**Journal of Supercomputing manuscript No.**
(will be inserted by the editor)

# On the Use of Many-core Marvell ThunderX2 Processor for HPC Workloads

**Víctor Soria-Pardos · Adrià Armejach**
**Darío Suárez · Miquel Moretó**

Received: date / Accepted: date

**Abstract** Marvel's ThunderX2 has been the first Arm-based processor with deployments in large-scale HPC production systems, challenging the dominance that x86 processors had in the last decades. While x86 processors and its software stack have been characterized in detail, the behaviour of Arm counterparts is not well known, limiting its adoption.

This work methodically characterizes performance and power efficiency of the ThunderX2 running different HPC workloads compiled with two state-of-the-art compilers, GCC and Arm HPC Compiler. We study the maturity of available compilers and find that the Arm HPC Compiler is able to apply additional optimizations, resulting in better performance than GCC. In addition, we also compare both performance and power with respect to an Intel Skylake processor. Despite the faster single thread performance of Skylake, ThunderX2 is able to match performance on multi-threaded workloads due to its superior memory bandwidth. However, power efficiency of ThunderX2 is far from matching Skylake-based processors when AVX512 extensions are used.

**Keywords** Arm · ThunderX2 · Skylake · Power · Arm HPC Compiler · GCC

## 1 Introduction

Delivering an exaflop requires to advance the current capabilities of computers in many directions, such as energy efficiency, that were not the traditional focus of HPC

Víctor Soria-Pardos
E-mail: victor.soria@bsc.es

Adrià Armejach
E-mail: adria.armejach@bsc.es

Darío Suárez Gracia
E-mail: dario@unizar.es

Miquel Moretó
E-mail: miquel.moreto@bsc.es

computers. These new requirements provided an opportunity for companies that were centered around mobile computing, such as ARM, to develop its newer Instruction Set Architecture (ISA), microarchitecture, and tools with the HPC segment in mind. Clear steps in this direction are: the first Arm HPC system, that demonstrated a functional software stack [22]; the announcement of Arm's Scalable Vector Extension (SVE) at HotChips 2016 [25]; or the development of Post-K, a top-tire system developed by Fujitsu and deployed at RIKEN [29]. The latter will become the first Arm-based pre-exascale system with enough compute capacity to challenge other vendors. Taking these developments into account, Arm has become a competitive option to build HPC systems and is being considered for large-scale deployments.

To become a reference HPC system, Arm needs to provide mature tools such as compilers, system libraries for parallelization, and math routines. Such a software ecosystem allows developers to run existing HPC applications on Arm hardware and achieve the expected performance without extensive modifications.

In this paper, we evaluate the current state of Arm's compiler ecosystem and extensively analyze parallelization and vectorization of HPC benchmarks on a real Arm-based processor, ThunderX2. To achieve this goal, we carry out an evaluation campaign in which we evaluated the two state-of-the-art compilers: GNU Compiler Collection (GCC) and the Arm HPC Compiler. In the experiments, we take into account compiler optimization directives, the use of Arm Performance Libraries, and shared memory multiprocessing via OpenMP. We characterize ThunderX2 system performance over a comprehensive set of HPC benchmarks and analyse the differences in compiler-generated code, providing insight on the performance differences observed. Additionally, we perform a performance and power comparison with an Intel Skylake processor.

In particular, Section 4 contains the performance results of ThunderX2 in GFlops using a roofline model, which enables categorization of the different workloads. The model helps illustrate what are the main limitations to achieve higher performance for each benchmark. Section 5 compares the differences between the two major Armv8 ISA compilers. We use speed-up metrics in order to determine which compiler generates faster code, for both single-threaded and multi-threaded executions. Additionally, in this section we also analyze the execution traces of the benchmarks that behave differently. Section 6 analyses the code generated and applied optimizations by both compilers in order to explain compiler related performance differences. We have found that simple compiler optimizations can have an important impact on performance.

Furthermore, we perform a performance and power comparison between ThunderX2 and an Intel Skylake processor. Section 7 shows the performance of both processors using different compilers. We find that Skylake beats ThunderX2 in single-threaded performance, but ThunderX2 nearly matches Skylake in multi-threaded performance. In Section 8 we leverage available power monitoring facilities to perform a power-efficiency study comparing both processors within a similar environment. We find that ThunderX2 has a lower power efficiency compared to the optimized architecture of Skylake.

## 2 Related Work

This section describes the state-of-the-art in the two areas that intersect with our work: the adoption of Arm cores in HPC and the comparison of performance and power efficiency between Arm and x86 architectures.

*Arm in High Performance Computing:* There is a growing interest in evaluating Arm-based platforms for HPC. A number of studies have used architectural simulators to study the trade-offs that appear with the design freedom associated to tailored designs enabled by license-based IP technology [3,18,19]. While others employed real platforms. For example, Petrogalli *et al.* [21] evaluate the *-fsimdmath* command line option from Arm HPC Compiler, which enables auto-vectorization of math functions in C and C++ codes. Garcia-Gasulla *et al.* [8] apply system software techniques to improve runtime support of HPC applications in ThunderX2. Banchelli *et al.* [4] present an evaluation of the Arm software ecosystem, focusing on the Arm HPC Compiler and the Arm Performance Libraries. Rico *et al.* [24] discus the state of the Arm HPC ecosystem and provide details on SVE as a future HPC technology. Limited performance and usability studies for SVE have been performed using architectural simulators [1,2].

Yokoyama *et al.* [28] provide a comprehensive analysis and discussion of the state-of-the-art of HPC on ARM architectures. The authors did a commending effort analyzing the contributions of more than 100 papers. While some articles include ThunderX2 in their experiments, there is no indepth study comparing multiple compilers with Neon support with respect to Intel's AVX512 extension. In this paper, we analyze the code generation and vectorization performed by two Armv8 compilers. We accurately measure the impact on performance of the vectorization and highlight why performance differences occur. In addition, we perform a performance-power comparison with an x86-based platform.

*Comparison of Arm and x86 Architectures:* Blem *et al*. [5] focus on the specific microarchitectural implementations of Arm and x86. It shows that the Atom processor can achieve similar energy consumption than a Cortex-A9. Laurenzano *et al.* [17] characterize the performance and energy of HPC computations on Cortex A9 and A15 processors and compares them to a Sandy Bridge Xeon processor. Jundt *et al.* [12] investigate the key architectural factors that impact power and performance on an Armv8 X-Gene 1 and Intel's Sandy Bridge processors.

