Characterization of HPC applications for ARM SIMD instructions

Víctor Soria Pardos

Director: Adrià Armejach (BSC-CNS)

Co-director: Darío Suárez (UNIZAR)

Responsible: Miquel Moretó (DAC)

Degree in Informatics: Computation
facultat d'informàtica de barcelona (fib)

Universitat Politècnica de Catalunya
(UPC) BarcelonaTech

July 2019

Abstract

Nowadays, most Intruction Set Architectures (ISA) include Single Instructions that process Multiple Data (SIMD) to speed up High Performance Computing (HPC) applications. The first part of this work aims to characterize HPC applications optimized using the NEON extension, which is the actual SIMD extension supported by ARMv8 processors. For this purpose, we have two high-end ARMv8 processors, ThunderX and ThunderX2, and two mainstream comercial ARMv8 compilers, GCC and Arm HPC Compiler. With this set up we have characterized a collection of benchmarks extracted from RAJAPerf, HACCKernels and HPCG benchmarks. The characterization includes experimental work in order to obtain speed-up, scalability, energy efficiency and power efficiency measurements for all benchmarks. Moreover, we have taken a look into the assembly code to identify what optimizations are used by each compiler that makes benchmarks run faster or slower.

The second part of this work focuses on the novel Scalable Vector Extension (SVE) specified in the ARMv8.2 ISA. This SIMD specification introduces a Vector-Length Agnostic programming model, which enables implementation choices for vector lengths that scale from 128 to 2048 bits. To this day, no real processor implements this new ISA, therefore we have used the Arm Instruction Emulator (ArmIE), an emulation tool developed by Arm, that allows the execution of SVE compiled binaries running in an ARMv8 processor. Our work analizes how compilers that support SVE (GCC and Arm HPC Compiler) vectorize the benchmarks and what is the quality of the generated assembly code. We also propose some low level optimizations to improve code generation.

Resum

Avui en dia, la majoria dels conjunts d'instruccions (ISA) inclouen instruccions que processen múltiples dades en una única instrucció. Aquestes instruccions s'utilitzen per accelerar aplicacions d'alt rendiment (HPC). La primera part d'aquest treball busca caracteritzar aplicacions HPC que han estat optimitzades utilitzant NEON, que és l'actual subconjunt d'instruccions vectorials suportat pels processadors basats en l'ISA ARMv8. Per aconseguir aquest objectiu tenim a la nostra disposició dos processadors de alta gamma basats en ARMv8, que són ThunderX i ThunderX2, i dos dels principals compiladors del mercat, GCC i Arm HPC Compiler. Amb aquests hem caracteritzat una col·lecció de benchmarks obtinguts del conjunt de benchmarks RAJAPerf i les aplicacions HACCKernels i HPCG. Aquesta caracterització inclou experiments per obtenir la millora de rendiment, l'escalabilitat, l'eficiència energètica i el consum de potència. A més, hem analitzat el codi assemblador per tal d'identificar quines optimitzacions s'han fet i quines caracterıstiques fan que uns experiments siguin més ràpids que altres.

La segona part d'aquest treball es centra en la nova extensió vectorial escalable (SVE) de Arm, la qual està especificada en la ISA ARMv8.2. Aquesta especificació introdueix el model de programació independent de la longitud dels registres vectorials (VLA). Aquest model permet que els fabricants de processadors puguin triar diferents longituds de vectors entre 128 i 2048 bits par la implementació de las seves micro-arquitectures. A dia d'avui, no existeix cap processador que implementi aquest nou repertori d'instruccions, per tant hem estat forçats a utilitzar una eina de emulació (ArmIE) desenvolupada per Arm. Aquesta eina ens permet executar binaris compilats amb suport per SVE en processadors de la ISA ARMv8. En aquest treball s'analitza com els compiladors GCC y Arm HPC Compiler vectoritzen aquestes benchmarks, i a més es proposa una optimització de baix nivell per tal de millorar la generació de codi.

Resumen

Hoy en día, la mayoría de repertorios de instrucciones (ISA) incluyen instrucciones que procesan multiples datos en una única instruccion. Éstas instrucciones se utilizan para acelerar aplicaciones de alto rendimiento (HPC). La primera parte de este trabajo busca caracterizar aplicaciones HPC que han sido optimizadas utilizando NEON, que es el actual subcojunto de instrucciones vectoriales soportado por los procesadores basados en la ISA ARMv8. Para alcanzar este objetivo tenemos a nuestra disposición dos procesadores tope de gama basados en ARMv8, que son ThunderX y ThunderX2, y dos de los principales compiladores del mercado, GCC y Arm HPC Compiler. Con ellos hemos caracterizado una colección de benchmarks extraidos del conjunto de benchmarks RAJAPerf y las aplicaciones HACCKernels y HPCG. Esta caracterización incluye una serie de experimentos que buscan calcular el speed-up, la escalabilidad, la eficiencia energética y de consumo de potencia. Además, hemos analizado el código ensamblador para identificar que optimiaciones se han llevado a cabo y qué caracteristicas hacen que unos experimentos sean más rápidos que otros.

La segunda parte de este trabajo se centra en la nueva extensión vectorial escalable (SVE) de Arm, la cual está especificada en la ISA ARMv8.2. Esta especificación introduce el modelo de programación independiente de la longitud de los registros vectoriales (VLA). La cual permite que los fabricantes de procesadores puedan elegir diferentes longitudes de vectores entre 128 y 2048 bits, para la implementación de sus microarquitecturas. A día de hoy, no existe ninguna máquina que implementa este nuevo repertorio de instrucciones, por lo tanto hemos tenido que usar una herramienta de emulación (ArmIE) desarrollada por Arm. Esta herramienta nos permite ejecutar binarios compilados con soporte para SVE en procesadores de la ISA ARMv8. Nuestro trabajo analiza cómo los compiladores GCC y Arm HPC Compiler vectorizan estos benchmarks y además propone ciertas optimizaciones de bajo nivel para mejorar la generación de código.

Preface

Dear reader, the document you are reading right now is my final bachelor thesis that concludes my four years degree in informatics engineering at Univeristy of Zaragoza, which I have developed and defended at the Polytechnic University of Catalonia. This project has been developed with the colaboration of the Mont-Blanc 2020 project.

I would like to thank Adrià Armejach, Darío Suárez and Miquel Moretó, for their dedication and support. Without them this project would not have been possible. My gratitude also goes to my office partners for the support, answering all my questions. I would finally like to dedicate this work to my parents and friends, whom have encouraged me during this long path.

Contents

Chapter 1

Context and Scope of the Project

This chapter analyzes HPC research lines on Single Instruction, Multiple Data architectures focusing in ARM-based proposals. Then, it introduces problem formulation, project objectives and principal stakeholders. Finally, describes state-of-the-art proposals, project scope and development practices.

1.1 Context

Nowadays, one of the most used methods of speeding up HPC (High Performance Computing) and machine learning applications, is through SIMD (Single Instruction, Multiple Data) [1]. These instructions increase data parallelism, since the same operation is executed on different data points at the same time. This approach dominated supercomputing during 80's, until they were surpassed by inexpensive MIMD approaches (Multiple Instructions, Multiple Data), where different processors compute different instructions to different data [2]. Modern supercomputers architectures are dominated by clusters of MIMD computers, each of them implementing SIMD.

A current trend in the server & HPC segments is the popularization of Arm's (Advanced RISC Machines) Instruction Set Architecture (ISA). The ARM ISA has become with the years the most used and the most produced in quantity terms [3]. One of the key factors for the adoption of ARM ISA is its power efficient RISC chips, which is a key factor in the mobile market segment. If we classify the ISAs according to their complexity, we usually distinguish between RISC and CISC. Reduced

Instruction Set Computing (RISC) is a family of processors architectures that have a small set of general purpose instructions, while on the other extreme of the design space, instructions in a Complex Instruction Set Computing (CISC) machine can execute several low level operations. In fact most current CISC processors internally execute RISC micro-operations.

On one hand, RISC cores are well known for addressing embedded systems and battery powered platforms where power efficiency is critical, due to their power efficient design they consume less for the same task. On the other hand CISC cores have been dominating supercomputing and server markets, because of high performance designs. However, with the years, these differences have been reducing and Arm has focused towards HPC and server systems introducing its SIMD extensions (NEON and SVE), while Intel has introduced its new mobile chips based on the Atom product like, for smartphones [4].

In this context, the Mont-Blanc project [6] was born to develop a supercomputer based on mobile market technology; in order to take advantage of the higher manufacturing volumes, faster design cycles and better economic factors [5]. To achieve this, Mont-Blanc focuses on ARM-based processors. An initial prototype was based on the Samsung Exynos 5 Dual processor, and in 2017 Mont-Blanc announced a new prototype based on Marvell's ThunderX2 CPUs [7].

1.2 Problem Formulation

This project aims to characterize SIMD generated code of the two main ARM ISA compilers: GCC (GNU Compiler Collection) and the Arm HPC compiler. We cover two different SIMD extensions: NEON and the Scalable Vector Extension (SVE), for the ARM 64 bits architecture ARMv8.2-A.

The NEON extension was first introduced in the ARMv7-A specification. With a vector length of 128-bit, its instructions can perform two operations of double precision floating point or 4 operations of single precision floating point [11]. SVE was first introduced within ARMv8.2-A in 2016. This extension is a novel approach for SIMD instruction specifications. It allows developers to generate and optimize SIMD code and forget about the vector length. In other words the ISA is vector

length agnostic and enables SVE binaries to run on machines with different vector lengths without recompilation. Apart from the vector length agnostic paradigm, SVE supports gather and scatter instructions that allow SIMD instructions to operate on non contiguous data [12]. Nowadays, there is no machine that implements the SVE extension, and it is planned that Fujitsu will deploy the first in 2021.

To perform the characterization we use a set of HPC and Machine Learning benchmarks, namely: RAJAPerf loops [8], HACCK [9] and HPCG [10]. These benchmarks will be compiled and executed for using the OpenMP programming model. A benchmark is a program which the main goal is to quantify the performance of a computer and to be able to compare it with other computers. This is achieved by executing the same binary under the same conditions on different computers and analyzing the metrics obtained from these executions. Metrics help analysts quantify and characterize program's behaviour, e.g. GFlops and speed-up are two well known metrics that measure a program performance [25].

The selected benchmarks employ OpenMP to parallelize execution. OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs. OpenMP is used to parallelize benchmarks, via shared memory multiprocessing programming. It offers an easy way to parallelize code in order to scale application at the node level, and it is widely used in supercomputers.

Each benchmark will be executed mainly in the two prototype clusters from the Mont-Blanc project that are called ThunderX and ThunderX2. The two clusters are based on ARM 64 bit processor, that are manufactured by Cavium (Marvell Technology Group). One is based on a ThunderX model with 48 in-order cores and the other a ThunderX2 with 32 out-of-order cores. Both processors use a custom implementation of ARM 64 bit cores, focusing its features for enterprise server solutions and HPC workloads.

Eventually, we will make some experiments in a third machine based on Intel's Skylake architecture. These experiment will help us to contrast the results obtained in both Cavium machines.

1.3 Objectives

Once we have the statement of the problem, we can focus on the different objectives this project aims to achieve. These can be summarized as:

1. Implement and maintain a work space for experiments.
2. Develop an evaluation methodology and the necessary infrastructure to conduct experiments for the research work. This infrastructure can be used in future experiments and by other researches.
3. Measure performance and power consumption differences between GCC and ARM-HPC compilers, between SIMD and no SIMD assembly code and between ThunderX and ThunderX2 nodes in machine learning and HPC benchmarks.
4. Check correctness of assembly code generated by both compilers for the new vector extension SVE.
5. Analyze how each compiler optimizes the same code. And seek for specific compiler options that optimize concrete applications.

1.4 Stakeholders

1.4.1 Developer and Research Intern

The developer will be responsible of all the technical part. From modifying the benchmarks code to be suitable for experiments to implement an automatized pipeline for the execution of the experiments. But the most important task for the researcher is to analyze and document the results.

1.4.2 Director, Co-director and Ponent

The director of this project is Adrià Armejach, he is senior researcher at Barcelona Supercomputer Center (BSC-CNS). The co-director of the project is Darío Suárez, assistant professor at Universidad de Zaragoza. And the responsible of the project is Miquel Moretó, Ramon y Cajal

research fellow at UPC and senior researcher at BSC. Their roles will be essential for the correct development of the project and their knowledge will be key for the analysis of the results.

1.4.3 Scientific Community

The results published after this work can be used by the scientific community in future research. The direct beneficiary will be the BSC. The usage of novel technologies like SVE and the Arm HPC compiler is still in its early stages compared to other technologies, so the results of this project may help developers in the future.

1.5 State-of-the-art

As mentioned before, the ARM ISA has seen a steep growth in popularity. This has lead academia to start researching ARM based solutions for HPC.

1.5.1 ThunderX and ThunderX2

On 2014, Cavium announced ThunderX, its first 64 bit multi-core ARM server microprocessor. Relying on the ARMv8 ISA, this processor was able to integrate 48 in-order cores. On 2016, Cavium announced its new server generation, ThunderX2. As well as ThunderX, is a 64 bit multi-core ARM server microprocessor, but with a radical new architecture based on 32 out-of-order cores implementing the ARMv8.1 ISA. Both processors are suitable for vector architecture experiments since they support NEON and have high memory bandwidth. Moreover, both processors have been used on several academic articles [13] [14].

1.5.2 The Mont-Blanc European Project

Since 2011, the Mont-Blanc European project has encouraged its vision of an exascale supercomputer based on technology coming from the mobile market segment. The last Mont-Blanc prototype is called Dibona, which is based on ThunderX2 processors. The whole system is suited for HPC workloads, like simulation of geophysics, fusion, materials, particle physics, life sciences, combustion, aerodynamics, weather forecast,

computational fluid dynamics, biophysics, and a large etc. As consequence of project researching, numerous reports have been published in the past years. During stage three of the Mont-Blanc project, the team chose a set of HPC applications for evaluating their performance in ARM-based architectures. The benchmarks were ported and tuned for ARM mini-cluster prototypes, that includes ThunderX processor [15]. After that, Dibona behaviour was studied using a heterogeneous set of benchmark compiled with a collection of commonly used compilers. These benchmark were chosen in order to reach the throughput and memory bandwidth peaks of ThunderX2. In those experiments the main objective instead of comparing possible ARM ISA compilers, was chose the best generated code that could achieve the highest performance. [16]. Another interesting report is [17], that compared GOMP (GCC OpenMP implementation) and IOMP (intel and LLVM OpenMP implementation). Nevertheless, these experiments only focused in OpenMP instead of full compiler capabilities.

While all these works are profoundly related with HPC on ARM based machines, none of them analyzes differences between compiler generated assembly.

1.5.3 LLVM and GCC Comparisons

The Arm HPC compiler is based on LLVM tool chain technology. Since LLVM was released first on 2003 there has been many comparisons between both compilers. Almost all these comparisons are focused on the x86 architecture [18], and only few ones that take a look at ARM assembly are outdated [19]. On these papers GCC usually has a better performance over LLVM while on Mont-Blanc's papers LLVM outperforms GCC, specially when using optimized Arm Performance Libraries.

Arm has also published some posts in its blog, in which they shown some small cases of study. Nevertheless, this experiments come from Arm itself and they don't give enough detailed information [20].

1.5.4 SVE Research

Until the first SVE implementation arrives, developers and the scientific community have only two main tools to port and tune applications for the new ISA. One based on simulation (gem5 [21]) and one based on emulation (ArmIE [22].) Simulations give detailed reports about execution performance, but require a high computational cost. On the contrary, emulators can run application's binaries in less time, but losing all possible information about execution time and performance. The major part of scientific research has focused on simulations in last years. These simulations are essential for the development of future chips, but emulators can obtain useful information, specially about compiler code generation and correctness.

1.6 Scope

The purpose of this section is to frame the contents of this project and to explain what is out of scope, either due to time limitations or divergence over the main objectives.

As said before, this project aims to characterize Arm's Single Instruction Multiple Data ISA extensions on HPC applications. There are two main extensions: NEON and SVE. The focus will be on the NEON extension, which can be evaluated using real machines. We will evaluate performance and power consumption. Since, there is no real hardware implementing SVE yet, its characterization would require the use of simulations tools with a steep learning curve, and a low level of maturity which does not fit with the time duration of this project. However, SVE will be analyzed using an emulator.

Initially, a set of HPC applications will be chosen. This benchmarks have to be suitable for vectorization and have different characteristics in order to cover a broad spectrum of HPC applications. For example, some of them could be compute intensive, that is when the CPU operations define the execution time, or memory intensive, where data movements represent a major part in execution time. The set of applications should take into account actual scientific problems, machine learning or genomics.

For NEON characterization, applications will be instrumented in order

to capture the execution time and hardware counters, that would give the performance information of its execution. Power measurement scripts will be written to catch energy consumption and calculate energy efficiency. All these tasks are a small part in the development of a fully automatized experiment pipeline. This necessity of such a high level of automatization is caused by long execution times and high number of experiments that restricts human intervention.

During the experiments, binaries will be evaluated on three machines, one based on ThunderX processors, other in ThunderX2 processors and the third in Skylake processors. These two latter machines are placed inside Dibona, which is a prototype developed within the Mont-Blanc project and employs ThunderX2 and Skylake processors. With data gathered from experiments, a two way analysis is done. On one side, assembly code generated is analyzed. On the other, performance metrics are calculated so binaries can be compared between each other. With the analysis, some conclusions should be obtained, like what compiler is doing a better job of auto-vectorization for HPC, or the most energy efficient processor and the associated reasoning.

For SVE analysis, a new version of every program is complied allowing SVE vectorization. The correctness of generated code is checked using the ArmIE emulator on the new binaries. Emulation only gives information about which instructions have been executed, but there is no useful information about execution time or cycles executed. As a consequence of performance information absence, both compilers will be compared on their ability to vectorize each loop and on the correctness of the code.

1.7 Development Practices and Validation

As said before, the short time available for finishing the project implies the use of agile and strict work methodologies.

1.7.1 Short Development Cycles

With the use of short cycle methodologies (small tasks defined each week) an accurate control of the project is granted. This methodology makes easy to check if the project is on track and reschedule tasks if necessary.

1.7.2 Tools for Development

To analyze the execution of the benchmarks, specific tools like power measurement scripts, paraver [23] and extrae [24] will be used. All of them are based on hardware counters and were developed at BSC to help developers understand the behavior of parallel applications. Additionally, the progress of the project will be registered using BSC's gitlab infrastructure.

1.7.3 Project Tracking

A detailed tracking on project progress will be achieved by performing weekly meetings with project advisors. These meetings will be documented in their respective records and documentation.

1.7.4 Final Evaluation

Once all the experiments and analysis have been done, we will obtain and redact what are the main differences between compilers. Then, the level of completeness of project is evaluated checking how many of the initial objectives are full filled.

Chapter 2

Project Planning

This chapter is a summary of the initial project management course done in March of 2019. It specifies project duration and individual tasks. Then, it explains project schedule and their associated resources. Finally, it describes an action plan for eventual deviations of project schedule.

2.1 Estimated Project Duration

The estimated project duration was approximately 4 months. The project started on February 18^{Th}, 2019 and the deadline was expected to be on July 3^{rd}, 2019. However, we extended it to July 17^{Th} of 2019.

2.2 Project Tasks

In this section, we explain what are the main project tasks. We explain which steps must be done to accomplish each task.

2.2.1 T1 Project Planning and Feasibility

This task pertains to Project Management Course and includes the next four stages:

I Project scope.

II Project planning.

III Project budget.

IV Initial state of the art.

2.2.2 T2 System Set Up and Benchmark Analysis

The main objective of this task is to make an analysis of possible benchmarks that are suitable for the project and set up the necessary software and tools needed to achieve project objectives.

Benchmarks are selected from Mont-Blanc 2020 project (MB2020) initial prospections. At the moment, this report is confidential until its final version is published. This report searches and analyzes a set of real life applications that exhibit the characteristic demands of Big Data and HPC applications. These programs stress all the segments targeted by MB2020. One of the principal characteristics of these applications its that allow exploring the vector ISA in terms of manual code vectorization, auto-vectorization, and the use of generic performance libraries. Therefore, the benchmarks used in this project are a subset of the MB2020 benchmarks

To develop the project, we need a modern version of GCC (version 8.2.0) and the ARM compiler for HPC (version 19.0) as well as their respective implementations of OpenMP installed, for compiling and executing the parallel versions of the benchmarks. Extrae and paraver are also needed in order to generate traces of the execution and visualize them. These traces are used to understand the behavior of the different benchmarks. All these tools are already installed in the test machines but the developer needs to test and learn how to use them for the project.

The test clusters use as operating system a Linux distribution adapted to ARM, so no learning phase is needed. The executions in the Mont-Blanc clusters are done through the SLURM queue system. SLURM (Simple Linux Utility for Resource Management) is an open-source resource manager designed for Linux clusters of all sizes. Learning how to use SLURM is needed for the project, because SLURM lets the user allocate exclusive access to resources, bind processes to cores, specify shell scripts to run before or after an execution, and automatize experiments.

2.2.3 T3 Compilation and Experiment Development

Benchmark compilation is the central part of the project. Initially all benchmarks are compiled in six different versions: NEON-ArmHPC,

NEON-GCC, SCALAR-ArmHPC, SCALAR-GCC, SVE-ArmHPC and SVE-GCC. Prefix refers to type of vectorization: SCALAR is a non-vectorized binary, NEON uses NEON vectorization and SVE uses SVE vectorization. The suffix refers to compiler used for the compilation. After that step, assembly code must be checked in order to assert that is correct and vectorized.

The first step to perform the experiments is to instrument all benchmarks so we can measure execution time and hardware counter values of the regions of interest. These regions execute the most important part of the benchmark, the part that stresses the processor. At same time shell scripts are written in a way that they launch experiments and capture energy consumption automatically. Another parallel task, is programming a python scripts for data visualization and metrics calculation, that help during analysis.

2.2.4 T4 Performance and Code Analysis

Next step consists in doing a deep analysis with all the information compiled from benchmarks executions. Benchmark bounds are identified and performance is compared between the two compilers and hardware prototypes, i.e. one benchmark may be limited by number of floating point operations done per second (FLOPS) and its vectorized version has an higher performance than scalar version. The best binary version is selected by comparing all binaries within the same workload. Performance differences between performing vectorization and not, are shown for each compiler.

Then, the power and energy consumption of both processors is measured in order to compute the energy and power efficiency. However, during the project development we could not measure this metrics in ThunderX, because the tool used for this purpose was broken. In the place of ThunderX, we finally measured Skylake.

Then assembly code is used to justify all these observations, checking for optimizations performed by the compiler. If each compiler selects a different set of instructions for the same task, they can be compared because both codes are executed on the same hardware.

2.2.5 T5 SVE Compilation and Code Analysis

Last step consist in benchmarking all SVE binaries for the same experiments used in past steps. Then we review the assembly code of the binaries in order to find possible errors, unnecessary parts or possible optimizations. We compare how much vectorization is done by each compiler using SVE. Then we show how SVE vectorization reduces the number of instructions executed when increasing the vector length from 128 bits to 2048 bits, and what weight have these SVE instructions. We also compare SVE to Intel SIMD instructions set in order to compare how well SVE vectorize for different vector lengths.

2.2.6 T6 Final Stage

The final stage consists on preparing the delivery of the project. This includes document all experiments, make the necessary graphs to illustrate the results, make an analysis over the results and draw conclusions. With these elements we build the memory of the bachelor thesis. Finally, we prepare the presentation for the thesis defense.

2.3 Task Dependencies

The first task to be done is project planning (T1). This task is key for the correct organization and development of the project. Tasks T2 and T5 depend on T1. Then Task T3 depends on T2 and T4 depends on T3. Finally, T6 depends on all of them. The final stage must be done once the project is finished.

2.4 Estimated Time

In this section we present a resume with the estimated cost in hours of each task and what has been its real duration.

Stage	Estimated dedication (hours)	Final dedication (hours)
Planning and feasibility	75	62
Benchmark analysis and system set up	50	67
Compilation and experiment development	160	177
Performance and code analysis	75	83
SVE compilation and code analysis	110	122
Final Stage	80	84
Total	550	595

Table 2.1: Estimated and dedicated hours per group of tasks.

2.5 Resources

2.5.1 Hardware Resources

- Laptop PC (with an Intel i7-8550U CPU @ 1.80GHz, 8 of RAM): used as work station.
- Thunder cluster: Cavium ThunderX with 128GB of RAM and 128GB of SSD. Each node has 2 sockets, each with 48 ARMv8 in-order cores.
- ThunderX2 cluster: Marvell ThunderX2 with 256GB of RAM and 3.3TB of disk. Each node has 2 sockets, each with 32 ARMv8.1 out-of-order cores.
- Skylake cluster: Intel Xeon Platinum 8176 Processor with 28 Skylake out-of-order cores.

2.5.2 Software Resources

- GNU Compiler Collection (GCC), version 8.2.0 : for compilation.

- Arm compiler for HPC, version 19.0: for compilation.
- Arm Instruction Emulator, version 19.0: for emulation of SVE instructions.
- Arm Performance Libraries: for compilation of scientific applications.
- CMake: for automatization of compilation and linking.
- Perfmon2: for capturing hardware counters.
- Python: for data parsing.
- Extrae: for obtaining OpenMP and hardware counter traces
- Paraver: for visualizing extrae traces.
- Latexmk: for compiling project documentation.
- PDF viewer: for visualizing project final documentation.
- Matplotlib: for building graphs.
- ICC: for compilation.
- Libcount: for counting instructions with ArmIE.

2.5.3 Human Resources

The development of this project involves Adrià Armejach, Darío Suárez and Miquel Moretó as advisors and Víctor Soria as main developer and research intern of the project.

2.6 Gantt chart

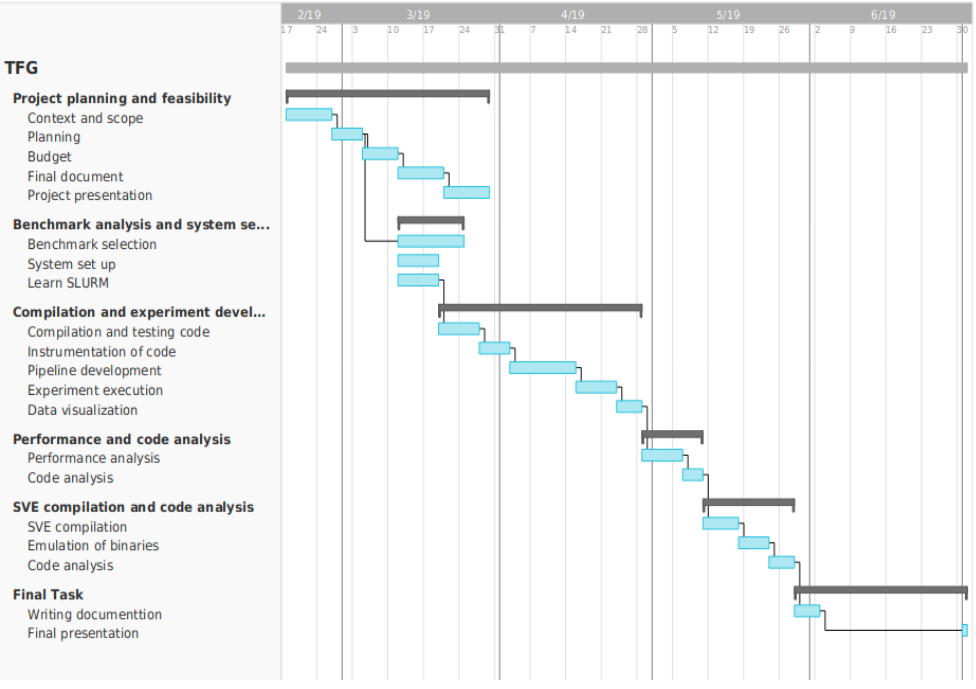


Figure 2.1: Gantt chart of project schedule

2.7 Action Plan

The Gantt chart listed above in Figure 2.1 depicts the initial project plan to achieve all the objectives, but in most projects, it is hard to follow planning as initially thought, and in this case we have had to delay the project duration. This was caused by an underestimation of the task duration, but in other cases can be due to unexpected problems. In these cases, the project team has to find a solution in order to minimize the negative impact over the original task scheduling.

