor use of memory, a minor execution time or minor consumption of energy for the processor when running the output program.

"Myriad transformations have been invented to improve the time or space requirements of executable code. Some, such as discovering loop-invariant computations and moving them to less frequently executed locations, improve the running time of the program. Others make the code more compact. Transformations vary in their effect, the scope over which they operate, and the analysis required to support them" [7].

Back-end phase

The back-end phase takes as input the IR returned by the optimization phase to construct the target program. The compiler chooses the order of execution of the instructions given by the previous phase and the management of cache and memory in the more efficient way. All the intermediate instructions are transformed in machine code operations.

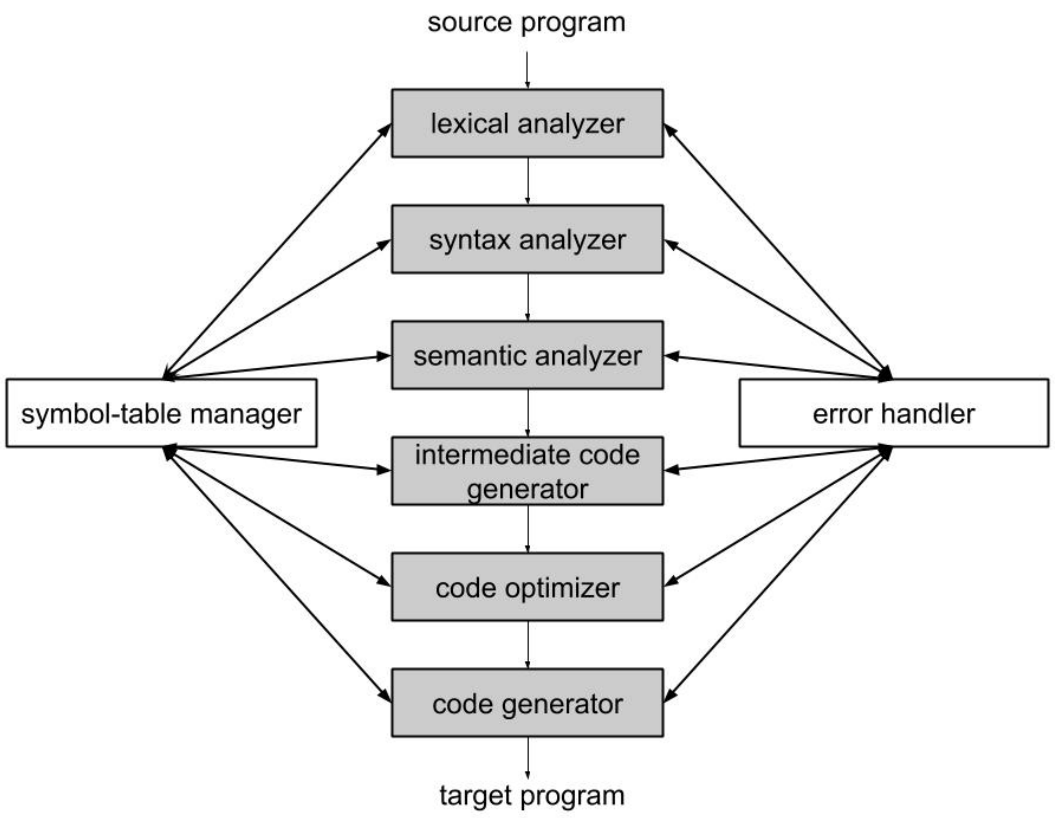


FIGURE 1.5: Phases of a compilers.[1]

Error detection

Each phase can encounter some errors, the compiler should be able to proceed when it finds one error in order to detect other errors, a compiler that stops at every little error is not efficient. The syntactic and semantic analysis handle the most part of the errors that the compiler detects. [1]

1.2 Optimization for locality and parallelism

"A compiler can enhance parallelism and locality in computationally intensive programs involving arrays to speed up target programs running on multiprocessor system" [1].

1.2.1 Locality

The memory stores data that programs need when executing and generally retrieving data from memory is the most time-consuming operation a program performs. In order to have an high computational speed engineers would need a fast memory big enough to store all the data required for program execution. The memory to be fast should be inserted directly inside the CPU but this would be very expensive and would also mean increasing the

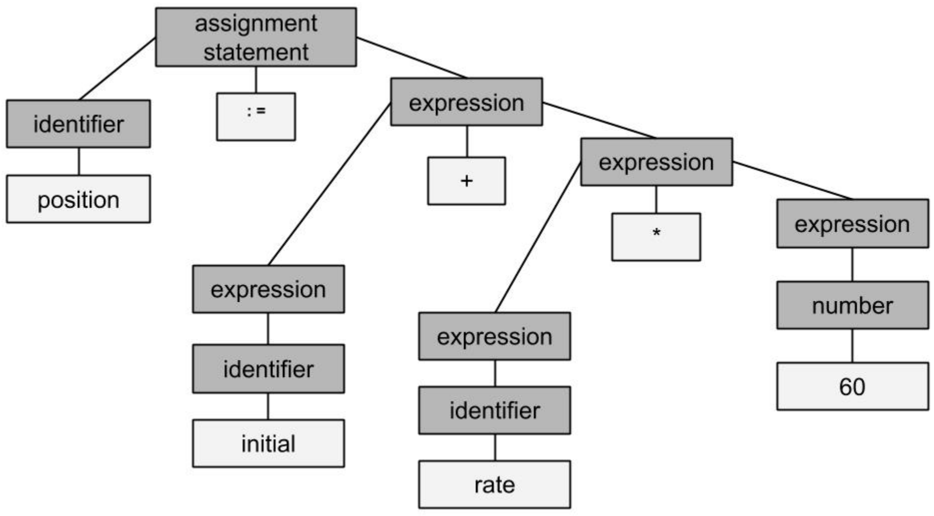


FIGURE 1.6: Parse tree for `position:=initial+rate*60` [1].

dimension of the CPU, which leads to heat dispersion issues. The most common solution adopted by architects is the combination of a fast little memory (cache) and a bigger and slower (but cheaper) main memory. When the CPU needs data for a computation it searches into the cache, if the data is available it is immediately used ("cache hit"). In the case the request to the cache fails, CPU looks for the data inside the main memory ("cache miss"). See Fig. 1.7.

This model allows to drastically decrease the time spent by accessing memory, because cache hits are much less expensive than calls to normal memory - they can be a hundred time faster [1].

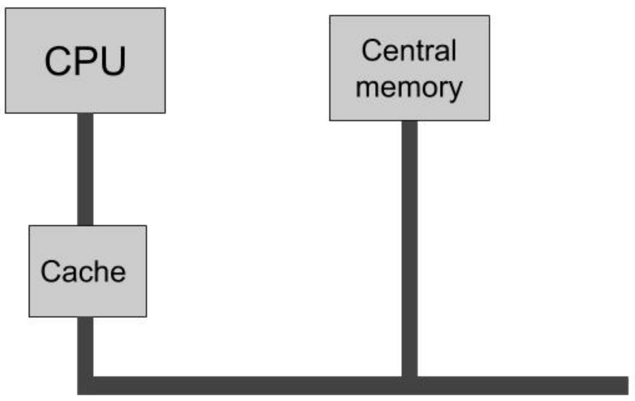


FIGURE 1.7: Cache is logically between CPU and memory [24].

Programs do not access the memory in a random way: if at some point it asks for the content in the address A, with high probability the next request to the memory will be near the address A [24]. The "locality principle", on which caches are built, states programs, in a little amount of time, tend to access neighboring areas of memory. The idea is that, when the CPU refers a data inside the memory, the data on its neighborhood will be loaded into the cache for the next use [24]. Memories and caches are divided into fixed size blocks, or cache lines, to take advantage of the locality principle. When a cache miss happens, the entire line containing the required data is loaded from the central memory into the cache, thus making neighboring data available for the next operations. In general the cache is organized in two or three nested levels: L1, L2, L3 which are hierarchically queried for data by programs from top to bottom.

A multiprocessor is a system of many CPUs sharing the same memory. To avoid conflicts between the processors one possible solution is to have an architecture providing a local memory for each processor, not accessible by others [24], see Fig. 1.8.

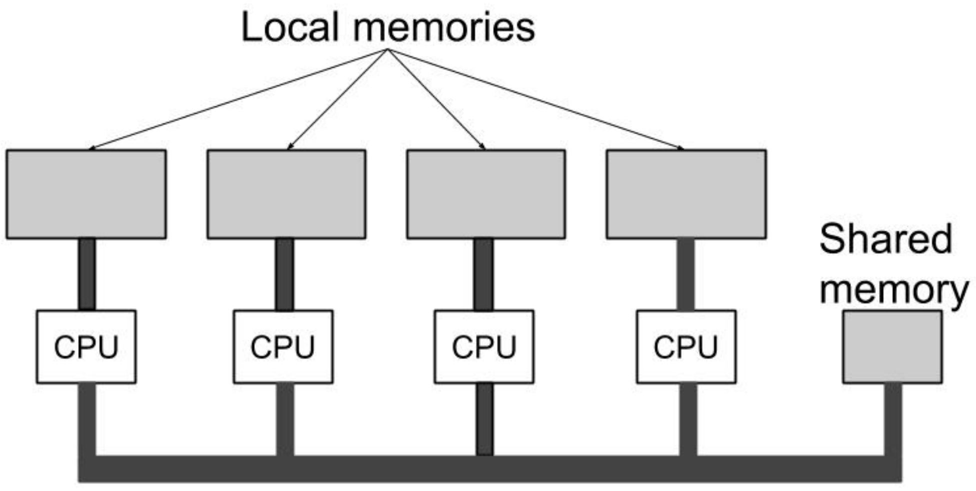


FIGURE 1.8: A multiprocessor [24].

Increasing the locality allows to minimize the communication between processors that can affect the performances making the running time of a program executed in parallel bigger than the sequential ones. A good data locality is achieved if the processor access the same data used recently [1]. So, in order to exploit in the optimal way the principle of cache, compilers should assign to each processor all related operations.

1.2.2 Parallelism

PIPS is an automatic parallelizer for scientific programs: it transforms the sequential input code into an equivalent parallel code, executable on many threads.

As described by the Intel User's Guide [14], threads are small tasks that can run independently inside the scope of a process. In other words, each thread is a basic unit of a CPU that shares the resources and the data access with other threads belonging to the same processor.

An automatic parallelizer computes dependencies among variables to identify the tasks that can be distributed among threads without errors and, during the code generation phase, the compiler inserts the appropriate parallel instructions inside the IR of the code. PIPS automatically marks the portions of code it identifies as "parallel regions" using the parallel instructions implemented by the OpenMP library.

OpenMP

"OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments" [23].

```
#pragma omp parallel.
```

Consider this example, taken from the book [23].

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[])
    {
        /* sequential code */
        #pragma omp parallel
        {
            printf("I am a parallel region.");
        }
        /* sequential code */
        return 0;
    }
```

OpenMP follows the Fork/Join model. At first, the execution of the program is serial, meaning that it is performed inside just one thread, the master thread. When OpenMP encounters the directive

```
#pragma omp parallel
```

it creates as many threads as the number of processing cores in the system or the number specified by the developers (fork). Those threads are called slave threads. Then, every thread executes simultaneously the parallel portion of code. When the execution of every slave thread ends the program execution comes back to the master thread (join). Fig. 1.9 shows the Fork/Join model.

OpenMP allows running loop in parallel with the directive

```
#pragma omp parallel for.
```

This feature is particularly useful for parallelizing loops. It is also possible to set the variables as private, meaning that threads cannot share them avoiding computational mistakes. Additionally, OpenMP handles the creation and management of threads, which is a major advantage for the programmers that will not need to take care of those issues.

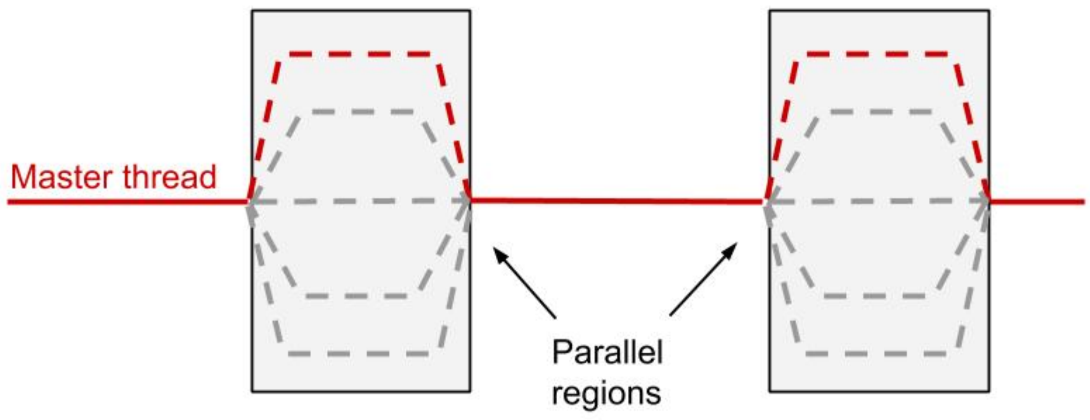


FIGURE 1.9: Fork/join model.

Criteria of performance of an automatic parallelizer

Aditi Athavale, Priti Randive and Abhishek Kambale [3] give some criteria useful to understand the quality of automatic parallelizers.

- Performance and scalability. The point of performing automatic parallelization is that the execution time of the output parallel program should be lower than serial time. The scalability property implies that the execution time of parallel code on n+1 processors should be less than or equal to the execution time on n processors.
- Memory and time complexity. The output program should have a minor memory usage and run time respect to the initial one.
- Parallelization overhead. The insertion of new lines containing the parallel directives should not impact too much on the performances. As good rule, the execution time of the parallel code on one core should be almost the same as serial time.
- User interaction. An automatic parallelizer should ask to the user the less effort and interaction possible.

1.3 Polyhedral compilation

After considering the dependencies among variables, but before marking parallel instructions, PIPS applies transformations based on polyhedral methods on the code. Without that phase, parallelizing instructions would just verify the legitimacy of splitting the instructions between the available logical threads, without taking into account the amount of work assigned to each of them. Without a clever reorganization of the instructions the workload could vary significantly between different tasks, thus wasting hardware resources.

The polyhedral model provides an abstraction to perform high-level transformations such as loop-nest optimization (loop tiling) and parallelization on loops. Those compilation techniques rely on the representation of programs thanks to parametric polyhedra. Polyhedral techiniques used in compilation exploits combinatorial and geometrical optimizations on those polyhedra to analyze and optimize programs [22].

This method is largely used in the context of automatic parallelization because it makes it possible to optimize programs which involve huge size arrays and nested loops with huge iteration spaces.

The iteration space is the set of values of the iteration vector for which the statement has to be executed. In the majority of cases, we can express the iteration space with a set of linear inequalities defining a polyhedron [4]. A polyhedron is a convex set of points in a lattice i.e. a set of points in a $\mathbb{Z}^n$ vector space bounded by affine inequalities: $D = \{x | x \in \mathbb{Z}^n, A \cdot x \geq c\}$

where x is the iteration vector, A is a constant matrix and c is a constant vector [4].

Consider this example, taken from [4] :