Ramirez-Gargallo *et al*. [23] evaluate modern TensorFlow workloads for image recognition on clusters based on different CPU architectures: x86 Intel Skylake, Armv8 Marvell ThunderX2, and PowerPC IBM Power9. McIntosh-Smith *et al.* [20] present performance results from Isambard, a system based on Marvell ThunderX2 and compare them with Intel Skylake and Broadwell, as well as Xeon Phi processors. They also compare performance across three major software tool-chains available for Arm: Cray's CCE, Arm HPC Compiler, and GCC. Jackson *et al*. [11] perform a thorough comparison between ThunderX2 and different Intel-based architectures using Message Passing Interface (MPI) benchmarks while comparing different interconnection networks. In this paper, the focus when comparing between Arm and x86

Table 1: Processor features: Instruction Set Architecture (ISA), # cores, frequency, power, memory hierarchy, architectural features, and operating system.

| Characteristic | ThunderX2 (TX2) | Skylake (SKX) |
|---|---|---|
| Architecture | Armv8.1 | x86_64 |
| Num. of cores | 32 | 28 |
| Frequency (GHz) | 2.0 | 2.1 |
| TDP (Watts) | 180 | 165 |
| I-L1 (KiB) | 32 | 32 |
| D-L1 (KiB) | 32 | 32 |
| L2 (KiB per core) | 256 | 1024 |
| L3 (MiB total chip) | 32 | 38.5 |
| Instruction Queue | 60 | 97 |
| Load/Store Queue | 64/36 | 72/56 |
| Re-order Buffer | 180 | 224 |
| SIMD | 2 units of 128 bits | 2 units of 512 bits |
| Bandwidth (GiB/s) | 170 | 119.21 |
| Operating System | Red Hat 4.8.5-28 | Red Hat 4.8.5-16 |
| Kernel version | 4.14.0-49.2.2.el7a.aarch64 | 3.10.0-693.2.2.el7.x86_64 |

is on energy efficiency and performance improvements achieved by the utilization of SIMD instructions.

## 3 Experimental Methodology

*Description of systems under test:* The majority of the experiments have been conducted on a node with a ThunderX2 CN9980 processor. While the rest of experiments have been executed on an Intel Xeon Platinum 8176 processor (Skylake microarchitecture). Both nodes are mounted on the same rack, and, thereby, they share the same file system, the power measuring infrastructure, and part of the software stack. Table 1 lists the principal characteristics for each system.

*Benchmarks:* We have considered fourteen benchmarks. Two of them are well known mini-applications representative of the HPC domain: HPCG [14] and HACCKernels [15]. The rest are 12 benchmarks from the RAJAPerf suite from Lawrence Livermore National Laboratory [16]. These benchmarks range from simple HPC-oriented loops to real kernels extracted from relevant HPC applications. Table 2 lists the inputs used with each benchmark. Sizespec, sizefact, and repfact specify the size of data structures, the size of inner loop iterations, and the amount of repetitions of the outer loop, respectively. Moreover, the table includes the operational intensity of each benchmark-input pair, which is calculated as follows:

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

The inputs have been chosen in order to represent typical large data structures of HPC workloads and exhibit representative last-level cache behaviour in terms of

Table 2: Benchmark inputs and operational intensity (OI).

| RAJAPerf Benchmark | Sizespec | Sizefact | Repfact | OI |
|---|---|---|---|---|
| COPY | – | 60 | 1 | 0 |
| MULADDSUB | – | 192 | 1 | 0.08 |
| FLOYD WARSHALL | large | 1 | 1 | 0.13 |
| INT PREDICT | – | 96 | 1 | 0.19 |
| HYDRO 1D | – | 48 | 1 | 0.21 |
| GEMM | extralarge | 1 | 1 | 0.25 |
| JACOBI 1D | extralarge | 1 | 25 | 0.38 |
| EOS | – | 72 | 1 | 0.50 |
| JACOBI 2D | extralarge | 1 | 600 | 1.13 |
| VOL3D | – | 6 | 1 | 1.39 |
| FIR | – | 192 | 1 | 2.00 |
| LTIMES | – | 92 | 1 | 4.49 |
| **Benchmark** | **Input** | | | **OI** |
| HPCG | 192 | | | 0.13 |
| HACCK | 100000 | | | 1.50 |

misses and reuse. Therefore, in most cases, data structures do not fit in the last level cache. All benchmarks allow parallel execution. Vectorization and parallelization is achieved by the compiler with the help of pragma hints (*#pragma omp parallel for simd*).

*Compiler Infrastructure:* We have compiled the selected benchmarks for ThunderX2 using the only two representative available compilers: GCC 8.2.0 [7] and the Arm HPC Compiler 19.0 [10]. The latter is based on the LLVM 7.0.2 tool chain. Both compilers support auto-vectorization, and both implement the OpenMP standard and runtime. In particular, OpenMP 4.5 is fully supported for C/C++ in GCC through the *libgomp* library (GOMP). On the other hand, the Arm HPC 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.

With these two compilers, we have generated four binaries for each benchmark-input pair. For the sake of clarity, a binary is a compiled benchmark-input pair with a particular combination of vectorization and compiler that we summarize with two labels. The first label, *vectorization*, has two values: NEON if the binary has support for Neon 128 bit SIMD technology, or SCALAR, if the binary does not execute any SIMD instruction. The second label corresponds to the compiler: GCC or ArmHPC for Arm HPC Compiler. Therefore, SCALAR-GCC is a binary compiled without SIMD support using GCC, and NEON-ArmHPC is a binary compiled with Neon SIMD support using the Arm HPC Compiler. Table 3 lists the flags used to compile each benchmark. Note that we have used the same flags for GCC and for Arm HPC Compiler. In order to obtain the four binaries mentioned before, we have used four different sets of flags (see Table 4), one for each binary.

For the experiments in Skylake, the benchmark-input pairs do not change while one compiler does. Arm HPC compiler is replaced by Intel ICC version 18.0.0, while GCC remains with version 8.2.0 and enables to have a common reference point.

Table 3: Compilation flags for GCC, Arm HPC Compiler, and ICC.

| Benchmark | GCC and Arm HPC Compiler Flags |
|---|---|
| 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` |
| **Benchmark** | **ICC Flags** |
| HACCK | `-O3 -qopenmp -ffast-math -funroll-loops -ffp-contract=fast` |
| HPCG | `-O3 -qopenmp -ffast-math -funroll-loops -ffp-contract=fast` |
| RAJAPerf | `-O3 -qopenmp -ffast-math -ansi` |

Table 4: Compilation flags used for ThunderX2 binaries.

| Binary Version | Flags for ThunderX2 |
|---|---|
| SCALAR-GCC | `-march=armv8-a+fp+nosimd -fno-tree-vectorize` |
| SCALAR-ArmHPC | `-march=armv8-a+fp+nosimd -fno-vectorize` |
| NEON-GCC | `-march=armv8-a+fp+simd -ftree-vectorize` |
| NEON-ArmHPC | `-march=armv8-a+fp+simd -fvectorize` |

Table 5: Specific flags used for Skylake binaries.

| Binary label | Flags for Skylake |
|---|---|
| SCALAR-GCC | `-O3 -march=skylake -mno-sse4 -mno-ssse3 -mno-sse2 -mno-avx -mno-avx2 -mno-avx512f -mno-avx512pf -mno-avx512er -mno-avx512cd -mno-avx512vl -mno-avx512bw -mno-avx512dq -mno-avx512ifma -fno-tree-vectorize` |
| AVX512-GCC | `-march=skylake -msse4 -mssse3 -msse2 -msse -mavx -mavx2 -mavx512f -mavx512pf -mavx512er -mavx512cd -mavx512vl -mavx512bw -mavx512dq -mavx512ifma -ftree-vectorize -mprefer-vector-width=512` |
| AVX512-ICC | `-O3 -march=corei7 -qopt-zmm-usage=high -xSKYLAKE-AVX512 -qopenmp -vec` |

Since ICC does not support disabling SIMD instructions, we have 3 binaries for each benchmark-input pair in Skylake: SCALAR-GCC, AVX512-GCC for scalar and SIMD (with AVX-512 instructions) binaries compiled with GCC, and AVX512-ICC for SIMD ICC compiler.