This bachelor thesis must be delivered before the 24Th of June, 7 seven days before the presentation. However, this project was planned to finish

the 3rd of June. Nevertheless, we have finished it the 17Th of June. We have been able to delay the project, because we planned the delivery 21 days before the real deadline. This gave us some days to accommodate the deviations.

Following we explain the project deviations and the mitigation actions we have taken.

2.7.1 Machine Failures

Normal servers, and specially prototypes, can have hardware errors. Also, machines may not be available for other reasons like high utilization on the server, network problems or maintenance works.
Solution: This problems can only be resolved by system administrators. In our case, we had to change the test machines in the phase of energy and power measurement, because the tools used for this purpose were broken. We changed ThunderX for Skylake, so we could compare ThunderX2 metrics with another architecture.

2.7.2 Timetable

Because we have special conditions for the development of this project (programs licenses), the timetable was a critical part of the project.
Solution: To overcome it, a very rigid and realistic timetable was followed with weekly meetings with the project advisors to ensure the project was following the timetable.

2.7.3 Bad Documentation

Working with machine prototypes usually leads to working with incomplete documentation.
Solution: We faced situations where we needed information that did not appear in the documentation. The developer have had to ask to experts or developers with knowledge on the problems for help.

Chapter 3

Budget and Sustainability

In this chapter the budget and sustainability of the project are explained. The first part contains a detailed description of software, hardware and human costs and an analysis of how the budget was affected by deviations. The second part evaluates the sustainability of the project.

3.1 Budget Estimation

In order to develop the project successfully, it requires a set of resources listed in the previous report, Section 2.5. In this section we estimate the costs of the human, hardware, software resources and the indirect costs. The amortization of each investment will be calculated using two factors; the first one is the useful life of the investment, and the second one is the amount of time is used in the project.

3.1.1 Human Resources

This project is mainly carried by only one person that does all the experiment and documentation work. He counts with the help of three advisors that have guided him. Table 3.1 specifies the number of hours estimated and the final hours consumed for every role and the economic value of them.

3.1.2 Software Resources

In this section, we list the costs of the software tools that were used during the development of the project. All tools are free for use for academic purposes or are open source.

Role	Price per hour(€)	Estimated Hours	Cost (€)	Real Hours	Cost (€)
Analyst / Advisor	20.00	32	640.00	32	640.00
Analyst / Intern	8.00	550	4400.00	595	4760.00
Total			5040.00		5400.00

Table 3.1: Human resources budget.

Product	Price (€)	Useful life	Amortization (€)
GCC (8.2.0)	0.00	–	0.00
Arm compiler for HPC (19.0)*	0.00	–	0.00
Arm Instruction Emulator*	0.00	–	0.00
Arm Performance Libraries*	0.00	–	0.00
CMake	0.00	–	0.00
Perfmon2	0.00	–	0.00
Python	0.00	–	0.00
Extrae	0.00	–	0.00
Paraver	0.00	–	0.00
Latexmk	0.00	–	0.00
PDF viewer	0.00	–	0.00
SLURM	0.00	–	0.00
Matlplotlib	0.00	–	0.00
ICC*	0.00	–	0.00
Libcount	0.00	–	0.00
Total	0.00		0.00

*Need licence but are free for academia

Table 3.2: Software budget.

3.1.3 Hardware Resources

In this section, we list the costs of the hardware clusters that were used during the development of the project. All this hardware has been purchased before the beginning of the project, because is part of the Mont-Blanc project. These machines are not exclusive for this bachelor thesis, so the only cost this hardware has for the project is their amortization. Servers and personal computers have an estimated life of 5 years. Table 3.3 lists the amortization of each hardware resource. The total cost is 1210,68 €, but we split this value in two because at the time initial estimation was done Skylake cluster was not supposed to be used.

Investment	Price (€)	Useful life	Amortization (€)
Dell Inspireon 13 7030	900.00	5 years	60.00
Thunder cluster node*	3500.00	5 years	233.34
Dibona ThunderX2 cluster node*	6000.00	5 years	400.00
Dibona Skylake cluster node*	7760.19	5 years	310.00
Total	18160.00		693.34 + 310.00

*The cost of the Dibona clusters is an estimation because they are not publicly available

Table 3.3: Hardware budget.

3.1.4 Indirect Costs

In addition to the costs related to the explicitly required resources specified up to this point, other costs that affect project budget must be considered. Even if they are not directly related with the tasks. These costs are derived from project execution, i.e the cost of electricity, internet access, office rent, etc. However, as the project was developed in its totality at the BSC-CNS installations, it is not possible to specify the real cost of the services. In its place we provide estimated data, see Table 3.4.

Product	Price per month(€)	Number of months	Cost (€)
Electricity	75.00	4	300.00
Internet	37,90	4	151.60
Office rent	90.00	4	360.00
Transport	35.00	4	140.00
Total			951.60

Table 3.4: Estimated indirect costs.

3.1.5 Contingency and Unexpected Events

Budget estimation as project planning can suffer possible deviations. Therefore, contingency was calculated as much as 15% of the total cost. This value permitted us to extend the project 14 days. At the end, we did not exceed the total budget, see Table 3.5

3.1.6 Total Budget

Finally, by consolidating all tables above, the total budget is obtained. Taxes that apply to resources cost are included in the investment price.

Type	Estimated Cost (€)	Real Cost (€)
Human resources	5040.00	5400.00
Software resources	0.00	0.00
Hardware resources	693.34	1003.34
Indirect costs	951.60	951.60
Total without contingency	6684.94	- -
Contingency (15%)	+ 802.20	- -
Total	7487.14	7354.94

Table 3.5: Total budget.

3.2 Budget Control

In this section we explain the control mechanism developed to prevent project deviations. One of the most frequent deviations in computer science projects is task duration. It is highly important to account the number of hours spent on every project task, and to have a comparison with the amount of estimated hours. Verification of project objectives is also a key part of project tracking.

A control mechanism for measuring and avoiding deviations is obtaining the difference between resources consumed and resources estimated for each task. This procedure has to take special care of tasks that have a high risk of deviation. The principal motivation for following such a strict tracking is that, the earlier a deviation is detected, the earlier could a corrective action be taken. Therefore, the deviation would have a lower impact on task scheduling. This tracking is assured to be executed because there is a weekly meeting between developer and advisors.

During project execution we did some more resources than the initially estimated, since we had to use extra hardware that was not planned. In the case of software programs there are a lot of open source alternatives that covered project needs. Human resources was the main resource deviation and the one who most budget increment supposed, that was the reason to include a contingency item in the budget.

The possible project deviations were measured with the following metrics:

- Efficiency deviations:
  Cost of human resources deviation = (number of work hours estimated - number of work hours realized) * estimated cost
  Cost of product consumption deviation = (estimated consumption of product - real consumption of product) * estimated product cost

- Total deviations:
  Total deviation on human resources = estimated human resources cost - real human resources cost
  Total fixed cost deviation = estimated total fixed costs - real total fixed costs

3.3 Sustainability

The sustainability of this project is discussed from three points of view: economic, social and environmental.

3.3.1 Self-assessment of the Current Domain of Sustainability Competence

After answering an interactive pool provided by UPC, I have obtained the following conclusions:

Throughout the survey, I realized that I have an intermediate level of knowledge about sustainability of a project. All knowledge I have about this topic comes from self learning and personal interests, because in my university of origin this competence does not exist. I think that I am able to analyze correctly the sustainability of a project in all three dimensions: economic, social and environmental. I can identify possible impact of an IT project. Moreover I can find new ideas and solutions in order to make these IT projects more sustainable in any of these three areas.

To me, economic and environmental aspects of IT projects are easy to cover since their impact can be easily measured and actions can be taken in consequence. For example, development teams can choose product materials taking into account how big are its associated costs or environmental footprints. However, to me social sustainability is harder to analyze. I do not know accessibility, ergonomics, on safety metrics; nor possible solutions or actions to modify project shortcomings on these aspects. Despite this lack of knowledge in these topics, I consider social impact of my research projects, so they can help people improve their quality of life.

On the other hand, during the survey I found some terms that I was not familiar with. For example, the existence of *deontological ethics* about sustainability for computer scientists. So I had to find information about this logic. In the future I want to know more about them so I can apply them to future projects.

3.3.2 Environmental Sustainability

During the development this project it consumed only one environmental resource, electricity. Test clusters (ThunderX and Dibona ones) were running during all 4 months. In fact, they are running all year, because they are a shared resource for different scientific research groups. Dibona's ThunderX2 has a known power of 180 watts and ThunderX 120 watts. However, these metrics are not exact because the whole system consumes more power, and both processors have lower operation frequencies, because they are not always working at full load. Since the start of the project to its finish date, there have been 2520 hours. That supposes 756 kWh consumed by the clusters.

Besides these two clusters, a laptop was used for development tasks, but its energy consumption is negligible, it spent around 500 Wh. All this energy consumption could not be reduced because the laptop was used only when it was necessary and the cluster cannot be turned off.

Even though, the resulting conclusions extracted from this project can be used in future works, so wasted energy would be further amortized. When programmers know what compiler flags generate best power efficient code helps to waste less energy.

Therefore its environmental sustainability is awarded with an 8. Despite its high energy consumption during development, it could save energy in posterior applications, and has a low risk for the environment.

3.3.3 Economic Sustainability

Project costs have been already detailed in section 3.1. There is no estimation for maintenance within budget summary, since the only project outcome is the results report. However, the project's useful life is expected to last for a while, because it may be useful for future research teams or developers.

Reducing project budget maintaining the project scope could have been only achieved by reducing the number of dedicated hours taken to develop it, but this would have required a more experienced developer who's salary would be higher. Both hardware and software resources are clearly justified in the planning section, so every euro spent in this resources could not be saved.

The cost of this project compared to other state-of-the-art and similar projects is more affordable, due to a shorter project duration, lower waste in human resources and lower use of software and hardware resources. This similar projects invest from 17933€ to 23026€ during development [26][27][27]. While the total cost of the project is 7487.14€.

This project is awarded an 8 in the economic sustainability area, since most of the software tools have no cost at all, there is no maintenance, and even though the Mont-Blanc clusters are expensive, they are not used exclusively for this research.

3.3.4 Social Sustainability

At personal level, the accomplishment of the projected plan has supposed my firsts steps in research. As said many times before, this project will engross Academia publications and may contribute to have a better understanding of today's compilers. At the same time I have learned about how compilers generate assembly code and optimize it. This project also aimed to support the development of high performance computing models for Arm technology. This may help to perfect HPC applications like medical, engineering and scientific simulations.

Actually, there is a lot of literature published about GCC and Clang compilers (Arm HPC Compiler is based on Clang). But all this articles focus on Intel architectures, therefore there is not enough information of code generated by these two compilers on Arm architectures. HPC research, which was treated indirectly in this project, has a high impact in society because it helps scientist find solutions for social problems like air pollution, medical treatments or product development.

Hence, it is awarded a 7 in the social sustainability area, since it will benefit the scientific community and has the potential to improve people quality of life, not directly but indirectly.

Chapter 4

Methodology

This chapter describes the experiment infrastructure built and the methodology we follow. First, a description of the software stack used, followed by selection of benchmarks, an architectural description of the test machines, and finally the experimentation environment and metrics.

4.1 Benchmark Selection

This section describes the selection process we have followed to choose the set of benchmarks we want to study. Then, the chosen benchmark suites are described. A benchmark suite is a collection of benchmarks that are grouped by its characteristics. As mentioned in section 2.2.2, we have picked up benchmark suites employed in the Mont-Blanc 2020 project. The project team has followed this criteria:

I Benchmarks represent a computational pattern that consumes significant resources on today's massively-parallel HPC systems.

II Benchmarks are foreseen to be long-term candidates for relevant HPC computations.

III Benchmarks allow porting in terms of manual code vectorization, auto-vectorization, and by using generic performance libraries. Benchmarks are suitable to SVE porting.

IV Benchmarks demand high memory bandwidth or represent relevant HPC computational patterns.

V Benchmarks are written in C/C++ and use OpenMP.

Therefore, we have choose the following suites of benchmarks:

Name of the suite	Type	Number of benchmarks
HACCK	n-body methods	1
HPCG	sparse linear algebra	1
RAJAPerf	set of HPC kernels	11

Table 4.1: Suite description and number of benchmarks per suite.

4.2 Metrics

To make an accurate and deep analysis we need a set of metrics that detail the behaviours of the experiments. Moreover, these metrics help us to compare different experiments between them. In this section we present the metrics and how they are obtained.

First and most basic metric is speed-up. It help us to calculate how much faster or slower is one experiment respect to other. Speed-up is calculated as:

$$Speedup = \frac{\text{Base Execution Time}}{\text{Improved Execution Time}}$$

When we calculate the speed-up respect to the same application but changing the number of cores, we usually talk about parallel efficiency. It measures how an application speeds-up respect to the number of cores used and is calculated as:

$$\text{Parallel Efficiency} = \frac{Speed - up}{\text{Number of Threads}}$$

Another performance metric commonly used when comparing different processors with different frequency is GFlops. It measures the number of floating operations executed by time unit. In most benchmarks the main part of the computation work is done by floating point operations, therefore measuring the number of floating point instructions is more accurate than measuring the total number of instructions. It is calculated as:

$$\text{GFlops} = \frac{\text{Number of Floating Point Instructions Executed}}{\text{Execution Time} \times 10^9}$$

When an in-depth analysis is made is common to measure the number of misses per kilo instruction (MPKI). This metric is hard to understand because it describes part of the architectural behaviour hidden to the programmer. This metric is also ambiguous because some times higher is worse (a bad code manages worst data and therefor generates more misses) and in others is better (a good code executes in less instructions and thus with higher misses per kilo instructions). MPKI is calculated as

$$\text{MPKI} = \frac{\text{Number of Misses}}{\text{Number Kilo of Instructions Executed}}$$

A common way of measuring power efficiency is calculating performance (GFlops) per Watt. This metric measures performance per power unit. It is calculates as follows:

$$\text{GFlops/Watt} = \frac{\text{Number of Floating Point Instructions Executed}}{\text{Average Power Consumption} \times \text{Execution Time} \times 10^9}$$

Finally we also want to measure energy efficiency with two distinct metrics. First one is total energy consumed by processor during the experiment. We only have power measuring tools, therefore we will obtain total energy consumed as follows:

$$\text{Energy (Joules)} = \text{Average Power Consumed} \times \text{Execution Time (Seconds)}$$

The last experimental metric is energy delay product (EDP), it relates time to the energy consumed. Sometimes EDP is a better metric than total energy consumed because some codes can be slower in order to consume less energy. But we are interested not only in the energy consumption reductions but also about the performance.

$$\text{EDP} = \text{Energy consumed(Joules)} \times \text{Execution Time (Seconds)}$$

Finally, we present a theretical metric that we will compute for every benchmark using only the high level code. This metric measures the amount of floating point operations a benchmark does for every byte of data brougth from memory, it is called Arithmemtic Intensity. It can help us clasify benchmarks and it is calculated as follows:

$$\text{Arithmetic Intensity (AI)} = \frac{\text{Number of floating point operations}}{\text{Number of bytes read from memory}}$$

4.3 Software Stack

In this section we will describe the main software packages used to generate HPC benchmark binaries and profile them.

4.3.1 Compilers

There are many different compiler solutions in the market, like Fujitsu or Cray compilers. However, for compiling our applications we have used the only two representative and available compilers: GCC 8.2.0 [30] and Arm Compiler for HPC 19.0 [31]. The latter is based on the LLVM 7.0.2 tool chain. Both compilers support SVE and NEON vectorization and implement the OpenMP standard and runtime. In particular, OpenMP 4.5 is fully supported for C/C++ in GCC through the libgomp library (GOMP), while Arm compiler offers C/C++ support for OpenMP 3.1 and some features of OpenMP 4.0/4.5 through the libiomp library. Both compilers have entirely different processes to generate optimized code. In fact, Arm has been working with to development teams two improve both compilers ARM ISA assembly generation [20].

4.3.2 Performance Libraries

Some applications like HPCG use Arm Performance Libraries (PL) that provide BLAS, LAPACK, FFT, and standard math routines. PL routines are tuned specifically for ARM processors. We have used version 19.0 of the PL for both GCC 8.2.0 and Arm HPC Compiler 19.0.

4.3.3 Hardware Counters

Because we want to characterize application behaviour we need more information than just execution time. Today's processors implement a set of registers and counters that can log architectural events inside the processor. Each processor can register a different subset of architectural events but almost all include cycle and instruction count.

There are several libraries that implement system calls that are available to developers so they can read performance counters. The two most famous are Performance Application Programming Interface (PAPI

[33]) and libpfm [32]. We will use the former with extrae, explained in next subsection. We will use the latter to instrument benchmark code, capturing events during the execution of the region of interest (ROI). The concrete events we will capture are specified in section 4.3. We use both tools, instead only PAPI, because extrae generates big trace files that are difficult to understand and have a lot of information. Therefore, we will use libpfm for all the initial experiments, so we can obtain ROI performance counters. For the interesting cases, we will use PAPI with extrae for a finer grain level of detail with traces.

4.3.4 Extrae and Paraver

Extrae and Paraver are two tools developed by BSC. The former is a dynamic instrumentation package to trace programs already compiled and run with shared memory programming model (OpenMP or pthreads), the message passing (MPI) programming model, or both programming models. Extrae generates trace files that can be later visualized with Paraver. Paraver is a flexible parallel program visualization and analysis tool. Paraver was developed responding to the need of having a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. We will use extrae with OpenMP and PAPI, so we can visualize the hardware counters and OpenMP events through time.

4.3.5 ArmIE

Arm Instruction Emulator (ArmIE [22]) is an emulation tool that allows the execution of binaries compiled for ARMv8.2 ISA on machines based on ARMv8.0 ISA. Therefore, we can emulate the execution of a SVE binary without a machine that implements SVE. However, emulation is not suitable to obtain performance metrics and execution times, because ArmIE's results are not related with the behaviour of a real machine implementing ARMv8.2 ISA.

One of the characteristics ArmIE implements is instruction counting, that allow us to count how many instructions are executed, and what number of them is emulated. With both numbers we can check how

increasing the machine vector length used by SVE instructions the total number of instructions is reduced.

4.4 Test Machines Architecture

In this section we describe what are the processors that we will evaluate in our experiments. The description include the architectural characteristics of the processors.

4.4.1 ThunderX

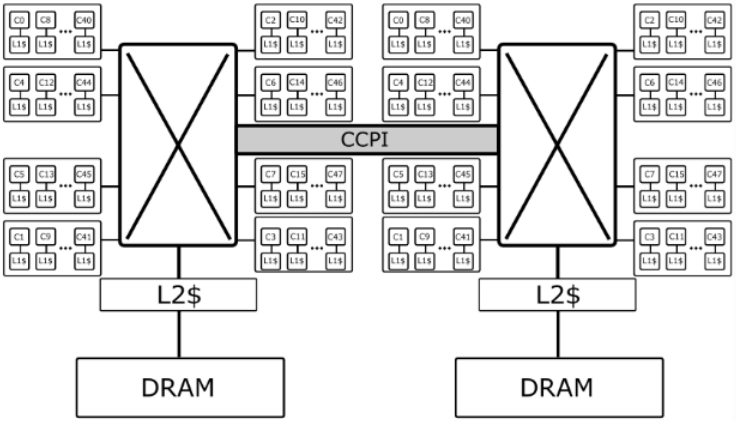


Figure 4.1: ThunderX1 cluster diagram [36]

The ThunderX node is dual socket system, this means that the node has two ThunderX processors. However, we will only use on the ThunderX processors. ThunderX processor is a 64-bit ARMv8 server SoCs, that enables servers and appliances optimized for compute, storage, network and secure compute workloads in the cloud and HPC datacenters [35]. One ThunderX processor has 48 in-order cores, manufactured in 28nm process technology under architectural license from Arm. It can run up to 2.5Ghz but our cluster is configured at 1.8Ghz, so the TDP is under its 120W. ThunderX memory hierarchy is described in Table 4.2 and Figure 4.1 describes processor layout.

L1 Data Cache	Policy	Write-through
	Type	Private
	Size	32KiB
	Associativity	32-way
	Block	128-bytes
L1 Instruction Cache	Size	72KiB
	Associativity	39-way
	Block size	128-bytes
L2 Cache	Policy	Write-back
	Type	Shared
	Size	16MiB
	Associativity	16-way
	Block	128-bytes

Table 4.2: ThunderX memory hierarchy [34].

The ThunderX pipeline is still unknown today, but several articles [37] and compilers development logs [38] take into account that there are two execution pipelines inside each ThunderX core as shown in Figure 4.2. This pipeline comes from the Octeon III router core, but we can figure out that ThunderX's one is very similar. ThunderX SIMD and floating point instructions can only be executed in one of the execution pipelines.

Finally, hardware counters supported by this processor are not documented and most libraries cannot read most of them. Despite, this problem we have been able to read instructions, cycles and L1-data misses.

4.4.2 ThunderX2

The second generation ThunderX2 product family introduced by Cavium was released for general availability in early 2018. ThunderX2 is a family of 64-bit ARMv8.1 processors re-branded by Cavium based on the original design of Broadcom's Vulcan. In the ThunderX2 CN99XX series, the ARM based SoC integrates high-performance custom out-of-order (OoO) cores, supporting up to 32 ARMv8.1 cores in a single-socket configuration and

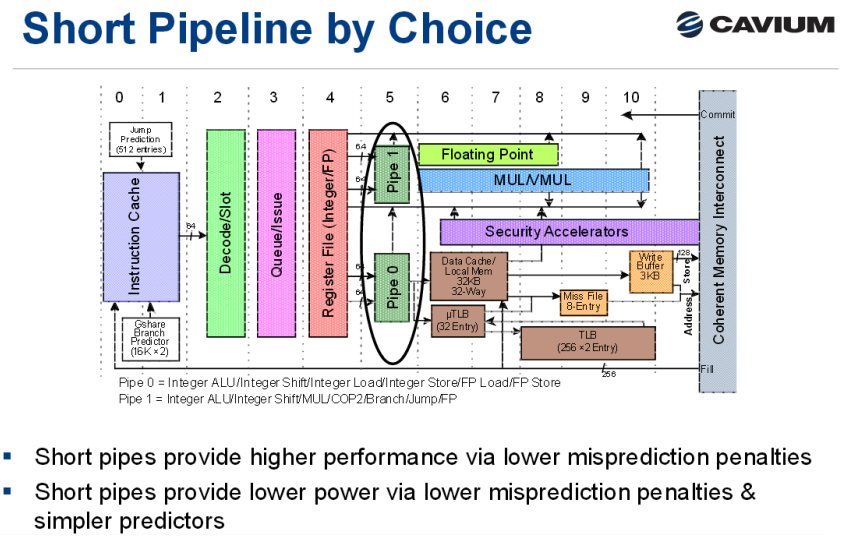


Figure 4.2: Pipe diagram of a ThunderX1 like processor (Octeon III) [37]

64 cores in a dual-socket configuration, with a frequency of up to 2.5GHz in nominal mode and 3GHz in Turbo mode, but in the Dibona prototype frequency is fixed at 2.0GHz. ThunderX2 has a Thermal Design Power (TDP) of 180 Watts. Simultaneous Multithreading (SMT) is supported which allows up to 4 threads per physical core, but we will not use it, SMT is deactivated in Dibona prototype by default. Each core has 32 KiB L1 instruction and data cache, as well as 256 KiB L2 cache. The 32 MiB L3 cache is distributed among the cores. Each ThunderX2 processor provides multiple, up to 8, DDR4-2666 memory controllers per chip with a maximum bandwidth of up to 170 GB/s [16].

In Figure 4.3 we can see the block diagram of one ThunderX2 core. The most interesting aspects of this core for our project is that there are 2 SIMD execution units and the double load bandwidth, two 128-bits load execution units. ThunderX2 memory hierarchy specs are shown in Table 4.3.

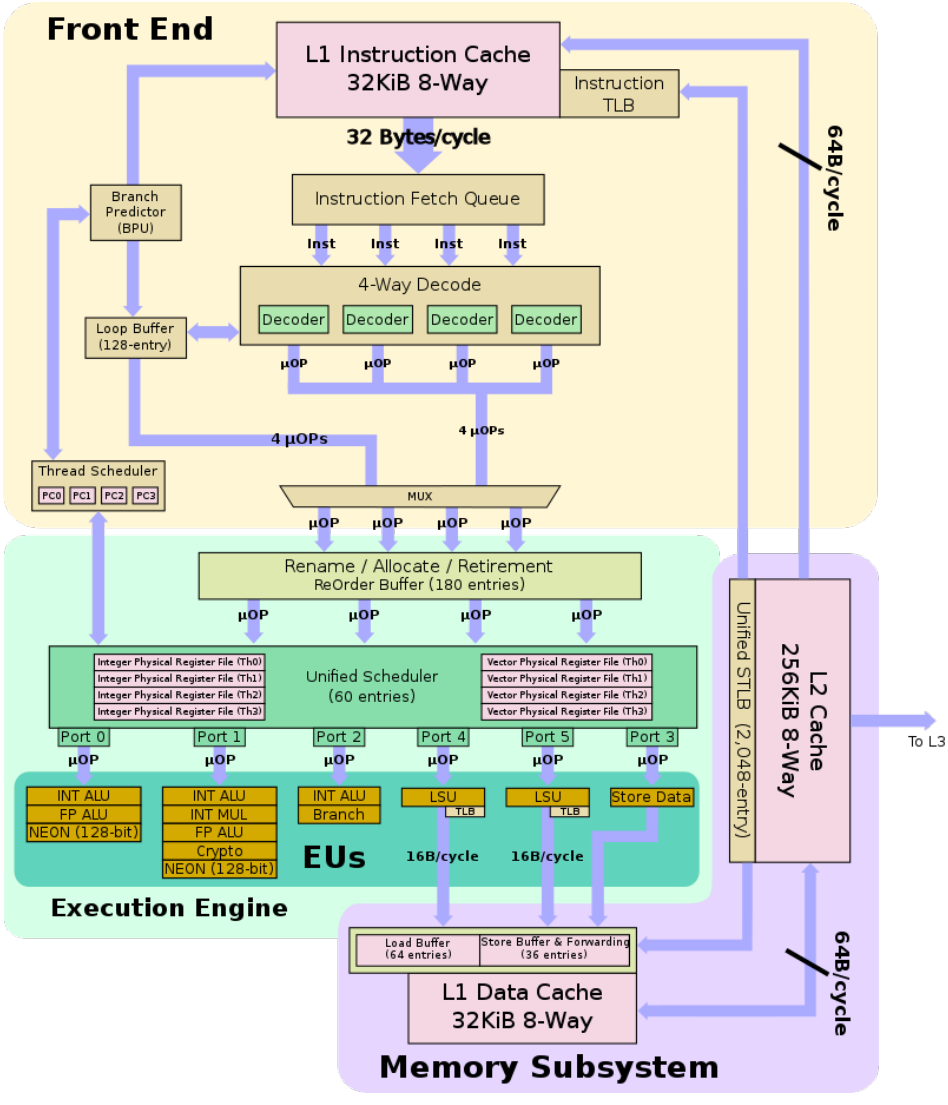


Figure 4.3: ThunderX2 block diagram [39]

Characteristic	L1 D-Cache	L2 Cache	L3 Cache
Type	Private	Private	Shared, Exclusive
Size	32KiB	256KiB	16MiB
Associativity	8-way	8-way	N/A

Table 4.3: ThunderX2 memory hierarchy.

4.4.3 Skylake

Because BSC power measurement tools broke before we could measure power consumption of ThunderX, we have decided to use another test node inside the same cluster for power consumption experiments. This machine is built with 2 Intel® Xeon® Platinum 8176 Processor [43]. Skylake is a family of x86˙64 processors. This processor contains 28 Out-of-Order cores that support Intel® SSE4.2 (128 bits SIMD), Intel® AVX (256b SIMD), Intel® AVX2 (245b SIMD), Intel® AVX-512 (512b SIMD) instructions set extensions. Thanks to multithreading each core support 2 threads that suppose a total number of 56 threads. Base frequency is 2.1 Ghz but on turbo mode can reach 3.80 GHz. Last level of cache is L3 that has a size of 38.5 MB. The TDP is 165 Watts.