```
do i=1, n
   x = a(i,i)
   do j=1, i-1
      x = x - a(i,j)**2
```

Fig.1.10 shows the iteration space for the example and the corresponding polyhedron. The polyhedron (see Fig. 1.11) is the set of inequalities defined by the bounds of the loops in the example.

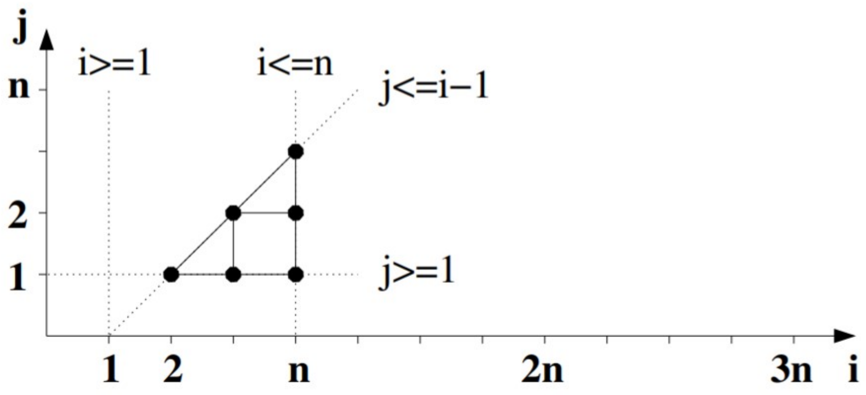


FIGURE 1.10: Iteration space of example code [4].

In conclusion, optimization algorithms based on polyhedral information focus on their geometrical structure and not on the amount of elements belonging to them. To avoid dealing with potentially huge loop iteration spaces we consider them as a lattice of points wrapped into a polyhedral structure. That approach allows us to exploit the geometrical properties of the polyhedra by performing some affine transformations, such as loop tiling, on those mathematical objects.

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 1 \\ -n \\ 1 \\ 1 \end{pmatrix}$$

FIGURE 1.11: Polyhedron $Ax \geq c$ [4].

1.3.1 Loop tiling

"Loop tiling [..] transforms the iteration space of the loop nest by structuring the execution of the loop into blocks/tiles of iterations of the original loop" [6].

Examples of loop tiling

Consider the loop tiling applied to matrix multiplication proposed by Pedro C. Diniz and João M.P. Cardosov [6].

The classic implementation of matrix multiplication is the following one:

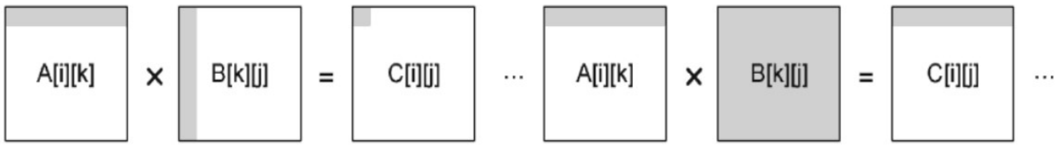
```
...
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            C[i][j]= A[i][k] * B[k][j];
...
```

By dividing the iteration space into smaller chunks we obtained a tiled loop, structured as two loops. The outermost loop scans the blocks in which the iteration set is divided while the innermost loop scans the elements in each tile. Here the block size is B1 x B2 x B3.

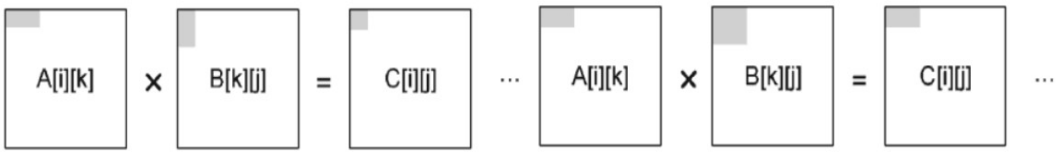
```
...
for (ii = 0; ii < N; ii+=B1)
    for (jj = 0; jj < N; jj+=B2)
        for (kk = 0; kk < N; kk+=B3)
            for (i = ii; i < min(ii+B1,N); i++)
                for (j = jj; j < min(jj+B2, N); j++)
                    for (k = kk; j < min(kk+B3,N); j++)
                        C[i][j]= A[i][k] * B[k][j];
 ...
```

In Fig. 1.12 we can see the graphical representation of the code analyzed above. The tiled code generates far fewer cache misses than the non-tiled version, mainly because the accesses to matrix elements of B are local.

Tiles can have a regular or not regular shape, depending on the dependencies in the code. In the following example [2] we have hexagonal tiles.

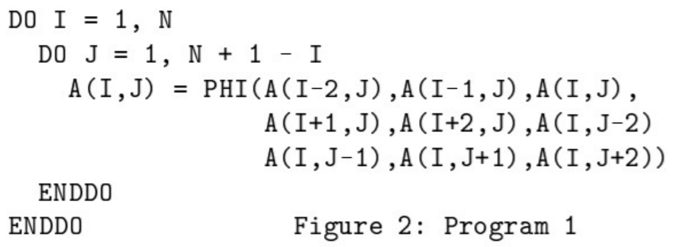


(A) Original source code



(B) Tiled code

FIGURE 1.12: Layout of data being read/write using using the original (A) and tiled (B) code [6]



```
DO I = 1, N
  DO J = 1, N + 1 - I
    A(I,J) = PHI(A(I-2,J),A(I-1,J),A(I,J),
                 A(I+1,J),A(I+2,J),A(I,J-2)
                 A(I,J-1),A(I,J+1),A(I,J+2))
  ENDDO
ENDDO                Figure 2: Program 1
```

FIGURE 1.13: Example code [2].

The iteration space is tiled with hexagonal tiles, see fig 1.14. The hexagonal tiles are defined by a system of inequalities:

$$\begin{cases} -3 \leq j < 3 \\ 0 \leq i + j < 6 \\ 0 \leq i < 6 \end{cases}$$

In fig 1.15 the iteration set of one tile is shown.

Choice of tiling matrix

Loop tiling is characterized by tile shape and tile size, but they are hard to choose. Primarily, tiling has to respect the correctness principle: the output of the code must not vary after tiling procedure. In order to preserve correctness, dependencies among variables must be evaluated. In other words, the tile shape should not cut the cone of dependencies but be inscribed in it. Otherwise, we could have mistakes on computations, that would be split in an illegal way. The cone of dependencies is the fraction of iteration domain bounded by the most external dependencies. To clarify this concept refer Fig. 1.16: the dependencies among variables inside the iteration space are represented with arrows, in particular the most external dependencies (in

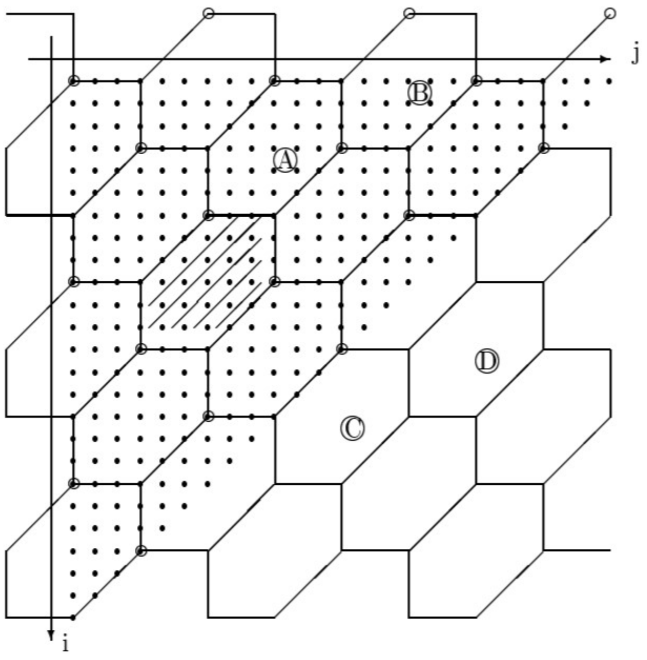


FIGURE 1.14: Hexagonal tiling [2].

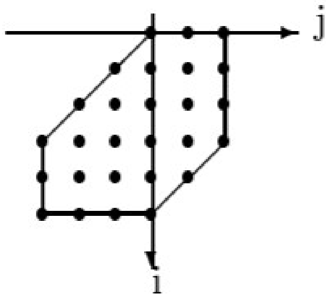


FIGURE 1.15: Hexagonal tile [2].

red) define the dependencies cone. Any tile chosen in the dependencies cone (light red area) is allowed.

Correctness is not the only issue we must consider when we decide the shape of the tiles. Choosing a regular shape, such as a square or rhomboid, allows to simplify the computation of loops bounds at the moment of compilation of the generated code by a traditional compiler.

The size of tiles is another element of primary importance to maximize locality and improve parallelism. Assuming that the iteration space is cut in a regular way, tile size is inversely proportional to their number. To avoid wastes of computational resources we have to keep all cores busy, and choosing the right number of tiles is a way to do that.

Unfortunately, there are not precise procedures to tile the iteration space but there are some considerations that can help.

The tile size should be small enough to exploit the fastest memories i.e. registers, L1 cache or L2 cache. If tiles are too small we are wasting some

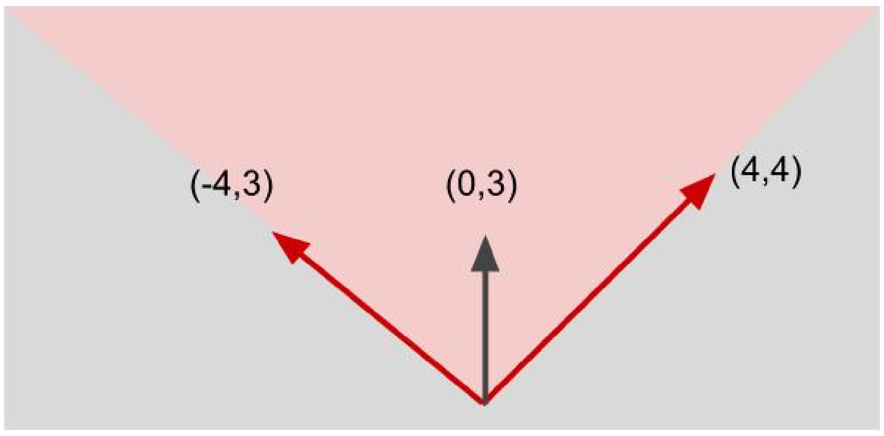


FIGURE 1.16: Cone of dependencies.

potential because more data could be stored into the cache. On the other hand, having tiles too big leads to inefficiency because tiles go beyond the size of the cache and they are stored inside the main memory.

A good tiling also helps parallelism by making the innermost dimension (i.e. the innermost loop) have enough iterations, especially if it is vectorial. This leverages execution on many threads by balancing the computational load.

Chapter 2

Experiments

2.1 Test procedure

Since several measurements were necessary to identify the inefficiencies inside the PIPS code generation phase, we designed an automated testing procedure. It allowed us to better organize the results and to significantly speed up the experimental phase.

The automated procedure for testing PIPS code generation is composed of three bash scripts in Appendix A.

The first script makes PIPS generate tiled code from the input code.

The second script compiles the output of the first script using a traditional compiler (gcc).

The third script executes the input of the second script to measure the performances of the codes.

2.1.1 Setting PIPS parameters

Initially, it is necessary to set the PIPS parameters, which means modifing the .tpips file. Inside the .tpips file resides the tiling matrix, thus by changing it one can specify the tiles shape and size and the scanning directions of tiles and elements inside each tile.

2.1.2 First phase

Run the command

```
sh first_phase.sh
```

to apply the tiling specified in the file my_program.tpips. It creates all the PIPS versions obtained by scanning tiles and elements inside each tile in every possible direction, see 3.2.4. The code resulting from the first phase is where code generation optimizations can be performed. Initially, those experimental optimizations were implemented manually on the code.

2.1.3 Second phase

Run the command

```
sh second_phase.sh
```

to compile all the different versions of the code with gcc compiler [9]. By adding the flag *-fopenmp* during the compilation phase the gcc compiler will consider the parallel OpenMP directives. By default, the gcc compiler would just ignore the directives, as they are comments. Notice that we compile with the flag -O3, meaning that the compiler gcc performs three level of optimizations [10].

2.1.4 Third phase

Run the command

```
sh third_phase_average.sh
```

to execute the compiled code from the second phase.

This script prints on the terminal the average of ten execution times for each version. The execution of the compiled code was done on a different number of threads: 1, 2, 4, 8, 12, 16. It is separated by the second phase because it can be run several times to check the stability of time execution measurements. It is not necessary to repeat the other phases, it is sufficient to have the compiled codes and this script.

2.2 Preliminary work

To be able to obtain meaningful results from our automated procedure of testing, defining appropriate starting conditions was critical. To achieve that, a significant preliminary work has been made to determine the environment, the implementation of the input code and the tiling matrix to use for the tests.

2.2.1 Environment definition

Optimizations, in general, can be machine-dependent and machine-independent. One of the reasons we want machine-dependent optimizations is when we seek excellent performance on a specific hardware [18]. Generally speaking, the optimizations we want to search for are machine-independent optimizations. These in fact allow for greater versatility.

Anyway, the optimization process, although we have in mind the objective that our optimization should be machine-independent, is strongly influenced by the target machine.

As an example, at the beginning of my internship I was conducting the experiments on my personal laptop. Below the machine specifications are reported:

Personal laptop:

- CPU: Intel® Core^{TM} i7-8550U Processor @ 1.80GHz
  - Microarchitecture: Kaby Lake R
  - Physical cores: 4
  - Logical cores: 8

  - Architecture: x86_64

- RAM: 8GB

The results showed a constant and increasing overhead at each consecutive run for no apparent reason - the input code and the parameters of the compilation were in fact the same - making the results not reliable at all.

Eventually, we found that this behaviour was due to the overclocking on my personal computer. Modern laptops often have an already integrated overclocking mechanism. So, when required, they increase the clock rate to obtain much better performance in terms of speed. Increasing the clock rate artificially alters the number of operations the CPU can perform in a unit of time. Those increased performances put the system under a stress that goes beyond the actual physical capability of the machine and cannot be sustained for a long time. In our specific case my machine started to overheat so much that the system was substantially decreasing the performance in order to cool down and avoid irreparable damage to the CPU. That mechanism, present in laptops, is not easy (and safe) to work around, so we decided to conduct the experiments on a more stable environment. For that purpose Sienne, a computer belonging to the research center with good features, was a perfect choice. Below the machine specifications are reported:

Sienne:

- CPU: Intel® Core^{TM} i7-8700 Processor @ 3.20GHz
  - Microarchitecture: Coffee Lake
  - Physical cores: 6
  - Logical cores: 12
  - Architecture: x86_64
  - Frequency (min/base/max): 0.8/3.2/4.6 GHz
  - D-Caches (L1/L2/L3): 32/256/12288 KB (L1/L2 per cores; L3 shared among cores)
  - SIMD: mmx, sse, sse2, ssse3, sse4_1, sse4_2, avx, avx2
- RAM: DDR4 DIMM 32GB @ 2666Mhz

Thanks to that stable and high performance machine, I was able to obtain reliable and comparable results from my tests and to concentrate on machine independent optimization. Additionally, the high number of threads of Sienne allowed us to do some interesting exploration into the parallelization process.

2.2.2 Input code

Before proceeding it should be emphasized that the type of code we focus on in this thesis is code involving nested loops. A nested loop is a loop in a loop, an inner loop within the body of an outer one. There are two kinds of nested loops:

- Perfectly nested loops, see Algorithm 1
- Multiple nested loops, see Algorithm 2

Algorithm 1 Perfectly nested loops