When compiling with GCC for Skylake, we have used the same flags used for ThunderX2, and we have found the equivalents flags in ICC (See Table 3). Again we have use specific flags for each binary, see Table 5.

*Power Measurements:* Average power consumption measurements use the facilities available at the rack, where both nodes are mounted. The cluster uses the High Definition Energy Efficiency Monitoring (HDEEM) library [9], which is a software interface used to measure power consumption of HPC clusters. Measurements are made via the BMC (Baseboard Management Controller) and a FPGA (Field-Programmable Gate Array) located at each compute node motherboard. This configuration allows us

to capture the average power consumed without interfering with processor performance. The system is able to measure different devices, but since our focus in on the processor, we capture the processor power rails. Before the execution of our experiments the processor activates GPIO signals to start and stop FPGA data collection. The FPGA is constantly monitoring the energy drain with a sampling rate of 1ms for the global board and processor.

*Performance Analysis Tools:* During our analysis, we have used Extrae [6] and Paraver [13]. The former is a dynamic instrumentation package to trace programs that employ shared memory programming models like OpenMP. Extrae generates trace files that can be later visualized with Paraver. Paraver is a flexible parallel program visualization and analysis tool. Paraver provides a qualitative global perception of the application behavior by visual inspection.

## 4 Roofline Model of the ThunderX2 Processor

The Roofline model is a simple and visual way to understand program performance on a given system [27]. A roofline model ties together floating point performance, operational intensity (OI), and memory performance in a two-dimensional graph. The Y-axis represents the GFlops per second (performance). Theoretical ceilings can be derived using the hardware specifications. The X-axis is OI, floating point 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 OI. The following 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:

$$\begin{aligned} \text{Peak GFlops} = (&\text{CPU speed in GHz}) \times (\text{number of CPU cores}) \times \\ &(\text{SIMD element wide}) \times (\text{flops per operation}) \times \\ &(\text{number of SIMD units}) \end{aligned} \quad (1)$$

We have used the fused multiply-add (FMA) instruction as reference to compute peak performance. The instruction performs two floating point operations, thus an FMA using Neon technology performs 4 double precision floating point operations. ThunderX2 can deliver up to two Neon SIMD FMA instructions of 128 bits every cycle, which is equivalent to 8 double precision floating point operations. We calculate two performance ceilings, one for binaries that only use non-vectorized instructions, named SCALAR, and one for binaries that use Neon SIMD technology,

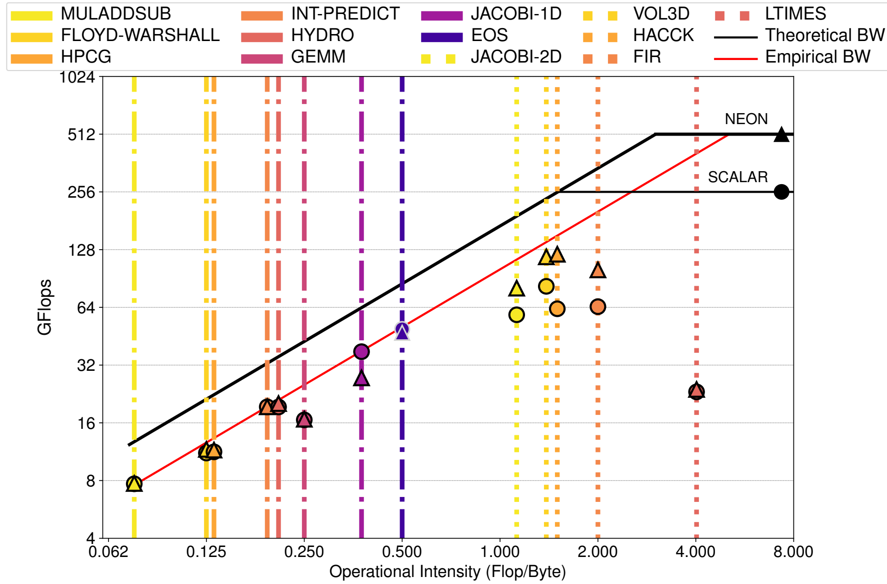



Figure 1: Roofline model for 32 ThunderX2 cores. For each benchmark we plot a vertical line indicating its OI (x-axis). For each vertical line we plot two markers that indicate performance (y-axis): the triangle indicates the performance obtained with Neon, and the circle the performance without SIMD support.

named as NEON.

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: 512 GFlops.

To complete the model the peak memory bandwidth is used. We also plot two memory ceilings: the theoretical memory bandwidth, which is 170 GB/s; and the empirically measured memory bandwidth reached using the Stream-COPY benchmark, 100 GB/s. Empirical peak memory bandwidth is necessary to understand what is the real bandwidth achievable in the system.

Figure 1 presents the roofline model of ThunderX2 using 32 cores. This roofline has the two compute ceilings labeled as SCALAR and NEON and also two memory bandwidth ceilings. The black line in the figure represents the theoretical bandwidth and the red line the empirical bandwidth.

Once the ceiling have been plotted we can add to the model different experiments that have been run on the machine. For each benchmark, we have run two experiments using two different binaries compiled with the Arm HPC Compiler. One of the binaries has support for SIMD (labeled as NEON), and the other one does not support any SIMD instructions (labeled as SCALAR). For each of the experiments, we plot a marker, a circle for SCALAR and a triangle for NEON. These same markers are

plotted over the corresponding performance ceiling. The Y-axis is the performance achieved by the experiment (in GFlops) and the X-axis corresponds with the operational intensity of the benchmark. We have discarded the experiments with GCC binaries from Figure 1 for clarity. In fact, GCC performance is lower in most of the experiments as we detail in Section 5.

We can categorize the benchmarks into two groups, those that are below 0.5 Flops/byte and those that are above. The low OI group is formed by benchmarks that are memory bandwidth bound, since they perform right at the empirical memory bandwidth threshold. The higher OI group of benchmarks, those above 0.5 Flops/-byte, are in most cases under the memory bandwidth ceiling, which means that memory bandwidth is not the main performance-limiting factor.

The only way to speed-up the benchmarks from the low OI group is to increase the memory bandwidth available. From this group, the only benchmark that is not restricted by memory bandwidth is JACOBI-1D, due to suboptimal code generation. For most of these benchmarks, both SCALAR and NEON versions yield the same performance. In the case of GEMM, SCALAR binaries achieve better performance than NEON binaries, we explain this performance difference in Section 5.

