Synergy between Compiler Optimizations and Partitioning on the Cell processor

Ramon Bertran[*,1], John Cavazos[†,2], Marisa Gil[*,1], Nacho Navarro[*,1], Mike O'Boyle[†,2]

[*] *Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Jordi Girona, 1-3. (Mòduls D6 i C6), 08034 Barcelona, Spain*
[†] *School of Informatics, Institute for Computing Systems Architecture, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland*

ABSTRACT

The Cell Broadband Engine provides to developers a high computational power by using the synergistic processing elements. In order to take advantage of them, the programs are partitioned manually or using user-guided automatic approaches. Consequently, developers are in charge of selecting the best partitioning scheme. Additionally, compiler optimizations are also crucial for achieving the optimal performance, i.e. exploiting data-level parallelism (SIMD).

The aim of this work is to explore the synergy between the different partitioning schemes and the compiler optimizations.

KEYWORDS: cell processor, compiler optimizations, partitioning and programming models.

1 Introduction

The Cell Broadband Engine (referred to thereafter as Cell)[cel05] includes a Power-pc processor and eight attached streaming processors (SPE's). In order to take advantage of the computational power provided, applications are partitioned spreading the computation on the processing units. Implementing optimal algorithms for this architecture from scratch is a time-consuming and error-prone task. Therefore, this approach is not suitable for non-expert programmers because they have to deal with low-level data such as branch hints, prefeching, profiling, DMA transfers, intrinsics and alignment issues.

[1]E-mail: {rbertran,nacho,marisa}@ac.upc.edu
[2]E-mail: {jcavazos,mob}@inf.ed.ac.uk

In order to hide the complexity of the partitioning problem developers use semi-automatic approaches provided by tools such as IBM Octopiler[Eic05], Rapidmind[MD06] or CellSS[BPBL06]. Although these approaches hide the data transfer (DMA) and alignment issues, they still rely on user directives to partition algorithms. Consequently, the user still has to explore the different partitioning schemes to achieve the optimal performance.

Additionally, compiler optimizations also have a significant role due to the Cell architecture characteristics. Compiler optimizations such as auto-vectorizing (SIMD), prefetching, branch hinting or instruction scheduling have significant impact on performance, specially on SPE's.

As the applicability of these optimizations depend on the partitioning scheme, the aim of this project is to study the synergy between them. This will be done by exhaustively searching the exploration space composed by a set of compiler flags and a set partioning scheme of each benchmark.

2 Cell Architecture: SPE characteristics overview

The Cell BE processor is composed of a 64-bit multi-threaded PowerPC processor element (PPE) and eight synergistic processor elements (SPE's). The most important characteristics to take into account when generating code for the SPE are: *(1)* Most of the SPE's instructions are SIMD using 16 bytes registers, including all memory instructions; *(2)* The SPE's have no dynamic branch prediction at all, although they provide a special branch hint instruction; *(3)* Each SPE have their own 256Kb fast local store (LS). These LS's are not coherent with the PPE main memory; and finally, *(4)* The SPE's support dual issuing of independent instructions on two pipelines: the even pipe and the odd pipe. The even pipe instructions must be at an even-word address and the odd pipe instructions must be at odd-word address. Additionally, each instruction have a preferred pipe for execution.

3 Code and Data Partitioning for the Cell

The main issue that developers must solve when porting applications to the Cell is to decide the optimal code and data layout. The approach for getting the best results is to manually analyse algorithms and use low-level libraries and intrinsics provided with the Cell SDK[PS06], such as the SIMD math library or the MASS library.

Another approach is to use specialized libraries and run-times such as ALF[PS06] or RapidMind. These solutions offer programmers an interface to partition data across a set of parallel processes easily without having to write architecturally dependent code. As an example, provided features are: data transfer management, parallel task management, double buffering, and data partitioning and SIMD operations.

The Cell SuperScalar framework (CellSS), which is based in a source to source compiler and a runtime library, is another solution to solve the partitioning by exploiting parallelism. The only requirement for programmers is to provide annotations before pieces of code to indicate that this part must be executed on the SPE's.

4 The importance of Compiler optimizations

The compiler optimization can play a significant role by taking into account the architectural characteristics summarized in Section 2.

Due to the fact that most of the instructions are SIMD on the SPE's units, scalar code is not adequate for the SPE's. As example, scalar store memory operations should be adapted to be a read, modify and write operations as stores necessarily store 16 bytes of data at once. Consequently, auto-vectorization techniques are important for SPE's codes.