```
// Outer loop.
for a in 1 2 3 4 5 do
    print "Pass %a in outer loop."
    // Inner loop.
    for b in 1 2 3 4 5 do
        print "Pass %b in inner loop."
        // Innermost loop.
        for c in 1 2 3 4 5 do
            print "Pass %c in innermost loop."
        end for
    end for
end for
```

Algorithm 2 Multiple nested loops

```
// Outer loop.
for a in 1 2 3 4 5 do
    print "Pass %a in outer loop."
    // First inner loop.
    for b in 1 2 3 4 5 do
        print "Pass %b in inner loop."
    end for
    // Second inner loop.
    for c in 1 2 3 4 5 do
        print "Pass %c in inner loop."
    end for
end for
```

Since our work was primarily focused on improving the tiling transformation, the loops contained in the input code have to involve vectors big enough to greatly exceed the caching capabilities. In fact, if the size of the arrays is too small the data would all be stored inside the cache, and tiling would not produce beneficial results.

In addition, the implementation of the input code has a major impact on the observability of optimization effect. For that purpose, I selected the Jacobi method of Pluto's benchmarks [21] as a case study and implemented it in different ways (see the codes below).

To show what guided the final choice on what code to test on, we did some profiling of code performances with Intel VTune. The insight it offers is not perfectly accurate because profiling tools do not really execute the program but just sample the program providing some approximate information about the memory usage, the execution time...etc

I chose to use different implementations of Jacobi code as starting point because there are enough dependencies among variables to observe the impact of tiling on the parallelization but not so many to make impossible to identify them by hand.

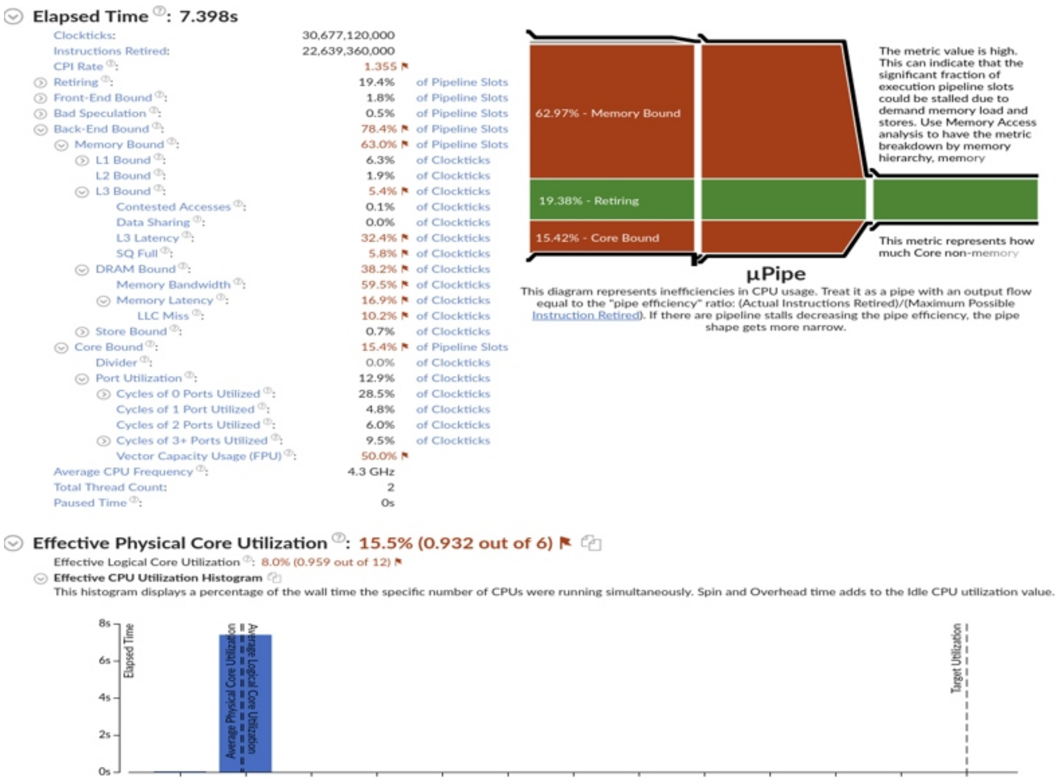
Jacobi method - copy versions

```c
...
#define N 2000
#define T 1000

double a[N][N]; //We will use two matrices
double b[N][N];
...
int main()
{
  int t, i, j, t2, i2, j2;
  double t_start, t_end;
  init_array();
  t_start = rtclock();

l0:
  for (t = 0; t<T; t++) {
    for (i = 2; i<N - 1; i++) {
      for (j = 2; j<N - 1; j++) {
        b[i][j] = 0.2*(a[i][j] + a[i][j - 1] +
          a[i][1 + j] + a[1 + i][j] +
          a[i - 1][j]);
      }
    }

    for (i2 = 2; i2<N - 1; i2++) {
      for (j2 = 2; j2<N - 1; j2++) {
        a[i2][j2] = b[i2][j2];
      }
    }
  }
  t_end = rtclock();
  fprintf(stdout, "%0.6lfs\n", t_end - t_start);
  return 0;
}
```

Fig. 2.1 shows the profiling of Jacobi code (copy version) on a single thread. The memory bound percentage is noticeably high. More than 62% of CPU resources are wasted waiting for memory operations to complete. This behaviour is mainly due to an inefficient management of the cache, that is filled and overwritten before the data stored can be reused. The average execution time is about 6 seconds.

Jacobi method - modulo version

```c
...
#define N 2000
#define T 1000
```



```
Elapsed Time: 7.398s
    Clockticks:                                   30,677,120,000
    Instructions Retired:                         22,639,360,000
    CPI Rate:                                              1.355 ⚑
    Retiring:                                              19.4%    of Pipeline Slots
    Front-End Bound:                                        1.8%    of Pipeline Slots
    Bad Speculation:                                        0.5%    of Pipeline Slots
    Back-End Bound:                                        78.4% ⚑  of Pipeline Slots
        Memory Bound:                                      63.0% ⚑  of Pipeline Slots
            L1 Bound:                                       6.3%    of Clockticks
            L2 Bound:                                       1.9%    of Clockticks
            L3 Bound:                                       5.4% ⚑  of Clockticks
                Contested Accesses:                         0.1%    of Clockticks
                Data Sharing:                               0.0%    of Clockticks
                L3 Latency:                                32.4% ⚑  of Clockticks
                SQ Full:                                    5.8% ⚑  of Clockticks
            DRAM Bound:                                    38.2% ⚑  of Clockticks
                Memory Bandwidth:                          59.5% ⚑  of Clockticks
                Memory Latency:                            16.9% ⚑  of Clockticks
                    LLC Miss:                              10.2% ⚑  of Clockticks
            Store Bound:                                    0.7%    of Clockticks
        Core Bound:                                        15.4% ⚑  of Pipeline Slots
            Divider:                                        0.0%    of Clockticks
            Port Utilization:                              12.9%    of Clockticks
                Cycles of 0 Ports Utilized:                28.5%    of Clockticks
                Cycles of 1 Port Utilized:                  4.8%    of Clockticks
                Cycles of 2 Ports Utilized:                 6.0%    of Clockticks
                Cycles of 3+ Ports Utilized:                9.5%    of Clockticks
                Vector Capacity Usage (FPU):               50.0% ⚑
    Average CPU Frequency:                               4.3 GHz
    Total Thread Count:                                        2
    Paused Time:                                              0s
```

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. Use Memory Access analysis to have the metric breakdown by memory hierarchy, memory

This metric represents how much Core non-memory

µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Effective Physical Core Utilization: 15.5% (0.932 out of 6) ⚑

Effective Logical Core Utilization: 8.0% (0.959 out of 12) ⚑

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

8s
6s
4s
2s
0s
Elapsed Time
Average Physical Core Utilization
Average Logical Core Utilization
Target Utilization

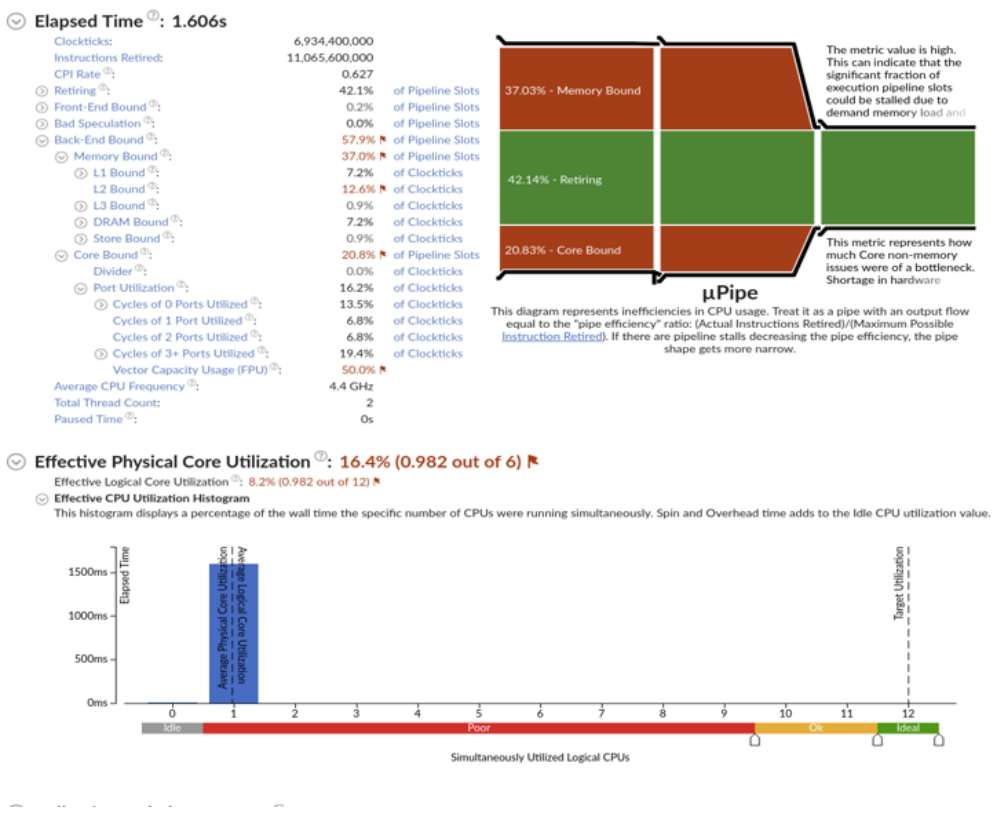
FIGURE 2.1: Profiling of the code Jacobi.

```c
double a[2][N][N];
double b[2][N][N];
...
int main()
{
  int t, i, j;
  double t_start, t_end;
  init_array(); //initialization of the array
  t_start = rtclock();

//The loop on which I want to apply tiling
l0:
  for (t = 0; t<T; t++) {
    for (i = 2; i<N - 1; i++) {
      for (j = 2; j<N - 1; j++) {
        a[(t + 1) % 2][i][j] = 0.2*(a[(t) % 2][i][j] +
          a[(t) % 2][i][j - 1] +
          a[(t) % 2][i][1 + j] +
          a[(t) % 2][1 + i][j] +
          a[(t) % 2][i - 1][j]);
      }
    }
  }
  t_end = rtclock();
  fprintf(stdout, "%0.6lfs\n", t_end - t_start);

  return 0;
}
```

Fig. 2.2 shows the profiling of Jacobi code (modulo version) on a single thread.

Here the memory bound shows an impact on CPU resources of 37% spent waiting for memory operations. It is still high, meaning this is probably not the best implementation of Jacobi method. Still it shows a drastic improvement respect to the preceding version.

The execution time, on average, of this code is 3.05 seconds.