On the other hand, the group of benchmarks with higher OI is not memory bandwidth bound. Increasing available bandwidth would not have an immediate impact on the performance of these benchmarks. We can see how FIR and LTIMES are further away from the performance ceiling. In order to speed-up these applications, many different optimizations can be carried out in hardware (pipeline redesign, wider vector lengths, higher core count, etc.) and software (compiler optimizations). We have found that by rewriting the code of FIR and LTIMES and simplifying self-defined types, the compiler is able to apply additional optimizations, which leads to $1.51\times$ and $1.98\times$ speed-up, respectively. Further details can be found in Section 6. In addition, for high OI benchmarks, SIMD provides significant performance improvements with almost linear scaling in HACCK. The LTIMES benchmark is the exception, because the compiler is not able to vectorize the code. Manual vectorization is the only solution in these cases.

We have seen how the roofline model helps to characterize the workloads and can explain where performance limitations lay. Even though ThunderX2 has abundant memory bandwidth with 8 memory channels, bandwidth is still a main performance constraint.

## 5 Compiler Performance Comparison

This section looks at the performance differences when employing two different compilers, GCC and Arm HPC Compiler, for scalar and vectorized (NEON) code. We quantify these performance differences in terms of speed-up between four compiled binaries: GCC with scalar instructions (SCALAR-GCC), GCC enabling NEON support (NEON-GCC), Arm HPC Compiler with scalar instructions (SCALAR-ArmHPC), and Arm HPC Compiler enabling NEON support (NEON-ArmHPC).

Figure 2 shows speed-ups for single-threaded executions normalized to SCALAR-ArmHPC. In single-threaded executions we can observe that vectorization signific-

antly improves performance in most benchmarks, with average speeds-up of 1.47× for NEON-ArmHPC and 1.50× for NEON-GCC with respect to SCALAR-ArmHPC. We recall that the theoretic maximum speed-up of NEON vectorization for double precision floating point programs is 2×. On average, there are no significant differences between compilers; however, specific benchmarks do have large differences.

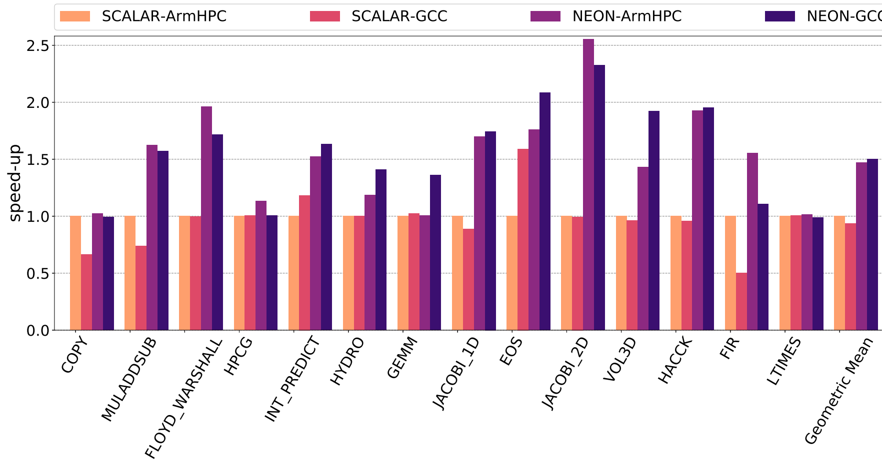



Figure 2: Speed-up of single-threaded executions, normalized to SCALAR-ArmHPC, on a ThunderX2 processor. Benchmarks are sorted by increasing OI.

These differences are caused by different optimization techniques being applied. For example: (i) the SCALAR-GCC versions of COPY and FIR are significantly slower than SCALAR-ArmHPC versions; and (ii) NEON versions of JACOBI_2D have additional optimizations when compared to their scalar counterparts, leading to superlinear speed-ups well above the 2× mark. We explain in detail the reasons behind these differences in Section 6.

There are two benchmarks where compilers are unable to auto-vectorize the code, i.e., HPCG and LTIMES. In the former, irregular access patterns due to sparse data structures prevent vectorization of many loops. In the latter, both compilers are not able to vectorize the loops because NEON does not have gather and scatter instructions.

The same experiments using 32 threads are shown in Figure 3. The speed-up is again computed with respect to the single-threaded execution of SCALAR-ArmHPC. In this second graph, we can see how the memory bandwidth ceiling affects performance. Only five applications out of 14 have a parallel efficiency of 95%. All these applications have an OI higher than 1 Flop/byte. On the other hand, 6 benchmarks are below 50% of parallel efficiency. Since benchmarks are sorted by their operational intensity, we can see the correlation between scalability and OI, and between vectorization and OI. Benchmarks with a low OI, from COPY to EOS, do not benefit from vectorization, because the available memory bandwidth limits scalability. While the high OI benchmarks are able to benefit from it; JACOBI_2D, VOL3D, HACCK and

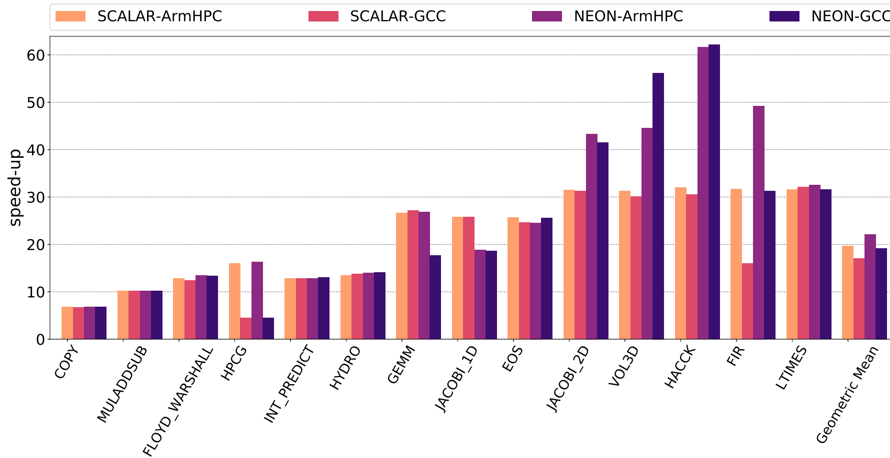



Figure 3: Speed-up of 32-threaded executions, normalized to SCALAR-ArmHPC, on a ThunderX2 processor. Benchmarks are sorted by increasing OI.

FIR. Despite having high OI, LTIMES is not vectorized as mentioned in the previous section.