Its most important architectural characteristics are the 224 ROB entry, two 512 bits Vector Floating Point ALU.

4.5 Workload and Environment

Since most HPC systems run over MIMD and SIMD paradigms we have done our experiments over different combinations of them. First, we have compiled four basic binaries that correspond with: NEON supported binary compiled with Arm HPC compiler (NEON-ArmHPC), NEON supported binary compiled with GCC, no-SIMD support binary compiled with Arm HPC Compiler (SCALAR-ArmHPC) and no-SIMD support binary with GCC Compiler (SCALAR-GCC). And for the emulation experiments we have compiled two more binaries that are SVE support binary compiled with Arm HPC Compiler (SVE-ArmHPC) and SVE support binary compiled with GCC (SVE-GCC). We have compiled two

no-SIMD versions only to compare them with SIMD ones. MIMD paradigm has been tested executing these binaries with different number of threads and binding each thread to only one core of the machine.

4.5.1 Compilation

Tables 4.4 and 4.5 detail the base compiler flags enabled for GCC and Arm HPC Compiler. Tables describe only two sets of compiler flags that are shared by all binaries. There are only two compiler flags that change for every binary, they are listed in Table 4.6. These two flags must be concatenated to the end of the basic flags in order to achieve the correct compilation. Respect to the rest of the compiler flags, we only employ the default ones that came with the vanilla sources. Flags have only been modified to generate specific ARMv8 code and allow or deny vectorization. Skylake flags are listed in appendix B.

Application	Flag SCALAR-GCC
HACCK	-O3 -ffast-math -fopenmp -funroll-loops -ffp-contract=fast
HPCG	-O3 -ffast-math -fopenmp -funroll-loops -std=c++11 -ffp-contract=fast -larmpl_lp64_mp
RAJAPerf	-O3 -fopenmp -ffast-math

Table 4.4: Flags for GCC scalar version.

Application	Flag SCALAR-ArmHPC
HACCK	-O3 -ffast-math -fopenmp -funroll-loops -ffp-contract=fast
HPCG	-O3 -ffast-math -fopenmp -funroll-loops -std=c++11 -ffp-contract=fast -larmpl_lp64_mp
RAJAPerf	-O3 -fopenmp -ffast-math

Table 4.5: Flags for Arm HPC Compiler, scalar version.

Binary Version	-march Flag
SCALAR-GCC	-march=armv8-a+fp+nosimd -fno-tree-vectorize
SCALAR-ArmHPC	-march=armv8-a+fp+nosimd -fno-vectorize
NEON-GCC	-march=armv8-a+fpsimd -ftree-vectorize
NEON-ArmHPC	-march=armv8-a+fpsimd -fvectorize
SVE-GCC	-march=armv8-a+fp+sve -ftree-vectorize
SVE-GCC	-march=armv8-a+fp+sve -fvectorize

Table 4.6: Flags for GCC, NEON supported version.

Chapter 5

Benchmark Vectorization

This chapter describes each kernel code and its main characteristics. Moreover, it details the modifications we have made to the code in order to achieve auto-vectorization, and what problems we had during compilation.

5.1 RAJAPerf

For all the benchmarks we have added *simd* clause to OpenMP *parallel for* clause in order to vectorize the loops automatically. For some of the kernels we have also added *schedule(dynaamic,CHUNK)* clause in order to partition the workload in blocks of *CHUNK* iterations of the parallel loop. This second clause is necessary to balance the workloads in ThunderX core, since some processor slowed down during execution.

5.1.1 MULADDSUB

This first kernel executes a Hadamard product of two vectors, addition and subtraction. We have modified the original source code, because both compilers produce extra and not necessary load operations. The original MULADDSUB code and the modified version to ease vectorization are shown in Listing 1. With the original code compilers distrust the accesses performed by other threads; thus, they generate one load and one store for each of the computation variables instead of storing them to registers. We have resolved this by adding the *firstprivate* clause, see Sublisting 1b. This kernel, has a very low Arithmetic Intensity (AI) because it makes only 3 floating point operations (flops) per 5 data elements of 8 bytes, therefore the AI is 0.075 flops/bytes.

Listing 1: MULADDSUB code

```
#pragma omp parallel for simd
for (int i = begin; i < end; ++i ){
  out1[i] = in1[i] * in2[i];
  out2[i] = in1[i] + in2[i];
  out3[i] = in1[i] - in2[i];
}
```

a: Original MULADDSUB code

```
#pragma omp parallel for simd \
 firstprivate(out1,out2,out3, \
              in1,in2)
for (int i = begin; i < end; ++i){
  double in1_i = in1[i];
  double in2_i = in2[i];
  out1[i] = in1_i * in2_i;
  out2[i] = in1_i + in2_i;
  out3[i] = in1_i - in2_i;
}
```

b: Optimized MULADDSUB code

5.1.2 EOS

The EOS kernel, shown in Listing 2, has an AI of 0.5 flops/byte. In each iteration, it takes 4 data elements of 8 bytes from memory and executes 16 floating operations. Despite having 10 memory accesses, only 4 of them are fetched from memory. This kernel has a good memory locality as the remaining accesses have already been fetched from memory and they are stored in cache.

Listing 2: EOS code

```
#pragma omp parallel for simd schedule(dynamic,CHUNK)
for (int i = ibegin; i < iend; ++i ) {
  x[i] = u[i] + r*( z[i] + r*y[i] ) +
         t*( u[i+3] + r*( u[i+2] + r*u[i+1] ) +
         t*( u[i+6] + q*( u[i+5] + q*u[i+4] ) ) );
}
```

5.1.3 HYDRO

Listing 3 shows HYDRO kernel. HYDRO has a low arithmetic intensity of 0.2083 flops/byte. HYDRO executes 3 accesses to main memory and 5 flops.

Listing 3: HYDRO code

```
#pragma omp parallel for simd schedule(dynamic,CHUNK)
for (int i = ibegin; i < iend; ++i ) {
  x[i] = q + y[i]*( r*z[i+10] + t*z[i+11] );
}
```

5.1.4 INT_PREDICT

The INT_PREDICT kernel is a one dimensional stride stencil solver, see Listing 4. INT_PREDICT accesses a matrix stored in a column major order. This kernel is thought to stress memory systems by non-contiguous pattern accesses, and has a low AI of

$$\frac{17 \text{ flops}}{11 \text{ accesses} \times 8 \text{ bytes}} = 0.19318 \text{ flops/byte}$$

.

Listing 4: INT_PREDICT code

```
#pragma omp parallel for simd schedule(dynamic,CHUNK)
for (int i = ibegin; i < iend; ++i ) {
  px[i] = dm28*px[i + offset * 12] + dm27*px[i + offset * 11] +
          dm26*px[i + offset * 10] + dm25*px[i + offset *  9] +
          dm24*px[i + offset *  8] + dm23*px[i + offset *  7] +
          dm22*px[i + offset *  6] +
          c0*( px[i + offset *  4] + px[i + offset *  5] ) +
          px[i + offset *  2];
}
```

5.1.5 COPY

COPY is a memory bandwidth stress kernel from the well-known Stream benchmark [41], Listing 5. No floating point operations are executed in this kernel, thus its AI is 0.

Listing 5: COPY code

```
#pragma omp parallel for simd schedule(dynamic,CHUNK)
for (int i = ibegin; i < iend; ++i ) {
  c[i] = a[i] ;
}
```

5.1.6 VOL3D

VOL3D is a very interesting application that manipulates points in a 3D space, Listing 6. In contrast to the previous kernels, has a high AI of 1.388 flops/byte. This value has been obtained experimentally. We have measured the number of data bytes requested by the processor and the number of flops executed. If we divide the number of memory accesses by the number of iterations we get 8 memory accesses requested in each iteration:

Listing 6: VOL3D code

```
#pragma omp parallel for simd schedule(dynamic,CHUNK)
for (int i = ibegin ; i < iend ; ++i ) {

    double x71 = x7[i] - x1[i] ;      double y71 = y7[i] - y1[i] ;
    double x72 = x7[i] - x2[i] ;      double y72 = y7[i] - y2[i] ;
    double x74 = x7[i] - x4[i] ;      double y74 = y7[i] - y4[i] ;
    double x30 = x3[i] - x0[i] ;      double y30 = y3[i] - y0[i] ;
    double x50 = x5[i] - x0[i] ;      double y50 = y5[i] - y0[i] ;
    double x60 = x6[i] - x0[i] ;      double y60 = y6[i] - y0[i] ;

    double z71 = z7[i] - z1[i] ;      double z30 = z3[i] - z0[i] ;
    double z72 = z7[i] - z2[i] ;      double z50 = z5[i] - z0[i] ;
    double z74 = z7[i] - z4[i] ;      double z60 = z6[i] - z0[i] ;

    double xps = x71 + x60 ;
    double yps = y71 + y60 ;
    double zps = z71 + z60 ;

    double cyz = y72 * z30 - z72 * y30 ;
    double czx = z72 * x30 - x72 * z30 ;
    double cxy = x72 * y30 - y72 * x30 ;
    vol[i] = xps * cyz + yps * czx + zps * cxy ;

    xps = x72 + x50 ;
    yps = y72 + y50 ;
    zps = z72 + z50 ;

    cyz = y74 * z60 - z74 * y60 ;
    czx = z74 * x60 - x74 * z60 ;
    cxy = x74 * y60 - y74 * x60 ;
    vol[i] += xps * cyz + yps * czx + zps * cxy ;
```

```
    xps = x74 + x30 ;
    yps = y74 + y30 ;
    zps = z74 + z30 ;

    cyz = y74 * z60 - z74 * y60 ;
    czx = z74 * x60 - x74 * z60 ;
    cxy = x74 * y60 - y74 * x60 ;
    vol[i] += xps * cyz + yps * czx + zps * cxy ;

    xps = x74 + x30 ;
    yps = y74 + y30 ;
    zps = z74 + z30 ;

    cyz = y71 * z50 - z71 * y50 ;
    czx = z71 * x50 - x71 * z50 ;
    cxy = x71 * y50 - y71 * x50 ;
    vol[i] += xps * cyz + yps * czx + zps * cxy ;
    vol[i] *= vnormq ;
}
```

5.1.7 FIR

Finite Impulse Response (FIR) filter is a signal processing application that applies a filtering function in different time steps, Listing 7. This application has great locality and reuse. On every iteration, 16 elements of 8 bytes of *in* array are used, but only one is brought from memory, the rest are stored in cache. There are 16 multiplications and 15 additions for every 2 elements of 8 bytes loaded, thus the AI is 1.938 flops/byte.

Listing 7: FIR code

```
double coeff_array[FIR_COEFFLEN] = {  3.0, -1.0, -1.0, -1.0,
                                     -1.0,  3.0, -1.0, -1.0,
                                     -1.0, -1.0,  3.0, -1.0,
                                     -1.0, -1.0, -1.0,  3.0 };
#pragma omp parallel for simd
for (int i = ibegin ; i < iend ; ++i ) {
  double sum = 0.0 ;
  for (int j = 0 ; j < coefflen ; ++j ) {
    sum += coeff[j]*in[i+j] ;
  }
  out[i] = sum ;
}
```

5.1.8 JACOBI_1D

Original JACOBI_1D kernel, see Sublisting 8a, is a one-dimensional convolution benchmark with an arithmetic intensity 0.1875 flops/byte (3 flops and 16 accessed bytes), similar to HYDRO and INT PREDICT, so we decided to change the original code for a new one-dimensional convolution, see Sublisting 8b. This convolution tries to find the local maxima in a one dimension data stream. AI improves up to 0.375 flops/byte (6 flops 16 read bytes). Filtering weights are parameterized so it can be easily modified. The Original coefficient of 0.333 must be changed for 0.5 in order to follow image filtering standards. This coefficient is $\frac{1}{K}$, where K is the sum of the positive weights.

Listing 8: JACOBI_1D code

```
for (int t = 0; t < tsteps; ++t) {
  #pragma omp parallel for
  for (int i = 1; i < N-1; ++i ) {
    B[i] = 0.33333 * (A[i-1] +
                      A[i]   +
                      A[i + 1]);
  }
  #pragma omp parallel for
  for (int i = 1; i < N-1; ++i ) {
    A[i] = 0.33333 * (B[i-1] +
                      B[i]   +
                      B[i + 1]);
  }
}
```

a: Original JACOBI_1D code

```
double weights[3] = { -1.0,
                       2.0,
                      -1.0 };

for (int t = 0; t < tsteps; ++t){

  #pragma omp parallel for simd
  for (int i = 1; i < N-1; ++i){
    B[i] = 0.5 *
          (A[i-1] * weights[0] +
           A[i]   * weights[1] +
           A[i+1] * weights[2]);
  }
}
```

b: Modified JACOBI_1D code

5.1.9 JACOBI_2D

We find the same situation in JACOBI_2D. Sublisting 9a shows original JACOBI_2D code, which does an image convolution or box filtering, and again we have the same problem this time with an IA of 0.3125 flops/byte. We have followed the same approach, and we have replaced the original kernel with a Laplace filter (Sublisting 9b) to recognise corners or points

on a gray scale image. The IA is 1.125 flops/byte (18 flops and 16 read bytes). Both JACOBI codes are very similar to the FIR kernel. The main difference is that FIR has a nested loop and an accumulator variable that can make compilers generate different code.

Listing 9: JACOBI_2D code

```
for (int t = 0; t < tsteps; ++t) {

  #pragma omp parallel for
  for (int i = 1; i < N-1; ++i ) {
    for (int j = 1; j < N-1; ++j ) {
      B[j + i*N] = 0.2 *
                   (A[j   + i*N]  +
                    A[j-1 + i*N]  +
                    A[j+1 + i*N]  +
                    A[j +(i+1)*N] +
                    A[j +(i-1)*N]);
    }
  }

  #pragma omp parallel for
  for (int i = 1; i < N-1; ++i ) {
    for (int j = 1; j < N-1; ++j ) {
      A[j + i*N] = 0.2 *
                   (B[j   + i*N]  +
                    B[j-1 + i*N]  +
                    B[j+1 + i*N]  +
                    B[j +(i+1)*N] +
                    B[j +(i-1)*N]);
    }
}
```

a: Original JACOBI_2D code

```
double weights[9] = { 1, 4, 1,
                      4,-20, 4,
                      1, 4, 1 };

for (int t = 0; t < tsteps; ++t) {

 #pragma omp parallel for simd
 for (int i = 1; i < N-1; ++i ) {
  for (int j = 1; j < N-1; ++j ) {
   B[i*N+j] = 0.05 *
     (A[(i-1)*N +j-1]*weights[0] +
      A[(i-1)*N +j  ]*weights[1] +
      A[(i-1)*N +j+1]*weights[2] +
      A[ i  * N +j-1]*weights[3] +
      A[ i  * N +j  ]*weights[4] +
      A[ i  * N +j+1]*weights[5] +
      A[(i+1)*N +j-1]*weights[6] +
      A[(i+1)*N +j  ]*weights[7] +
      A[(i+1)*N +j+1]*weights[8]);
  }
 }
}
```

b: Modified JACOBI_2D code

5.1.10 GEMM

The GEMM benchmark is a common matrix multiplication algorithm with time cost $O(n^3)$. Original code in Listing 10a. We have removed the *collapse* clause because it inhibited compiler vectorization. The *alpha* constant has been removed too, because we wanted a lower IA. Original is 0.375 flops/byte, the same as JACOBI˙1D. Final code is shown in Listing

10b. GEMM arithmetic intensity is hard to calculate as there is a some reuse in the code. Because each matrix has a size of 2400x2400 elements, all three matrices cannot fit in memory, but a row or a column can. First, we only consider the inner loop. Only k is modified in each iteration, so we can take A as a constant because only a row is stored in memory and B as the read bytes source. Thus, AI is 0.25 flops/byte (2 flops and 8 read bytes). We can repeat this computation and be more precise for the middle loop. Only k and j are modified in each loop iteration, again A is a constant and this time B and C are a source of data: $\frac{2400*2 \text{ flops}}{(2400+1)*8 \text{ read bytes}} = 0.249$ flops/byte.

Therefore, we can consider a theoretic AI of 0.25 flops/byte, however we are running this benchmarks on real machines, and therefore we should consider that memory is brought in blocks and stored in cache. Therefore if the assembly code generated accesses only one element of the block and then requests another block, the AI may be lower depending on the cache block size or the optimizations performed by the compilers.

Listing 10: GEMM code

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < ni; ++i ) {
  for (int j = 0; j < nj; ++j ) {
    C[j + i*nj] *= beta;
    double dot = 0.0;
    for (int k = 0; k < nk; ++k ) {
      dot += alpha * A[k + i*nk] *
             B[j + k*nj];
    }
    C[j + i*nj] = dot;
  }
}
```

a: Original GEMM code

```
#pragma omp parallel for simd
for (int i = 0; i < ni; ++i ) {
  for (int j = 0; j < nj; ++j ) {
    C[j + i*nj] *= beta;
    double dot = 0.0;
    for (int k = 0; k < nk; ++k){
      dot += A[k + i*nk] *
             B[j + k*nj];
    }
    C[j + i*nj] = dot;
  }
}
```

b: Modified GEMM code

5.1.11 FLOYD_WARSHALL

Floyd-Warshall algorithm is a well known polynomial algorithm that can find all pair short paths in a graph in time $O(|V|^3)$. As GEMM algorithm,

this kernel has low locality and reuse, as can be seen in Listing 11. In this case, *pin[k + i*N]* is a row of the input matrix. We can consider it as a constant is reused in the inner loop through different iterations. We consider the < operation as a floating point operation. Thus, its IA is 0.125 (2 flops and 16 read bytes).

Listing 11: FLOYD_WARSHALL code

```
for (int irep = 0; irep < run_reps; ++irep) {
  for (int k = 0; k < N; ++k) {

    #pragma omp parallel for simd
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        pout[j + i*N] = pin[j + i*N] < pin[k + i*N] + pin[j + k*N] ?
                        pin[j + i*N] : pin[k + i*N] + pin[j + k*N];
      }
    }
  }
}
```

5.2 HACCK

The Hardware/Hybrid Accelerated Cosmology Code (HACC [9]) , is a cosmology N-body-code designed to run efficiently on diverse computing architectures and to scale to millions of cores and beyond. It can simulate gravity forces produced between particles at cosmological scale.

The first time we compiled the program we found that the Arm HPC Compiler could vectorize the code whereas GCC could not. To enable both compilers to vectorize the benchmark we modified the source code of the benchmark, see Sublisting 12a. We found that the *continue* clause was the problem and removed it using Boolean variables. Arm HPC Compiler made this transformation using a bit mask approach. We also changed single precision variables to double precision, because the rest of the benchmarks used double precision. Then, we recompiled and verified that both compilers have vectorized the code. Such modifications are shown in Sublisting 12.

Listing 12: HACCK code

```
#pragma omp simd \
    reduction(+:lax,lay,laz)
for (int i = 0; i < n; ++i) {
  float dx = x[i] - x0;
  float dy = y[i] - y0;
  float dz = z[i] - z0;
  float r2 = dx * dx +
             dy * dy +
             dz * dz;

  if (r2 >= MaxSepSqrd || r2 == 0.0f)
    continue;


  float r2s = r2 + SofteningLenSqrd;
  float f=PolyCoefficients[PolyOrder];
  for (int p=1; p<=PolyOrder; ++p)
    f = PolyCoefficients[PolyOrder-p]
        + r2*f;

  f = (1.0 / (r2s * sqrt(r2s)) - f) *
       mass[i];

  lax += f * dx;
  lay += f * dy;
  laz += f * dz;
}
```

a: Original HACCK code

```
#pragma omp simd \
    reduction(+:lax,lay,laz)
for (int i = 0; i < n; ++i) {
  double dx = x[i] - x0;
  double dy = y[i] - y0;
  double dz = z[i] - z0;
  double r2 = dx * dx +
              dy * dy +
              dz * dz;

  bool bigger, zero;
  bigger = r2 < MaxSepSqrd;
  zero = r2 != 0.0f;

  double r2s = r2 + SofteningLenSqrd;
  double f=PolyCoefficients[PolyOrder];
  for (int p = 1; p <= PolyOrder; ++p)
    f = PolyCoefficients[PolyOrder-p]
        + r2*f;

  f = (1.0 / (r2s * sqrt(r2s)) - f) *
      mass[i];