```
Elapsed Time: 1.606s
    Clockticks:                                  6,934,400,000
    Instructions Retired:                       11,065,600,000
    CPI Rate:                                            0.627
    Retiring:                                            42.1%    of Pipeline Slots
    Front-End Bound:                                      0.2%    of Pipeline Slots
    Bad Speculation:                                      0.0%    of Pipeline Slots
    Back-End Bound:                                      57.9%    of Pipeline Slots
        Memory Bound:                                    37.0%    of Pipeline Slots
            L1 Bound:                                     7.2%    of Clockticks
            L2 Bound:                                    12.6%    of Clockticks
            L3 Bound:                                     0.9%    of Clockticks
            DRAM Bound:                                   7.2%    of Clockticks
            Store Bound:                                  0.9%    of Clockticks
        Core Bound:                                      20.8%    of Pipeline Slots
            Divider:                                      0.0%    of Clockticks
            Port Utilization:                            16.2%    of Clockticks
                Cycles of 0 Ports Utilized:              13.5%    of Clockticks
                Cycles of 1 Port Utilized:                6.8%    of Clockticks
                Cycles of 2 Ports Utilized:               6.8%    of Clockticks
                Cycles of 3+ Ports Utilized:             19.4%    of Clockticks
                Vector Capacity Usage (FPU):             50.0%
    Average CPU Frequency:                              4.4 GHz
    Total Thread Count:                                      2
    Paused Time:                                            0s
```

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and

This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware

µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Effective Physical Core Utilization: 16.4% (0.982 out of 6)

Effective Logical Core Utilization: 8.2% (0.982 out of 12)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

FIGURE 2.2: Profiling of the code Jacobi, in the modulo version.

That memory access issue is not limited to performance on mono-thread execution, but has an important impact on execution time improvements after tiling and optimization on multi-thread runs. In fact, as we can see on the Jacobi tables in [19], the copy version shows very weak improvements on parallel executions compared to the modulo version. In conclusion, it is important to be able to appreciate the effects of loop tiling optimization, to work with a appropriate version of our code that takes into account the minor memory usage and the least possible dependencies between variables.

2.2.3 Tiling matrix selection

When we select the tiling matrix we have to take into consideration the shape and the size of the tiles.

Tile shape

We had to perform manual calculations to determine the cone of dependencies within the confines of we are allowed to choose the tiling matrix. Generally we preferred to use simple, regular shapes as tiling matrix, in order to make the subsequent loop bound computation by the compiler easier to manage.

Tile size

Experimentally we found that a good size for tiling our pieces of code was 16 or 32. That choice is confirmed by other groups of researchers working on compilers, see reference [21]. In fact, having too many tiles would congestion the twelve threads of Sienne, because the large number of tiles to support does not allow a simultaneous computation by threads. On the other hand too large tiles would go beyond cache capacity, thus deteriorating the locality.

While this approach is clearly not precise in finding the most suitable tiling matrix, it gave us good results. Anyway, finding the best possible tiling matrix is not a trivial task. The CRI team itself is dedicating resources to explore that interesting topic with new approaches, like for example by using machine learning to find the best option. They also collaborate with the University of Urbana-Champaign where a tool has been developed to test tiling with all possible matrices.

2.2.4 Parallel code versions

PIPS, starting from the same tiling matrix, is able to generate seven different versions of parallel code. Having so many different versions makes not so obvious the choice of the best version.

There are two different ways to scan the tiles, either following the "TP" direction (i.e. orthogonal direction respect to the hyperplan sequential direction) or following the partitioning vector direction "TS". Fig. 2.3 shows that the hyperplan sequential direction is parallel to the sum of the dependency vectors. The sequential direction holds all the dependencies, while along the "TP" direction the data are independent and so computable in parallel. The "TS" direction instead is parallel to the partitioning vector direction, see fig. 2.4

Furthermore there are three different ways to scan the elements inside each tile, following:

- LI - the initial layout of the data
- LS - partitioning vector direction
- LP - orthogonal vector respect to hyperplan sequential direction.

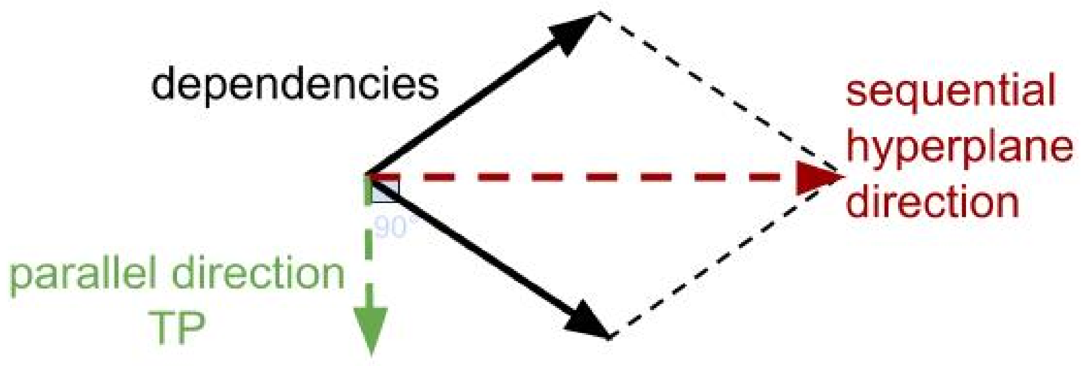


FIGURE 2.3: Tiles scanning direction TP.

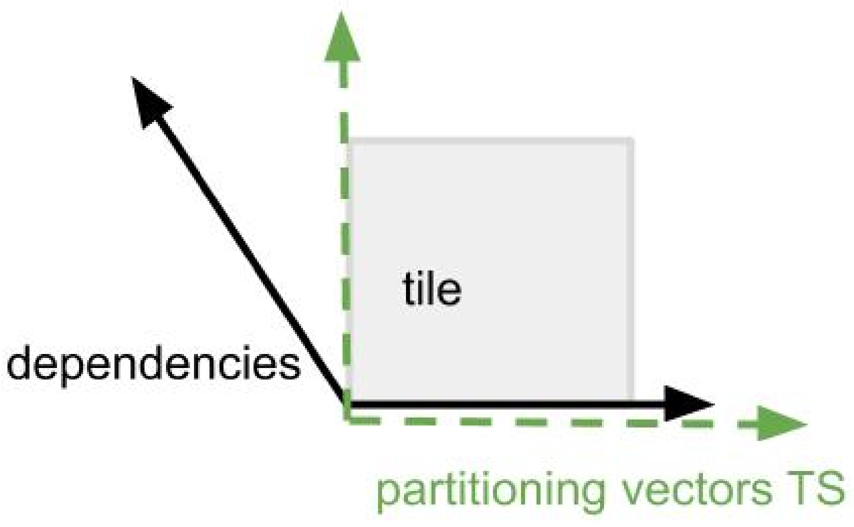


FIGURE 2.4: Tiles scanning direction TS.

Fig. 2.5 shows the scanning directions of the elements in each tile.

From the experiments emerges that in the vast majority of cases, the best option was the TS-LI, that means scanning the tiles following the partitioning vector direction and scanning the elements in each tile following the initial order of the elements. One possible explanation could be that following this order we respect the order in which data are stored into the cache, the “array layout”. To understand if there is a preferential scanning direction and if we can discard some directions is not the scope of this thesis and many other experiments are required for this purpose.

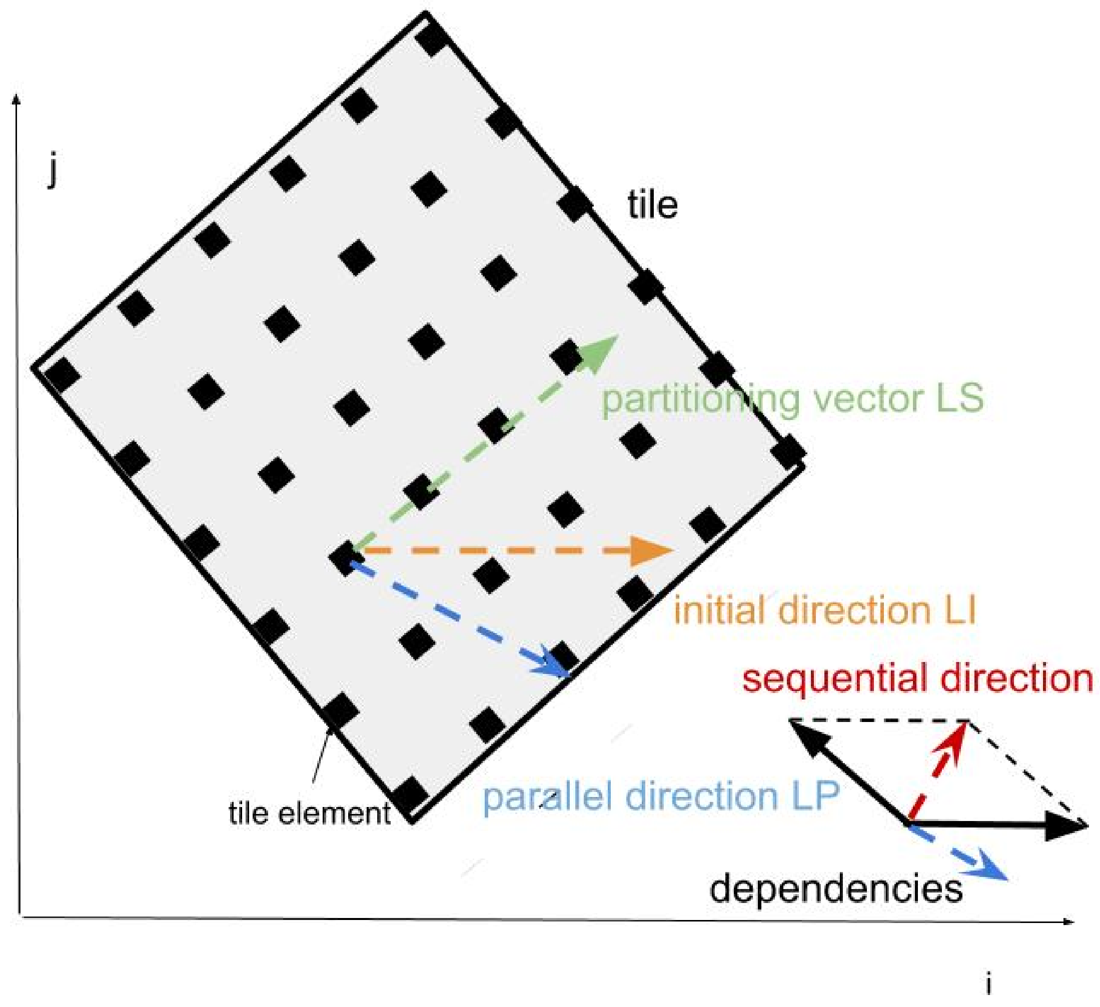


FIGURE 2.5: Scanning directions of elements inside each tile.

2.3 Optimization work

At this point we have all we need to find the inefficiencies inside PIPS code generation phase. Every choice we made in order to optimize the PIPS loop tiling process was evaluated in terms of speed-up. The speed-up measures the improvement of performances from the old version of code to the new one.

$$\text{Speed-up} = \frac{\text{execution time previous version}}{\text{execution time current version}} \tag{2.1}$$

2.3.1 Generation of sequential code

This section is dedicated to the evaluation of the generation of sequential code. The sequential code is the code generated by PIPS, compiled with gcc ignoring the OpenMP directives and executed on a single thread. Every comparison will be made in terms of speed-up of the optimized sequential generated code compared to the input code.

Invariant code motion

We noticed that some loop bounds were computed several times. While that behaviour is not efficient, saving every invariant loop bound inside memory would not be optimal as well. Saving the loop bounds of the outermost parallel loop and of the innermost loop (if vectorial) into variables resulted to be the solution showing the best results. Consider the example below to clarify what we mean with "invariant code motion". The loop bounds that have been moved are highlighted.

```c
int lbp, ubp, lbv, ubv;
for (t_t = 0; t_t <= 12; t_t += 1) {
  lbp = t_t / 2;
  ubp = (t_t + 16) / 2;
#pragma omp parallel for private (lbv, ubv,j_t,k_t,t_l,i_l,j_l,k_l )
  for (i_t = lbp; i_t <= ubp; i_t += 1)
    for (j_t = t_t / 2; j_t <= (t_t + 16) / 2; j_t += 1)
       for (k_t = t_t / 2; k_t <= (t_t + 16) / 2; k_t += 1)


        for (t_l = pips_max_4(32 * i_t - 255, 32 * j_t - 255, 32 * k_t - 255, 16 * t_t); t_l <= pips_min_2(198, 16 * t_t + 15); t_l += 1)
          for (i_l = pips_max_2(1, 32 * i_t - t_l + 1); i_l <= pips_min_2(256, 32 * i_t - t_l + 32); i_l += 1)
            for (j_l = pips_max_2(1, 32 * j_t - t_l + 1); j_l <= pips_min_2(256, 32 * j_t - t_l + 32); j_l += 1) {
              lbv = pips_max_2(1, 32 * k_t - t_l + 1);
              ubv = pips_min_2(256, 32 * k_t - t_l + 32);
#pragma ivdep
#pragma vector always
              for (k_l = lbv; k_l <= ubv; k_l += 1)
                A[(t_l + 1) % 2][i_l][j_l][k_l] = alpha * A[t_l % 2][i_l][j_l][k_l] +
                beta * (A[t_l % 2][i_l - 1][j_l][k_l] + A[t_l % 2][i_l][j_l - 1][k_l] +
                  A[t_l % 2][i_l][j_l][k_l - 1] + A[t_l % 2][i_l + 1][j_l][k_l] +
                  A[t_l % 2][i_l][j_l + 1][k_l] + A[t_l % 2][i_l][j_l][k_l + 1]);

            }
}
```

The experiments showed very promising results in some cases. With Jacobi method, for example, we saw a speed-up improvement of 2, see Table 2.1 and Table 2.2.

Jacobi modulo - parallel tiling TS-LI		
	Execution time	Speed-up
Before ICM	4.57 s	0.67
After ICM	2.50 s	1.22

TABLE 2.1: The speed-up is doubled after ICM

.

Jacobi copy - parallel tiling TS-LI		
	Execution time	Speed-up
Before ICM	5.94 s	1.00
After ICM	2.78 s	2.18

TABLE 2.2: The speed-up is doubled after ICM.

Although we obtained substantial improvements in terms of speed up in some cases, we noticed that applying ICM to other kinds of code showed minor or negligeable changes. Let's consider Table 2.3 and Table 2.4.

3dheat modulo- parallel tiling TS-LI		
	Execution time	Speed-up
Before ICM	6.31 s	0.54
After ICM	4.90 s	0.71

TABLE 2.3: The speed-up improvement after ICM is modest.

Advect3d modulo- parallel tiling TS-LI		
	Execution time	Speed-up
Before ICM	0.44 s	0.97
After ICM	0.44 s	0.96

TABLE 2.4: The speed-up improvement after ICM is negligeable.

The ICM performed by default by gcc could explain that erratic behaviour. Consider this code, generated by PIPS, as it was, from the Jacobi-copy version method.

```
...
        lb = pips_max_4(3 * i_l - 32 * i_t + 32 * j_t - 33, i_l - 1998, -2 * i_l + 32 * j_t + t_l + 32 * t_t + 2, t_l + 2);
    ub = pips_min_4(3 * i_l - 32 * i_t + 32 * j_t - 2, i_l - 2, -2 * i_l + 32 * j_t + t_l + 32 * t_t + 33, t_l + 1998);
    for (j_l = lb; j_l <= ub; j_l += 1)
      b[i_l - j_l][j_l - t_l] = 0.2*(a[i_l - j_l][j_l - t_l] + a[i_l - j_l][j_l - t_l - 1] + a[i_l - j_l][j_l - t_l + 1] + a[i_l - j_l + 1][j_l - t_l] + a[i_l - j_l - 1][j_l - t_l]);
...
```

Performing tiling here leaded to complex loop bounds, so complex that even gcc is not able to isolate ad move them to variables. The complexity of those loop bounds makes the ICM operation even more necessary, since recalculating them is an heavy operation both resources and time wise.

Now, consider this other code. It is the code generated by PIPS, as it was, by performing tiling on advect3d code.

```
...
    lb = 4 * k_t_t + 4;
    ub = pips_min_2(306, 4 * k_t_t + 7);
    for (k_l_l = lb; k_l_l <= ub; k_l_l += 1)
        al[j_l_l][i_l_l][k_l_l] = (0.2*(a[j_l_l][i_l_l - 1][k_l_l] + a[j_l_l][i_l_l][k_l_l]) + 0.5*(a[j_l_l][i_l_l - 2][k_l_l] + a[j_l_l][i_l_l + 1][k_l_l]) + 0.3*(a[j_l_l][i_l_l - 3][k_l_l] + a[j_l_l][i_l_l + 2][k_l_l]))*0.3*uxl[j_l_l][i_l_l][k_l_l];
...
```

As we can see, the loop bounds are simple and that explains why the invariant code motion we performed on the generated code does not change substantially the performances (gcc was already doing it). Since we relied on gcc to perform the invariant code motion on the code, and since this issue did not manifest systematically on all codes, it was not obvious to identify this inefficiency. At that point, we made a lot of experiments and trials to understand if ICM should be performed and in which cases. Finally we decided to apply the invariant code motion to all the code. That transformation in fact is safe: it never worsens the speed-up and it removes a critical risk for performances. For that reason we decided to implement a new compilation phase to preemptively apply ICM on the parallel tiled code generated by PIPS. The pseudo code for this new phase, now implemented in PIPS, can be found into the Appendix B.

Minimum and maximum inline function

We substituted the calls to functions for the computation of the PIPS maximum and minimum with references to an external script, where we defined them recursively, see Appendix C.