The HPCG benchmark is a special case, because the performance of GCC binaries is specially low (see Figure 3). To have a better insight, we have plotted the scalability of the benchmark in Figure 4. The X-axis is the number of threads used, and the Y-axis represents the speed-up achieved with respect to 1-threaded SCALAR-ArmHPC execution. We can see that the scalability of GCC drops below 12% parallel efficiency for 32 threads, while Arm HPC Compiler achieves a 50% parallel efficiency. GCC is performing almost four times slower than Arm HPC Compiler. Regarding vectorization, GCC cannot vectorize the sparse matrix operations. On the other hand, the Arm HPC compiler slightly benefits from vectorization, because SCALAR-ArmHPC version is already memory bound. We have analyzed the traces obtained with Extrae [6] and visualizing them with Paraver [13]. Paraver allows us to plot only the parallel regions executed during the experiment. Each function is represented with a rectangle, while idle execution is represented as white spaces. With this analysis we have concluded that performance is limited by the fraction of code that is sequential. This sequential code is part of the OpenMP runtime, and represents a significant part of the total execution. In the case of Arm HPC Compiler this sequential code is executed $2\times$ faster than in GCC, which explains its higher scalability.

Another interesting case is the JACOBI_1D benchmark: with 32 threads NEON versions are slower than SCALAR ones. However, for single-threaded executions, NEON versions are over $1.50\times$ faster than SCALAR. Figure 5 depicts two traces of JACOBI_1D obtained with Extrae and visualized with Paraver. Figures 5a and 5b depict the same representative time window for SCALAR-ArmHPC and NEON-ArmHPC executions with 32 threads, respectively. By inspecting both traces, we can see that in the NEON version there is a thread that in each parallel function takes more time to finish its work (thread number 29), leading to a significant amount of

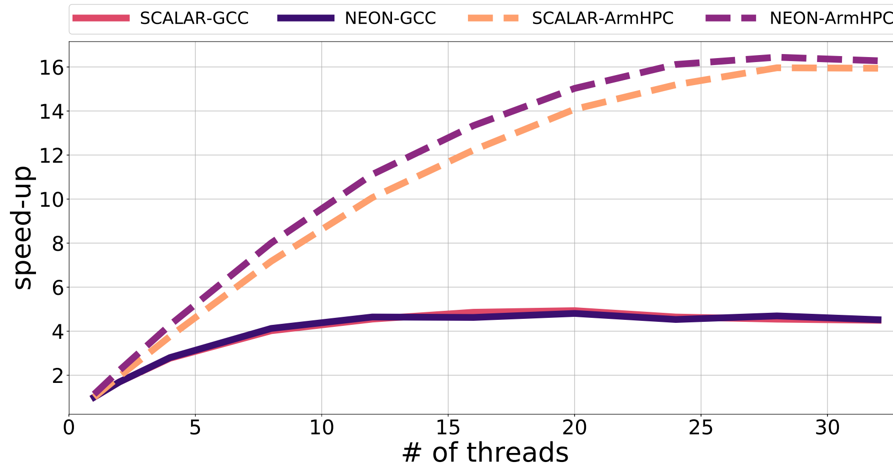



Figure 4: Speed-up of HPCG, normalized to 1-thread SCALAR-ArmHPC, using different number of threads on ThunderX2

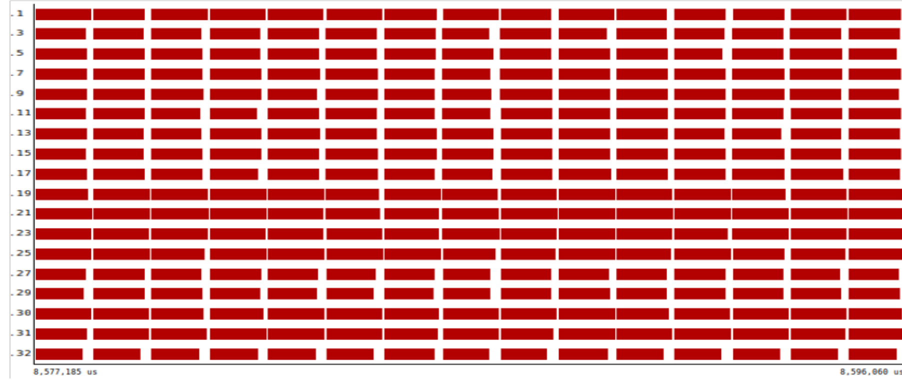
idle time due to this load imbalance. Load imbalance happens when some threads have more work than others or one processor takes more time to finish its work. This problem can typically be solved by changing the scheduling function in the parallel loops.

## 6 Compiler Code Generation

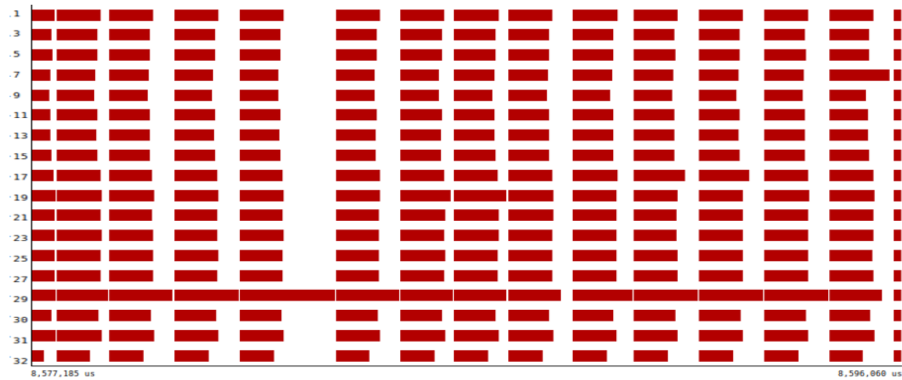
In this section, we provide insights on the different optimizations applied by the Arm HPC Compiler and GCC. As we have seen in Figure 2, there are significant differences between the two compilers in single threaded performance. Hence, we have analyzed the assembly code generated by both compilers. We have found two important optimizations that explain the observed performance differences:

### 6.1 Loop Unrolling and Multiple Memory Access Instructions.

After analyzing the assembly code of all the binaries, we have found that Load Pair (*ldp*) and Store Pair (*stp*) instructions can have a significant impact. The former instruction loads two registers with two consecutive data elements from memory, instead of using two common load instructions. The latter stores two registers into two consecutive data elements in memory. Arm HPC compiler exploits this functionality as it usually applies a two-iteration loop unrolling optimization, and then it reduces the number of memory access instructions using *ldp* and *stp*. 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 and NEON-ArmHPC binaries. In FLOYD_WARSHALL and HACCK this optimization is only applied for vectorized versions. But the Arm HPC Compiler does not only apply




(a) SCALAR-ArmHPC execution trace




(b) NEON-ArmHPC execution trace

Figure 5: Traces of JACOBI_1D binaries compiled with and without SIMD on the Arm HPC compiler. The binaries run on the ThunderX2 with 32 threads (only 18 are plotted for clarity). Y-axis represents the number of threads. X-axis represents time, both share the same time window. Each rectangle represents one parallel function. White represents idle.

*ldp*-optimization to loop unrolling, it also uses it whenever there are two consecutive accesses in the loop body, such as EOS and JACOBI_2D. The latter only applies this in vectorized versions, which explains the superlinear speed-up shown in Figure 2.