Another important set of optimizations for SPE's codes are the loop optimizations. On one hand, they help the compiler to vectorize code. And on the other hand, they reduce the number of branch instructions, which are expensive when are miss-predicted. Additionally, the compiler can be aggressive introducing branch hint instructions for reducing the negative effect of not having a dynamic branch predictor. The combination of these optimizations can also impact the performance. However, loop-unrolling should take into account the local store size constraint.

Finally, instruction scheduling is another quite important optimization because each instruction have a preferred execution pipeline (even or odd). The scheduling algorithm must be aggressive introducing nop's for modifying code layout and reducing pipeline stalls.

5 Looking for synergy

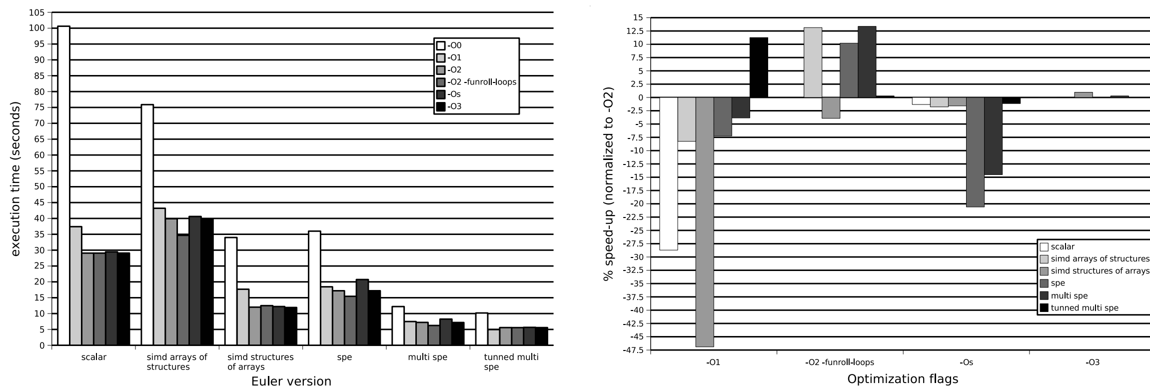
After having studied the principal characteristics for programming the Cell, we are exploring how affect the partitioning scheme on the compiler optimizations. At the end, we will detect what partitioning schemes introduce more barriers to the compiler optimizations and what combination of partitioning scheme and optimizations are the most suitable for the Cell. We expect this study will be useful for improving partitioning approaches by taking into account compiler optimizations. Additionally, this study is the preliminary work to design iterative and machine-learning compilation techniques for the Cell afterwards.

The starting exploration space is bi-dimensional, just combining different compiler flags with different partition schemes of each benchmarks. To reduce the search space, we only use flags related to the characteristics commented on Section 4. Afterwards we will extend the exploration space by adding more dimension such as: number of SPE's used, applying different set of flags on each partition or using different compilers (xlc vs gcc).

6 Case Study: Euler

As a preliminary experiment, we used the toy particle system simulation using Euler integration algorithm provided with the Cell SDK. Six versions of the algorithm are provided (see [PS06] for details). We compiled the versions using then GNU gcc 4.1.1 compiler with a basic set of optimization flags. Then, we executed each version 30 times on a PlayStation3 to retrieve the average execution time.

The left chart on the next page, shows that selecting a good partitioning scheme can improve the execution by at least one order of magnitude. On the right chart, which shows the percentage of speed-up respect to the -O2 execution time, we realize that the flags have different impact depending the scheme used. For example, the -O1 flag produces 11% faster



code than -O2 on tunned version, because the compiler optimizations undo the hand-made tunning. Moreover, the -Os perform worse, specially on the SPE codes, due to alignment issues and the no introduction of nop instructions by the instruction scheduler. Finally, we can see that -floop-unrolling flag can improve algorithm performance, specially on non hand-tunned SPE codes.

7 Future work

Future work include the collection of more benchmarks using different partitioning schemes and the realization of an exhaustive search with more fine grained compilation flags. Further extensions of these work can be to extend the exploration space i.e. applying different compilation flags on each algorithm partition. The conclusions of these study will let us to know which code characteristics are relevant for designing iterative compilation and machine learning techniques for the Cell in the future.

References

[BPBL06] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Memory—cellss: a programming model for the cell be architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM Press.

[cel05] *The Cell Processor Architecture*, Washington, DC, USA, 2005. IEEE Computer Society.

[Eic05] A. Eichenberger. Optimizing compiler for the cell processor, 2005.