```
#include "define_script.h"
```

That allows the C preprocessor to include them directly on the code, avoiding the calls to functions during the compilation.

In Table 2.5 and Table 2.6 there are two examples of code on which this operation leads on a slightly better speed-up.

Jacobi modulo- parallel tiling TS-LI		
	Execution time	Speed-up
Before in-lining	2.50 s	1.22
After in-lining	2.44 s	1.25

TABLE 2.5: Speed-up improvement.

Jacobi copy - parallel tiling TS-LI		
	Execution time	Speed-up
Before in-lining	2.78 s	2.18
After in-lining	2.78 s	2.19

TABLE 2.6: Speed-up improvement.

2.3.2 Scalability

Selection of the OpenMP parallel directions

PIPS is able to detect the parallel loops in the code, but simply marking with OpenMP directives all the parallel loops is not efficient because on nested parallel loops it leads to a quasi-sequential execution due to the overhead of OpenMP thread creation. Marking only the outermost parallel loop as parallel for OpenMP is the best choice to efficiently exploit the computational resources of each thread in the machine.

Any innermost parallel loop which does not contain nested loops should be marked as vectorial. The OpenMP directives for vectorial loops are:

```
#pragma always
#pragma ivdep
```

Consider the example below to clarify the concept of OpenMP directive selection.

```
...
#pragma omp parallel for private(i_t_t_t)
for(j_t_t_t = 0; j_t_t_t <= 25; j_t_t_t += 1)
#pragma omp parallel for private(k_t_t_t)
  for(i_t_t_t = 0; i_t_t_t <= 25; i_t_t_t += 1)
#pragma omp parallel for private(j)
    for(k_t_t_t = 0; k_t_t_t <= 75; k_t_t_t += 1)
#pragma omp parallel for private(i)
      for(j = 12*j_t_t_t+4; j <= pips_min(2, 306, 12*j_t_t_t
    +15); j += 1)
#pragma omp parallel for private(k)
        for(i = 12*i_t_t_t+4; i <= pips_min(2, 306, 12*
    i_t_t_t+15); i +=1)
#pragma omp parallel for
          for(k = 4*k_t_t_t+4; k <= 4*k_t_t_t+7; k += 1)
            af[j][i][k]=(0.2*(a[j][i][k-1]+a[j][i][k])+0.5*(
    a[j][i][k-2]+a[j][i][k+1])+0.3*(a[j][i][k-3]+a[j][i][k
    +2]))*0.3*uzf[j][i][k];

...
```

In order to maintain correctness the threads should not share the variables that were marked as parallel. So, we mark them as private variables for the parallel loop we keep.

The changes for the above example are highlighted in orange for the modified lines and in gray for the deleted lines.

```
...
#pragma omp parallel for private(i_t_t_t,k_t_t_t,j,i,k)
for(j_t_t_t = 0; j_t_t_t <= 25; j_t_t_t += 1)
```

```
#pragma omp parallel for private(k_t_t_t)
  for(i_t_t_t = 0; i_t_t_t <= 25; i_t_t_t += 1)
#pragma omp parallel for private(j)
    for(k_t_t_t = 0; k_t_t_t <= 75; k_t_t_t += 1)
#pragma omp parallel for private(i)
      for(j = 12*j_t_t_t+4; j <= pips_min(2, 306, 12*j_t_t_t
   +15); j += 1)
#pragma omp parallel for private(k)
        for(i = 12*i_t_t_t+4; i <= pips_min(2, 306, 12*
   i_t_t_t+15); i +=1)
#pragma omp parallel for
#pragma ivdep
#pragma always
          for(k = 4*k_t_t_t+4; k <= 4*k_t_t_t+7; k += 1)

            af[j][i][k] = (0.2*(a[j][i][k-1]+a[j][i][k])
   +0.5*(a[j][i][k-2]+a[j][i][k+1])+0.3*(a[j][i]
           [k-3]+a[j][i][k+2]))*0.3*uzf[j][i][k];

...
```

This operation is already implemented in the new PIPS phase which pseudo code is in Appendix B.

Number of threads

When run in parallel a scalable code execution time decreases linearly. So, if we double the number of cores the execution time should halve. We executed all our codes on a various number of threads, see Appendix A. One of the issues I focused on was to understand the best number of threads for our target machine Sienne. Tests were conducted on 1, 2, 4, 8, 12, 16 threads.

Executing the code on a single thread is useful to understand if an high overhead has been introduced by the parallel directives.

Testing the code on more than 12 threads (the maximum for Sienne), had the purpose to confirm that performances worsen due to the presence of tasks assigned to exceeding threads and forced to wait.

Chapter 3

Results

In this chapter we consider the data gathered during our tests on PIPS code generation before and after the optimizations described on Chapter 2. Tables summarizing the experiments results can be found in [19].

A comparison will be made between PIPS former and actual version; subsequently PIPS will be compared with Pluto [21], a well known source-to-source compiler.

Finally some words will be spent about the Intel compiler [14] as an alternative to gcc [9].

3.1 Considerations on PIPS's improvements

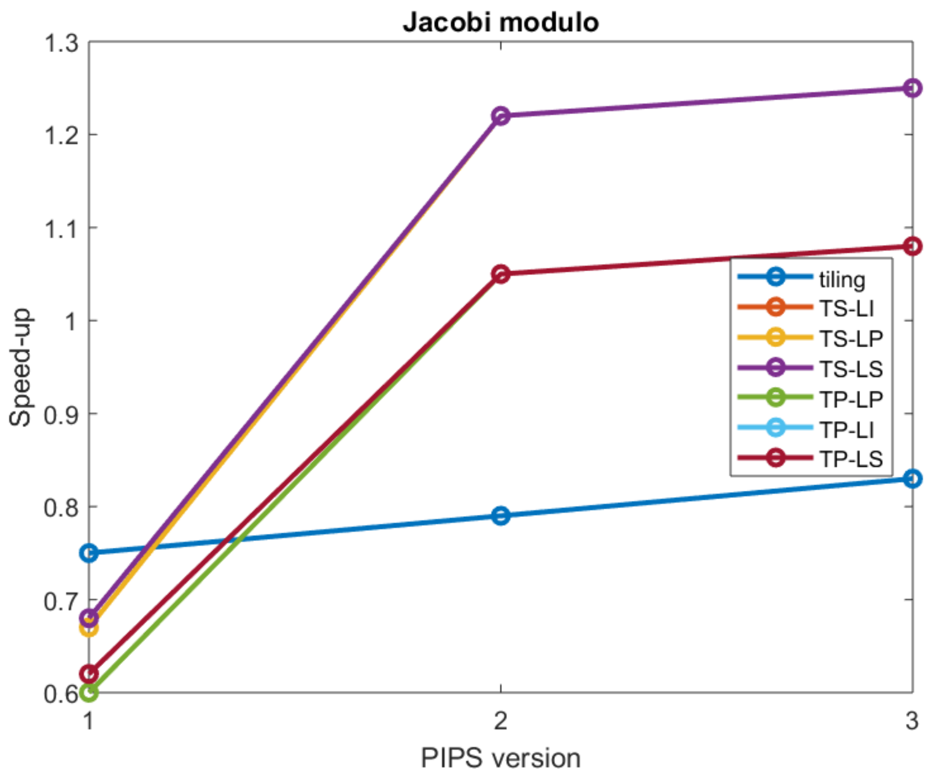
To evaluate if the optimization work led to significant results, without compromising data locality and parallelism, we will consider the speed-up improvements, the parallel directives overhead and the scalability of generated code compared to PIPS former version.

3.1.1 Speed-up improvement

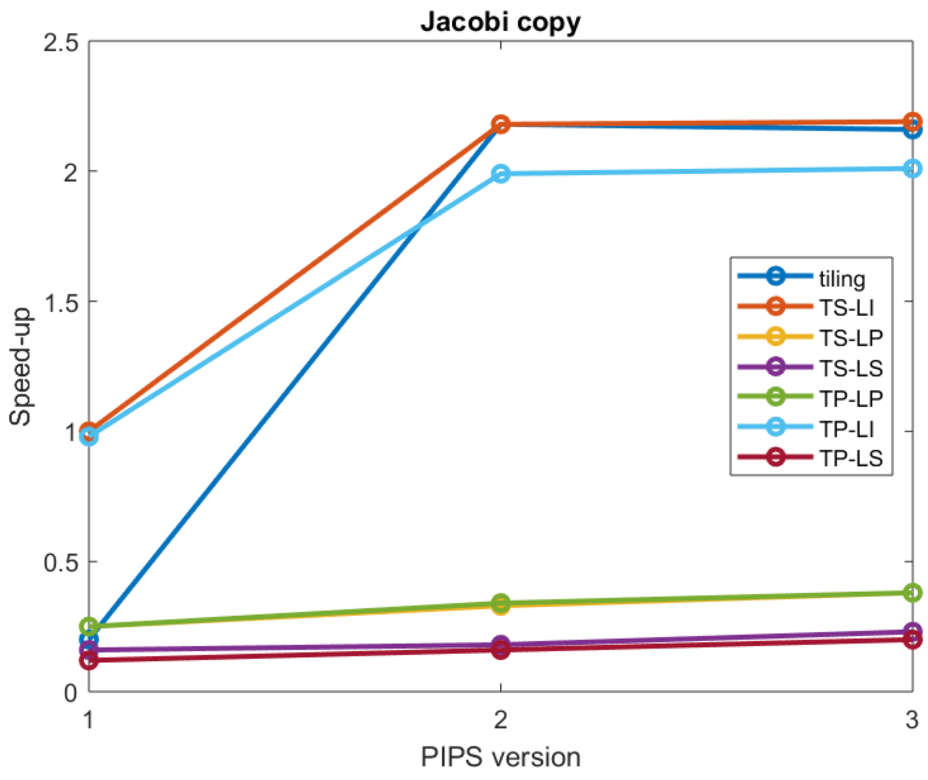
As shown on the following graphs, the speed-up of the new version of PIPS presents improvements over the former on every version of generated code. On the sequential tiled code of Jacobi modulo version the speed-up improvement is greater than 2.

On the x-axis:

- **1:** PIPS as it was before optimization
- **2:** PIPS after invariant code motion and OpenMP directive selection
- **3:** PIPS final version i.e. the version at point 2 with the in-lined minimum and maximum function



(A) Jacobi modulo



(B) Jacobi copy

FIGURE 3.1: Speed-up improvement

3.1.2 Parallel directive overhead

To be sure that data locality has not been compromised by the insertion of parallel OpenMP directives, we must compare the execution time of the sequential version (on which gcc ignores OpenMP directives) and the parallel version (with OpenMP directives) executed on a single thread.

If data locality has been preserved, the values should be similar, see Chapter 1.

Table 3.1 shows as, for our example codes, the overhead introduced by OpenMP directives is modest.

TABLE 3.1: Parallel directives overhead is small

	Jacobi modulo		Jacobi copy	
PIPS parallel tiling	sequential	parallel on 1 thread	sequential	parallel on 1 thread
tiling	3.69	3.92	2.83	2.56
TS-LP	2.44	2.56	15.93	17.25
TS-LI	2.44	2.58	2.78	2.57
TS-LS	2.44	2.58	26.80	29.49
TP-LP	2.82	2.62	16.02	16.95
TP-LI	2.82	2.97	3.03	2.85
TP-LS	2.81	2.98	30.29	34.90

	3dheat		advect3d	
PIPS parallel tiling	sequential	parallel on 1 thread	sequential	parallel on 1 thread
tiling	4.78	4.83	0.65	0.63
TS-LP	5.13	4.90	0.93	0.93
TS-LI	4.90	4.92	0.44	0.45
TS-LS	5.20	4.95	0.61	0.62
TP-LP	6.85	6.70	-	-
TP-LI	6.80	6.62	-	-
TP-LS	6.57	6.68	-	-

3.1.3 Scalability

To evaluate scalability we compare the execution time of the generated code on different number of threads; as a general rule the speed-up of a scalable code increases depending on the number of threads (with the caveats illustrated on Chapter 2).

On Jacobi modulo code we notice worst performances on 16 threads compared to 12. That is not unexpected, considering the maximum number of threads on Sienne is 12. Avect3d and 3dheat codes show a different behaviour, with a better speed-up on 8 and 16 threads compared to 12. but that behaviour is probably to ascribe to peculiar features of the code, which appears to be more optimized on 8 (and multiples) threads.

As shown on the following graph (Fig 3.2), every version of code is scalable, and the improvements are particularly noticeable on TS-LI version, which we will use as reference version for comparisons. See Table 3.2 for scalability data on TS-LI versions.

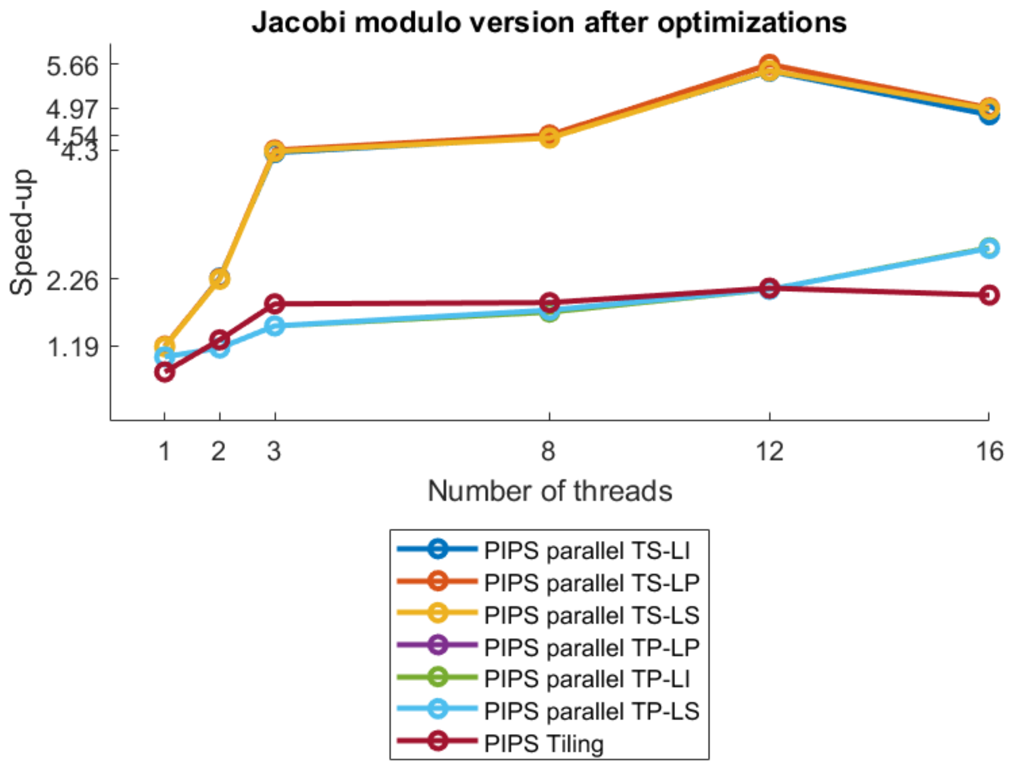


FIGURE 3.2: TS-LI is the best direction.

Speed-up improvement - parallel tiling TS-LI compiled with -fopenmp				
Threads number	advect3d	Jacobi modulo	Jacobi copy	3dheat
1	0.95	1.18	2.09	0.70
2	1.77	2.27	4.02	1.16
4	3.08	4.26	7.48	1.74
8	4.73	4.50	6.93	1.71
12	4.53	5.55	7.20	1.66
16	4.90	4.86	6.57	1.70

TABLE 3.2: Scalability of TS-LI direction.

3.2 Comparison with Pluto

"[Pluto is] an automatic polyhedral source-to-source transformation framework that can optimize regular programs (sequences of possibly imperfectly nested loops) for parallelism and locality simultaneously. [...][Pluto's approach is] driven by an integer linear optimization framework that takes an explicit view of finding good ways of tiling for parallelism and locality using affine transformations.[...] [Pluto is able] to automatically generate OpenMP parallel code from C program sections." [5]

In order to evaluate the quality of PIPS performances in terms of scalability we chose to compare it to Pluto, because both tools perform optimizations, making use of polyhedral methods and OpenMP library, for locality and parallelism.

The following graphs show the comparison between PIPS (after the optimization) and Pluto generated codes speed-up on different numbers of threads. Pluto's tiled generated codes were compiled with the gcc compiler and executed on Sienne, the usual machine.

We can observe from the data regarding execution on one thread how the additional overhead is similar for both tools in every example code.

The similarities between the two tools are particularly noticeable on the generated code, that is almost the same, for Jacobi method (modulo version) which performances are illustrated in Fig. 3.3.

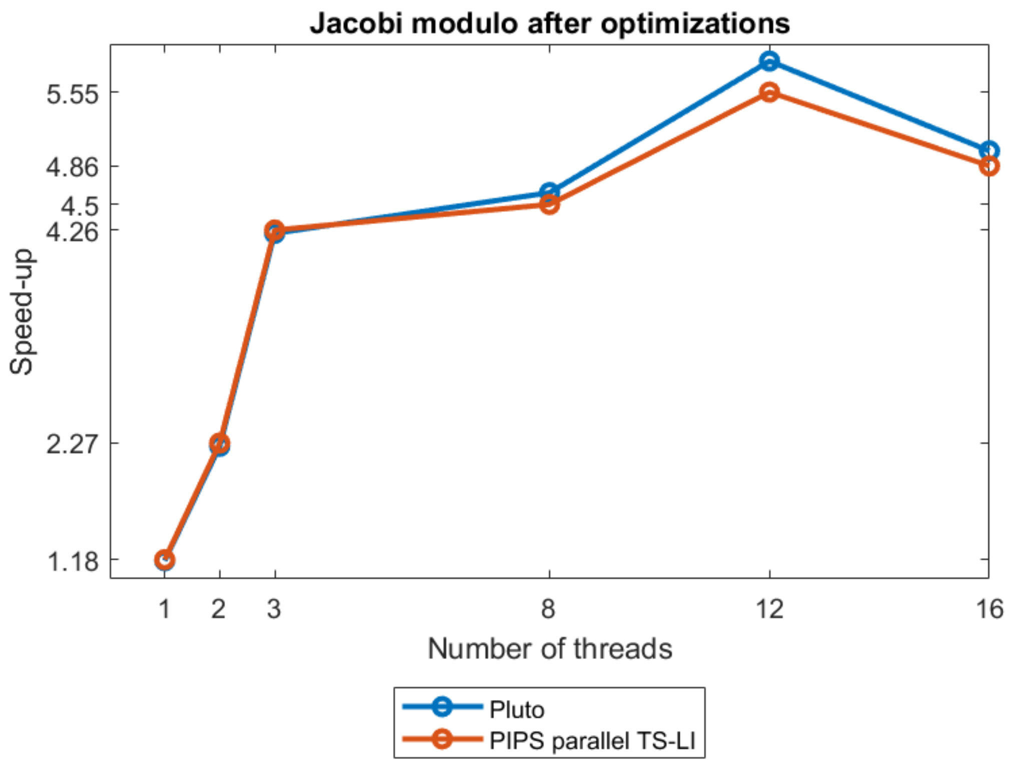


FIGURE 3.3: Comparison PIPS-Pluto performances on Jacobi modulo code.

Advec3d generated code reflects the choice PIPS developers made of conserving only non-redundant inequalities of the polyhedron representing the iteration space of loops. The result of that simplification will be a polyhedron with less faces but describing a larger iteration space for some cuts and projections.

On advec3d code the simplification of loop bounds leads to major advantages on PIPS compared to Pluto, see Fig. 3.4.

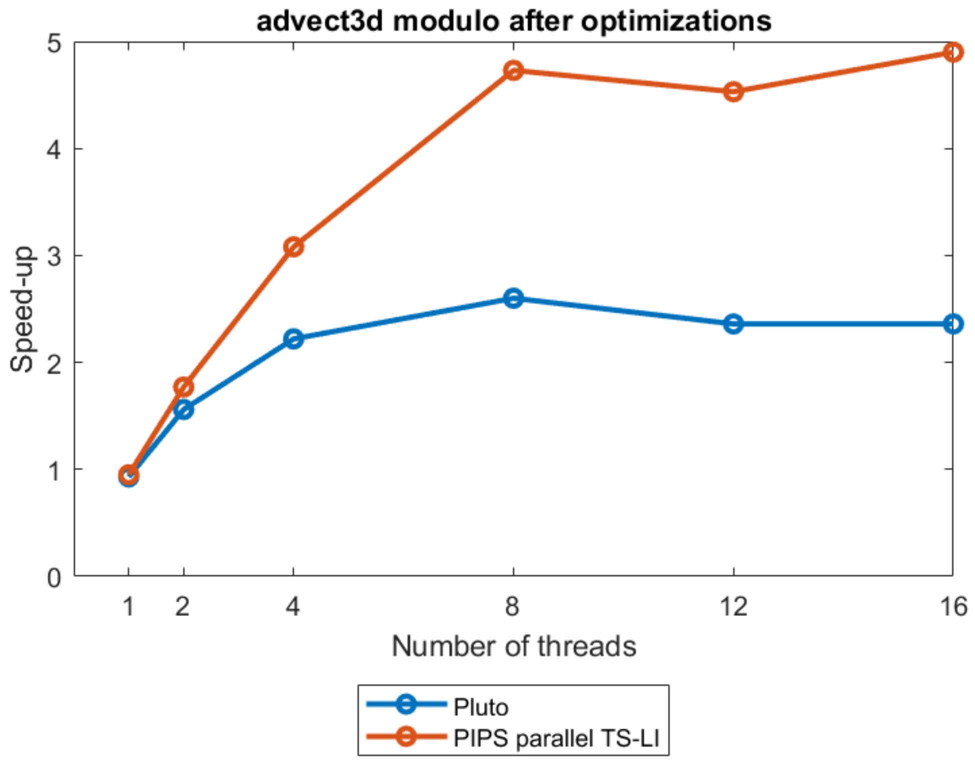


FIGURE 3.4: Comparison PIPS-Pluto performances on advect3d modulo code.

That simplifies loop constraints but, as a drawback, the number of iterations is greater.

The drawback of having a larger than necessary iteration space for loops becomes evident on heat3d generated code. On highlighted lines of the following PIPS code, the bounds are simple.