  lax += f * dx * zero * bigger;
  lay += f * dy * zero * bigger;
  laz += f * dz * zero * bigger;
}
```

b: Optimized HACCK code

There are three different versions of the HACC Kernel depending on the degree of the polynomial used (order 4, 5 or 6). We will analyze the performance of all of them.

5.3 HPCG

The last application we have selected is High Performance Conjugate Gradient (HPCG) benchmark. HPCG was designed to replace the Linpack benchmark, whose main objective is to serve as a metric for ranking high performance computing systems [10]. HPCG resolves systems of sparse linear equations using an iterative method. This algorithm makes use of many known linear functions like matrix-vector multiplication or library calls that make high level code hard to read, therefore we. The main modification done to the code is the addition of the *simd* clause To parallel loops.

Chapter 6

Results and Analysis

In this chapter we present and analyze the results of our experiments on ThunderX and ThunderX2. First, we expose what are the performance bounds of our applications using Roofline Models. Then, we divide the benchmarks in different sets by their behaviors and analyze their performance. Moreover, we compare both processors and detail power efficiency and energy consumption of the applications. Finally, we analyze what are the causes of these behaviors by inspecting the assembly code generated by Arm HPC Compiler and GCC.

6.1 Roofline Models

Roofline models [40] are a simple and visual way to understand program behaviour. A roofline model ties together floating point performance, arithmetic intensity, and memory performance in a two-dimensional graph, see Figure 6.1. The Y-axis is GFlops per second (performance). Theoretical ceilings can be derived using the hardware specifications. The X-axis is operational intensity, operations per byte of DRAM traffic. Therefore, we measure traffic between the caches and memory rather than between the processor and the caches. We can then plot memory performance by calculating the maximum floating-point performance that the memory system of that computer can support for a given operational intensity. This formula drives the two performance limits in the roofline model:

$$\text{GFlops/s} = min \begin{cases} \text{peak floating point performance} \\ \text{peak memory bandwidth } \times \text{ operational intensity} \end{cases}$$

We have calculated peak performance with the following formula:

$$\begin{aligned}\text{Peak GFlops} = {} & (\text{CPU speed in GHz}) \times (\text{number of CPU cores}) \times \\ & (\text{SIMD element wide}) \times (\text{flops per operation}) \times \\ & (\text{number of SIMD units})\end{aligned} \tag{6.1}$$

We have used fused multiply-add (FMA) instruction as reference to compute peak performance. It performs two floating point operations, thus an FMA using NEON performs 4 double precision floating point operations. Next we list the different performance peaks we will use for the roofline models:

For 48 ThunderX cores using scalar instructions in a double precision program we have:

$$1.8\ \text{GHz} \times 48\ \text{cores} \times 1\ \text{element wide} \times 2\ \text{flops} \times 1\ \text{exec. unit} = 172.8\ \text{GFlops}$$

For 48 ThunderX cores using SIMD in a double precision program:

$$1.8\ \text{GHz} \times 48\ \text{cores} \times 2\ \text{elements wide} \times 2\ \text{flops} \times 1\ \text{SIMD unit} = 345.6\ \text{GFlops}$$

ThunderX can deliver up to one SIMD FMA instruction of 128 bits every cycle.

For one ThunderX2 core using scalar instructions in a double precision program:

$$2.0\ \text{GHz} \times 1\ \text{core} \times 1\ \text{element wide} \times 2\ \text{flops} \times 2\ \text{exec. unit} = 8\ \text{GFlops}$$

For one ThunderX2 core using SIMD in a double precision program:

$$2.0\ \text{GHz} \times 1\ \text{core} \times 2\ \text{elements wide} \times 2\ \text{flops} \times 2\ \text{SIMD unit} = 16\ \text{GFlops}$$

For 32 ThunderX2 cores using scalar instructions in a double precision program:

$$2.0\ \text{GHz} \times 32\ \text{cores} \times 1\ \text{element wide} \times 2\ \text{flops} \times 2\ \text{exec. unit} = 256\ \text{GFlops}$$

For 32 ThunderX2 cores using SIMD in a double precision program:

$$2.0\ \text{GHz} \times 32\ \text{cores} \times 2\ \text{elements wide} \times 2\ \text{flops} \times 2\ \text{SIMD unit} = 512\ \text{GFlops}$$

ThunderX2 can deliver up to two SIMD FMA instructions of 128 bits every cycle. This is equivalent to 8 double precision floating point operations.

Peak theoretical memory bandwidth of ThunderX is 150,0 GB/s. While, ThunderX2 has a peak memory bandwidth of 170 GB/s.

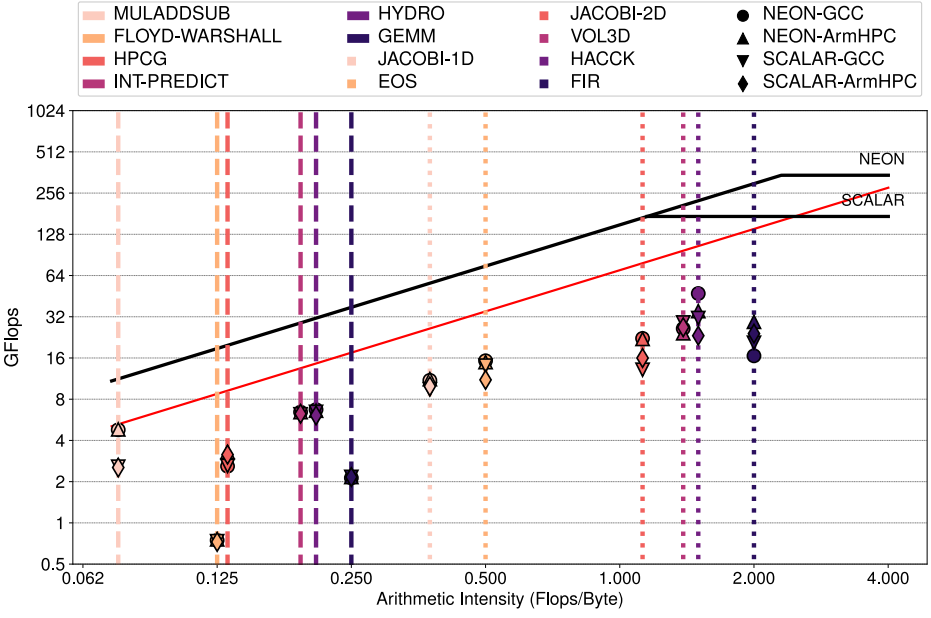


Figure 6.1: Roofline model for 48 ThunderX cores

Figure 6.1 represents the roofline model for a ThunderX processor with 48 cores. This roofline has two CPU performance ceilings: SCALAR and NEON double precision performance ceilings. There are also two memory bandwidth ceilings: the black line represents the theoretical peak bandwidth (150 GB/s) and the red line the maximum bandwidth achieved using the STREAM benchmark (70 GB/s).

For every benchmark we make 4 measurements corresponding to all possible compiler and flag flavors. These are the two compilers GCC and Arm HPC Compiler (labeled ArmHPC) and no-SIMD (labeled as SCALAR) or SIMD (labeled as NEON).

We can see that all benchmarks are far from the theoretical bandwidth limit and only MULADDSUB is near from the experimental memory peak. This means that the performance bottleneck is the computational CPU capacity. The performance (GFlops/s) of FLOYD_WARSHALL and GEMM benchmarks is really poor. In the former, we can only obtain 0.746 GFlops/s with 48 cores (less than 0.2% of the peak), and in the latter, about 2.166 GFlops/s (around 0.6%). In both benchmarks a matrix is traversed in row major order. Despite the rest of the benchmarks have a better performance none of them is above 47.30 GFlops (13.6% of the peak). Therefore we can say that arithmetic units are underused because of data dependencies and pipeline stalls. These are due to microarchitectural details of a ThunderX core, that features an in-oreder pipeline that is not well suited for HPC tight loops. We can also distinguish between compilers and between SCALAR and NEON. We will explain this in Section 6.2.

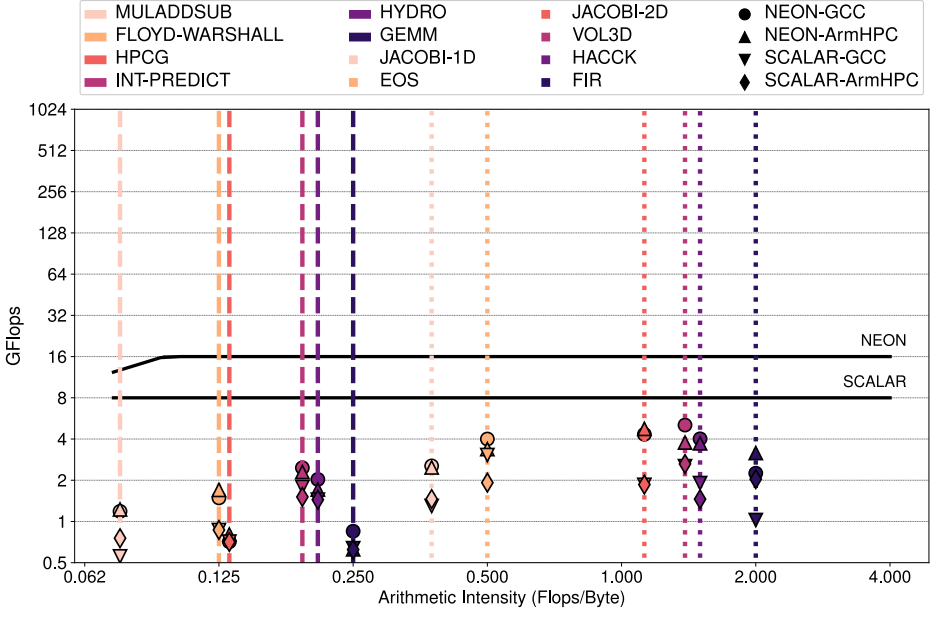


Figure 6.2: Roofline model for 1 ThunderX2 core

Figure 6.2 is the ThunderX2 roofline model for **only one core**. Again

we can see that all applications are bounded by CPU performance. In this model memory bandwidth ceilings do no represent any constraint. FLOYD_WARSHALL executed on only one core of ThunderX2 has a better performance than executed on 48 cores on the ThunderX. The peak performance is 5.05 GFlops/s (31.6%). In summary ThunderX2 exploits its resources better than ThunderX due to its out-of-order pipeline; however, there is still room for improvement.

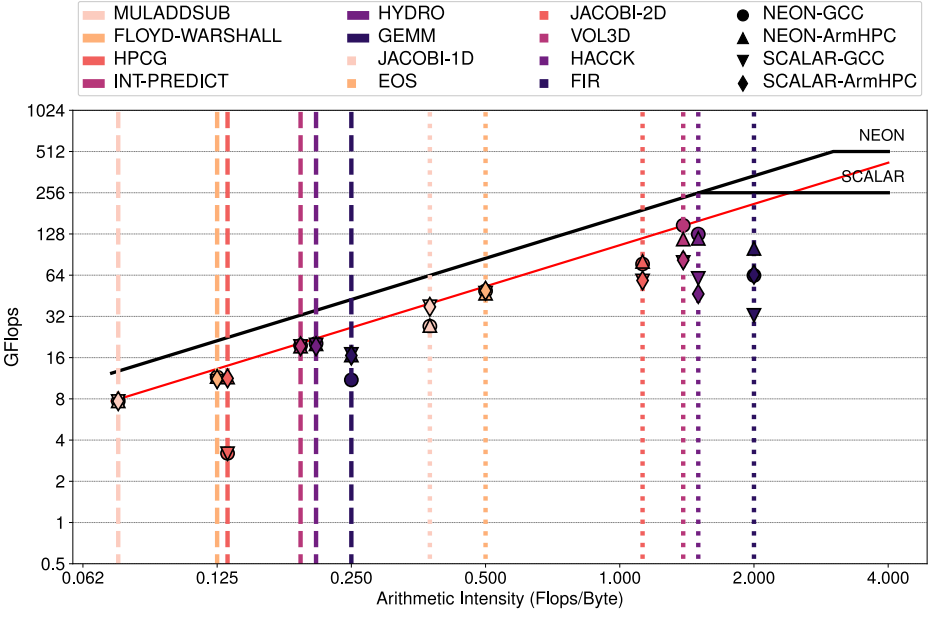


Figure 6.3: Roofline model for 32 ThunderX2 cores

Figure 6.3 shows a second roofline model for ThunderX2, this time using all 32 cores. Again, the black memory bandwidth ceiling represents the theoretical limit (170 GB/s) and the red one represents the peak achieved by STREAM benchmark (100 GB/s). This third model is completely different from the past two because many benchmarks reach the experimental memory bandwidth limits. No matter how fast the CPU we use is, it will be limited by memory bandwidth. Only HPCG, JACOBI_2D, FIR and some versions of HPCG are below the memory bound ceiling, and

therefore can improve their performance.

These behaviour differences between test machines can be explained. The ThunderX processor is built with in-order cores that make it perform far from its theoretical peak. In contrast, the ThunderX2 processor is based on out-of-order cores that yield superior performance. With only one thread and one core running, performance is not constrained by memory bandwidth, but when the processor is running with all the cores, it is.

6.2 Performance, Scalability and Bounds

This section contains the performance differences (speed-up) between the four compiled binaries, GCC using scalar instructions (SCALAR-GCC), GCC compiled with NEON support (NEON-GCC), Arm HPC Compiler using scalar instructions (SCALAR-ArmHPC) and Arm HPC Compiler with NEON support (NEON-ArmHPC). We will remark what are the main limitations in benchmarks with poor scalability.

Figure 6.4 shows the speed-up between the four binaries on 48 ThunderX cores. The speed-up is normalized to the execution time using one thread and the SCALAR-ArmHPC version. The differences between versions do not change with the number of threads, thus we only need one graph.

On average there are no major differences between both SCALAR binaries and both NEON binaries. The speed-up is about 1.25x for NEON versions compared to SCALAR. Nevertheless, each benchmark behaves differently. For example MULADDSUB vectorization speeds up 2x SCALAR performance, however, vectorization does not improve performance in multiple benchmarks, e.g. FLOYD_WARSHALL, GEMM, or VOL3D. We also observe that the compiler can play a major role in performance, in EOS benchmark, SCALAR-GCC is 1.5 times faster than SCALAR Arm HPC compiler.

The selected applications scale really well on the ThunderX processor, it has an average parallel efficiency of 80%. Only two benchmarks (COPY and HPCG) do not scale well and are under 40% of parallel efficiency, while 5 benchmarks scale up to 95%. However, this behavior is due to poor single-core performance and the abundant amount of memory bandwidth that does not become a limiting factor for performance.

Same information about one thread running on ThunderX2 processor is

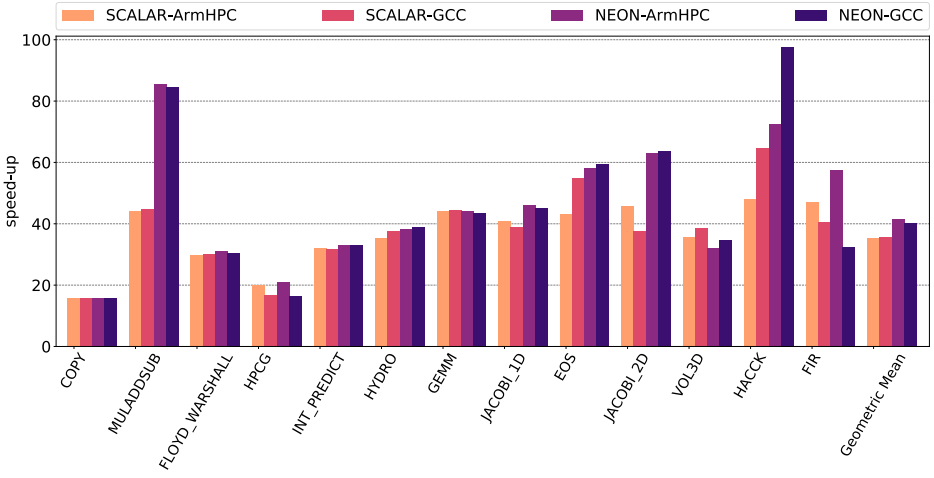


Figure 6.4: Speed-up (w.r.t 1-thread SCALAR-ArmHPC) of all benchmarks with 48 threads on a ThunderX node.

shown in Figure 6.5. We show speed-up for 1 and 32 threads separately because they behave differently. We can see how vectorization speeds up our applications 1.6x for NEON-ArmHPC and 1.63x for NEON-GCC with respect to SCALAR-ArmHPC. A larger gain than that observed for the ThunderX. According to the geometric mean there are no significant differences between compilers. The most remarkable cases are JACOBI_2D, which has a speed up of 2.5x for NEON-ArmHPC binary, and FIR, where SCALAR-GCC is two times slower than SCALAR-ArmHPC. Other interesting cases that we will analyze later in Section 6.5 are: VOL3D in which NEON-GCC speeds up 1.5x with respect to NEON-ArmHPC binary due to a better and shorter code, and EOS in which SCALAR-GCC speeds-up 1.5x with respect to SCALAR-ArmHPC due to a better management of the registers.

Figure 6.6 shows the speed up on 32 cores. In this second graph, we can see how the memory bandwidth wall/ceiling affects our applications. Four applications out of 13 have a parallel efficiency of 95%, but 6 benchmarks are below 50%. Since benchmarks are sorted by their arithmetic intensity, we can see that the ones with an arithmetic intensity (AI) below 0.5, from

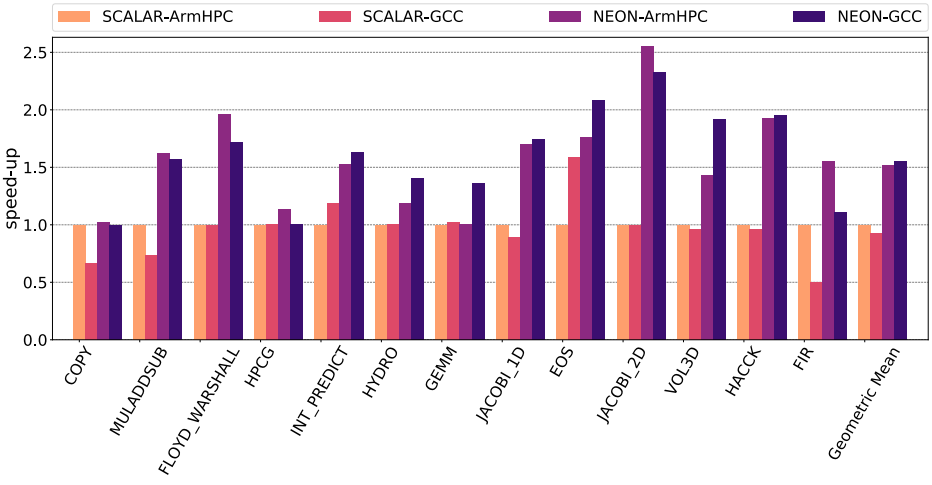


Figure 6.5: Speed-up (w.r.t 1-thread SCALAR-ArmHPC) with one thread on ThunderX2 processor

COPY to EOS, do not benefit from vectorization and do not scale well. While the ones with a higher AI do benefit; JACOBI_2D, VOL3D, HACCK and FIR.

At first sight, Arm HPC Compiler versions scale better than GCC, due to HPCG benchmark, in which GCC has a poor scaling. To compare both compilers, we have to make a deeper analysis on each benchmark, because there is a memory bandwidth bound. There are also some benchmark-dependent behaviours we want to investigate, such as low performance of NEON for JACOBI_1D benchmark (Section 6.2.3) or previous mentioned low scalability of GCC for HPCG the benchmark (see Section 6.2.4).

6.2.1 Low AI Benchmarks

We have seen how AI is related with scalability, therefore we have selected a set of benchmarks that have a low AI. Figure 6.7 shows the speed-up of 6 applications run on ThunderX for 4 compilers. X-axis corresponds to the number of threads and Y-axis corresponds to the speed-up. The speed-up is calculated with respect to SCALAR-ArmHPC binary execution time with only one thread. The benchmarks are sorted by arithmetic intensity,

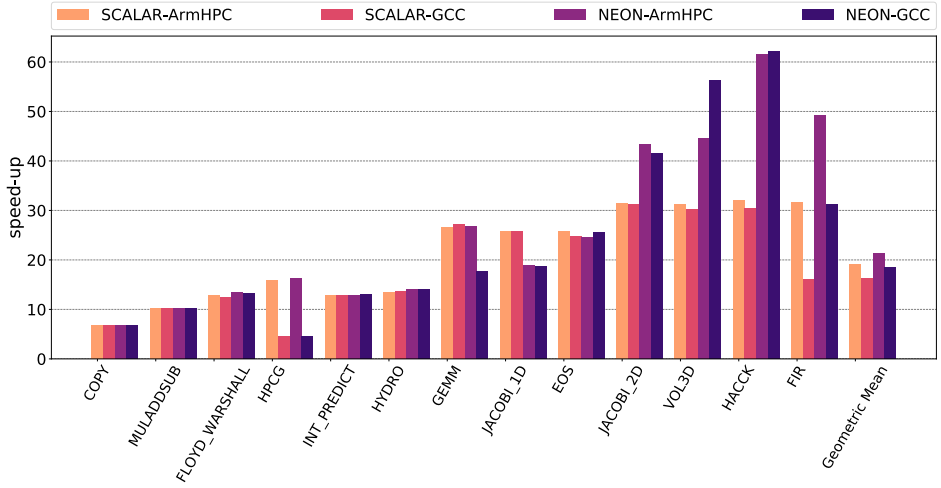


Figure 6.6: Speed-up (w.r.t 1-thread SCALAR-ArmHPC) running 32 threads on ThunderX2 node

thus COPY has the lowest AI, then MULADDSUB, and finally EOS has the highest AI.

The COPY benchmark is clearly limited by memory bandwidth, because it has an AI of 0. In most of the benchmarks there are no differences between binary versions, except for MULADDSUB and EOS in which NEON obtains a speed-up of 2x with respect to SCALAR versions, achieving a speed-up of over 48x. Between these two extremes we find FLOYD_WARSHALL, INT_PREDICT and HYDRO that do not scale perfectly, but achieve a 65% of parallel efficiency. Despite these benchmarks having a low AI, and significant memory bandwidth consumption, they are not memory bound. This is because ThunderX is an in-order core, so any memory stall can significantly reduce single thread performance. All 48 threads barely consume all the available memory bandwidth.

Figure 6.8 shows the same benchmarks running on ThunderX2. All these benchmarks scale well until they saturate memory bandwidth and then stall on that performance. It is easy too see the correlation of the results with the ThunderX2 32 cores roofline model, higher AI means higher memory ceiling and therefore better speed-up. Vectorization only takes an important role

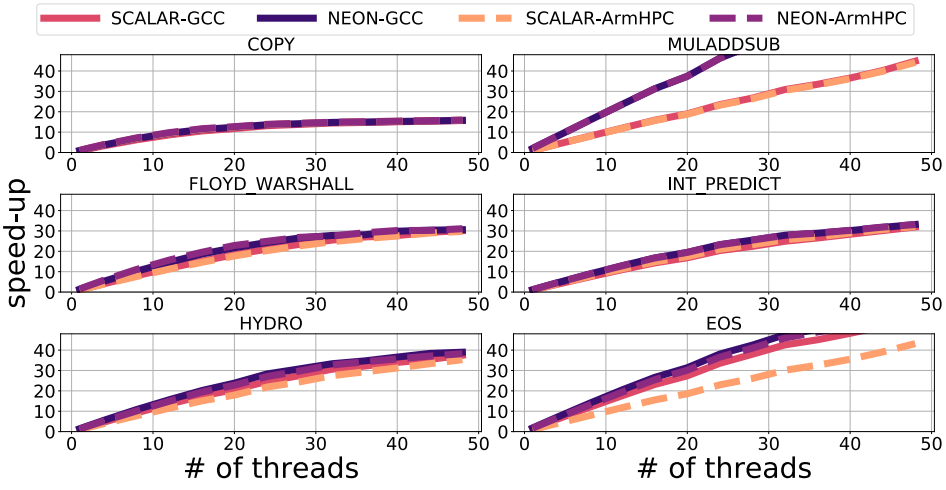


Figure 6.7: Speed-up and performance scalability (w.r.t 1-thread SCALAR-ArmHPC) of low arithmetic intensity benchmarks executing on ThunderX

in EOS benchmark, where NEON versions hit the speed-up ceiling with less cores than the SCALAR versions.

6.2.2 High AI Benchmarks

The second set of applications is formed by the four ones AI above 0.5. Figure 6.9 shows the speed-up of the four binaries respect to 1-thread SCALAR-Arm HPC execution time on ThunderX. In all four benchmarks we can see a very strong scalability, specially for JACOBI_2D and HACCK. Both benchmarks achieve a parallel efficiency of almost 100% for SCALAR binaries.

VOL3D has a slightly worse scalability than the other three benchmarks and does not benefit from NEON. This is due to a high overhead in some NEON instructions like *MLA* and *MLS*, which appear mainly in VOL3D (see Section 6.5.4). These instructions take more cycles than the SCALAR versions [38]. Arm HPC Compiler doubles the performance of GCC in the FIR benchmark thanks to a loop unrolling of two iterations (see Section 6.5.1) .

Same experiments under ThunderX2 (Figure 6.10) show again a great

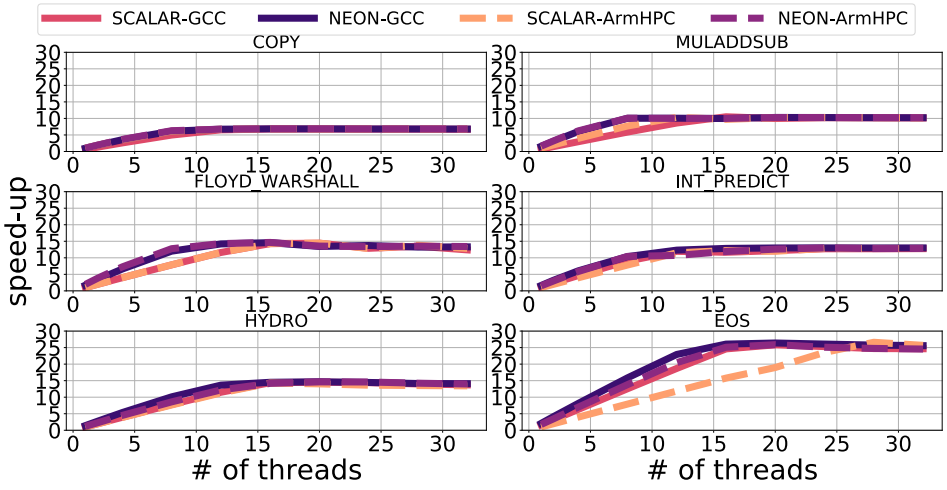


Figure 6.8: Speed-up and performance scalability (w.r.t 1-thread SCALAR-ArmHPC) of low arithmetic intensity applications executing on ThunderX2

scalability for the four applications. In JACOBI_2D, the two SCALAR versions scale well, but the NEON ones have a higher memory bandwidth consumption and exhaust the memory bandwidth. NEON speeds-up VOL3D around 2x for GCC and 1.5x for Arm HPC Compiler. GCC generates a shorter and simpler assembly code for VOL3D, this is explained in Section 6.5.4. Again Arm HPC Compiler beats GCC in FIR benchmark by 2x, in this case the OpenMP runtime has a big impact on performance, we look into this in Section 6.5.3.

6.2.3 JACOBI_1D

The JACOBI_1D benchmark is a special case because NEON versions of the benchmark behave completely different compared to SCALAR versions when ran on ThunderX2, see Figure 6.11. At first sight, we can see how NEON versions scale worse than SCALAR versions at high core counts. NEON versions stall at 22x speed-up while SCALAR versions scale up to 26x. With only one thread, NEON binaries have a 1.65x speed-up with respect to SCALAR-ArmHPC running with one thread. However, with 32 threads NEON is 30.7% slower than SCALAR. In contrast, the execution

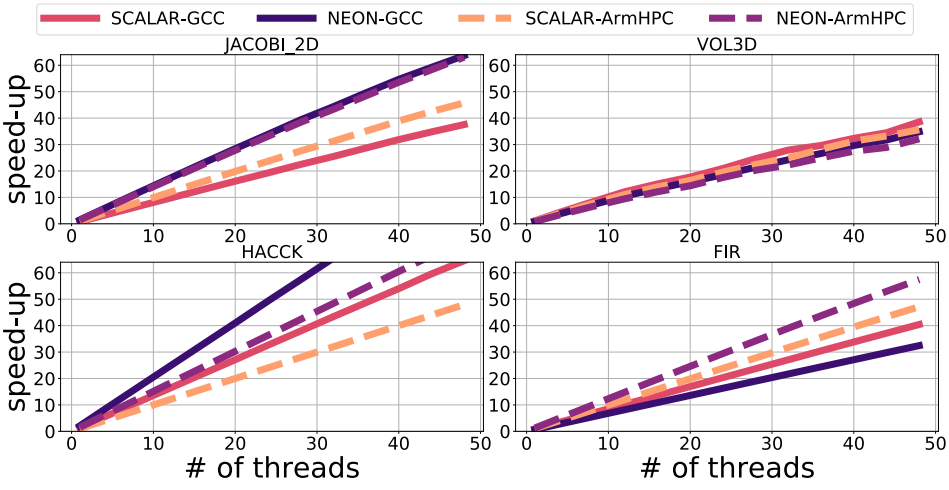


Figure 6.9: Speed-up and scaling (w.r.t 1-thread SCALAR-ArmHPC) of high arithmetic intensity applications executing on ThunderX

of the same binaries on ThunderX does not present this behaviour.

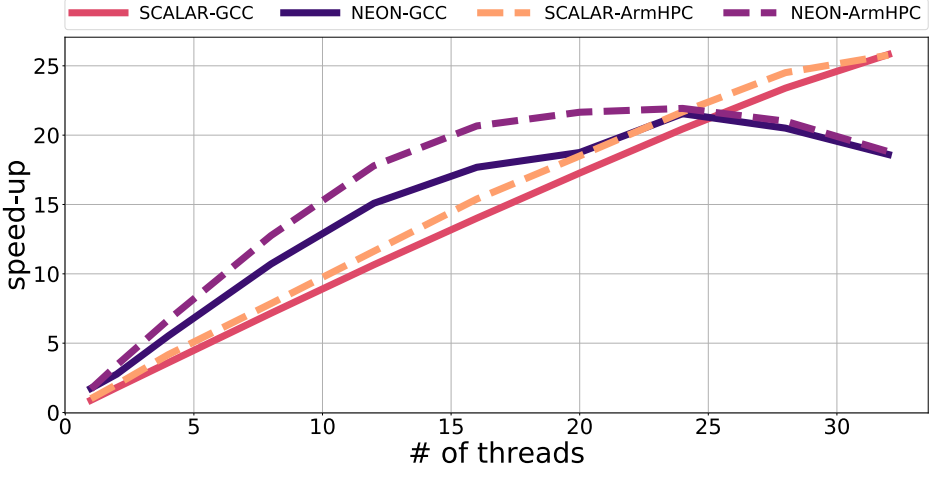


Figure 6.11: Speed-up (w.r.t 1-thread SCALAR-ArmHPC) of JACOBI_1D running on ThunderX2