[MD06] Michael D. McCool and Bruce D'Amora. M08—programming using rapidmind on the cell be. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 222, New York, NY, USA, 2006. ACM Press.

[PS06] Michael P Perrone and Tanaz Sowadagar. Cell be—cell be software programming and toolkits. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 10, New York, NY, USA, 2006. ACM Press.

Synergy between Compiler Optimizations and Partitioning on the Cell Processor

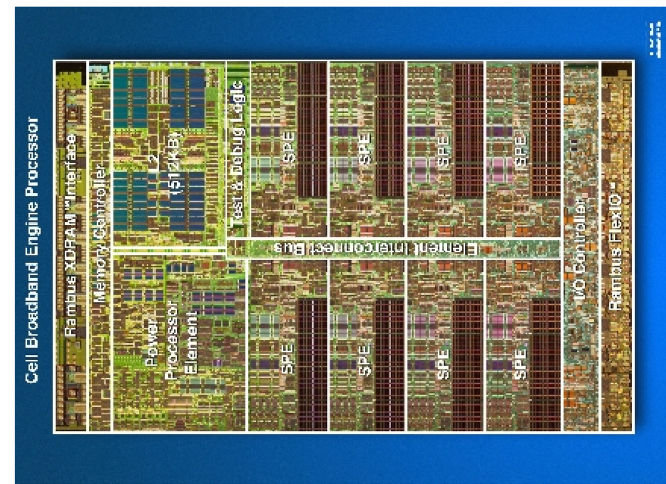
HiPEAC COMPILATION ARCHITECTURE

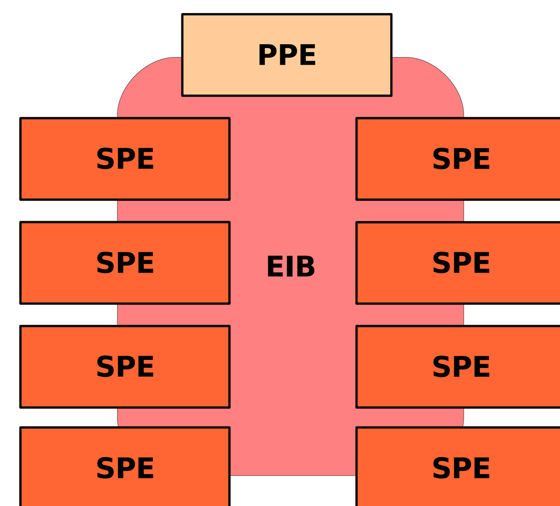
Ramon Bertran   John Cavazos   Marisa Gil   Nacho Navarro   Mike O'Boyle

UPC Departament d'Arquitectura de Computadors
UNIVERSITAT POLITÈCNICA DE CATALUNYA
Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh

Cell Architecture Overview

- Cell Broadband Engine (Cell)
  - 1 Power Processing Element
  - 8 Synergistic Processing Element
  - 1 Element Interconnection Bus
  - 512Kb L2 Cache
- Power Processing Element (PPE)
  - Power 970 architecture compliant core (64 bits)
  - Two-way SMT
  - 32Kb L1 Instruction and Data Cache
  - Altivec Extension
- Synergistic Processing Element (SPE)
  - Synergistic Processing Unit (SPU)
    - RISC processor
    - 128-bit SIMD Organization
  - Memory Flow Controller (MFC)
    - DMA Controller
  - 256Kb Local Store
  - Not dynamic branch predictor
  - Large register file (128 registers)
- Element Interconnection Bus (EIB)
  - Connects PPE,SPEs and Memory
  - 4 Channel Ring structure
  - 3 concurrent transactions per channel





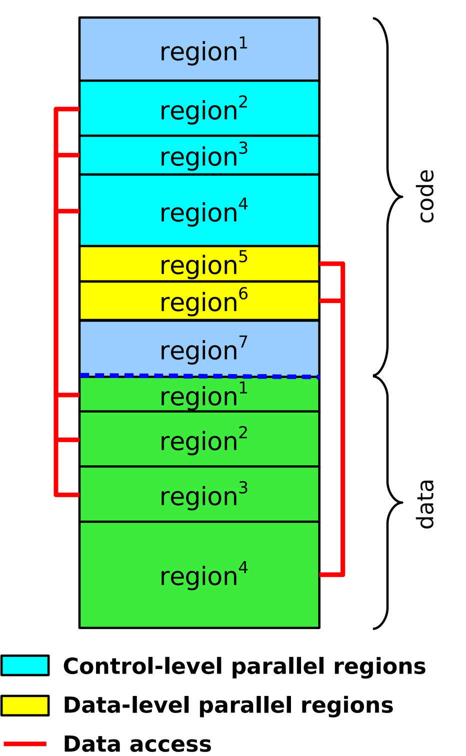
The Importance of Compiler Optimizations