```
...
int  lbp, ubp, lbv,ubv;
for(t_t = 0; t_t <= 12; t_t += 1){
     lbp=t_t/2;
     ubp=(t_t+16)/2;
#pragma omp parallel for private (lbv,ubv,j_t,k_t,t_l,i_l,
    j_l,k_l )
     for(i_t =lbp ; i_t <=ubp; i_t += 1)
       for(j_t = t_t/2; j_t <= (t_t+16)/2; j_t += 1)
       for(k_t = t_t/2; k_t <= (t_t+16)/2; k_t += 1)
          for(t_l = pips_max_4( 32*i_t-255, 32*j_t-255, 32*
    k_t-255, 16*t_t); t_l <= pips_min_2( 198, 16*t_t+15); t_l
     += 1)
              for(i_l = pips_max_2( 1, 32*i_t-t_l+1); i_l <=
     pips_min_2( 256, 32*i_t-t_l+32); i_l += 1)
                  for(j_l = pips_max_2( 1, 32*j_t-t_l+1);
    j_l <= pips_min_2( 256, 32*j_t-t_l+32); j_l += 1){
                    lbv= pips_max_2( 1, 32*k_t-t_l+1);
                    ubv= pips_min_2( 256, 32*k_t-t_l+32);
#pragma ivdep
#pragma vector always
                    for(k_l =lbv ; k_l<= ubv ; k_l += 1)
                        A[(t_l+1)%2][i_l][j_l][k_l] = ...
                    }
}
...
       for(k_t = t_t/2; k_t <= (t_t+16)/2; k_t += 1)
          for(t_l = pips_max_4( 32*i_t-255, 32*j_t-255, 32*
    k_t-255, 16*t_t); t_l <= pips_min_2( 198, 16*t_t+15); t_l
     += 1)
              for(i_l = pips_max_2( 1, 32*i_t-t_l+1); i_l <=
     pips_min_2( 256, 32*i_t-t_l+32); i_l += 1)
.lastline.lastline
                  for(j_l = pips_max_2( 1, 32*j_t-t_l+1);
    j_l <= pips_min_2( 256, 32*j_t-t_l+32); j_l += 1){
                    lbv= pips_max_2( 1, 32*k_t-t_l+1);
                    ubv= pips_min_2( 256, 32*k_t-t_l+32);
#pragma ivdep
#pragma vector always
                    for(k_l =lbv ; k_l<= ubv ; k_l += 1)
                        A[(t_l+1)%2][i_l][j_l][k_l] = ...
                    }
}
...
```

Noticeably, Pluto more complex generated code has a finer tuning on the highlighted loops bounds.

```
...
```

```c
int t1, t2, t3, t4, t5, t6, t7, t8;
int lb, ub, lbp, ubp, lb2, ub2;
for (t1=-1;t1<=12;t1++) {
  lbp=ceild(t1,2);
  ubp=floord(t1+16,2);
#pragma omp parallel for private(lbv,ubv,t3,t4,t5,t6,t7,t8)
  for (t2=lbp;t2<=ubp;t2++) {
    for (t3=max(0,ceild(t1-1,2));t3<=floord(t1+17,2);t3++) {
      for (t4=max(max(0,ceild(t1-1,2)),t3-8);t4<=min(floord(t1+17,2),t3+8);t4++) {
        for (t5=max(max(max(max(0,16*t1),32*t3-256),32*t4-256),32*t1-32*t2+1);t5<=min(min(min(min(198,16*t1+31),32*t2+30),32*t3+30),32*t4+30);t5++) {
          for (t6=max(max(32*t2,t5+1),-32*t1+32*t2+2*t5-31);t6<=min(min(32*t2+31,t5+256),-32*t1+32*t2+2*t5);t6++) {
            for (t7=max(32*t3,t5+1);t7<=min(32*t3+31,t5+256);t7++) {
              lbv=max(32*t4,t5+1);
              ubv=min(32*t4+31,t5+256);
#pragma ivdep
#pragma vector always
              for (t8=lbv;t8<=ubv;t8++) {
                A[(t5 + 1) % 2][(-t5+t6)][(-t5+t7)][(-t5+t8)] = ...
              }
            }
          }
        }
      }
    }
  }
 }
```

```c
      for (t4=max(max(0,ceild(t1-1,2)),t3-8);t4<=min(floord(t1+17,2),t3+8);t4++) {
        for (t5=max(max(max(max(0,16*t1),32*t3-256),32*t4-256),32*t1-32*t2+1);t5<=min(min(min(min(198,16*t1+31),32*t2+30),32*t3+30),32*t4+30);t5++) {
          for (t6=max(max(32*t2,t5+1),-32*t1+32*t2+2*t5-31);t6<=min(min(32*t2+31,t5+256),-32*t1+32*t2+2*t5);t6++) {
```