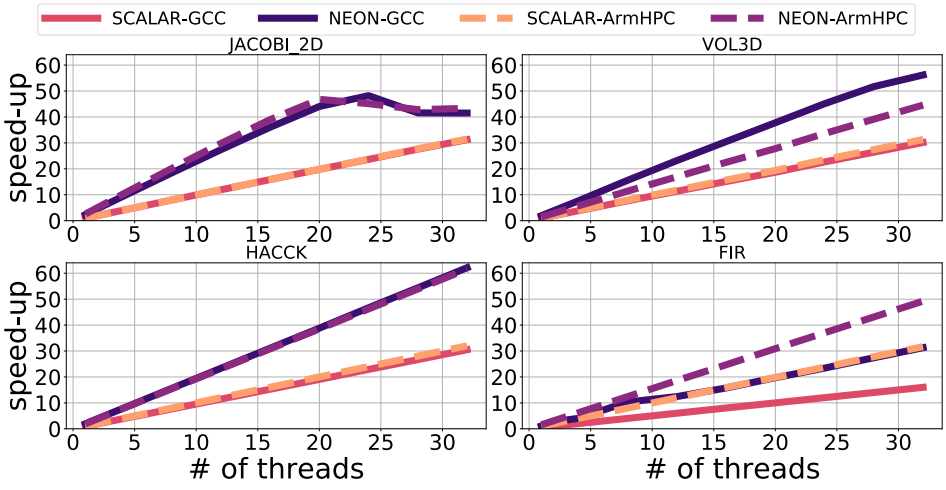
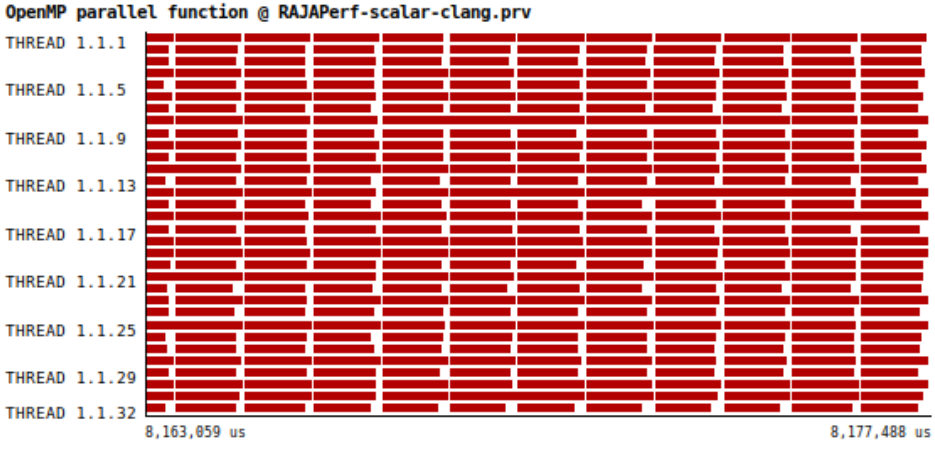
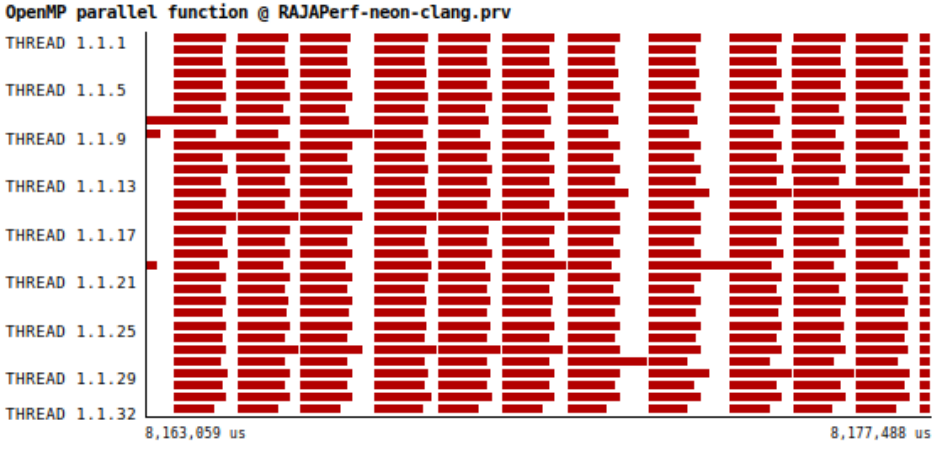


Figure 6.10: Speed-up and scaling (w.r.t 1-thread SCALAR-ArmHPC) of high arithmetic intensity applications in ThunderX2

We have used Extrae tool to take a picture of what is happening during the execution at runtime level. Figure 6.12 depicts two traces of JACOBI_1D obtained with Extrae and visualized with Paraver. Paraver allow us to plot only the runtime function calls executed during the experiment, and also select a time window of all the trace. Each function is represented with a colored rectangle while idle execution is represented as white spaces. Subfigure 6.12a an represents arbitrary but representative, time window execution of SCALAR-ArmHPC running with 32 threads and Subfigure 6.12b represents the same time window but of NEON-ArmHPC running 32 threads. If we compare both traces, it is easy to see that in the NEON version a large part of the execution trace is in idle state. In other words, many threads are not computing anything while they wait for other threads to finish their respective jobs. This behaviour is known as load imbalance and happens when some threads have much more work than others and take more time to finish their work. The problem can be mitigated if the scheduling function is changed to another more suitable.



(a) SCALAR-ArmHPC execution trace



(b) NEON-ArmHPC trace

Figure 6.12: Traces of JACOBI_1D compiled by Arm HPC Compiler with and without SIMD, executed on ThunderX2 with 32 threads (only a few are tagged for clarity). Y-axis represents the number of thread. X-axis represents time, both share the same time window. Each colored rectangle represents one parallel function. White represents idle.

6.2.4 HPCG

As JACOBI_1D, the HPCG benchmark has a special behaviour. The scalability on ThunderX (see Figure 6.13) is poor, it has a parallel efficiency of 40% for 48 threads. This low scalability results from its sparse matrix-vector multiplications that stresses the memory bandwidth. As mentioned before ThunderX has a blocking cache that is very sensitive to HPCG's access patterns.

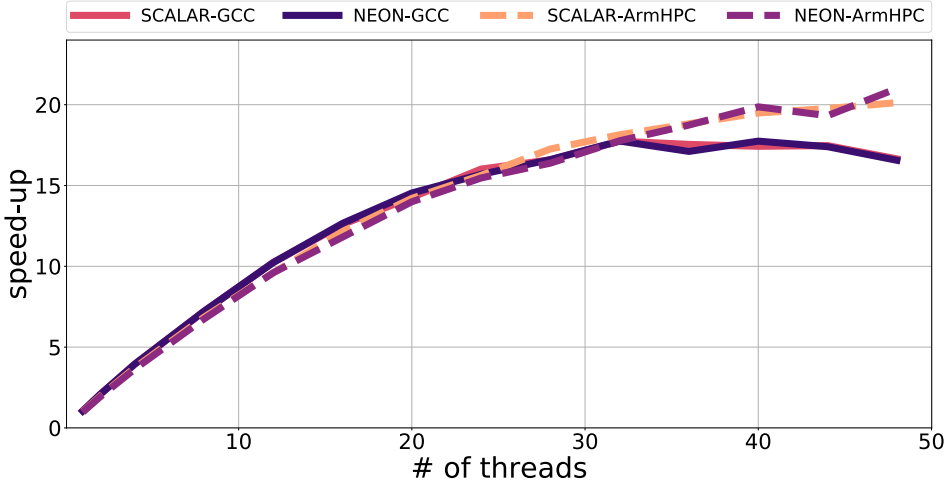


Figure 6.13: Speed-up (w.r.t 1-thread SCALAR-ArmHPC) of HPCG running on ThunderX

When we run this application on ThunderX2 (Figure 6.14), scalability of GCC drops below 12% parallel efficiency, while Arm HPC Compiler achieves a 50% parallel efficiency. GCC is performing almost four times slower than Arm HPC Compiler. Regarding vectorization, GCC cannot benefit from it, while Arm HPC compiler does, but discreetly because SCALAR-ArmHPC version is already memory bound.

As in previous section, the traces obtained using Extrae can help us to explain GCC poor scalability. Subfigures 6.15a and 6.15b show the traces of two executions of HPCG, the former is SCALAR-ArmHPC and the latter is SCALAR-GCC. Again the trace consists on the parallel functions executed by each thread. In this case there are different parallel functions but we

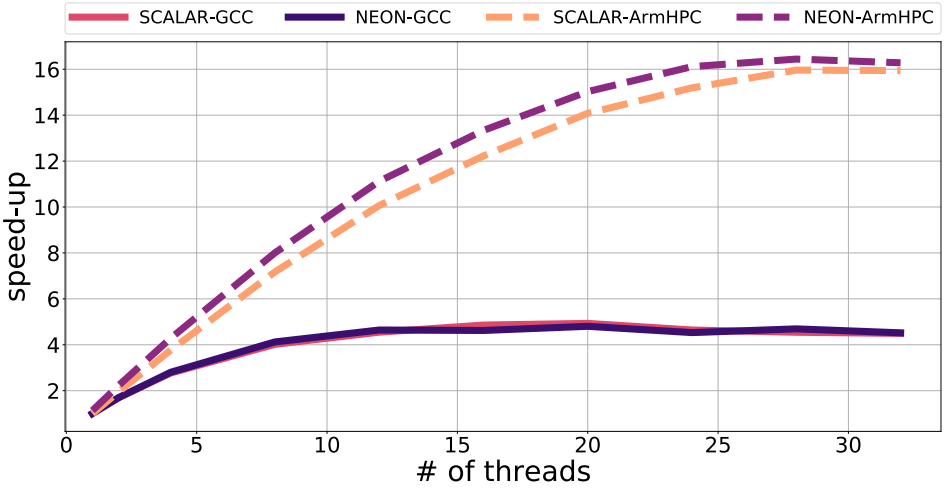
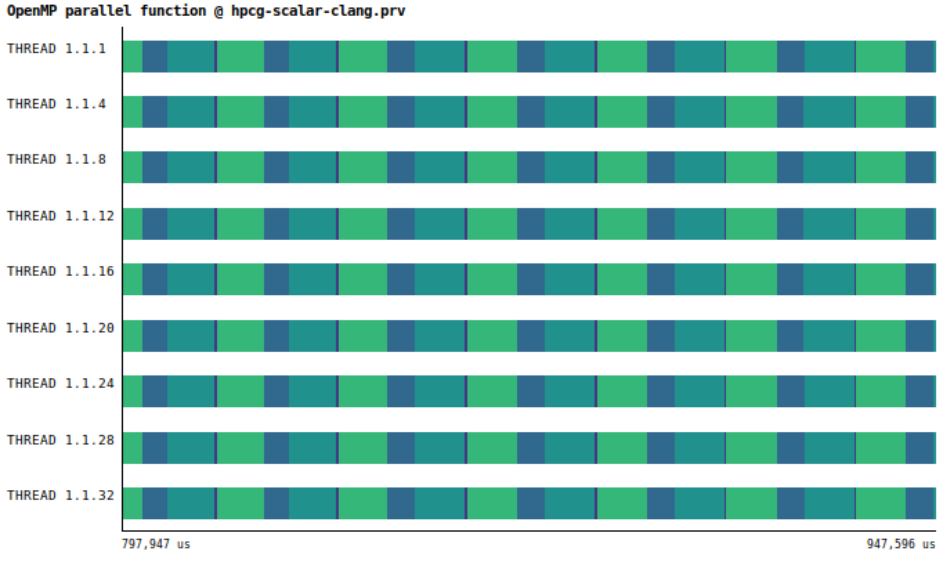


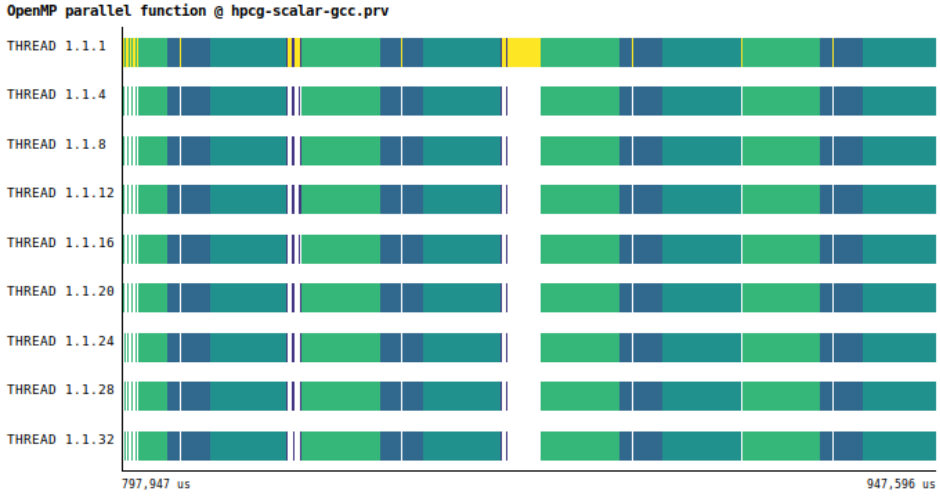
Figure 6.14: Speed-up (w.r.t 1-thread SCALAR-ArmHPC) of HPCG on ThunderX2

do not need to distinguish them because they are not interesting for the analysis. Two parallel functions with the same color are the same parallel function, and white represents idle state. The two most interesting details are that GCC parallel functions are slower than Arm HPC Compiler ones (rectangles are wider), and that GCC has a bigger sequential execution time. This happens when only master thread (thread number one) is working, and the rest of the threads are idle. This sequential function is part of the runtime and it is a call to a shared library.

This sequential function is only a small part of the execution time, and the slow-down of the parallel functions is more important (around 2x), we have decided to look closer at the traces. Paraver usually hides fine grained details when the time window is big, because it interpolates the colors of the functions. Figure 6.16a and 6.16b show a close up of the traces. Paraver shows us with these traces, that the sequential function of the runtime is the main reason that made the total execution of GCC 4x slower. Figure 6.16a shows that master thread (thread one) is idle, and not executing a any function. This happens because sequential function cannot be traced by Extrae, libiomp library is not fully supported by Extrae.



(a) SCALAR-ArmHPC trace



(b) SCALAR-GCC

Figure 6.15: Traces of HPCG compiled by GCC and Arm HPC Compiler, executed on ThunderX2 with 32 threads (only shown the forth thread of every four and number one). Y-axis represents the number of thread. X-axis represents time, the same for both traces. Each colored rectangle represents one parallel function. White represents idle.

(a) SCALAR-ArmHPC trace

(b) SCALAR-GCC trace

Figure 6.16: Traces of HPCG compiled by GCC and Arm HPC Compiler, executed on ThunderX2 with 32 threads (only shown the forth of every four thread and number one). Y-axis represents the number of thread. X-axis represents time, the same for both traces. Each colored rectangle represents one parallel function.

6.3 Test Machines Performance Comparison

This section compares ThunderX and ThunderX2 processors. Figure 6.17 shows the speed-up of the four binaries executed using one thread on both processors. The reference time used is 1-thread SCALAR-ArmHPC running on ThunderX. We use logarithmic scale for the speed-up for clarity. The geometric mean shows a speed up of 7x between both processor for SCALAR versions and 11x for NEON versions. These values are high compared to typical comparisons between in-order and out-of-order cores. Two examples are [44] in which speed-up values range from 1.21X to 3.25x and [45] with speed-ups between 1.66x and 4x. However, we are not considering that ThunderX2 has a 10% higher frequency, and one level of cache more. Furthermore, these papers only consider in-order cores with buffered caches, but ThunderX has a blocking cache that is very sensitive to memory bandwidth consuming benchmarks like RAJAPerf or HPCG.

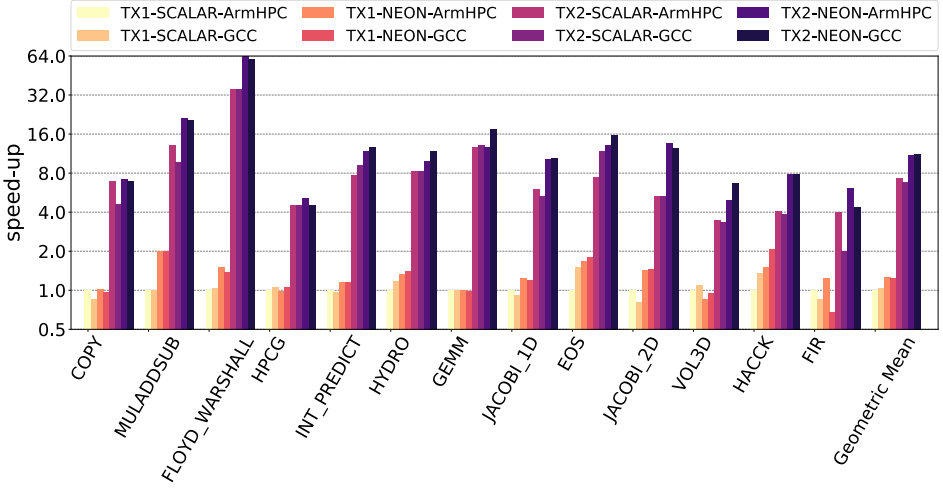


Figure 6.17: Speed-up using the logarithmic scale (w.r.t 1-thread SCALAR-ArmHPC on ThunderX) comparison of one thread between ThunderX (TX1) and ThunderX2 (TX2)

The highest speed-up ThunderX2 achieves is 16X for MULADDSUB and FLOYD_WARSHALL. Again, we remark that ThunderX does not benefit

from NEON as much as ThunderX2 does. In benchmarks with higher arithmetic intensity (see the three benchmarks with highest AI: VOL3D, HACCK and FIR) speed-up reduces to common values like 4x for SCALAR and below 8x for NEON.

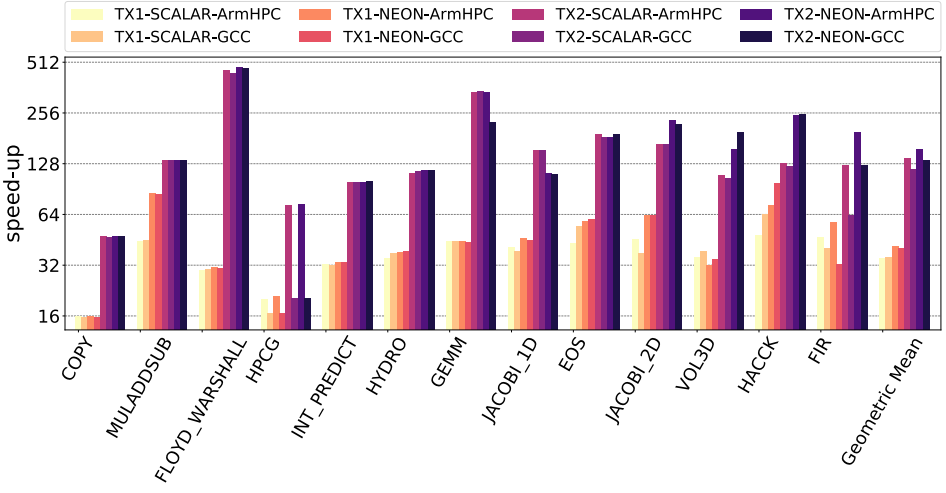


Figure 6.18: Speed-up using the logarithmic scale (w.r.t 1-thread SCALAR-ArmHPC on ThunderX) comparison of 48 threads on ThunderX (TX1) and 32 threads on ThunderX2 (TX2)

When we run the benchmarks with 48 threads for ThunderX and 32 threads for ThunderX2, we see how speed-ups achieved by ThunderX2 over ThunderX reduce to an average of 4x for SCALAR and NEON versions. Again both nodes have a mild benefits from SIMD utilization, due to in-order execution that stalls ThunderX pipelines and memory boundness in ThunderX2. High arithmetic intensity benchmarks are the exception, NEON binaries present significatn speed-ups, as we showed in Section 6.2. The most interesting benchmark is HPCG, in which GCC running on ThunderX2 has the same performance as both binaries of Arm HPC Compiler running on ThunderX, as disscused in Section 6.2.4.

6.4 Power and Energy Efficiency of ThunderX2

Another aspect we want to analyze is the energy consumption, energy efficiency and power efficiency of both processors. However, only ThunderX2 measurements were available. Power measurement tool let us read the average power consumption of the processor during the execution of the region-of-interest (ROI) for each benchmark, see Figure 6.19. All benchmarks are far away of the nominal TDP (180 watts), the highest value is 132 watts achieved by NEON-GCC running VOL3D. Which means that the power consumption is always below the 73.33% of the nominal TDP. Both, HPCG and HACCK are below 100 Watts, very low values if we consider that idle consumption of the processor is 67 watts.

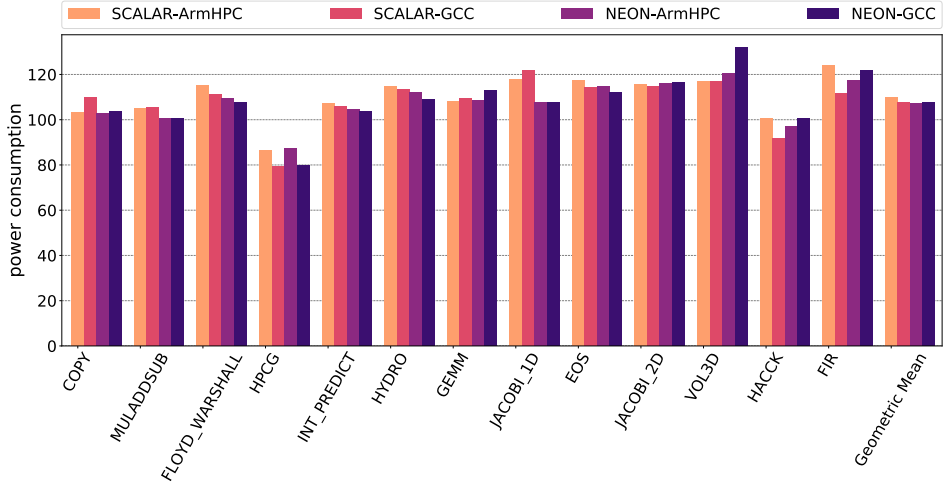


Figure 6.19: Absolute power values in watt for every benchmark running on ThunderX2 processor

With the average power consumption we can calculate the energy consumed multiplying by the execution time. Figure 6.20 shows the normalized energy consumed of all the benchmarks. We normalize the values because each benchmark has a different magnitude and we only want the relationships between different experiments of the same benchmark. First thing we identify is two outliers, one is GCC binaries running the

HPCG benchmark and the second is SCALAR-GCC running the FIR benchmark. Both, experiments have a high execution time (see Figure 6.6 in Section 6.2). A part from this, we can see how NEON reduces the energy consumed a 10% on average and up to a 50% for HACCK. This reduction is higher for the benchmarks with high AI. Low AI benchmarks spend most of the time waiting for data to arrive to the cache, reducing the potential energy savings. Only JACOBI_1D and GEMM consume more energy when NEON is used, and in both cases the performance and scalability of NEON is worse than SCALAR versions.

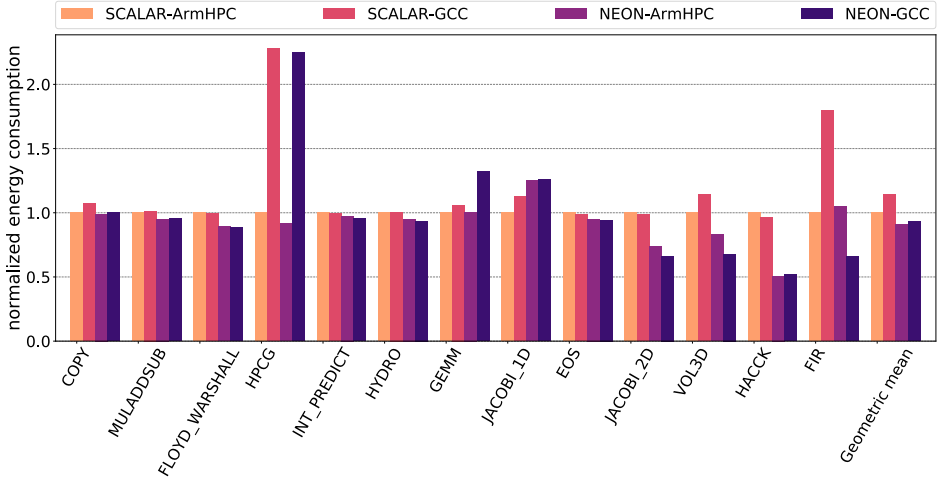


Figure 6.20: Normalized energy consumption (w.r.t 32-threads SCALAR-ArmHPC) for every benchmark running on ThunderX2 processor

Figure 6.21 shows the Energy Delay Product (EDP) of all the benchmarks. EDP is a metric that considers not only the energy consumed but also the execution time. Therefore, experiments that have a higher execution time, but low energy consumption have higher EDP values than experiments that present low execution time and low energy consumption. As in previous figure, we can see HPCG and FIR outliers. If we ignore the outliers SCALAR-GCC version has a higher EDP than SCALAR-ArmHPC version, for instance, 1.3x for VOL3D, 1.2x for JACOBI_1D and 1.1x for COPY and GEMM. In contrast NEON-GCC version has lower EDP than

NEON-ArmHPC version, 60% less for FIR, 38% for VOL3D and 21% for JACOBI˙2D. If we do not take into account HPCG, GCC benefits more than Arm HPC Compiler from vectorization, reducing on average EDP a 10% with respect to SCALAR-ArmHPC. As in previous figure AI is related to EDP reduction for NEON binaries.

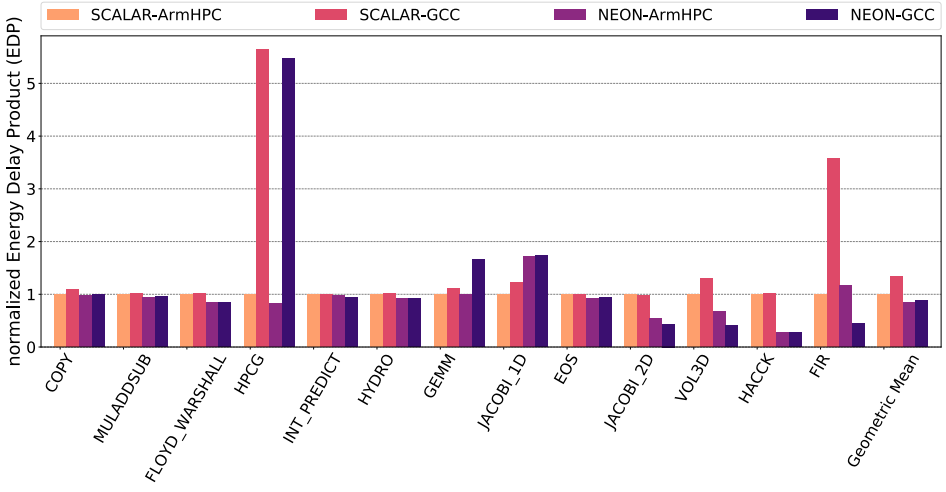


Figure 6.21: Normalized EDP (w.r.t 32-thread SCALAR-ArmHPC) for every benchmark running on ThunderX2 processor

After measuring energy efficiency, we proceed with power efficiency. We use a performance/power metric (GFlops/Watt metric) because processors tend to consume more power in order to achieve a better performance (speculation, wider vector lengths, superscalar, etc), but at same time this higher power consumption harms scalability because higher power and dissipation must be ensured. For this reason we use GFlops/watt, which penalizes extra power consumption to achieve better performance. Figure 6.22 depicts the GFlops per Watt achieved by every application. As in previous sections, benchmarks are sorted by arithmetic intensity. We can appreciate how AI is correlated with performance per watt. In low AI benchmarks, processors waste most of their cycles waiting for data to arrive. In contrast high AI benchmarks have higher values, because they spend most of the time executing floating point operations. In general all

benchmarks have a poor energy efficiency, being the geometric mean below 0.2 GFlops/Watt and all below 1.4 GFlops/Watt. The theoretic maximum GFLOPS/Watt is $2.84 = \frac{512 \text{ peak GFlops}}{180 \text{ watts TDP}}$. This values are far away of the top ten of the Green500 List (17.604 GFlops/Watt [46]), however these HPC systems use graphic cards that have different trade-offs for power consumption. HPCG is the less power efficient benchmark, around 0.003 GFlops/Watt for GCC and 0.007 for Arm HPC. Similar values for HPCG appear in Montblanc Project experiments [16].

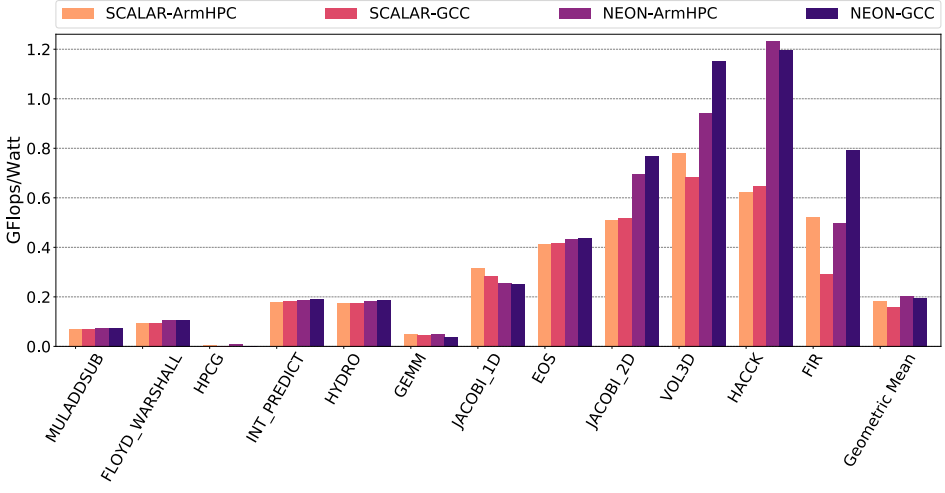


Figure 6.22: GFlops per Watt consumed for every benchmark executed in ThunderX2 node

Because this values by themselves are hard to understand we have done the same experiments with a Skylake cluster that is inside the Dibona infraestructure. Figure 6.23 shows the comparison of performance/watt between ThunderX2 (labeled as TX2) and Skylake (labeled as SKX) processors. We have only compiled SCALAR and 128 bit wide SIMD (SSE) binaries for fairness, despite Skylake has vector instructions of 256 bits and 512 bits. The complete comparison with all vector lengths can be found in appendix C. We consider two compilers for Intel: GCC and ICC. The details of compilation can be found in appendix B. The SCALAR-ICC version is excluded because we cannot force ICC to avoid vectorization

completely.

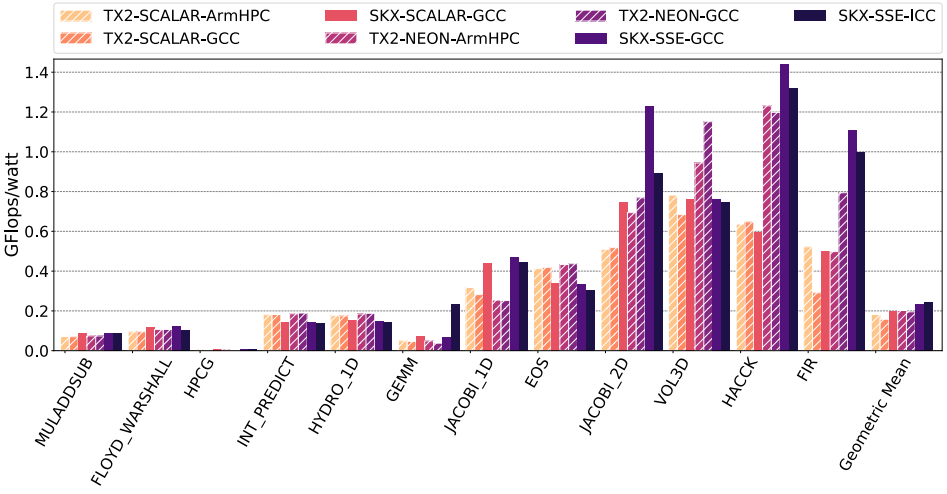


Figure 6.23: GFlops per Watt consumed for every benchmark executed using 32 threads in ThunderX2 (TX2, dashed bars) processor and 28 threads in Skylake (SKX) processor. First three bars are SCALAR versions and the last four are the 128 bits SIMD versions.

On the average both processors are not too far from each other in terms of power efficiency. Skylake is around 10% more efficient in SCALAR versions and a 20% for SIMD versions, because it has more internal resources (2 executions units more than ThunderX) that improve its performance without increasing too much the power consumption. Skylake power efficiency is interesting because its native vector length is 512 bits. This wider vector length implies a higher power consumption although only 128 of the 512 bits are used by binaries.

The power efficiency of both processors is similar for low AI benchmarks, both processors have nearly the same bandwidth. Skylake is clearly more efficient in GEMM, JACOBI˙2D, HACCK and FIR benchmarks. While ThunderX2 is better in EOS and VOL3D kernels. Part of this behaviour could be explained by the two extra execution units Skylake has. Three out of the eight execution units allow SIMD instructions.

6.5 Code Analysis

In this section we analyze the principal characteristics of our binaries focusing the analysis on the assembly code. We have four binary versions: compiling with Arm HPC Compiler and no SIMD support (SCALAR-ArmHPC), compiling with GCC and no SIMD support (SCALAR-GCC), compiling with Arm HPC Compiler and NEON support (NEON-ArmHPC) and compiling with GCC and NEON support (NEON-GCC). This section is structured by code optimizations and different behaviour characteristics binaries have in common. With this analisys we aim to understand how generated code affects performance.

6.5.1 Loop Unrolling and Reduction of Data Access Instructions

After analyzing all assembly code of the benchmarks we have found that Load Pair of Register (*ldp*) is one of the most relevant instructions in the ARMv8 ISA. This instruction loads two registers with two consecutive data elements in memory, instead of using two common load instructions. Arm HPC compiler benefits from this instruction because it ususally applies a two-iteration loop unrolling optimization, and then it reduces the number of memory access instructions using *ldp*. Therefore the total number of instructions is reduced. We can find this two-iteration loop unrolling in COPY, GEMM, JACOBI_1D and FIR in both SCALAR-ArmHPC an NEON-ArmHPC binaries, and in FLOYD_WARSHALL and HACCK only for vectorized versions. But the Arm HPC compiler does not only apply *ldp*-optimization to loop unrolling, it also uses it whenever there are two consecutive accesses, like in EOS and JACOBI_2D. GCC barely uses *ldp* because it rarely performs a loop unrolling unless programmer forces it with a compiler flag or a pragma. GCC uses *ldp* in EOS and JACOBI_1D.

To illustrate how important is this instruction, we will analize the COPY benchmark (See Listings 13a and 13b). In the SCALAR-ArmHPC version, Arm HPC compiler does a loop unrolling and then reduces the number of load instructions using *ldp*, as we mentioned before. While GCC generates a simpler assembly code.

Listing 13: COPY assembly code