GCC is not able to apply these optimizations because the RAJAPerf suite is written using self-defined (non-standard) types. Therefore, we have designed a simple experiment in which we modify FIR benchmark by removing all self-defined types. We have achieved 1.51× speed-up for SCALAR-GCC and 1.32× for NEON-GCC, matching the performance of the Arm HPC Compiler. Therefore, we can say that this instruction reduction is the responsible of an important part of the performance differences between Arm HPC Compiler and GCC. The few cases where GCC uses *ldp* and *stp* by itself are EOS and JACOBI_1D, in which for one loop iteration there are two consecutive accesses to a data array that are replaced by a *ldp*.

Listing 1: COPY assembly code generated with Arm HPC Compiler

```
1 #LOOP: ldp  x17, x18, [x14, #-8]
2 #I     subs x16, x16, #0x2
3 #I     add  x14, x14, #0x10
4        stp  x17, x18, [x15, #-8]
5 #I     add  x15, x15, #0x10
6        b.ne #LOOP
```

Listing 2: COPY assembly code generated with GCC

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

To illustrate the importance of this optimization, we analyse the COPY benchmark (see Listings 1 and 2). In the SCALAR-ArmHPC version, the Arm HPC compiler does a loop unrolling and then reduces the number of load instructions using *ldp*, as we mentioned before. GCC generates a simpler assembly code, but for two iterations of the high-level code, GCC executes 10 instructions and Arm HPC Compiler just 6. This instruction reduction has an impact in the utilization of the inner structures of each core, enabling more memory-level parallelism, i.e., more in-flight memory operations.

To get insights into this impact, we dig into the structure of our benchmarks and the memory access patterns they exhibit. All benchmarks have two nested loops, the outer loop counts the number of times the kernel is executed, and the inner loop(s) executes the body of the region of interest for all the elements present in the data structures. Therefore, in most benchmarks one data element is not accessed again until all the elements of the data structure, or a partitioned block, have been accessed. Because the size of the data structures is chosen to be representative of real HPC applications, these rarely fit in low-latency private caches, leading to long-latency misses. From the three types of misses (i.e., compulsory, capacity, and conflict), most of them belong to the first or the second categories. This memory access pattern is typically called streaming or non-temporal.

Programs with long-latency memory accesses have an interesting property. Imagine we have two different binaries of this long-latency access program, and one version executes less instructions on each iteration of the main loop. It is easy to see that every 1000 instructions, the binary with less instructions in the main loop has more memory accesses in-flight than the other binary. Due to the nature of the accesses, this exposes additional memory-level parallelism. Out-of-order processors are able to request multiple memory blocks in parallel to hide these latencies; therefore, having more outstanding requests in the short-loop binary allows hiding more efficiently these long-latency misses.

Figure 6 shows, for the COPY benchmark, that Arm HPC Compiler generates more misses per kilo instruction (MPKI) than GCC code. This allows for a better utilization of the available memory bandwidth by saturating the available Miss Status Holding Registers (MSHRs). MSHRs are usually limited to a few entries in L1 caches (i.e., 8 to 16 entries).

Analyzing the assembly code in Listings 1 and 2 we can measure MSHR utilization. However, the amount of MSHR entries and cache block size are required for the computation. To illustrate this example, we suppose a cache block size of 64

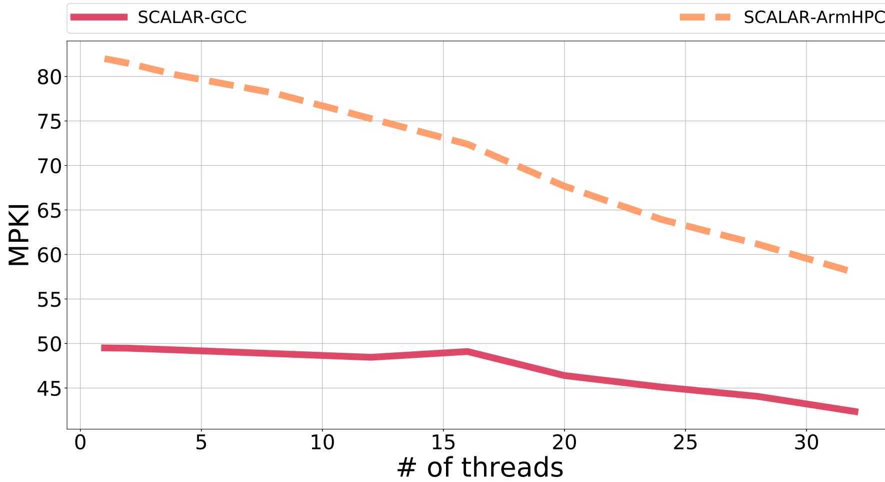



Figure 6: Misses Per Kilo Instruction of SCALAR-ArmHPC and SCALAR-GCC binaries for COPY benchmark. Executed on ThunderX2 with different thread counts.

bytes and 12 MSHR entries. We know that ThunderX2 has a ROB of 180 entries [26]. Therefore, we can compute how many iterations of the assembly loop can be stored in the ROB, which is 30 complete iterations for SCALAR-ArmHPC assembly ($180/6 = 30$) and 36 complete iterations for SCALAR-GCC ($180/5 = 36$).

Because we know the cache block size and the memory accesses pattern (sequential), we know every how many iterations there will be a miss. Thus, we have $\frac{64 \text{ bytes/block}}{8 \text{ bytes/element}} = 8$ elements per block. Notice that *ldp* and *stp* accesses two elements at same time. Therefore, every 4 instructions of *ldp* or *stp* there is a miss. In the case of simple *ldr* or *str* instructions, there is a miss every 8 instructions of this type. Thus, SCALAR-ArmHPC executes 15 misses for the 30 assembly iterations stored in the ROB ($\frac{30}{4} \times 2 = 15$). However, there are only 12 MSHR entries, so we can say that we are fully utilizing all MSHRs, exploiting memory-level parallelism. On the other hand, for SCALAR-GCC there are 9 cache misses ($\frac{36}{8} \times 2 = 9$), which is only a 75% utilization of the MSHRs. With this simple example we have seen how shorter codes lead to a better utilization of the resources of the core and of the available memory bandwidth.

## 6.2 Register Management.

Another important difference between GCC and Arm HPC Compiler is the management of registers. Register content can be classified in three groups: control data (indices, loop counters, addresses), constants (fixed values used on each iteration), and variables. Between the two compilers, most differences are in the first two groups.