```c
            for (t7=max(32*t3,t5+1);t7<=min(32*t3+31,t5+256);t7++) {
              lbv=max(32*t4,t5+1);
              ubv=min(32*t4+31,t5+256);
#pragma ivdep
#pragma vector always
              for (t8=lbv;t8<=ubv;t8++) {
                A[(t5 + 1) % 2][(-t5+t6)][(-t5+t7)][(-t5+t8)] = ...
              }
            }
          }
        }
      }
    }
```

Fig. 3.5 shows how the simplification of loop bounds affects the performances of PIPS on heat3d code in comparison to Pluto.

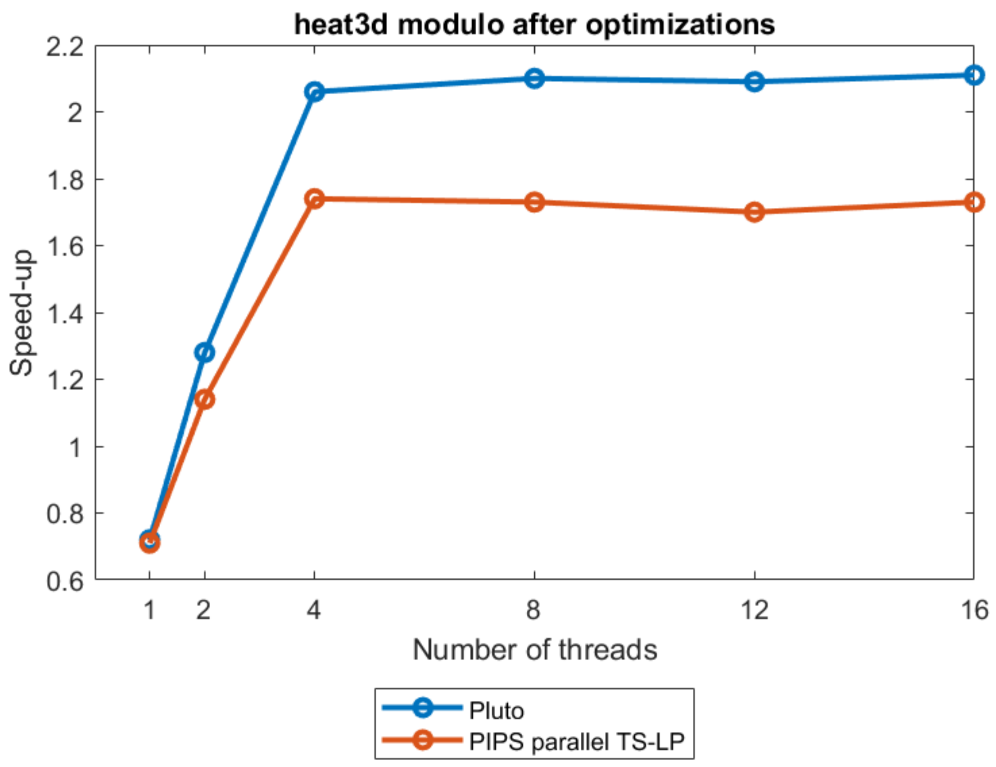


FIGURE 3.5: Comparison PIPS-Pluto performances on 3dheat modulo code.

3.3 Intel compiler

The Intel compiler represent the state of art for compilers optimized for Intel architecture.

> *"Intel C++ compiler is generally able to provide the best performance because it has a better picture of the target machine architecture, i.e., it knows how to exploit all available registers, minimize memory operations, etc. Intel C++ compiler also has good support [...] OpenMP standards."* [8]

Furthermore, Intel compiler can automatically vectorize a higher number of loops compared to the GNU compiler. [17]

Being Sienne a machine based on Intel architecture, we decided to test the speed-up of code generated by PIPS prior to our optimization effort compiled with icc, comparing it with the same code compiled with gcc and optimized code compiled with gcc. The results confirmed that when compiling unoptimized code, Intel compiler produces a better performing (on our Intel based-machine) output than GNU compiler. When compared to the performances of the output produced by GNU compiler from PIPS optimized code, anyway, Intel compiler's output is slower, even on an Intel architecture.

For the comparison on sequential code see Table 3.3.

Table 3.4 shows the comparison on parallel code on 12 threads.

	previous PIPS gcc	icc	optimized PIPS gcc
TS-LP	0.67	0.88	1.25
TS-LI	0.67	0.88	1.25
TS-LS	0.68	0.88	1.25
TP-LP	0.60	0.78	1.08
TP-LI	0.62	0.78	1.08
TP-LS	0.62	0.78	1.08

TABLE 3.3: Speed-up of Jacobi modulo - sequential code

	unoptimized PIPS gcc	unoptimized PIPS icc	optimized PIPS gcc
TS-LP	3.38	4.04	5.66
TS-LI	3.39	4.00	5.55
TS-LS	3.38	4.00	5.56
TP-LP	0.24	0.27	2.09
TP-LI	0.62	1.65	2.09
TP-LS	1.51	1.79	2.09

TABLE 3.4: Speed-up of Jacobi modulo - parallel code 12 threads

Conclusion

This research aimed to find an explanation to the lack of scalability of the tiled code generated by PIPS. Based on theoretical background and quantitative experience it can be concluded that performing invariant code motion on some specific loop bounds and in-lining the function of maximum and minimum used on loop bounds can diminish the tiled code execution time. This thesis also provided a rule to insert parallel OpenMP directives inside the tiled code to efficiently distribute the computational load amongst available threads. In essence, we were able to identify the inefficiencies we were looking for, their causes and to propose solutions.

The optimization we performed led to an overall better execution time of the code generated by PIPS, compared both with the previous version of PIPS and with Pluto, without impacting on machine independence, locality and parallelism. This work will contribute to the collective effort of the research aiming to optimize loop tiling performed by machine-independent compilers, nowadays fundamentals to exploit parallelism on super computers used in computational costly scientific fields.

Furthermore, some directions of work on loop tiling have already been identified for future consideration. For example, selecting the best tiling matrix, exploiting machine learning techniques. Understanding the correlation between the tiled code and the PIPS scanning direction of tiles and of elements inside tiles in order to select the best one or at least discard some inefficient directions would be another interesting topic to study. It would be necessary to implement the minimum and maximum function inline inside PIPS. This optimization requires a revision of the exiting PIPS code generation phase so it is still a project for the CRI team. Another critical point to confront is the improvement of PIPS dependence accuracy for codes containing modulo operation. For the moment, these dependencies are correct but remain an approximation.

In conclusion, besides having successfully pursued the objective of my internship, the procedure we designed to study this issue can be reused to extend these first experiments to other benchmarks, to validate the different stages of this new code generation process implemented in PIPS.

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools (1st ed.)* Addison-Wesley, 1986.

[2] C. Ancourt and F. Irigoin. "Scanning polyhedra with DO loops". In: *Principles and Pratice of Parallel Programming* (1991), pp. 39–50.

[3] A. Athavale, P. Randive, and A. Kambale. *Automatic Parallelization of Sequential Codes Using S2P Tool and Benchmarking of the Generated Parallel Codes*. 2011. URL: `https://pdfs.semanticscholar.org/ccf4/351a9f86a06d94cfc370d81872874a56722f.pdf`.

[4] C. Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques* (2004).

[5] U. Bondhugula et al. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *ACM SIGPLAN Programming Languages Design and Implementation (PLDI)* (2008).

[6] J.M.P Cardoso and P.C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. Springer US, 2009.

[7] K. D. D. Cooper and L. Torczon. *Engineering a Compiler (2nd ed.)* Morgan Kaufmann, 2001.

[8] Colfax research department. *A Practical Automatic Polyhedral Parallelizer and Locality Optimizer*. 2017. URL: `https://colfaxresearch.com/compiler-comparison/`.

[9] *GCC documentation*. URL: `https://gcc.gnu.org/`.

[10] *GCC optimize options*. URL: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[11] P. Gonzalez-de-Aledo et al. "An optimization approach for agent-based computational models of biological development". In: *Advances in Engineering Software* 121 (2018), pp. 262–275.

[12] S. Hack, P. Kelly, and C. Lengauer. "Loop Optimization (Dagstuhl Seminar 18111)". In: *Dagstuhl Reports* (2018), pp. 39–59.

[13] E. Ilyushin and D. Namiot. "On source-to-source compilers". In: *International Journal of Open Information Technologies* (2016).

[14] *Intel Hyper-Threading Technology, Technical User's Guide*. URL: `https://web.archive.org/web/20100821074918/http://cache-www.intel.com/cd/00/00/01/77/17705_htt_user_guide.pdf`.

[15] L. Kalms, T. Hebbeler, and D. Göhringer. "Automatic OpenCL Code Generation from LLVM-IR Using Polyhedral Optimization". In: PARMA-DITAM '18 (2018), pp. 45–50.

[16] W. Liu, F. Wang, and H. Zhou. "Parallel Seismic Modeling Based on OpenMP+AVX and Optimization Strategy". In: *Journal of Earth Science* 30.4 (2019), pp. 843–848.

[17] S. Maleki et al. "An Evaluation of Vectorizing Compilers". In: *International Conference on Parallel Architectures and Compilation Techniques* (2011).

[18] J. Nithyashri. *System Software*. Tata McGraw-Hill Education, 2010.

[19] *Numerical results of experiments*. URL: `https://drive.google.com/drive/folders/125xMkidVHoGg5ASA5mjERN2T1LU1fWva?usp=sharing`.

[20] *PIPS: Automatic Parallelizer and Code Transformation Framework*. URL: `https://pips4u.org/`.

[21] *Pluto GitHub documentation*. URL: `https://github.com/bondhugula/pluto/blob/master/doc/DOC.txt`.

[22] *Polyhedral Compilation*. URL: `http://polyhedral.info/`.

[23] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts (9th ed.)* Wiley, 2012.

[24] A. S. Tanenbaum. *Structured computer organization*. Prentice Hall, 1976.

[25] J. Zhao et al. "Revisiting Loop Tiling for Datacenters: Live and Let Live". In: *Proceedings of the 2018 International Conference on Supercomputing* (2018), pp. 328–340.

Appendices

Appendix A

Test procedure

Setting PIPS parameters

- Change "my_program" with the name of the input code.c
- Change the tiling matrix as discussed in Chapter 2

```
delete  my_program

setproperty ABORT_ON_USER_ERROR TRUE

create   my_program   my_program.c

setproperty PRETTYPRINT_LANGUAGE "C"
setproperty PRETTYPRINT_STATEMENT_NUMBER FALSE
setproperty FOR_TO_DO_LOOP_IN_CONTROLIZER TRUE

apply LOOP_TILING[main]

apply PARALLEL_LOOP_TILING[main]

10
32 0 0
-64 32 0
-64 0 32

display PRINTED_FILE[main]

apply PRIVATIZE_MODULE[main]

activate PRINT_PARALLELIZEDOMP_CODE

apply COARSE_GRAIN_PARALLELIZATION[main]

display PRINTED_FILE[main]

apply UNSPLIT

close
quit
```

First phase

```
#in this phase the shell  prepares the files for the compilation:
# 1)  change the option in the file 'my_program'.tpips
# 2)  apply pips on the c code
# 3) copy the the generated files form the folder database/src/ into the main folder
# 4) substitute  pips_max($j, with pips_max_$j( to avoid problems with macros



#echo Insert the name of the program to be executed
#read my_program

cp ${my_program}.tpips ${my_program}0.tpips

# STANDARD TILED VERSION
tpips ${my_program}.tpips > temp 2>&1

cp ${my_program}.database/Src/${my_program}.c ${my_program}_tiled.c

for j in 2 3 4 5 6 7 8 9 10
do
     sed -i 's/pips_max('$j', /pips_max_'$j'( /g'  ${my_program}_tiled.c
     sed -i 's/pips_min('$j', /pips_min_'$j'( /g'  ${my_program}_tiled.c
done


#VERSION PARALLEL TILING TS - LP

sed -i 's/LOOP_TILING/PARALLEL_LOOP_TILING/g' ${my_program}.tpips
sed -i  '/LOOP_TILING/ i\ setproperty TILE_DIRECTION "TS"' ${my_program}.tpips
sed -i  '/LOOP_TILING/ i\ setproperty LOCAL_TILE_DIRECTION "LP"'  ${my_program}.tpips

tpips ${my_program}.tpips > temp 2>&1

cp ${my_program}.database/Src/${my_program}.c ${my_program}_parall_tiled_TS_LP.c

for j in 2 3 4 5 6 7 8 9 10
do
        sed -i 's/pips_max('$j', /pips_max_'$j'( /g'  ${my_program}_parall_tiled_TS_LP.c
        sed -i 's/pips_min('$j', /pips_min_'$j'( /g'  ${my_program}_parall_tiled_TS_LP.c
done



```

```

#echo EXECUTION PARALLEL TILING TS - LI


sed -i 's/"LP"/"LI"/g' ${my_program}.tpips



tpips ${my_program}.tpips > temp 2>&1

cp ${my_program}.database/Src/${my_program}.c ${my_program}
    _parall_tiled_TS_LI.c

for j in 2 3 4 5 6
do
     sed -i 's/pips_max('$j', /pips_max_'$j'( /g'  ${
    my_program}_parall_tiled_TS_LI.c
     sed -i 's/pips_min('$j', /pips_min_'$j'( /g'  ${
    my_program}_parall_tiled_TS_LI.c
done




#VERSIO PARALLEL TILING TS - LS


sed -i 's/"LI"/"LS"/g' ${my_program}.tpips



tpips ${my_program}.tpips > temp 2>&1

cp ${my_program}.database/Src/${my_program}.c ${my_program}
    _parall_tiled_TS_LS.c

for j in 2 3 4 5 6
do
        sed -i 's/pips_max('$j', /pips_max_'$j'( /g'  ${
    my_program}_parall_tiled_TS_LS.c
        sed -i 's/pips_min('$j', /pips_min_'$j'( /g'  ${
    my_program}_parall_tiled_TS_LS.c
done



# VERSION  PARALLEL TILING TP - LP

sed -i 's/"TS"/"TP"/g' ${my_program}.tpips

sed -i 's/"LS"/"LP"/g' ${my_program}.tpips


tpips ${my_program}.tpips > temp 2>&1

cp ${my_program}.database/Src/${my_program}.c ${my_program}
    _parall_tiled_TP_LP.c
```

```

for j in 2 3 4 5 6
do
        sed -i 's/pips_max('$j', /pips_max_'$j'( /g'  ${my_program}_parall_tiled_TP_LP.c
        sed -i 's/pips_min('$j', /pips_min_'$j'( /g'  ${my_program}_parall_tiled_TP_LP.c
done


#VERSION PARALLEL TILING TP - LI



sed -i 's/"LP"/"LI"/g' ${my_program}.tpips


tpips ${my_program}.tpips > temp 2>&1

cp ${my_program}.database/Src/${my_program}.c ${my_program}_parall_tiled_TP_LI.c

for j in 2 3 4 5 6
do
        sed -i 's/pips_max('$j', /pips_max_'$j'( /g'  ${my_program}_parall_tiled_TP_LI.c
        sed -i 's/pips_min('$j', /pips_min_'$j'( /g'  ${my_program}_parall_tiled_TP_LI.c
done




#VERSION PARALLEL TILING TP - LS



sed -i 's/"LI"/"LS"/g' ${my_program}.tpips


tpips ${my_program}.tpips > temp 2>&1

cp ${my_program}.database/Src/${my_program}.c ${my_program}_parall_tiled_TP_LS.c

for j in 2 3 4 5 6
do
        sed -i 's/pips_max('$j', /pips_max_'$j'( /g'  ${my_program}_parall_tiled_TP_LS.c
        sed -i 's/pips_min('$j', /pips_min_'$j'( /g'  ${my_program}_parall_tiled_TP_LS.c
done

cp ${my_program}0.tpips ${my_program}.tpips #get the initial version of ${my_program}.tpips


```

```
#CREATION OF THE PLUTO TILED FILE

/home/gabriella/pluto/polycc  ${my_program}_pluto.c  --
    parallel --tile  --pet  -o  ${my_program}_pluto_tiled.c
```

Second phase

```

#echo Insert the name of the program to be executed
#read my_program


#compile  the program in the standard way


gcc -O3 ${my_program}.c -o ${my_program}_init

#compile  the program with PIPS

gcc -O3 ${my_program}_tiled.c -o ${my_program}_tiled  #one
    thread
gcc -O3 -fopenmp ${my_program}_tiled.c -o ${my_program}
    _tiled_omp  #with openmp

#compile  the program with PIPS parallel tiling TS LP
gcc -O3 ${my_program}_parall_tiled_TS_LP.c -o ${my_program}
    _parall_tiled_TS_LP #one thread
gcc -O3 -fopenmp  ${my_program}_parall_tiled_TS_LP.c -o ${
    my_program}_parall_tiled_TS_LP_omp #with openmp