```
#LOOP: ldp     x17, x18, [x14, #-8]
       subs    x16, x16, #0x2
       add     x14, x14, #0x10
       stp     x17, x18, [x15, #-8]
       add     x15, x15, #0x10
       b.ne    #LOOP // b.any
```

a: COPY assembly code generated with Arm HPC Compiler

```
#LOOP: ldr     x1, [x20, x0, lsl #3]
       str     x1, [x19, x0, lsl #3]
       add     x0, x0, #0x1
       cmp     x2, x0
       b.ne    #LOOP
```

b: COPY assembly code generated with GCC

Next, we explain how this optimization affects performance. We do not know the characteristics of the L1D cache of ThunderX2, so we have supposed that the size of the block is 64 bytes and the number of entries in the miss status holding register (MSHR) is 16. We know that ThunderX2 has a ROB of 180 entries. Therefore, we can calculate how many iterations of the loop can be stored in the ROB, which is 30 iterations for SCALAR-ArmHPC assembly ($\frac{180}{6} = 30$) and 36 iterations for SCALAR-GCC ($\frac{180}{5} = 36$). Thus, we can calculate how many cache misses are stored in MSHR, because the memory accesses are sequential and we have the hypothetical block size. In each cache block we have 8 elements of data

$$\frac{64 \text{ bytes/block}}{8 \text{ bytes/element}} = 8 \text{ elements per block}$$

Therefore, if *ldp* accesses to a pair double precision floating point numbers, every 4 *ldp* instructions we read or write an entire cache block, so we can compute the number of misses stored in cache MSHRs: 15 out of 16 entries for SCALAR-ArmHPC ($\frac{30}{4} \times 2 = 15$) and 9 cache misses stored in the MSHR for GCC ($\frac{36}{8} \times 2 = 9$). In this example many entries of the MSHR are underutilized by GCC and this can explain the difference in performance between SCALAR-ArmHPC and SCALAR-GCC shown in Figure 6.5, for COPY, JACOBI_1D and FIR. The latter has a speed-up of 1.8x over GCC.

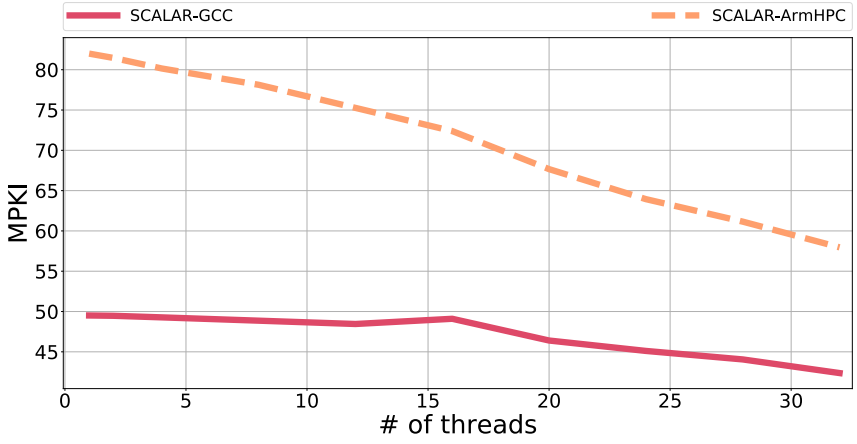


Figure 6.24: Number of misses per kilo instruction of COPY running on ThunderX2

Figure 6.24 shows the number of misses per kilo instructions for both binaries. L3 counters are not available, instead we show the L2 refill events. This event counts all misses at the L2 level plus some other events that imply bringing data blocks from L3 or main memory (prefetch or speculative misses included). It is interesting how SCALAR-ArmHPC has almost twice the ratio of misses than SCALAR-GCC, thus SCALAR-ArmHPC stresses more the memory and feeds the cores faster.

We can compute this MPKI values theoretically. For 1000 instructions we have 167 iterations of the SCALAR-ArmHPC assembly loop and 200 for SCALAR-GCC. As we have mention before, there are two misses every 4 iterations of the SCALAR-ArmHPC assembly loop, so we have 83 MPKI. If we repeat this computation for SCALAR-GCC, there are two misses every 8 iterations of SCALAR-GCC assembly loop, which is 50 MPKI. We can appreciate that the theoretical values are nearly the same as the observed ones in Figure 6.24 for one core. This number decreases when using more cores because the number of instructions executed out of the loop in order to control threads execution increases.

Both NEON binaries have twice the MPKI ratio of SCALAR due to vector instructions. Therefore, they fill the MSHRs registers, as SCALAR-ArmHPC, but do not achieve a higher performance because the MSHRs

are already saturated.

6.5.2 Keeping Constants Stored in Registers

Another important difference between GCC and Arm HPC compiler is the management of constants. GCC tries to keep constants stored in registers whenever is possible, so constants are loaded only once before start executing the kernel loop. On the other hand, Arm HPC Compiler tries to use the minimum number of registers, so in every iteration of the loop constants are loaded from memory. Those extra loads should only have impact on in-order cores like ThunderX, where they stall the pipe line.

Listing 14: EOS assembly code

```
#LOOP:
#Load  ldr     d0, [x21]
       lsl     x14, x9, #3
       ldp     d4, d2, [x13, #-16]
       ldr     d5, [x13]
       ldr     d6, [x10, x14]
       ldr     d7, [x11, x14]
       fmadd   d2, d4, d0, d2
#Load  ldr     d1, [x27]
       ldur    d3, [x13, #-24]
       ldp     d4, d16, [x13, #16]
       fmadd   d6, d7, d0, d6
#Load  ldr     d7, [x26]
       fmadd   d2, d2, d0, d5
       ldr     d5, [x13, #8]!
       fmadd   d0, d6, d0, d3
       cmp     x9, x8
       add     x9, x9, #0x1
       fmadd   d3, d5, d7, d4
       fmadd   d3, d3, d7, d16
       fmadd   d2, d3, d1, d2
       fmadd   d0, d2, d1, d0
       str     d0, [x12, x14]
       b.lt    #LOOP
```

a: EOS assembly code generated with Arm HPC Compiler

```
#LOOP: ldp     d4, d0, [x0, #24]
       ldp     d1, d5, [x0, #40]
       add     x0, x0, #0x8
       ldr     d2, [x0]
       ldr     d3, [x19, x1, lsl #3]
       fmadd   d0, d10, d0, d1
       ldr     d1, [x0, #8]
       fmadd   d2, d8, d2, d1
       ldr     d1, [x20, x1, lsl #3]
       fmadd   d1, d8, d1, d3
       ldur    d3, [x0, #-8]
       fmadd   d0, d10, d0, d5
       fmadd   d2, d8, d2, d4
       fmadd   d1, d8, d1, d3
       fmadd   d0, d9, d0, d2
       fmadd   d0, d9, d0, d1
       str     d0, [x21, x1, lsl #3]
       add     x1, x1, #0x1
       cmp     x2, x1
       b.ne    #LOOP
```

b: EOS assembly code generated with GCC

As an example we show the code of the EOS benchmark (see Listing 14), but this can be seen also in the HYDRO benchmark. In SCALAR-ArmHPC code we have remarked the extra loads with label #Load. We have counted the number of instruction of both codes and SCALAR-ArmHPC has a loop length of 23 instructions, and SCALAR-GCC has 20. This a negligible difference, but in Section 7.2.2 we will see that it is important for SVE assembly.

6.5.3 Library and Runtime Performance

We have seen in Sections 6.2.3 and 6.2.4 how the OpenMP runtime implementation libraries, libomp for Arm HPC compiler and libgomp for GCC, have an important impact in performance. Listing 15 presents the assembly code of SCALAR-ArmHPC and SCALAR-GCC for MULADDSUB. We can see that both codes are almost the same. But if we return to Figure 6.5, SCALAR-Arm-HPC is faster than SCALAR-GCC. We belive that this speed-up is caused by the runtime instead of the main loop.

Listing 15: MULADDSUB assembly code

```
#LOOP: lsl     x10, x9, #3
       ldr     d0, [x23, x10]
       ldr     d1, [x22, x10]
       cmp     x9, x8
       add     x9, x9, #0x1
       fmul    d2, d1, d0
       fadd    d3, d1, d0
       fsub    d0, d0, d1
       str     d2, [x21, x10]
       str     d3, [x20, x10]
       str     d0, [x19, x10]
       b.lt    #LOOP  // b.tstop
```

a: MULADDSUB assembly code generated with Arm HPC Compiler

```
#LOOP: ldr     d0, [x5, x0, lsl #3]
       ldr     d1, [x6, x0, lsl #3]
       fmul    d3, d0, d1
       fadd    d2, d0, d1
       fsub    d0, d0, d1
       str     d3, [x2, x0, lsl #3]
       str     d2, [x3, x0, lsl #3]
       str     d0, [x4, x0, lsl #3]
       add     x0, x0, #0x1
       cmp     x1, x0
       b.ne    #LOOP  // b.any
```

b: MULADDSUB assembly code generated with GCC

GCC allows programmers to link the assembly code against other

libraries that implement OpenMP runtime, which is the case of the libomp library. We have compiled a new binary following this technique, so our binary mixes GCC generated code with the Arm HPC Compiler runtime library. With this binary we want to test if libomp speeds-up GCC assembly code. However, we find that in most cases it does not improve GCC execution time, only in the NEON version of FIR it speeds-up a 1.3x, see SCALAR-CROSS and NEON-CROSS in Figure 6.25. This single positive result pushes us to think, that GCC and Arm HPC Compiler not only use different runtime libraries, but also use them in a different way.

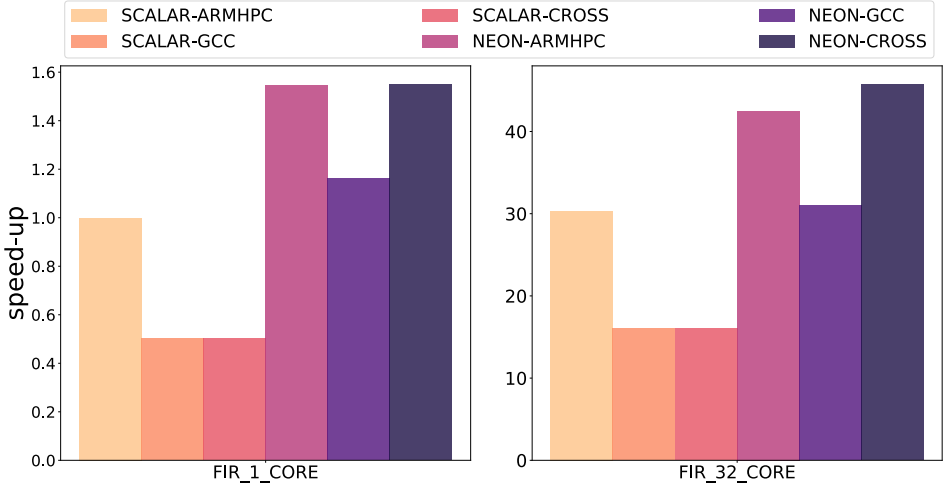


Figure 6.25: Speed up (w.r.t 1-thread SCALAR-ArmHPC) of cross compiling GCC against Arm HPC Compiler runtime in ThunderX2 with 1 thread (left) and 32 threads (rigth)

6.5.4 VOL3D

Finally we will study VOL3D benchmark in detail. In Sections 6.2 and 6.2.2 it has been shown how NEON-GCC binary outperforms NEON-ArmHPC binary when we run VOL3D (See Figures 6.5 and 6.6). To explain this speed-up we have compared both codes using labels to make a bidirectional association. This labels let us remark sections of code that are present in both versions and parts that are distinct.

We do not list all the code because it is too long and some parts are repetitive. Both codes have 3 sections one where the operands are loaded, the second where the kernel equation is resolved with the operands, and a third one where the result is stored and the indices are updated (See Listings 16a and 16b). In the first section both codes are almost identical, they execute 24 loads (label #T1) corresponding to the 24 operands needed for the kernel. This loads use the same addressing method of base address register plus index register.

Section two is different for both compilers, despite they are computing the same equation. Arm HPC Compiler uses 6 instructions more than GCC (GCC uses 59 instructions), this is 10% more instructions. Five of these instructions correspond to *fneg*, labeled with #T2. This instruction negates a register, and Arm HPC Compiler uses it to execute a subtraction with *fmla* (Floating-point fused multiply-add to accumulator). GCC instead of negating the content of a register and then using *fmla*, it uses *fmls* (Floating-point fused multiply-subtract from accumulator). Arm HPC compiler also uses *fmls*, but in other parts of the code. The other extra instruction used by the Arm HPC compiler is a store, labeled with #T3, which is not necessary at all. We also want to remark that, there is a constant in the original kernel that multiplies the result in order to normalize it. GCC uses a register to store this constant, and Arm HPC compiler loads it on every iteration, label #T6.

In the third section both codes store the result, label #T5, and update the addressing registers, label #T4. Arm HPC compiler updates the base address instead of the index. Thus, GCC code only does one "add" operation and Arm HPC compiler code does it **24 times**.

Finally, if we count the number of instructions executed for both versions in each iteration we have that Arm HPC compiler generates 120 instructions while GCC only generates 88, which means the Arm HPC Compiler generates 36% more instructions than GCC. This difference is likely the reason ro the differences in performance seen in Figures 6.5 and 6.6.

Listing 16: VOL3D assembly code

```
;;;;;; Operands load

#LOOP: ldur    x10, [x29, #-168]
       subs    x21, x21, #0x2
       lsl     x10, x10, #3
       ldr     q0, [x12, x10]
       ldr     q1, [x13, x10]
#T1    ldr     q2, [x14, x10]
       ldr     q3, [x15, x10]

;;;;;; Kernel body execution

       fsub    v1.2d, v0.2d, v1.2d
       fsub    v2.2d, v0.2d, v2.2d
       fsub    v0.2d, v0.2d, v3.2d
       ........
       fmul    v30.2d, v21.2d, v3.2d
       fmul    v31.2d, v7.2d, v3.2d
       fadd    v27.2d, v19.2d, v6.2d
       fmls    v30.2d, v23.2d, v2.2d
#T2    fneg    v31.2d, v31.2d
       fadd    v28.2d, v25.2d, v20.2d
       fmul    v29.2d, v21.2d, v17.2d
       fmla    v31.2d, v17.2d, v2.2d
       ........
#T3    str     q27, [x22, x10]
#T6    ldr     d0, [x19]
       fmul    v0.2d, v27.2d, v0.d[0]

;;;;;; Store Result

#T4    add     x23, x23, #0x10
#T4    add     x20, x20, #0x10
#T4    add     x27, x27, #0x10
#T4    add     x11, x11, #0x10
#T4    add     x22, x22, #0x10
#T5    str     q0, [x22, x10]
       b.ne    #LOOP          // b.any
```

a: VOL3D assembly code generated with Arm HPC Compiler

```
;;;;;; Operands load

       mov     x0, #0x0   // initial index #0
#LOOP: ldr     q19, [x24, x0]
       ldr     q9,  [x12, x0]
#T1    ldr     q23, [x25, x0]

;;;;;; Kernel body execution

       fsub    v7.2d, v20.2d, v7.2d
       fsub    v24.2d, v24.2d, v27.2d
       fsub    v21.2d, v18.2d, v21.2d
       fsub    v3.2d, v3.2d, v1.2d
       fsub    v6.2d, v18.2d, v6.2d
       fmul    v0.2d, v4.2d, v23.2d
        ........
       fmul    v7.2d, v27.2d, v21.2d
       fmla    v0.2d, v6.2d, v16.2d
#T2    fmls    v3.2d, v18.2d, v26.2d
       fadd    v4.2d, v4.2d, v24.2d
       fmla    v0.2d, v4.2d, v3.2d
       fmls    v6.2d, v2.2d, v1.2d
       fmul    v2.2d, v2.2d, v19.2d