For control data, GCC optimizes the use of registers and uses only one register to store loop control counters and indices. To illustrate this optimization, we use the

Listing 3: INT_PREDICT assembly code generated with Arm HPC Compiler

```
1  #LOOP:
2  #I     lsl   x2, x9, #3
3  #C     ldr   d0, [x22]
4         ldr   d1, [x11, x2]
5  #C     ldr   d2, [x20]
6         ldr   d3, [x12, x2]
7  #C     ldr   d4, [x19]
8         fmul  d0, d1, d0
9         ldr   d1, [x13, x2]
10        fmadd d0, d3, d2, d0
11 #C     ldr   d2, [x23]
12        ldr   d3, [x14, x2]
13        fmadd d0, d1, d4, d0
14 #C     ldr   d1, [x21]
15        ldr   d4, [x15, x2]
16        fmadd d0, d3, d2, d0
17 #C     ldr   d2, [x28]
18        ldr   d3, [x16, x2]
19        fmadd d0, d4, d1, d0
20 #C     ldr   d1, [x27]
21        ldr   d4, [x17, x2]
22        fmadd d0, d3, d2, d0
23        ldr   d2, [x18, x2]
24        ldr   d3, [x0, x2]
25        fmadd d0, d4, d1, d0
26        ldr   d1, [x1, x2]
27 #C     ldr   d4, [x26]
28        fadd  d2, d3, d2
29 #I     cmp   x9, x8
30        fadd  d0, d0, d1
31        fmadd d0, d2, d4, d0
32 #I     add   x9, x9, #0x1
33        str   d0, [x10, x2]
34        b.lt  #LOOP
```

Listing 4: INT_PREDICT assembly code generated with GCC

```
1  #LOOP:
2         ldr   d0, [x0, x21, lsl #3]
3         ldr   d1, [x0, x26, lsl #3]
4         ldr   d16,[x0, x28, lsl #3]
5         fmul  d0, d10, d0
6         ldr   d6, [x0, x23, lsl #3]
7         fmadd d1, d9, d1, d0
8         ldr   d5, [x0, x22, lsl #3]
9         ldr   d4, [x0, x24, lsl #3]
10        ldr   d3, [x0, x19, lsl #3]
11        ldr   d2, [x0, x20, lsl #3]
12        ldr   d0, [x0, x27, lsl #3]
13        ldr   d7, [x0, x25, lsl #3]
14        fadd  d1, d1, d16
15        fadd  d0, d0, d7
16        fmadd d1, d11, d6, d1
17        fmadd d1, d12, d5, d1
18        fmadd d1, d13, d4, d1
19        fmadd d1, d14, d3, d1
20        fmadd d1, d15, d2, d1
21        fmadd d0, d8, d0, d1
22        str   d0, [x0], #8
23 #I     cmp   x0, x1
24        b.ne  #LOOP
```

previous code of COPY (Listings 1 and 2) and also INT_PREDICT code (Listings 3 and 4). In the listings, we have labeled each ”index generation” or ”loop control” instruction with the marker #I at the beginning of the line. In these benchmarks, we can see how Arm HPC Compiler uses different registers for indexing addresses and loop control. In Listing 1, line 2 is the loop control instruction and lines 3 and 5 are the index calculations. In Listing 3, lines 29 and 32 are the loop control instructions and line 2 is the index calculation. On the other hand, GCC uses control loop counters as addresses or indices. In Listing 2, lines 3 and 4 are the control loop and the index calculation instructions. In 4, line 23 is the control loop instruction, and address calculation is done using post-increment indexing in line 22.

For COPY and INT_PREDICT benchmarks the amount of extra instructions due to control data is small, only one and two instructions, respectively. However, in the case of VOL3D, Arm HPC Compiler executes 24 instructions to update addresses

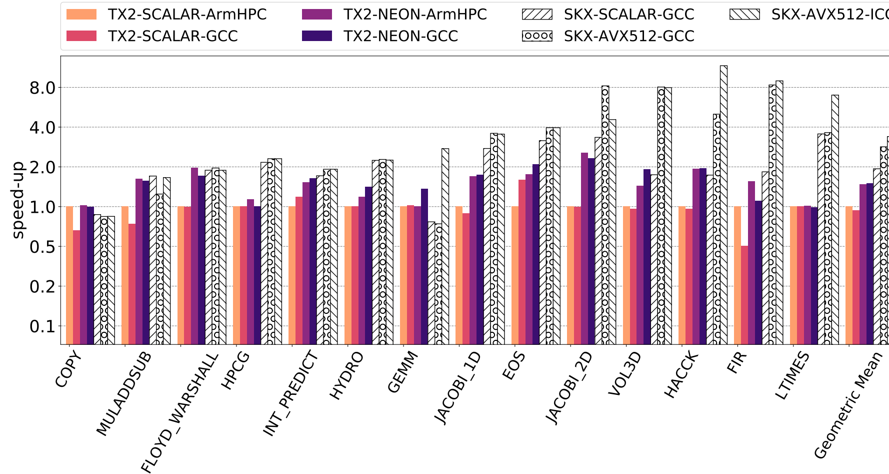



Figure 7: Speed-up of single-threaded runs with respect to TX2-SCALAR-ArmHPC on ThunderX2 and Skylake. Benchmarks sorted by increasing OI.

while GCC only uses one instruction, which represents a 20% increase on the number of instructions in the loop body.

For constants, GCC always tries to keep them stored in registers whenever possible. This makes the core access to constants only once before start executing the kernel loop. On the other hand, Arm HPC Compiler tries to use the minimum number of registers, so every iteration loads constants from memory. Out-of-Order execution hides the latency of accessing these constants from the L1 cache. However, those instructions consume entries in the ROB, slightly harming performance. In Listings 3 and 4 we have labeled the instructions that load constants in every iteration with the tag #C. As we can see, GCC executes 8 instructions less than Arm HPC Compiler (25% less) for constants on each iteration, at the expense of occupying 8 additional registers.

## 7 ThunderX2 and Skylake Performance Comparison

This section presents a quantitative comparison between ThunderX2 and Skylake-based Xeon processors. Both processors are high-end products designed for HPC and server workloads. Figure 7 shows the single-threaded performance speed-up comparison normalized to the execution time of ThunderX2 single-threaded SCALAR-ArmHPC experiments. For the sake of clarity, all ThunderX2 results are the same from Figure 2 with the prefix TX2, and Figure 7 adds the relative speed-ups using the Skylake processor as well. Skylake experiments correspond to 3 versions: GCC without SIMD support (SKX-SCALAR-GCC), GCC with support for AVX512 (SKX-AVX512-GCC), and ICC with support for AVX512 (SKX-AVX512-ICC).

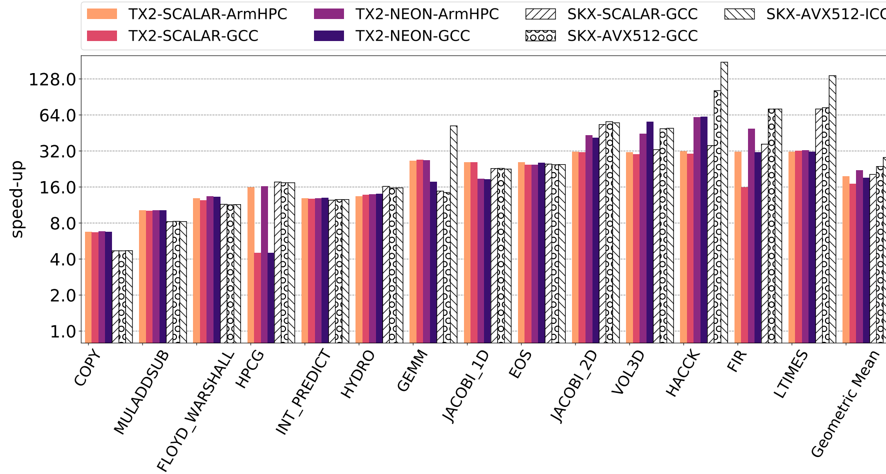



Figure 8: Speed-up of multi-threaded runs with respect to 1-thread TX2-SCALAR-ArmHPC on ThunderX2 and Skylake. Benchmarks sorted by increasing OI.