#compile  the program with PIPS parallel tiling TS LI
gcc -O3 ${my_program}_parall_tiled_TS_LI.c -o ${my_program}
    _parall_tiled_TS_LI #one thread
gcc -O3 -fopenmp  ${my_program}_parall_tiled_TS_LI.c -o ${
    my_program}_parall_tiled_TS_LI_omp #with openmp

#compile  the program with PIPS parallel tiling TS LP
gcc -O3 ${my_program}_parall_tiled_TS_LS.c -o ${my_program}
    _parall_tiled_TS_LS #one thread
gcc -O3 -fopenmp ${my_program}_parall_tiled_TS_LS.c -o ${
    my_program}_parall_tiled_TS_LS_omp #with openmp



gcc -O3 ${my_program}_pluto_tiled.c -o ${my_program}
    _pluto_tiled #one thread

gcc -O3 -fopenmp ${my_program}_pluto_tiled.c -o ${my_program
    }_pluto_tiled_omp #with openmp
```

Third phase

```




  rm initial_version.txt
  rm tiling.txt
  rm parallel_tiling_TS_LP.txt
  rm parallel_tiling_TS_LI.txt
  rm parallel_tiling_TS_LS.txt

  rm pluto_tiling.txt


  #INITIAL VERSION
  for i in 1 2 3 4 5 6 7 8 9 10
  do

./${my_program}_init>> initial_version.txt

  done

  export init=$( numaverage  initial_version.txt)
  echo the average execution time for initial version is:
  $init

  #STANDARD TILED VERSION
  for i in 1 2 3 4 5 6 7 8 9 10
  do

./${my_program}_tiled >> tiling.txt

  done

  echo the average execution time with tiling is:   $(
 numaverage  tiling.txt) and the speed up is
  echo  "scale=4 ;  $init/ $(numaverage  tiling.txt) " |
 bc


  #PARALLEL TILED VERSION  TS LP
  for i in 1 2 3 4 5 6 7 8 9 10
  do

./${my_program}_parall_tiled_TS_LP   >>
 parallel_tiling_TS_LP.txt
  done

  echo the average execution time  with parallel tiling TS
 -LP is:   $( numaverage  parallel_tiling_TS_LP.txt) and
 the speedup is
  echo  "scale=4 ;  $init /$( numaverage
 parallel_tiling_TS_LP.txt)" | bc


  #PARALLEL TILED VERSION  TS LI
```

```
    for i in 1 2 3 4 5 6 7 8 9 10
    do
  ./${my_program}_parall_tiled_TS_LI   >>
   parallel_tiling_TS_LI.txt
    done

    echo the average execution time  with parallel tiling TS
   -LI is:   $( numaverage  parallel_tiling_TS_LI.txt) and
   the speedup is
    echo  "scale=4 ; $init/$( numaverage
   parallel_tiling_TS_LI.txt)" | bc


    #PARALLEL TILED VERSION  TS LS
    for i in 1 2 3 4 5 6 7 8 9 10
    do
  ./${my_program}_parall_tiled_TS_LS   >>
   parallel_tiling_TS_LS.txt
    done

    echo the average execution time  with parallel tiling TS
   -LS  is:   $( numaverage  parallel_tiling_TS_LS.txt) and
    the speedup is
    echo  "scale=4 ; $init/$( numaverage
   parallel_tiling_TS_LS.txt) " | bc




    #PLUTO TILED VERSION

    for i in 1 2 3 4 5 6 7 8 9 10
    do
  ./${my_program}_pluto_tiled >> pluto_tiling.txt
    done

    echo the average execution time  of the tiled pluto
   version is:   $( numaverage pluto_tiling.txt) and the
   speedup is
    echo  "scale=4 ; $init/ $(numaverage pluto_tiling.txt) "
    | bc






for nt in 1 2 4 8 12 16
do
    export OMP_NUM_THREADS=$nt
    echo nombres de threads = $nt

    rm parallel_tiling_TS_LP_omp_${nt}.txt

    rm parallel_tiling_TS_LI_omp_${nt}.txt

    rm parallel_tiling_TS_LS_omp_${nt}.txt



  rm pluto_tiling_omp_${nt}.txt



  #STANDARD TILED VERSION WITH OPENMP
  for i in 1 2 3 4 5 6 7 8 9 10
  do

./${my_program}_tiled_omp  >> tiling_omp_${nt}.txt
  done

  echo the average execution time with tiling  with openmp is:     $( numaverage  tiling_omp_${nt}.txt) and the speedup is
  echo  "scale=4 ; $init/ $( numaverage  tiling_omp_${nt}.txt)" | bc


  #PARALLEL TILED VERSION  TS LP WITH OPENMP
  for i in 1 2 3 4 5 6 7 8 9 10
  do

./${my_program}_parall_tiled_TS_LP_omp   >> parallel_tiling_TS_LP_omp_${nt}.txt
  done

  echo the average execution time  with parallel tiling with openmp  TS-LP is:    $( numaverage parallel_tiling_TS_LP_omp_${nt}.txt) and the speedup is
  echo  "scale=4 ; $init/ $( numaverage parallel_tiling_TS_LP_omp_${nt}.txt) " | bc

  #PARALLEL TILED VERSION  TS LI WITH OPENMP
  for i in 1 2 3 4 5 6 7 8 9 10
  do
./${my_program}_parall_tiled_TS_LI_omp   >> parallel_tiling_TS_LI_omp_${nt}.txt
  done

  echo the average execution time  with parallel tiling TS-LI with openmp is:    $( numaverage parallel_tiling_TS_LI_omp_${nt}.txt) and the speedup is
  echo  "scale=4 ;$init/ $( numaverage parallel_tiling_TS_LI_omp_${nt}.txt) " | bc


  #PARALLEL TILED VERSION  TS LS WITH OPENMP
  for i in 1 2 3 4 5 6 7 8 9 10
  do
./${my_program}_parall_tiled_TS_LS_omp   >> parallel_tiling_TS_LS_omp_${nt}.txt
  done

```

```
    echo the average execution time  with parallel tiling TS-LS with openmp  is:    $( numaverage parallel_tiling_TS_LS_omp_${nt}.txt) and the speedup is
    echo  "scale=4 ; $init/$( numaverage parallel_tiling_TS_LS_omp_${nt}.txt)" | bc






    #PLUTO TILED VERSION WITH OPENMP
    for i in 1 2 3 4 5 6 7 8 9 10
    do
  ./${my_program}_pluto_tiled_omp >> pluto_tiling_omp_${nt}.txt

    done

    echo the average execution time  of the tiled pluto version with openmp is:    $( numaverage pluto_tiling_omp_${nt}.txt) and the speedup is
    echo  "scale=4 ;$init/ $( numaverage pluto_tiling_omp_${nt}.txt) " | bc
done
```

Appendix B

Pseudocode new PIPS's phase

```
/*Pseudo code of the optimization of the parallelization
    phase.
*We apply this phase after that the parallelization has been
     performed.
*We have as input a parallel code.
Loops in PIPS can be be sequential, parallel or vectorial
*/

list <String> PRIV_VAR=new list();//this global variable
    keeps the list of private variables in the code

static void main(){
  statement stm= <--from database;
  tree T=extract_tree(stm); %the tree containing only loops
    and statements


  arrayList result=new arrayList[2];

  /*
  *here we have gen_recurse that applies the function down
    going down on the branch,
  *and the function up going up.
  *N.B gen_recurse starts to scan the tree from the leftmost
     branch until the rightest one,
  *it is able to analyze the next branch when it meets a
    switch
  */


  gen_recurse(
        //---------------------------------
        //GOING DOWN
        arrayList A=down(T);
        list P=A[0]; //list of parallel loops
        list C=A[1]; //list of the containers of teh
    corresponding parallel loop

        //--------------------------------
        //GOING UP
         up(P, C, T);
        stm innermost_parallel_loop=P.get(P.length -1); //
    here we catch the innermost parallel
```

```
                                                          // loop that is the last one in the list of parallel loops that we built going down in the branch
                         stm container_of_innermost_parallel_loop=C.get(C.length-1), //it's the last element of the list of containers

            if (innermost_parallel_loop is vectorial){
          move_bounds(innermost_parallel_loop, container_of_innermost_parallel_loop, PRIV_VAR);
            }
          //----------------------------
        )





  //now we just move the bounds of the first loop
  move_bounds(result.get[0].get(0),result.get[1].get(0), PRIV_VAR);


}
//-----------------------------------------------------
/*
*This function takes in input one tree and returns the same tree
*but only keeping the loop and the statement nodes

*@param stm - the initial tree
*@return only loop and statement nodes
*/
static void extract_tree(statement stm){

  //here code to extract the desired  nodes

}

//-----------------------------------------------------

/*
*This function is the one applied to the nodes going down on the tree

*@param T- the tree that we process
*@param stm- the initial node of the tree T
*@param result-this array list contains two lists: the list of
*              parallel loops and the list of the their respectives containers
*@return the modified arrayList that we gave as input
*/
static arrayList down(tree T, statement stm, arrayList result  ){
    statement previous=null;
```

```
       statement current=stm;//current is initialized at the
    first stm
       list par_loops=new list();
       list containers =new list();


       /*we iterate on the tree until we reach the last
   level of leaves
       *we take also in consideration the case in which
   previous is null that corresponds
       *at the situation in which we have a loop as first
   node of the tree
       */
       while(current has children OR previous!=null) {
               if(current.type==loop){
      if(current.loop_is_parallel==true){
        par_loops.push(current); //save the parallel loop in
    the list
        containers.push(previous); //save the previus
   statement  in the list
      }
         }
         previous=current;
  }

  /*update the arrayList result, saving the paralle loop in
   the
         *first position and the containers in the second
   position
   */
  result[0]=par_loops;
  result[1]=containers;

  return result;
}

//------------------------------------------------------

/*
*This function is the one applied to the nodes going up on
   the tree.
*It checks if the body of the innermost parallel loops
   contain sequential loops,
*if not it marks the innermost loops as vectorial.
*After that it keeps only the first element in the list "
   par_loops" as parallel
*and it marks all the other parallel loops as sequential.
*
*@param T- the tree that we process
*@param par_loops- list of parallel loops
*@param containers-list of the containers of the loops
 *
*/

static void up (list par_loops, list containers,tree T){
       boolean VECT=true;//this boolean keeps note of the
   fact that the last parallel loop is vectorial
```

```
       statement current= last_children; // I assume that is
    possible to recognize the last level of the tree
       while(current has a father){
               if(current.type==loop){
      if(current.loop_is_sequential==true){
        VECT=false; //i note that the uinnermost paralell
   loop is not vectorial
      }
      elseif(current.loop_is_parallel==true AND current!= P.
   get[0]){
        if(VECT==true){
          current.loop_is_vectiorial=true;
          VECT=false; // i need to clean the variable in
   order to not enter in this if
                        //in the next iteration
        }
        else{
          current.loop_is_sequential=true;
          //I recover the index of the loop and I save it in
    the list of
          //private variables
          concatenate_private(l.getIndexVariable(), PRIV_VAR
   );
        }


      }
               //remove current from the list of parallel loop
   and remove also the corresponding container
      par_loops.remove(current);
      //here with par_loops.getIndex(current) I find out the
    index on the
      //parallel loops of the loop (current) that we are
   considering.
      int index=par_loops.getIndex(current);
      //After that i remove the corresponding container: it
   is in the list "containers"
      //at the place "index"
      containers.remove(containers[index]);
         }
       }
}

//--------------------------------------------------------
/*
*This function saves the bounds of the loop that it takes as
    input in variables declared in
*the statement that is the father of the considered loop
*@param  l- the loop of which we move the bounds
*@param  previous- the statement that contains l
*@param   PRIV_VAR-the list of private variables
*/
static void move_bounds(loop l, statement previous,PRIV_VAR)
   {

    //I assume that there is a function able to create a new
    variable never used before in the program

    //The function addDeclaration adds the declaration of a
    new variable, passed as input, to the statement 'previous
    '
    //passed as input to the function move_bounds
    previous.addDeclaration(int var1=new variable());
    previous.addDeclaration(int var2=new variable());


    //The function addInstructions adds the instruction,
    passed as input, to the statement 'previous'
    //passed as input to the function move_bounds

    previous.addInstructions(var1=l.get_lower_bound());
    previous.addInstructions(var2=l.get_upper_bound());

    //change the bounds into the loop
    l.get_lower_bound()="var1";
    l.get_upper_bound()="var2";

    //add the generated variables to the list of private
    variables
    concatenate_private("var1", PRIV_VAR);
    concatenate_private("var2", PRIV_VAR);
}
//-------------------------------------------------------
/*
*This function add the variable var to the list of private
    variable
*@param   var-variable to add to the list
*@param PRIV_VAR-list of private variables
*/

static void concatenate_private(String var, list PRIV_VAR){
   //here concatenate the string that we pass as first
    argument in the function to the second one that is the
    list of private variables
}
```

Appendix C

Minimum and maximum function script

```c
#define min(a,b) ((a<b)?a:b)
#define max(a,b) ((a>b)?a:b)
#define pips_min_2(a,b) min(a,b)
#define pips_max_2(a,b) max(a,b)
#define pips_min_3(a,b,c) min(pips_min_2(a,b),c)
#define pips_max_3(a,b,c) max(pips_max_2(a,b),c)
#define pips_min_4(a,b,c,d) min(pips_min_3(a,b,c),d)
#define pips_max_4(a,b,c,d) max(pips_max_3(a,b,c),d)
#define pips_min_5(a,b,c,d,e ) min( pips_min_4(a,b,c,d),e)
#define pips_max_5(a,b,c,d,e) max(pips_max_4(a,b,c,d),e)
#define pips_min_6(a,b,c,d,e,f) min( pips_min_5(a,b,c,d,e ),f)
#define pips_max_6(a,b,c,d,e,f) max( pips_max_5(a,b,c,d,e),f)
#define pips_min_7(a,b,c,d,e,f,g) min(pips_min_6(a,b,c,d,e,f),g)
#define pips_max_7(a,b,c,d,e,f,g) max(pips_max_6(a,b,c,d,e,f),g)
#define pips_min_8(a,b,c,d,e,f,g,h) min(pips_min_7(a,b,c,d,e,f,g),h)
#define pips_max_8(a,b,c,d,e,f,g,h) max(pips_max_7(a,b,c,d,e,f,g),h)
#define pips_min_9(a,b,c,d,e,f,g,h,i) min(pips_min_8(a,b,c,d,e,f,g,h),i)
#define pips_max_9(a,b,c,d,e,f,g,h,i) max(pips_max_8(a,b,c,d,e,f,g,h),i)
#define pips_min_10(a,b,c,d,e,f,g,h,i, j) min( pips_min_9(a,b,c,d,e,f,g,h,i),j)
#define pips_max_10(a,b,c,d,e,f,g,h,i, j) max( pips_max_9(a,b,c,d,e,f,g,h,i),j)
```