;;;;;; Store Result

#T5    str     q0, [x2, x0]
#T4    add     x0, x0, #0x10
       cmp     x4, x0
       b.ne    #LOOP
```

b: VOL3D assembly code generated with GCC

Chapter 7

SVE Analysis

This chapter details the experiments and preliminary analysis that we have carried on SVE vector instructions. In particular, we have measured how different vector lengths can reduce the number of instructions and what differences exist between both compilers. Finally we have analyzed the assembly code understand these differences.

7.1 Instruction Reduction

One of the most important features of the Scalable Vector Extension, defined in the ARMv8.2 ISA, is the adoption of a Vector-Length Agnostic (VLA) programming model, which means that vector instructions adapt to the available vector length. Therefore, with VLA we do not need different instruction sets in order to express different vector lengths. In this section, we want to answer the question of how different vector lengths affect our SVE binaries. However, as we stated on Section 1.5.4, we have only one available tool to answer this question before the project deadline. This tool is Arm Instruction Emulator (ArmIE), and we already explained how it works in Section 4.3.5. ArmIE allows us to count the total number of instructions that would have been executed by an ARMv8.2 processor during the execution of a benchmark. We have evaluated six different binaries for each benchmark:

7.1.1 Number of Instructions Executed

In this subsection, we present the experiments that measure the number of instructions executed in each benchmark.

Figure 7.1 shows the normalized number of instructions obtained from

Binary	Compiler	SIMD extension supported	Vector length (bits)
SCALAR-ArmHPC	Arm HPC Compiler	NONE	- -
SCALAR-GCC	GCC	NONE	- -
NEON-ArmHPC	Arm HPC Compiler	NEON	128
NEON-GCC	GCC	NEON	128
SVE128-ArmHPC	Arm HPC Compiler	SVE	128
SVE128-GCC	GCC	SVE	128
SVE256-ArmHPC	Arm HPC Compiler	SVE	256
SVE256-GCC	GCC	SVE	256
SVE512-ArmHPC	Arm HPC Compiler	SVE	512
SVE512-GCC	GCC	SVE	512
SVE1024-ArmHPC	Arm HPC Compiler	SVE	1024
SVE1024-GCC	GCC	SVE	1024
SVE2048-ArmHPC	Arm HPC Compiler	SVE	2048
SVE2048-GCC	GCC	SVE	2048

Table 7.1: List of experiments and its characteristics: compiler used, vector extenion that supports, vector length in bits of the extension.

the execution of SCALAR, NEON, and SVE128 binaries. Instructions are normalized with respect to the SCALAR-GCC. This figure illustrates the number of instructions SVE executes for the same vector length NEON has. We use a logarithmic scale with base 2 for the Y-axis, because NEON and SVE128 instructions work with two double precision floating point elements. On average NEON reduces the number of instructions by 47% for NEON-GCC, and by 45% for NEON-ArmHPC with respect to SCALAR versions. It is easy to see the effects of the two-iteration loop unrolling in the following benchmarks: COPY, HACCK and FIR for SCALAR-ArmHPC and COPY, FLOYD_WARSHALL, JACOBI_1D, HACCK and FIR for NEON-ArmHPC. In these experiments, the Arm HPC Compiler binary employs a significantly lower number of instructions than GCC.

Similarly, SVE128- GCC achieves a slightly lower 45%, that can be

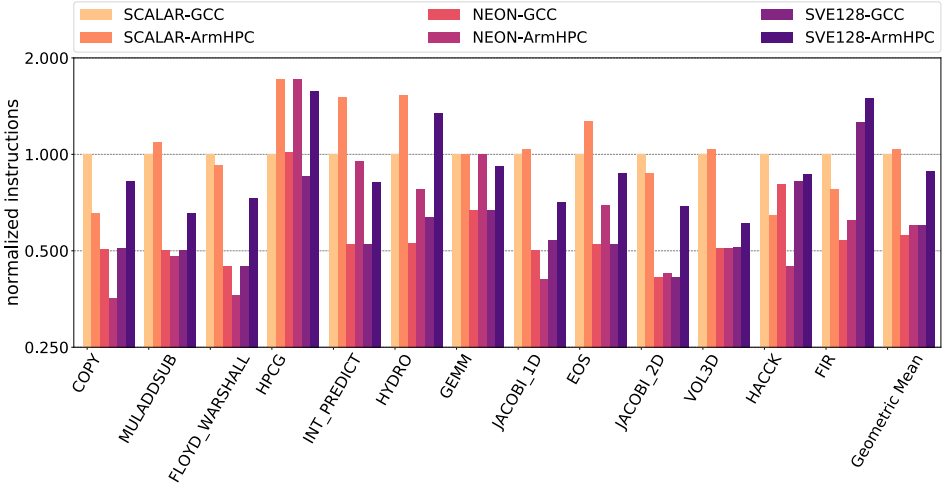


Figure 7.1: Normalized number of instructions executed (w.r.t SCALAR-GCC) using ArmIE. Y-axis uses a logarithmic scale. Only SCALAR, NEON and SVE-128 versions. All experiments use 32 OpenMP threads

explained by high number of instructions executed in FIR and HACCK benchmarks. In contrast, SVE128-ArmHPC gets a poor 27%, this increase of instructions with respect to NEON-ArmHPC can be seen in all benchmarks except GEMM, we will see in Section 7.2 that this is caused by some unnecessary extra instructions and the lack of loop unrolling optimizations.

After comparing SVE128 with SCALAR and NEON versions, we follow with the remaining SVE experiments. We have had to split the experiments in two figures (Figures 7.2 and 7.3) because the high number of experiments made graphs difficult to interpret. Both figures show the normalized number of instructions for SVE binaries with vector lengths of 128, 256, 512, 1024 and 2048 bits. Both graphs share same geometric means. Moreover, we omit SCALAR and NEON experiments for clarity, because all experiments are normalized with respect to SCALAR-GCC version. Therefore, 1.0 value represents the same number of instructions that SCALAR-GCC executed.

We can see how in almost all benchmarks, the number of instructions is

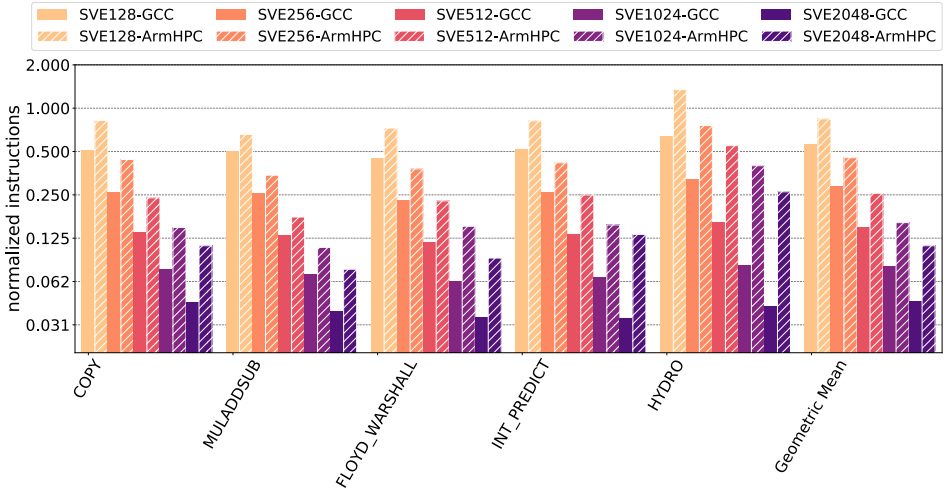


Figure 7.2: Normalized number of instructions executed (w.r.t SCALAR-GCC) using ArmIE. Y-axis uses a logarithmic scale. There are 5 different experiments with vector lengths of 128, 256, 512, 1024 and 2048 bits. All experiments use 32 OpenMP threads

reduced by a factor of 2x with respect to the previous vector length. But, there is one benchmark that stands out above the rest, it is FIR benchmark. This benchmark cannot reduce the number of instructions from 1024 to 2048 bits for both compilers. The way compilers vectorize this benchmark is not optimal, because it contains a inner loop that has 16 iterations, the same number of double precision floating point data elements that fit in 1024 bits. Therefore, for larger vector lengths there is a part of the vector registers that is not used. Further explanations can be found in Section 7.2.

Aside from this outlier, we can appreciate how instruction count is halved for each 2x increase of the vector length. A perfect example is INT_PREDICT's experiments with the SVE-GCC versions, it achieves reductions of 50% for 128 bits, 75% for 256 bits, 87.5% for 512 bits, 93.8% for 1024 bits and 96.9% for 2048 bits. The best reduction is achieved by SVE2048-GCC running JACOBI˙2D with a 97.2%. This value is higher than the expected maximum reduction performed by a vector length of 2048

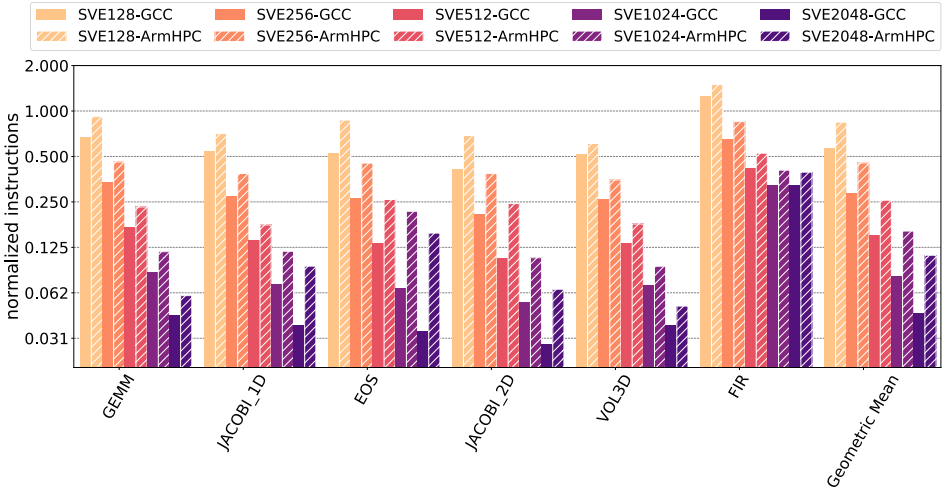


Figure 7.3: Normalized number of instructions executed (w.r.t SCALAR-GCC) using ArmIE. Y-axis uses a logarithmic scale. There are 5 different experiments with vector lengths of 128, 256, 512, 1024 and 2048 bits. All experiments use 32 OpenMP threads

bits ($\frac{32-1}{32} = 96.87\%$). The explanation is that SCALAR-GCC version uses much more instructions than NEON and SVE128-GCC bits experiments, they both achieve a reduction higher than the expected ($\frac{2-1}{2} = 50\%$) with a 59%. Arm HPC Compiler versions execute always a higher number of instructions than GCC versions. We can see how on average, SVE2048-GCC yields a reduction of 95.8% and SVE2048-ArmHPC achieves a lower reduction of 91.4%, both with vector lengths of 2048 bits. With these values seen through logarithmic scale, we can conclude that Arm HPC Compiler executes on average, twice as many instructions than GCC.

We have set apart HPCG and HACCK benchmarks because both benchmarks are significantly different than the rest and put a lot of noise into the geometric mean. In the former, both compilers can only vectorize a small part of code, and we cannot select a region of interest with ArmIE. In the latter, the GCC is not able to vectorize the main loop (GCC prompts that there are not enough data references in basic block), while Arm HPC Compiler is able to perfectly vectorize the loop. We can see these two

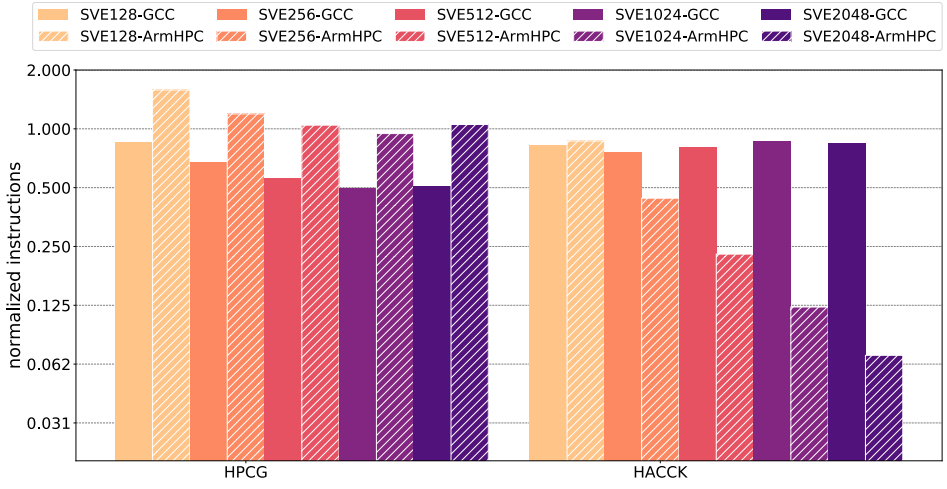


Figure 7.4: Normalized number of instructions executed (w.r.t SCALAR-GCC) using ArmIE. Y-axis uses a logarithmic scale. There are 5 different experiments with vector lengths of 128, 256, 512, 1024 and 2048 bits. All experiments use 32 OpenMP threads

behaviours in Figure 7.4.

7.1.2 Ratio of SVE Instructions

Finally we have use the ArmIE feature that counts the number of SVE instructions emulated, and we have computed the ratio of SVE instructions emulated with respect to the total number of instructions. Figures 7.5 and 7.6 depict the rate of SVE instructions emulated with respect to the total number of instructions. Again, we split the benchmark collection in two sets, and we do not consider either HPCG and HACCK benchmarks for the computation of the geometric mean.

When running the HACCK benchmark, the first thing we can observe is that GCC executes almost no SVE instructions (0.07% of the total instructions). We can conclude that the GCC binary is not vectorized for HACCK benchmark. In the HPCG benchmark, SVE code has at most a weight of 48% from the total number of instructions, which is a low ratio. We also remark FIR benchmark, because there is no difference between

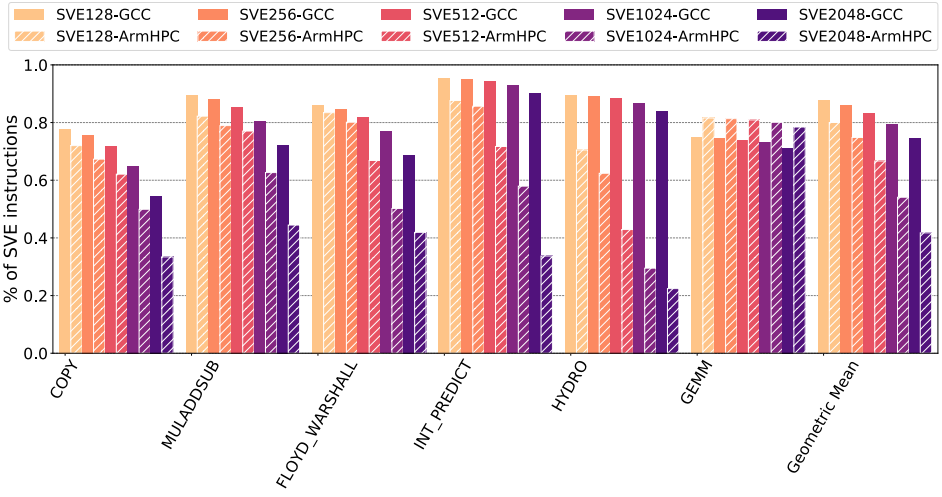


Figure 7.5: Normalized number of instructions executed (w.r.t SCALAR-GCC) using ArmIE. There are 5 different experiments with vector lengths of 128, 256, 512, 1024 and 2048 bits. All experiments use 32 OpenMP threads

1024 and 2048 bits.

On average, GCC is able to generate assembly code that has a higher rate of SVE instructions than Arm HPC Compiler. For 128 bits, GCC is able to vectorize the 87.5% of the code and Arm HPC compiler an 80%. Therefore, when we duplicate the vector length, the weight that SVE instructions lose (two iterations become one) is less for GCC than for the Arm HPC Compiler. From 128 to 2048 bits, GCC loses only a 12.5% of the SVE instructions weight, while Arm HPC Compiler loses 38.26%.

7.1.3 Comparision Between SVE and x86_64

In order to understand how good are these code reductions, we have performed experiments on Skylake in order to compare SVE against Skylake instruction set. We can compare the normalized number of instructions between both architectures because we normalize with respect to binaries without SIMD support, SCALAR-GCC compiled with the ARMv8 ISA and SCALAR-GCC compiled with the x86˙68 ISA. The

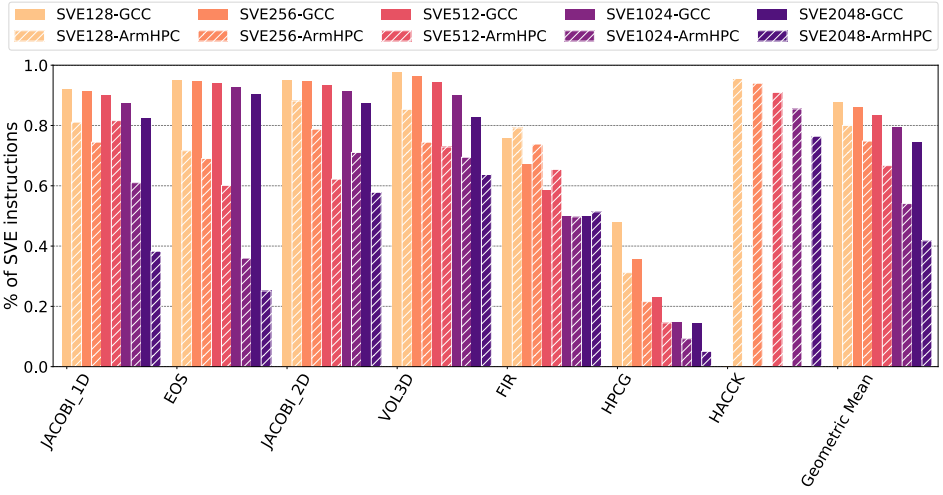


Figure 7.6: Normalized number of instructions executed (w.r.t SCALAR-GCC) using ArmIE. There are 5 different experiments with vector lengths of 128, 256, 512, 1024 and 2048 bits. All experiments use 32 OpenMP threads

Skylake microarchitecture implements 3 different vector lengths: 128 bits with SSE, 256 bits with AVX2, and 512 bits with AVX512. We have used the same binaries of Section 6.4, using GCC and ICC compilers. The main difference with Arm SVE, is that for every version we have to compile a new binary, this gives some extra information to compilers about which vector length they are compiling for. Therefore, both ICC and GCC optimize the assembly code they generate for each vector length. Right now, compiling SVE with optimization for an specific vector length is not allowed in GCC and Arm HPC Compiler, but in the future it is expected to be supported.

Figure 7.7 shows the normalized number of instructions executed by each binary version with respect to SCALAR-GCC version. On average SSE (128 bits) and AVX2 (256 bits) achieve a reduction of 45.3% and 75.3% respectively. In contrast AVX512 only reduces the number of instructions by 84%, a slightly lower than the expected 88%. We should remark that instruction reduction is highly benchmark-dependent. For example INT_PREDICT, HYDRO, JACOBI_1D, VOL3D and FIR benchmarks

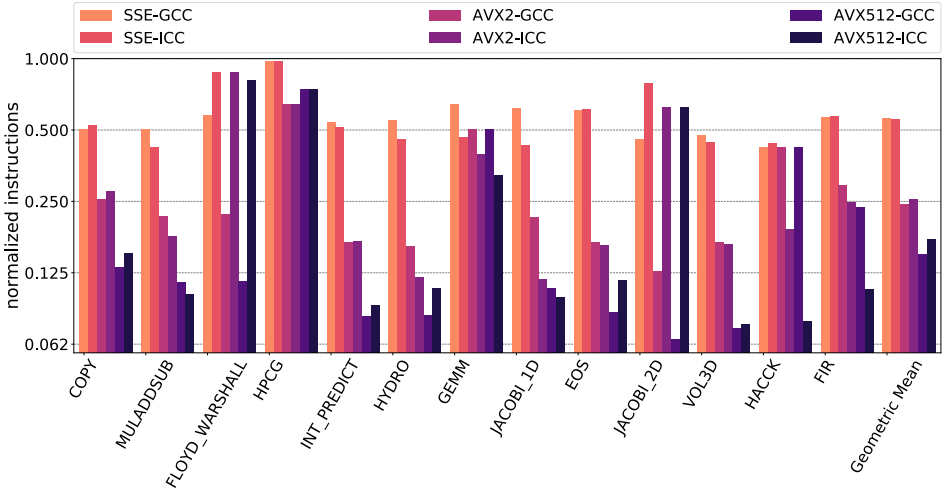


Figure 7.7: Normalized number of instructions executed (w.r.t SCALAR-GCC) running with 32 threads in Skylake processor. There are 3 different types of experiments with vector lengths of 128, 256, and 512 bits.

compiled with AVX512 can reduce the number of instructions above 88%, which is more than what one should expect for a vector length of 512 bits. As explained before, this is due to specific optimizations done for a vector length. Others benchmarks like FLOYD_WARSHALL, GEMM, HPCG, JACOBI_2D or HACCK present a large gap between the number of instructions ICC and GCC binries execute. In these cases, one or both compilers choose to generate code with smaller vector lengths, with respect to the given flags, because they consider that the performance would be better.

7.1.4 Final Remarks

We can conclude that SVE is capable of reducing the number of instructions with on par results as x86_64, but with a smaller instruction set. However, in the future ARMv8.2 compilers should be able to specify the vector length and unlock further optimizations. Regarding the quality of the code of the compilers, both Arm HPC Compiler and GCC should improve its assembly code generation, the former should improve reducing the size of instructions

in the loops and the latter should review the cost model in order to vectorize complex loops like HACCK benchmark. Finally we should remark that less instructions not always leads to better performance. However, we have already seen in Section 6.5.1 that shorter codes have an impact on core structures utilization.

7.2 Code Analysis

In this section we analyze the principal differences between both compilers and detail some assembly code that explains the behaviour shown in the previous section.

7.2.1 Data Dependencies

Across all the benchmarks, we have detected that Arm HPC Compiler always adds the same sequence of instructions at the end of loops. This sequence consists of a *brkns* and a *mov* instruction, see label #L in Sublisting 17a. *Brkns* instruction was designed to implement a high level instruction *break*. It is used, when in a loop a break is executed, then the rest of iterations are not executed and the processor jumps to the next code sequence after the loop. This translated to SIMD is done through predication, each *z* register has a predicate register that indicates if each of the stored elements are active or not. What *brkns* does with the predication is preventing the execution of the remaining iterations after the break, setting the corresponding elements as inactive. The *mov* instructions is needed because *brkns* cannot read and write to the same register. The way *brkns* instruction is used in these loops is far from its correct use, because all the iterations in the loop are executed and therefore there is no conditional execution. Using a whilelo as GCC does is enough, see Sublisting 17b.

Listing 17: HYDRO benchmark assembly code

```
#LOOP:
#I add    x14, x8, x11
#C ld1rd  {z0.d}, p2/z,[x22]
   ld1d   {z1.d}, p0/z,[x14]
#I add    x14, x10, x11
#C ld1rd  {z2.d}, p2/z,[x20]
   ld1d   {z3.d},p0/z,[x14,x27,lsl#3]
#C ld1rd  {z4.d}, p2/z,[x26]
   incd   x12
   whilelo p1.d, x12, x13
   fmul   z2.d, z3.d, z2.d
   ld1d   {z3.d},p0/z,[x14,x28,lsl#3]
#I add    x14, x9, x11
#I addvl  x11, x11, #1
   fmla   z2.d, p2/m, z3.d, z4.d
   fmla   z0.d, p2/m, z2.d, z1.d
   st1d   {z0.d}, p0,[x14]
#L brkns  p1.b, p2/z, p0.b, p1.b
#L mov    p0.b, p1.b
   b.mi   #LOOP
```

a: Assembly code generated with Arm HPC Compiler

```
#LOOP:
  ld1d   {z0.d},p0/z,[x3,x0,lsl #3]
  ld1d   {z2.d},p0/z,[x4,x0,lsl #3]
  ld1d   {z1.d},p0/z,[x5,x0,lsl #3]
  fmul   z0.d, z4.d, z0.d
  fmla   z0.d, p1/m, z5.d, z2.d
  fmad   z0.d, p1/m, z1.d, z3.d
  st1d   {z0.d},p0,[x1,x0,lsl #3]
  incd   x0
  whilelo p0.d, x0, x2
  b.ne   #LOOP
```

b: Assembly code generated with GCC

In order to measure the impact of these unnecessary instructions, we have computed their weight on each benchmark, except for HPCG due to its code complexity, see Table 7.2. First column contains the name of the benchmark, the second contains the number of instructions for one iteration of the main loop. In the cases of FLOYD_WARSHALL, GEMM and FIR, which contain nested loops, we consider only the inner loop. We use the inner loop because it represents the major part of the instructions that will be executed in these cases. On the third column, we show the reduction ratio that can be achieved by removing unnecessary *brkns* and *mov* instructions. This pair of instructions represent an important part of the total instruction budget in benchmarks that have short loops like COPY and FIR. Moreover, on average they represent a significant 9.60%.

Benchmark	# instructions ArmHPC	% Reduction
COPY	8	25.00%
MULADDSUB	13	15.38%
FLOYD_WARSHALL	12	16.66%
INT_PREDICT	35	5.71%
HYDRO	19	10.52%
GEMM	11	18.18%
JACOBI_1D	17	11.76%
EOS	32	6.25%
JACOBI_2D	39	5.12%
VOL3D	118	1.69%
HACCK	36	5.55%
FIR	8	25.00%
Geom. Mean	–	9.60%

Table 7.2: Number of instructions of each loop compiled by Arm HPC Compiler and the reduction achieved removing *brkns* and *mov* instructions.

7.2.2 Indices and Registers

We have seen in Section 6.5.2 that Arm HPC Compiler when compiling for ARMv8.0 does not store constants in registers in order to have more available registers for the rest of the loop. However, this implies loading the same constants from cache every iteration. In addition, we have shown that Arm HPC Compiler makes a worse management of address indices, see Section 6.5.4. In contrast, GCC is able to manage constants and indexes reducing the total number of instructions.

This two characteristic are inherited by ARMv8.2 code generation. In Sublisting 17a, we have seen how Arm HPC Compiler doubles the number of instructions generated with respect to GCC. We have labeled constant loading instructions with #C and index computing instructions with #I. This instructions have a potentially higher weight than the pair shown in

Benchmark	# instructions	# index instructions	# constant instructions	% Reduction
COPY	8	1	0	12.50 %
MULADDSUB	13	0	0	0.00 %
FLOYD_WARS.	12	0	1	8.33 %
INT_PREDICT	35	1	8	25.71 %
HYDRO	19	4	3	36.84 %
GEMM	11	2	0	18.18 %
JACOBI_1D	17	1	3	23.52 %
EOS	32	6	3	28.12 %
JACOBI_2D	39	5	9	35.89 %
VOL3D	118	27	0	24.57 %
HACCK	36	0	0	0.00 %
FIR	8	0	0	25.00 %
Geom. Mean	–	–	–	13.12 %

Table 7.3: Number of instructions of each loop compiled by Arm HPC Compiler, the number of instructions where an additional index computation or constant loading is performed and the reduction achieved by removing them.

the previous section, so again we have computed the reduction achieved by removing them, see Table 7.3.

This reduction cannot be done in all benchmarks, MULADDSUB and HACCK do not have any of this extra instructions. But, INT_PREDICT, HYDRO, JACOBI_1D, EOS, JACOBI_2D, VOL3D and FIR represent more than a 20 % of the total code. Thus, we can achieve a 13.12 % reduction on average. In all cases, except JACOBI_1D and HACCK, GCC successfully avoids these instructions.

7.2.3 Instruction Selection

The first processor that will implement SVE is the Fujitsu Post-k, arriving around 2021 [47]. This means that nowadays GCC and Arm HPC Compiler developers do not known the details of the michroarchitecture of SVE processors. Thus, developers must take design decisions on the fly. One such decisions is choosing between two sequence of assembly code that perform the same task but with different instructions. In Listings 18 and 19, we present the assembly code of two loops, where GCC chooses a set of instructions to solve a task and Arm HPC Compiler chooses a different set. In Listing 18, Arm HPC Compiler chooses the *fmla* instruction, while GCC chooses *fmul* and *fadd*. In this case, it is clear that *fmla* will have a better performance, because it implements a sequence of two floating point operations that are data dependent using only one instruction.

Listing 18: FIR benchmark assembly code

```
#LOOP:
   ld1d   {z2.d},p1/z,[x20,x13,lsl#3]
   ld1d   {z3.d},p1/z,[x12,x13,lsl#3]
   incd   x13
   whilelo p2.d, x13, x11
#L brkns  p2.b, p0/z, p1.b, p2.b
   fmla   z1.d, p1/m, z3.d, z2.d
#L mov    p1.b, p2.b
   b.mi   #LOOP
```

a: Assembly code generated with Arm HPC Compiler

```
#LOOP:
   ld1d   {z2.d},p6/z,[x3,x2,lsl#3]
   ld1d   {z1.d},p6/z,[x0,x2,lsl#3]
   incd   x2
   fmul   z1.d, z1.d, z2.d
   fadd   z0.d, p6/m, z0.d, z1.d
   whilelo p6.d, x2, x4
   b.ne   #LOOP
```

b: Assembly code generated with GCC

In Listing 19, we present the assembly code of FLOYD_WARSHALL. This benchmark selectes the minimum value between two memory addresses. Arm HPC Compiler uses a comparison (*fcmgt*) and a conditional data movement (*sel*) to perform this operation. On the other hand, GCC uses *fminnm* instruction, that implements directly the selection of the minimum.

Listing 19: FLOYD_WARSHALL benchmark assembly code

```
#LOOP:
   ld1d   {z0.d},p1/z,[x16,x4,lsl#3]
#C ld1rd  {z1.d},p0/z,[x3]
   ld1d   {z2.d},p1/z,[x18,x4,lsl#3]
   fadd   z1.d, z2.d, z1.d
   fcmgt  p2.d, p0/z, z1.d, z0.d
   sel    z0.d, p2, z0.d, z1.d
   st1d   {z0.d},p1,[x17,x4,lsl#3]
   incd   x4
   whilelo p2.d, x4, x10
#L brkns  p2.b, p0/z, p1.b, p2.b
#L mov    p1.b, p2.b
   b.mi   #LOOP
```

a: Assembly code generated with Arm HPC Compiler

```
#LOOP:
   ld1d   {z0.d},p0/z,[x5,x0,lsl#3]
   ld1d   {z1.d},p0/z,[x3,x0,lsl#3]
   fadd   z0.d, z2.d, z0.d
   fminnm z0.d, p1/m, z0.d, z1.d
   st1d   {z0.d}, p0,[x1,x0,lsl#3]
   incd   x0
   whilelo p0.d, x0, x2
   b.ne   #LOOP
```

b: Assembly code generated with GCC

7.2.4 FIR

We have seen in Section 7.1, that the minimum number of instructions achieved in FIR benchmark is obtained with a vector length of 1024 bits. In Listing 18 we can see FIR benchmark assembly code, which corresponds to the inner loop. This loop iterates over 16 floating point constants of double precision, each constant is represented with 64 bits. In total, we have 1024 bits. Therefore, if we increase the vector length over 1024 bits, when we load the vector of constants, there will be elements of the vector register that will be unused, and these positions will be set to inactive in the predicate register.

One possible solution to this bottleneck is, instead of loading all the constants into a vector register, we can have one register for each constant (this is 16 of the 32 vector registers). For every vector register all its elements ($\frac{\text{vector length}}{64bits}$) are set to the corresponding constant for that register. Then, each constant is multiplied by the corresponding element of the input array (see FIR source code in Section 5.1.7) and added to the accumulator. With these modifications we achieve a better utilization of the vector length.

7.2.5 Loop unrolling

We have shown in Section 6.5.1 how loop unrolling and replacing two contiguous loads for one *ldp* are two of the best optimization techniques. The increase of the number of misses per instruction rises the pressure on memory bandwidth, which speeds-up high arithmetic intensive benchmarks like VOL3D, HACCK and FIR. SVE only allows the former optimization, but not the latter, because load instructions only accept one destination vector register [48]. Despite, compilers can keep using loop unrolling, only JACOBI_2D, is optimized with loop unrolling. A likely reason is that compilers do not know what is the target execution machine and what vector length is implemented.

Incidentally, SVE implements a subset of memory access instructions, that are specific for loading structured data. They are known as structs, and they are a composite data type. Structs allow us to reference a set of variables using only one address. The way SVE allows loading and storing this type of data is through *ld2*, *ld3*, *ld4*, *st2*, *st3* and *st4* instructions. They load or store multiple N-element structures composed by basic data types of the same type. For example *ld2d* loads two vector registers with double precision floating point elements taken from memory. The only difference with a multiple contiguous load is interleaving. This means that the first double it reads is stored in the first vector register, and then the second double is stored in the second register, the next double goes to the first register, and so on (see Arm reference for more information [49]). This can be generalized to the rest of the instructions, so *ld3* loads the double number i in the register x, where x is obtained using the following equation $x = (i \bmod 3)$. And finally, *ld4* works similarly but using the equation $x = (i \bmod 4)$.

Listing 20: COPY benchmark different assembly code implementations

```
#LOOP:
  ld1d   {z0.d},p0/z,[x22,x3,lsl#3]
  st1d   {z0.d},p0,[x21,x3,lsl#3]
  incd   x3
  whilelo p0.d, x3, x1
  b.ne   #LOOP
```

a: Assembly code generated with GCC

```
#LOOP:
  ld1d   {z0.d},p0/z,[x0]
  ld1d   {z1.d},p0/z,[x0,#1,mul vl]
  ld1d   {z2.d},p0/z,[x0,#2,mul vl]
  ld1d   {z3.d},p0/z,[x0,#3,mul vl]
  st1d   {z0.d},p0,[x1]
  st1d   {z1.d},p0,[x1, #1,mul vl]
  st1d   {z2.d},p0,[x1, #2,mul vl]
  st1d   {z3.d},p0,[x1, #3,mul vl]
  incb   x0, all, mul #4
  incb   x1, all, mul #4
  incd   x5, all, mul #4
  whilelo p0.d, x5, x3
  b.ne   #LOOP
```

b: Arm assembly code proposal for loop unrolling

```
#LOOP:
      ld4d   {z0.d-z3.d}, p0/z, [x21, x3, lsl #3]
      st4d   {z0.d-z3.d}, p0, [x22, x3, lsl #3]
      incd   x3, all, mul #4
      whilelo p0.d, x3, x1
      b.ne   #LOOP
```

c: Our assembly code proposal for loop unrolling

With these instructions we have something similar to a load pair instruction, that we can exploit to reduce the number of instructions executed by a benchmark. In Listing 20, we have written three versions of COPY benchmark. The first version (Sublisting 20a) is the assembly generated by GCC. The second version (Sublisting 20b) is a handwriten assembly code we have obtained following the Arm proposition for four-iteration loop unrolling presented in Hot Chips Conference[50]. The third version (Sublisting 20c) is our proposal of a four iteration loop unrolling using *ld4d* instructions.

In Figure 7.8, we can see the normalized number of instructions of the three assembly versions: GCC as GCC-BASE, Arm loop unrolling

proposal as Arm-UNROLL and our proposal as LD4-UNROLL. We can see how our proposal uses 75% less instructions than the baseline, while Arm proposal is at 34.7%. In addition, for all vector lengths our proposal uses less instructions than Arm proposal and GCC generated code. This technique can be applied to other benchmarks with low complexity like MULADDSUB, FLOYD_WARSHALL, INT_PREDICT or FIR, where the interleaving of the memory access instructions is not an obstacle.

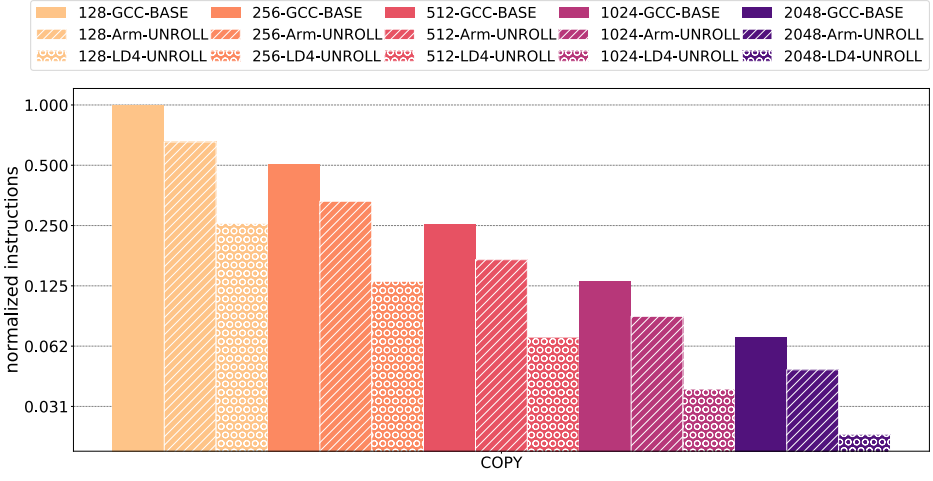


Figure 7.8: Normalized number of instructions executed (w.r.t GCC-BASE using 128 bits) using ArmIE. There are 5 different experiments with vector lengths of 128, 256, 512, 1024 and 2048 bits.

We finally remark the importance of this reductions, because in out-of-order processors that would implement short vector lengths (128 or 256 bits) it can imply a large improvement on single thread performance and for high arithmetic intensity benchmarks it would also improve the performance of one-thread-per-core execution. However, this are only speculations and in the futer we want to test these codes in simulators like *Gem5*.

7.2.6 Non-Temporal Vector Instructions

SVE includes two interesting instructions: *ldnt1d* and *ldnt1d*. These are non-temporal contiguous load and store instructions that include a hint to the memory system, so it does not need to keep referenced blocks in cache. Nevertheless, they are only a hint, so cache can continue storing the blocks anyways. This instructions are useful to code benchmarks with streaming memory access patterns, this pattern consists in executing operations over all the elements of a large data structure. Therefore, when a memory location is accessed, is not expected to be accessed again in a long period of time.

All of our benchmarks show this behaviour, but this is not enough to benefit from non-temporal instructions, benchmarks should also present some re-usability in the accesses. Streaming accesses usually evict other memory blocks from the data cache due to cache conflicts, if there are some memory blocks that are used frequently they could be evicted from cache, leading to performance penalties. From our collection of benchmarks, GEMM, FLOYD_WARSHALL and HPCG have re-usability patterns in memory accesses. Nevertheless, none of the assembly binaries use these instructions. In Listing 21, we propose a modification to the original GCC generated assembly code (see Sublisting 21a), so it could reduce the number of conflict misses and therefore improve performance.

Listing 21: FLOYD_WARSHALL benchmark modified assembly code