- The compiler still has a significant role
  - specially due to the special architecture characteristics of the SPEs
  - and due to the overhead associated to the separate compilation approach
- We study the following compiler optimizations:
  - Auto-vectorization
    - Take profit of the Altivec unit of the PPE
    - Take profit of the SPE units
      - Reducing the amount of scalar code which is not suitable for SPE's units due to:
        - All memory operations are 16B long
        - No dynamic branch predictor
  - Loop Optimizations
    - For reducing the number of branch instructions
    - Take profit of the large register file
    - Increase the opportunities for auto-vectorizers
    - i.e. loop unrolling, loop data arrays pre-fetching, move loop invariants, unswitch loops
  - Branch hints
    - Aggressive introduction of branch hint instructions to improve the static branch predictor results
      - Use of profiling information
      - Use of user feed-back information
  - Instruction scheduling
    - The SPE are in-order dual pipeline co-processors
    - Each instruction has a preferred execution pipeline
      - Introduction of NOPs to reduced pipeline stalls and penalties due to execution on non preferred pipeline

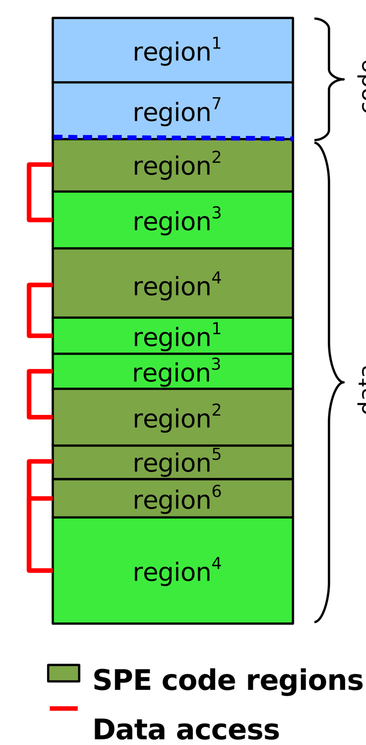
Code and Data Partitioning for the Cell

- Applications have to be ported for exploiting:
  - Data level parallelism
    - i.e. Use of SIMD units
  - Control level parallelism
    - i.e. Streaming applications
- Several approaches for exploiting parallelism:
  - Rewrite the applications from scratch
    - Time-consuming
    - Error-prone
    - Not-suitable for non-expert developers
    - Best results by using low-level intrinsics
  - Use of specialized programming models and compilers that hide the architectural complexity
    - Easy-porting
    - Hide low-level issues
      - Alignment constraints
      - DMA transfers
    - Still rely on user directives for partitioning applications
    - i.e. CellSS, RapidMind, Octopiler
- Selecting the optimal partitioning scheme is time-consuming
  - Need to implement similar approaches and evaluate the performance results
  - The code is split in several parts and then it is embedded in a fat binary
    - Separate compilation approach