```
#LOOP:
   ld1d   {z0.d},p0/z,[x5,x0,lsl#3]
   ld1d   {z1.d},p0/z,[x3,x0,lsl#3]
   fadd   z0.d, z2.d, z0.d
   fminnm z0.d, p1/m, z0.d, z1.d
   st1d   {z0.d}, p0,[x1,x0,lsl#3]
   incd   x0
   whilelo p0.d, x0, x2
   b.ne   #LOOP
```

a: Assembly code generated with GCC

```
#LOOP:
   ld1d   {z0.d},p0/z,[x5,x0,lsl#3]
#N ldnt1d {z1.d},p0/z,[x3,x0,lsl#3]
   fadd   z0.d, z2.d, z0.d
   fminnm z0.d, p1/m, z0.d, z1.d
#N stnt1d {z0.d}, p0,[x1,x0,lsl#3]
   incd   x0
   whilelo p0.d, x0, x2
   b.ne   #LOOP
```

b: Our proposed assembly code

We have changed the second load and the store (see Sublisting 21b,

changes labeled with #N) for non-temporal memory accesses. These instructions bring memory blocks to the cache that are used once and then they are evicted. Nevertheless, this new memory blocks can make the cache evict other memory blocks due to cache conflicts, and this evicted blocks are used frequently. More specifically, frequently used blocks are loaded by the first load, they have some re-usability and fit in cache. These frequently used blocks represent the row of a matrix that is used in different loop calls. With these changes we give hints to the cache to not bring other non-temporal memory blocks, and if cache uses the hints, we can reduce the possibility that the frequently used blocks are evicted from the cache.

Chapter 8

Conclusions

In this last chapter we summarize what are the most remarkable aspects of this work. We cover all the chapters and what are the conclusions we have extracted from them. And finally we evaluate the work we have done with respect to the initial project management course.

8.1 Summary

In Chapter 5, we have seen that a basic tool to auto-vectorize parallel loops, is the directive *#pragma omp simd*. In some cases, this directive is not enough to achieve the optimal code, and programmers have to use other directives, or modify the original code, in order to aid automatic vectorization, but without changing the correctness of the benchmark.

In Chapter 6, we analyzed the execution of four binaries: one compiled with Arm HPC Compiler and no SIMD support, one compiled with GCC and no SIMD suppport, one compiled Arm HPC Compiler and NEON support, and one compiled with GCC and NEON support. This analysis consisted on executing the four binaries in two different processors: ThunderX and ThunderX2; and with different number of threads. We saw how memory bandwidth or CPU performance limit the final performance of the binaries, and we correlated this with the arithmetic intensity of the benchmarks and architecture of the processor in which the benchmark was executed. We saw how vectorization plays a significant role in high arithmetic intensive benchmarks or single thread execution, and how Arm HPC Compiler yields a higher performance than GCC. In addition, we also compared both test processors and measured the energy and power efficiency of ThunderX2.

In the second part of the chapter we analyzed what were the

optimizations or techniques used in the assembly generation process that have a significant impact on performance. We found that Arm HPC Compiler exploits *ldp* to improve its loop unrolling optimization. Moreover, the Arm HPC Compiler has better implementation of the runtime library and uses it more efficiently than GCC. We have also found that GCC is more aggresive storing constants to registers, which leads to better performance for some evaluated benchmarks .

In Chapter 7, we used the Arm Instruction Emulator (ArmIE) to test the compilation support of SVE in GCC and Arm HPC Compiler. Moreover, we have used ArmIE to count the total number of instructions and the number of SVE instructions to obtain the relative instruction reduction achieved by increasing the vector length. Furthermore we have obtain what is the ratio of vectorization of each compiler using SVE. With these experiments we have found that GCC is not able to vectorize complex loops and that Arm HPC Compiler has a poorer assembly generation. Regarding assembly code, we have detailed why the Arm HPC Compiler generates sometimes inefficient assembly code, and we have proposed some optimizations, we believe are interesting and we want to study in the future.

8.2 Project Autoevaluation

From the set of objectives listed in Section 1.3 we have completely accomplished the objectives 1, 2, 4 and 5. Objective number 3 has been completed, except the part of comparing power consumption of ThunderX and ThunderX2. Similarly, from the scope defined in Section 1.6, we have accomplish everything except the power efficiency comparision of ThunderX and ThunderX2. Instead of this, we have compared Skylake and ThunderX2. Therefore, we can say that the scope and the objectives have been completed. Moreover, we have followed all the good practices listed in Section 1.7.

Regarding project planning, we have followed the schedule defined at the start of the project. However, there have been some divergences with respect to the duration of the tasks, in which the project is divided. The total amount of hours invested is 595, and 45 of them were not included in the initial plan. However, we arleady estimated that we may needed 3 weeks more than those planned, that we have used to cover this extra

work.

Finally, I would like to express what has been to work in a research group within the scope of an european project. During these 4 months, I have learned a lot about vectorization, compiler optimizations, OpenMP, microarchitecture, and performing experiments. This work has help me to start writing research reports and analying results. Moreover, this work has introduced me to some state-of-the-art topics in the world of computer architecture like HPC, Arm SVE and OpenMP supporting compilers. In the future, I would like to continue working with BSC on these topics.

Appendix A

Workloads

In this appendix we list what are the workloads we have used in our experiments. We differentiate four sets of benchmarks: simple input benchmarks from RAJAPerf, complex input benchmarks from RAJAPerf, HACCKernels and HPCG.

The set of simple imput benchmarks from RAJAPerf is made up of RAJAPerf benchmarks that only have three input arguments: sizefact argument, which is the number of iterations of the parallel loop, in other words the number of times the body of the loop is executed; repfact argument, which is the number of repetitions the parallel loop is executed; and npasses argument, which is the number of times the whole experiment is executed in order to obtain the arithmetic mean of the hardware counters and time execution of the experiments, its value is always 3. Table A.1 shows the workload used for each benchmark from and the inputs we have used. Apart from the three arguments we listed above this table lists the multiplicative constants that multiply sizefact and repfact. For example, for COPY benchmark we have executed 60 million iterations of the parallel loop and we have executed the parallel loop 1800 times. We have set the number of iterations of the inner loop, so the working set does not fit in cache. For VOL3D we could not obtain the multiplicative constant because the benchmark uses a complex function to obtain the size of the data structures and the number of iterations.

Benchmark	Sizefact	Mult. Sizefact	Repfact	Mult. Repfact
COPY	60	1000000	1	1800
MULADDSUB	192	100000	1	3500
INT_PREDICT	96	100000	1	4000
HYDRO	19.2	100000	1	12500
EOS	72	100000	1	5000
VOL3D	6.0	Unknown	1	300
FIR	192	100000	1	1 600

Table A.1: Workloads of simple input benchmarks in RAJAPerf suite.

The rest of the benchmarks of RAJAPerf suite have one more argument that takes different semantic for each benchmark. Therefore, we have opted to define them one by one.

- FLOYD_WARSHALL: we use a matrix of 3600 x 3600 double precision floating point elements. Parallel loop is executed only one time.
- GEMM: we use two matrix of 2400 x 2400 double precision floating point elements. Parallel loop is executed 6 times.
- JACOBI_1D: we use a vector of 9200000 double precision floating point elements. Parallel loop is executed 25000 times.
- JACOBI_2D: we use a matrix of 2800 x 2800 double precision floating point elements. Parallel loop is executed 400 times.

Finally we use the following parameters nx=192, ny=192 and nz=192 for HPCG benchmark and for HACCKernels we use 100000 iterations.

Appendix B

Skylake Flags

Application	Flag Intel GCC SCALAR
HACCKernels	-O3 -march=skylake -mno-sse4 -mno-ssse3 -mno-sse2 -msse -mno-avx -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -fno-tree-vectorize -ffast-math -funroll-loops -ffp-contract=fast
HPCG	-O3 -march=skylake -mno-sse4 -mno-ssse3 -mno-sse2 -msse -mno-avx -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -fno-tree-vectorize -ffast-math -funroll-loops -std=c++11 -ffp-contract=fast
RAJAPerf	-O3 -march=skylake -mno-sse4 -mno-ssse3 -mno-sse2 -msse -mno-avx -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -fno-tree-vectorize -ffast-math

Table B.1: Flags for GCC, no SIMD version.

Application	Flag Intel GCC SSE
HACCKernels	-O3 -march=skylake -msse4 -mssse3 -msse2 -msse -mno-avx -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -ftree-vectorize -ffast-math -mprefer-vector-width=128 -funroll-loops -ffp-contract=fast
HPCG	-O3 -march=skylake -msse4 -mssse3 -msse2 -msse -mno-avx -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -ftree-vectorize -ffast-math -mprefer-vector-width=128 -funroll-loops -std=c++11 -ffp-contract=fast
RAJAPerf	-O3 -march=skylake -msse4 -mssse3 -msse2 -msse -mno-avx -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -ftree-vectorize -ffast-math -mprefer-vector-width=128

Table B.2: Flags for GCC, SSE (128b SIMD) support version.

Application	Flag Intel GCC SSE, AVX and AVX2
HACCKernels	-O3 -march=skylake -msse4 -mssse3 -msse2 -msse -mno-avx -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -ftree-vectorize -ffast-math -mprefer-vector-width=256 -funroll-loops -ffp-contract=fast
HPCG	-O3 -march=skylake -msse4 -mssse3 -msse2 -msse -mavx -mavx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -ftree-vectorize -ffast-math -mprefer-vector-width=256 -funroll-loops -std=c++11 -ffp-contract=fast
RAJAPerf	-O3 -march=skylake -msse4 -mssse3 -msse2 -msse -mavx -mavx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -ftree-vectorize -ffast-math -mprefer-vector-width=256

Table B.3: Flags for GCC, AVX2 (256b SIMD) support version.

Application	Flag Intel GCC SSE, AVX, AVX2 and AVX512
HACCKernels	-O3 -march=skylake -msse4 -mssse3 -msse2 -msse -mno-avx -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -ftree-vectorize -ffast-math -mprefer-vector-width=512 -funroll-loops -ffp-contract=fast
HPCG	-O3 -march=skylake -msse4 -mssse3 -msse2 -msse -mno-avx -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -g -fopenmp -ftree-vectorize -ffast-math -mprefer-vector-width=512 -funroll-loops -ffp-contract=fast
RAJAPerf	-O3 -march=skylake-avx512 -msse4 -mssse3 -msse2 -msse -mavx -mavx2 -mavx512f -mavx512pf -mavx512er -mavx512cd -mavx512vl -mavx512bw -mavx512dq -mavx512ifma -g -fopenmp -ftree-vectorize -ffast-math -mprefer-vector-width=512

Table B.4: Flags for GCC, AVX512 (512b SIMD) support version.

Application	Flag Intel ICC SSE
HACCKernels	-O3 -std=c++11 -gcc-name=/usr/bin/gcc -march=corei7 -xSSE4.2 -g -qopenmp -vec -ffast-math -funroll-loops -ffp-contract=fast
HPCG	-O3 -gcc-name=/usr/bin/gcc -march=corei7 -xSSE4.2 -g -qopenmp -vec -ffast-math -funroll-loops -std=c++11 -ffp-contract=fast
RAJAPerf	-O3 -gcc-name=/usr/bin/gcc -march=corei7 -xSSE4.2 -g -qopenmp -vec -ffast-math -ansi

Table B.5: Flags for ICC, SSE (128b SIMD) support version.

Application	Flag Intel ICC SSE, AVX and AVX2
HACCKernels	-O3 -std=gnu++11 -gcc-name=/usr/bin/gcc -march=corei7 -xCORE-AVX2 -g -qopenmp -vec -ffast-math -funroll-loops -ffp-contract=fast
HPCG	-O3 -gcc-name=/usr/bin/gcc -march=corei7 -xCORE-AVX2 -g -qopenmp -vec -ffast-math -funroll-loops -std=c++11 -ffp-contract=fast
RAJAPerf	-O3 -gcc-name=/usr/bin/gcc -march=corei7 -xCORE-AVX2 -g -qopenmp -vec -ffast-math -ansi

Table B.6: Flags for ICC, AVX2 (256b SIMD) support version.

Application	Flag Intel ICC SSE, AVX, AVX2 and AVX512
HACCKernels	-O3 -std=gnu++11 -gcc-name=/usr/bin/gcc -march=corei7 -qopt-zmm-usage=high -xSKYLAKE-AVX512 -g -qopenmp -vec -ffast-math -funroll-loops -ffp-contract=fast
HPCG	-O3 -gcc-name=/usr/bin/gcc -march=corei7 -qopt-zmm-usage=high -xSKYLAKE-AVX512 -g -qopenmp -vec -ffast-math -funroll-loops -std=c++11 -ffp-contract=fast
RAJAPerf	-O3 -gcc-name=/usr/bin/gcc -march=corei7 -qopt-zmm-usage=high -xSKYLAKE-AVX512 -g -qopenmp -vec -ffast-math -ansi

Table B.7: Flags for ICC, AVX512 (512b SIMD) support version.

Appendix C

ThunderX2 and Skylake Comparision

In this appendix we show the complete graphs of the comparision between ThunderX2 and Skylake processors. First we list both machines performance, next energy efficiency and finally power efficiency.

C.1 Performance

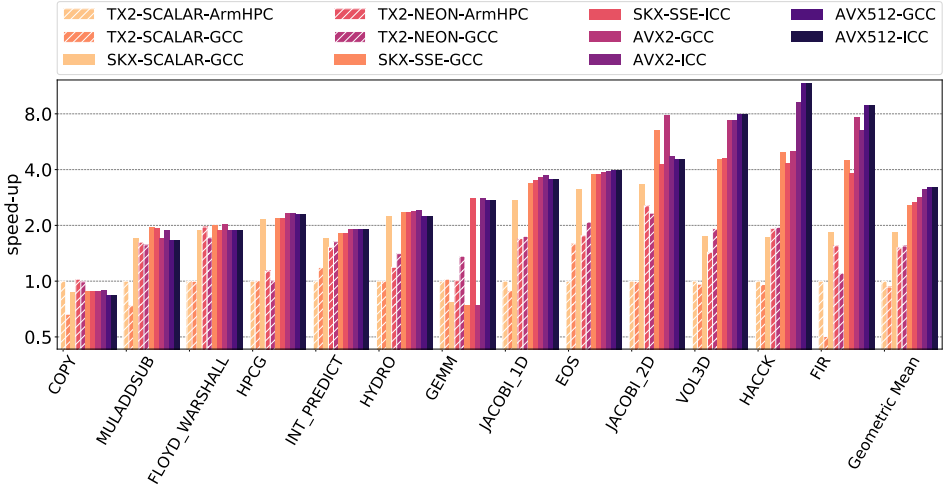


Figure C.1: Speed-up using the logarithmic scale (w.r.t 1-thread SCALAR-ArmHPC on ThunderX2) comparision of one thread between ThunderX2 (TX2) and Skylakae (SKX).

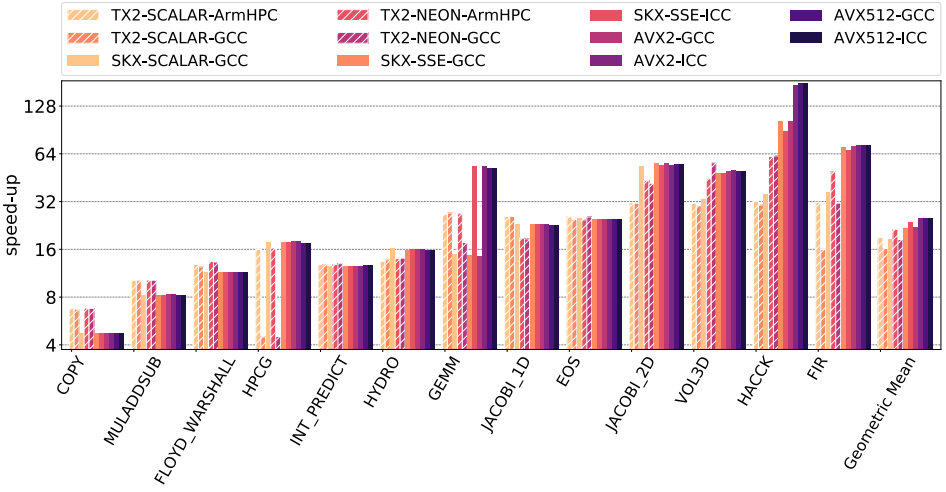


Figure C.2: Speed-up using the logarithmic scale (w.r.t 1-thread SCALAR-ArmHPC on ThunderX2) comparision of between 32 threads on ThunderX (TX2) and 28 threads on Skylake (SKX).

C.2 Energy Efficiency

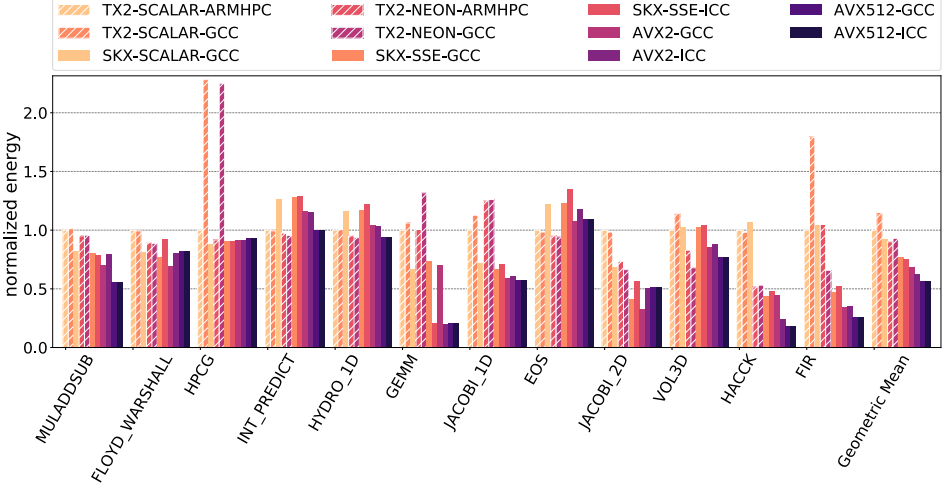


Figure C.3: Normalized energy (w.r.t 1-thread SCALAR-ArmHPC on ThunderX2) comparision between 32 threads in ThunderX2 (TX2) and 28 threads on Skylake (SKX).

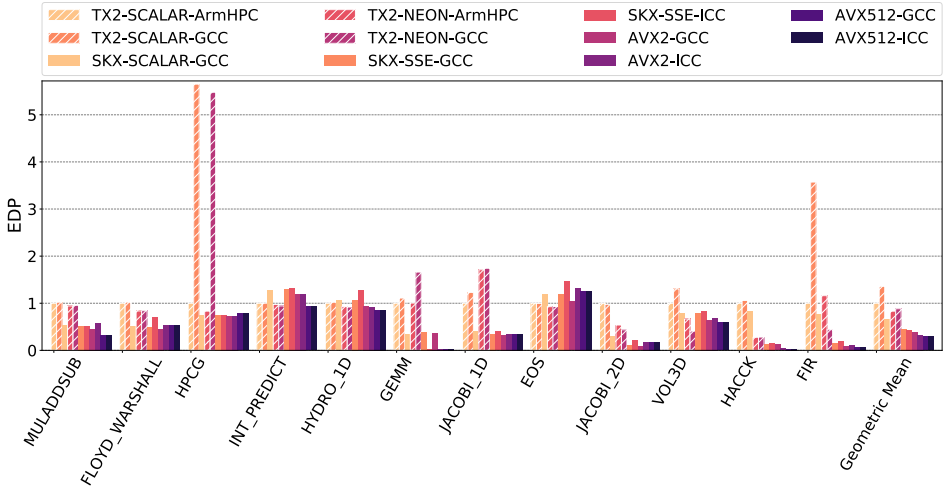


Figure C.4: Energy Delay Product (w.r.t 1-thread SCALAR-ArmHPC on ThunderX2) comparision between 32 threads ThunderX (TX2) and 28 threads on Skylake (SKX).

C.3 Power Efficiency

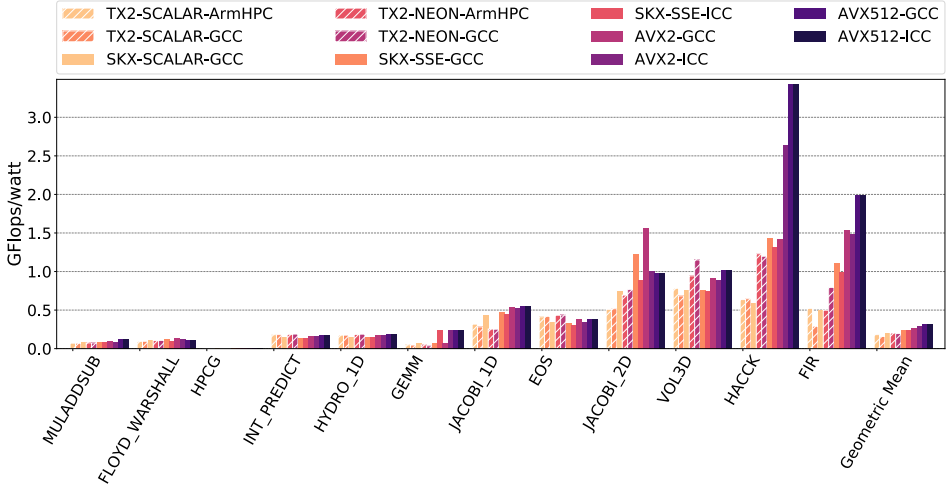


Figure C.5: GFlops/watt (w.r.t 1-thread SCALAR-ArmHPC on ThunderX2) comparision between 32 threads on ThunderX2 (TX2) and 28 threads on Skylake (SKX).

Bibliography

[1] Alejandro Rico, José A. Joao, Chris Adeniyi-Jones, and Eric Van Hensbergen. ARM HPC Ecosystem and the Reemergence of Vectors: *Invited Paper. In Proceedings of the Computing Frontiers Conference*, CF'17, pp. 329–334, New York, NY, USA, 2017.

[2] Robert A. Walker. MIMD computer sciences lectures, Kent State University, added on Feb. 25, 2019.

[3] Kerry McGuire Balanza (11 May 2010), ARM from zero to billions in 25 short years, ARM Holdings, added on Feb. 25, 2019.

[4] K. Roberts-Hoffman and P. Hegde. ARM Cortex-A8 vs. Intel Atom: Architectural and Benchmark Comparisons. In *report at University of Texas*, Dallas, USA, 2009.

[5] N. Rajovic, P. Carpenter, I. Gelado, N. Puzovic, A. Ramirez, and M. Valero. Supercomputing with Commodity CPUs: Are Mobile SoCs Ready for HPC?. in *SC*, 2013

[6] N. Rajovic, A. Rico, F. Mantovani, D. Ruiz, J.O. Vilarrubi, C. Gomez, L. Backes, D. Nieto, H. Servat, X. Martorell, et al. The Mont-blanc Prototype: An Alternative Approach for HPC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, USA, 13–18 November 2016; IEEE Press: Piscataway, NJ, USA, 2016.

[7] Mont-Blanc. Mont-Blanc Project History Page. Added on Feb. 25, 2019.

[8] RAJAPerf by Lawrence Livermore National Laboratory. Added on March 14, 2019.

[9] HACCKernels. Added on March 14, 2019.

[10] HPCG benchmark. Added on March 14, 2019.

[11] ARM Holdings. Introducing NEON Development Article. Added on Feb. 25, 2019.

[12] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The ARM Scalable Vector Extension. *IEEE Micro special issue on Hot Chips*, 2017.

[13] Enrico Calore, Filippo Mantovani, and Daniel Ruiz. Advanced performance analysis of HPCworkloads on Cavium ThunderX. International *Conference on High Performance Computing & Simulation* (HPCS 2018). IEEE, 2018.

[14] D. Ruiz, F. Mantovani, M. Casas, J. Labarta, & F. Spiga. The HPCG benchmark: analysis, shared memory preliminary improvements and evaluation on an Arm-based platform. (2018)

[15] Mont-Blanc. MB3 D6.4 – Report on application tuning and optimization on ARM platform. Added on Feb. 25, 2019.

[16] Mont-Blanc. MB3 D6.9 – Performance analysis of applications and mini-applications and benchmarking on the project test platforms. Added on Feb. 25, 2019.

[17] Mont-Blanc. MB3 D7.17– Final report on Arm-optimized Fortran compiler and mathematics libraries. Added on Feb. 25, 2019.

[18] Michael Larabel (December 28 2017), GCC 8 vs. LLVM Clang 6 Performance At End Of Year 2017, Added on Feb. 25, 2019.

[19] J.J. Kim, S.Y. Lee, S.M. Moon, and S. Kim, Comparison of LLVM and GCC on the ARM platform, 5th *Int. Conf. Embed. Multimed. Comput.*, 2010.

[20] WRF as a performance case study. Added on March 18, 2019.

[21] N. Binkertet et al., The GEM5 simulator, CAN, vol. 39, pp. 1–7, 2011

[22] Miguel Tairum (November 5 2018) Emulating SVE on existing Armv8-A hardware using DynamoRIO and ArmIE. Added on Feb. 25Th of 2019.

[23] https://tools.bsc.es/paraver. Added on March 18, 2019.

[24] https://tools.bsc.es/extrae. Added on March 18, 2019.

[25] Hennessy, John & A. Patterson, David. Computer Architecture: A Quantitative Approach / J.L. Hennessy, D.A. Patterson; page C-31. 1996

[26] Extending OmpSs programming model with task reductions: A compiler and runtime approach Added on March 14, 2019

[27] Dynamic vectorization of instructions Added on March 14, 2019

[28] Porting and tuning of the Mont-Blancbenchmarks to the multicore ARM 64bit architecture Added on March 14, 2019

[29] Characterization of applications in new architectures Added on March 14Th of 2019.

[30] GCC 8.1.0. Added on May 20, 2019.

[31] Arm HPC Compiler. Added on May 20, 2019.

[32] libpfm library. Added on June 8, 2019.

[33] Papi Performance Application Programming Interface. Added on May 20, 2019.

[34] ThunderX memory hierarchy. Added on May 20, 2019

[35] ThunderX product overview Added on March 27, 2019.

[36] PRACE report on ARM-based architectures Added on March 27, 2019.

[37] ThunderX pipeline article Added on March 27, 2019.

[38] ThunderX GCC support Added on March 27, 2019.

[39] ThunderX2 chip diagram Added on March 27, 2019.

[40] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4): 65–76, 2009.

[41] STREAM Benchmark. Added on June 3, 2019

[42] SIMD registers. Added on June 4, 2019

[43] Skylake Xeon Added on May 14, 2019.

[44] A. Armejach, M. Casas, and M. Moretó. Design trade-offs for emerging HPC processors based on mobile market technology. *The Journal of Supercomputing*, pp. 1-24. 2019

[45] Khubaib, M.A. Suleman, M. Hashemi, C. Wilkerson and Y.N. Patt, MorphCore: An Energy-Efficient Microarchitecture for High Performance ILP and High Throughput TLP, *45Th Annual IEEE/ACM International Symposium on Microarchitecture*, Vancouver, BC, 2012, pp. 305-316.

[46] Top 500 green list. Added on June 17, 2019.

[47] Post-k press release. Added on June 17, 2019.

[48] SVE documentation of LD1 Added on June 17, 2019.

[49] SVE documentation of multiple load and store. Added on June 17, 2019.

[50] Arm SVE presentation in Hot Chips Symposium. Page 25. Added on June 17, 2019.