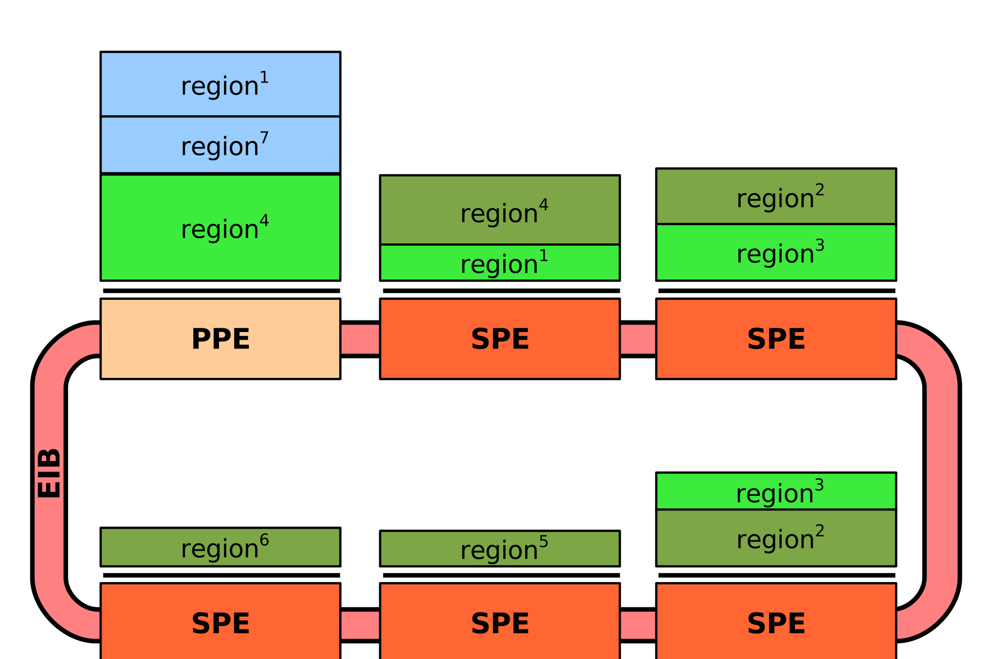
Original Scalar Application



Partitioned Application (Static View)



Partitioned Application (Dynamic View)



Motivation: Looking for Synergy

Search on the exploration space, using different partitioning schemes, different set of compiler optimizations and different compilers to look for special behaviors and identifying the synergies between compiler optimizations, partitioning and programming model used.

- Exploration space:
  - Application Partitioning scheme
  - Compiler Optimizations
  - Programming model
  - Compiler platform

Case Study: 6 versions of the Euler algorithm

- Euler: A toy particle system simulation using Euler integration.
  - Five versions of the same algorithm

1 - Euler scalar:
- Scalar version of the algorithm
- Runs only using the PPE main core

2 - Euler SIMD arrays of structures
- SIMDize the code (vectorize)
  - Use of the Altivec extension on the PPU core
- Particle system data organized as arrays of structures

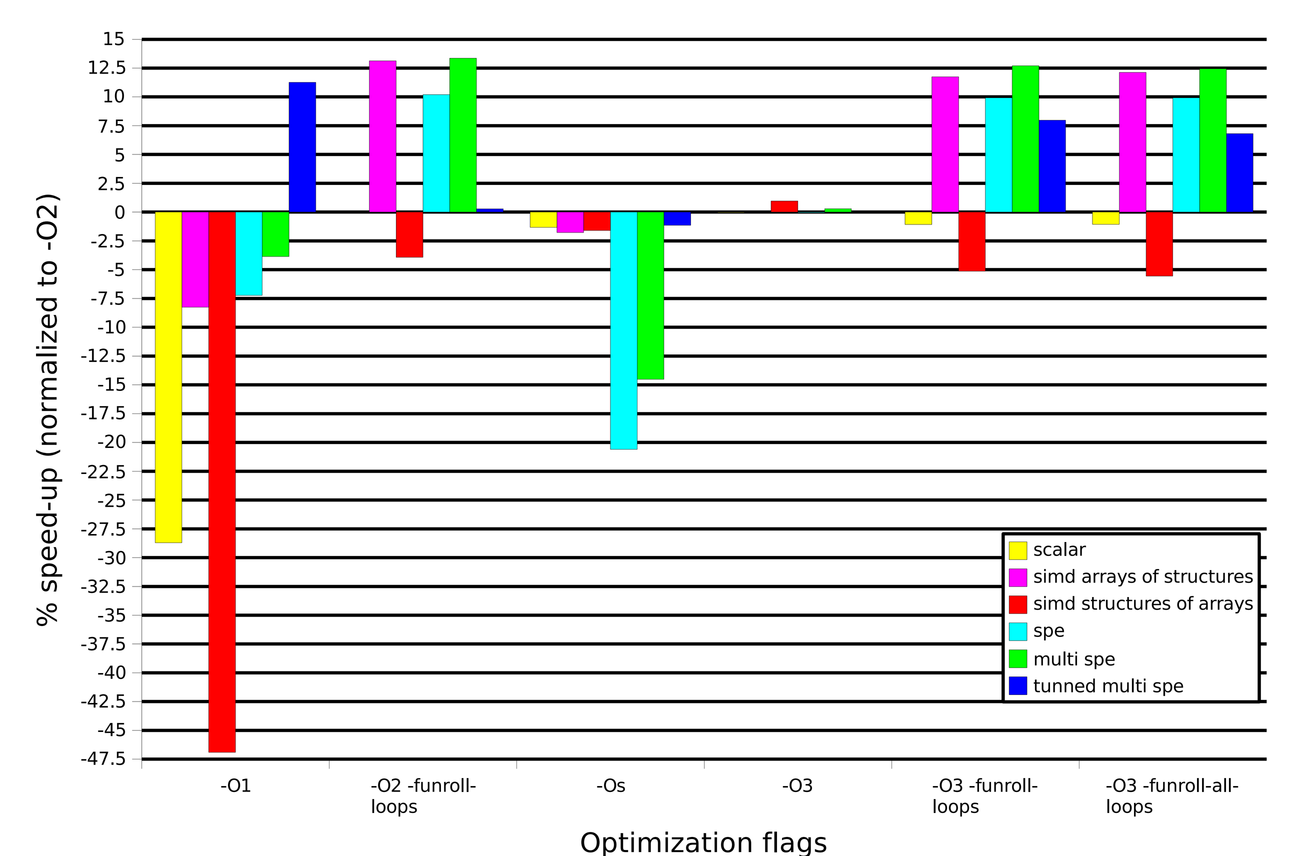
3 - Euler SIMD structures of arrays
- SIMDize the code (vectorize)
  - Use of the Altivec extension on the PPU core
- Particle system data organized as structures of arrays

4 - Euler SPE
- Port the code for execution on the SPE
  - Create a control structure for defining the computation on the SPE
  - Introduction of SPE intrinsics
    - Most are direct translation of SIMD vector instructions
  - Add an additional looping construct to partition the data array because all the data will not fit within the SPEs -local store
  - Add DMAs to move data in and out of the SPUs local store

6 - Euler multi SPE
- Parallelize the code across multiple SPEs
  - Data partitioning problem: easy due to the lack of dependencies

7 - Euler multi SPE tunned
- Optimize and tune for performance
  - Loop unroll
  - Remove scalar memory operations
  - Multi-buffering for interleave DMAs with the computation

Benchmark + Optimization Level	Time *(seconds)*
Euler multi spe tunned -O1	4.96
Euler multi spe tunned -O3 -funroll-loops	5.14
Euler multi spe tunned -O3 -funroll-all-loops	5.2
Euler multi spe tunned -O2 -funroll-loops	5.57
Euler multi spe tunned -O3	5.58
Euler multi spe tunned -O2	5.59
Euler multi spe tunned -Os	5.65
Euler multi spe -O2 -funroll-loops	6.25
Euler multi spe -O3 -funroll-loops	6.3
Euler multi spe -O3 -funroll-all-loops	6.31
Euler multi spe -O3	7.19
Euler multi spe -O2	7.21
Euler multi spe -O1	7.49
Euler multi spe -Os	8.26
Euler multi spe tunned -O0	10.19
Euler SIMD soa -O3	11.92
Euler SIMD soa -O2	12.03
Euler multi spe -O0	12.21
Euler SIMD soa -Os	12.22
Euler SIMD soa -O2 -funroll-loops	12.51
Euler SIMD soa -O3 -funroll-loops	12.65
Euler SIMD soa -O3 -funroll-all-loops	12.7
Euler spe -O2 -funroll-loops	15.44
Euler spe -O3 -funroll-loops	15.49
Euler spe -O3 -funroll-all-loops	15.49
Euler spe -O2	17.2
Euler spe -O3	17.21
Euler SIMD soa -O1	17.68
Euler spe -O1	18.44
Euler spe -Os	20.74
Euler -O2 -funroll-loops	29.06
Euler -O2	29.07
Euler -O3	29.08
Euler -O3 -funroll-all-loops	29.38
Euler -O3 -funroll-loops	29.38
Euler -Os	29.45
Euler SIMD soa -O0	33.99
Euler SIMD aos -O2 -funroll-loops	34.67
Euler SIMD aos -O3 -funroll-all-loops	35.07
Euler SIMD aos -O3 -funroll-loops	35.22
Euler spe -O0	35.99
Euler -O1	37.41
Euler SIMD aos -O3	39.9
Euler SIMD aos -O2	39.91
Euler SIMD aos -Os	40.62
Euler SIMD aos -O1	43.2
Euler SIMD aos -O0	75.89
Euler -O0	100.62



- Exploration space
  - Compiler: gcc
  - 6 versions of the Euler algorithm
  - Set of flags: -O0, -O1, -O2, -O2 -funroll-loops, -Os, -O3, -O3 -funroll-loops, -O3 -funroll-all-loops
- Preliminary results show that:
  - The partition scheme is the most important factor to improve application performance
    - Improving the application performance by at least one order of magnitude
  - The compiler optimizations still have a significant role
    - Achieving an average of 2x speed-up