On average, SKX-SCALAR-GCC is $1.93\times$ faster than SCALAR versions of ThunderX2, i.e., Skylake cores are much more performant due to their aggressive out-of-order design (see Table 1 for the details). When Skylake uses AVX512 extensions, it achieves a modest speed-up of $3.39\times$, while the potential speed-up of AVX512 is up to $8\times$. The modest speed-up is because most benchmarks are already memory bound, that is, the core is already saturating all the memory bandwidth with the scalar versions of the benchmarks. This is apparent in low operational intensity benchmarks (from COPY to HYDRO), for which SKX-SCALAR-GCC is on par with SKX-AVX512-GCC. This precludes the Skylake core from obtaining benefits due to vectorization. For the benchmarks with the lowest OI (COPY, MULADDSUB, FLOYD_WARSHALL), ThunderX2 is able to match Skylake performance. On the other hand, benchmarks with higher OI do benefit from the performance boost AVX512 can offer, which results in over $8\times$ speed-ups when compared to TX2-SCALAR-ArmHPC.

We have seen how Skylake beats ThunderX2 performance for single thread executions. However, ThunderX2 has a significant lead in peak memory bandwidth (170 vs. 120 GB/s), and a more balanced architecture in terms of memory bandwidth per flop. Figure 8 shows the speed-ups for multi-threaded executions. In this case, the lead of Skylake over ThunderX2 diminishes significantly. On average, both SCALAR versions have similar performance, and Skylake is only able to beat TX2-NEON-ArmHPC ($22\times$ speed-up) on high OI benchmarks, especially when using the AVX512 extension ($28.20\times$ speed-up). The additional memory bandwidth available in ThunderX2 has a significant impact in terms of performance. In low OI benchmarks such as COPY, MULADDSUB, FLOYD _WARSHALL, INT_PREDICT, and JACOBI_1D; ThunderX2 is able to outperform Skylake. For high OI benchmarks,

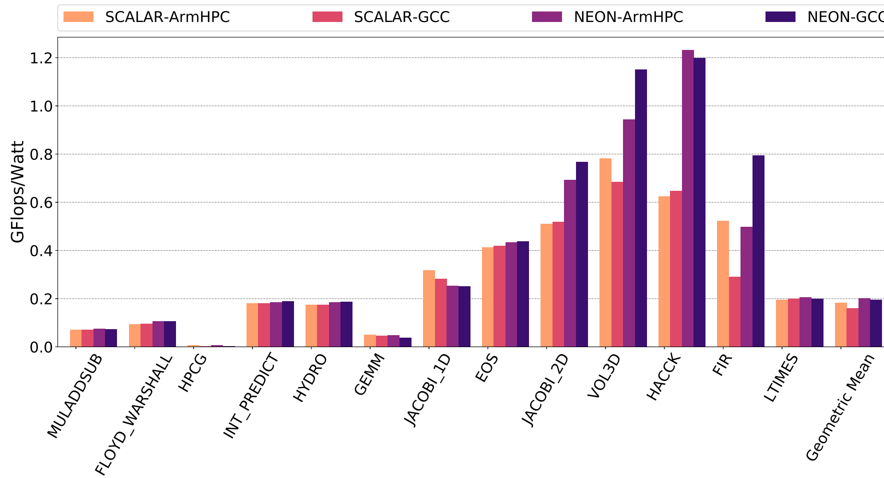



Figure 9: GFlops per Watt consumed for every benchmark executed in ThunderX2 node. Benchmarks are sorted by increasing OI. Benchmarks without floating point operations are omitted for clarity.

ThunderX2 is only able to outperform Skylake in VOL3D. To conclude, HPC application performance can be largely driven by the available memory bandwidth. Despite having a modest single-threaded performance, ThunderX2 is able to compete with a Skylake processor due to its 8 memory channels that yield a better memory bandwidth. In high OI benchmarks, the AVX512 extension makes a large difference as it is able to extract abundant data-level parallelism. The arrival of technologies such as SVE in Arm platforms can help close this performance gap.

## 8 Power Efficiency of ThunderX2 Processor

As mentioned before, power efficiency is of paramount importance in large-scale deployments. In this section, we first measure the power efficiency of ThunderX2, and then compare it with other commercial solutions, namely Skylake. We employ the power measuring infrastructure described in Section 3, which is shared by the two evaluated processors. Using this infrastructure we are able to compute performance-power metrics such as GFlops per Watt. This will enable us to gauge the relationships that exist between the additional power devoted to achieve better performance (i.e., aggressive speculation, wider vector lengths, etc) and the final throughput obtained in terms of GFlops.

Figure 9 depicts the GFlops per Watt achieved for every benchmark. As in previous sections, benchmarks are sorted by operational intensity. We can observe that OI is correlated with performance per power unit. In low OI benchmarks, processors waste most of their cycles waiting for data to arrive. In contrast, high OI benchmarks achieve significantly better performance per unit of power, as they spend most of the

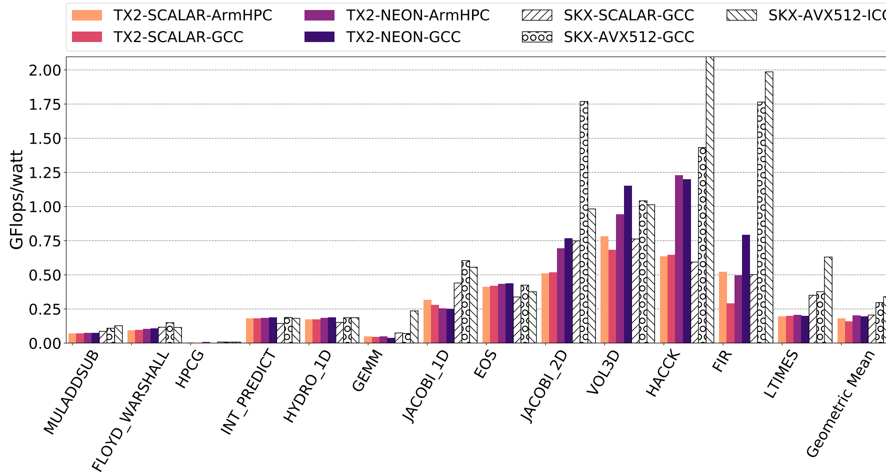



Figure 10: GFlops per Watt consumed for every benchmark executed using 32 threads in ThunderX2 (TX2, dashed bars) and 28 threads in Skylake (SKX). Benchmarks are sorted by increasing OI. Benchmarks without floating point operations are omitted for clarity.

time executing floating point operations. In general, all benchmarks are far from the theoretic maximum GFlops/W for ThunderX2, which is $2.84 = \frac{512 \text{ peak GFlops}}{180 \text{ watts TDP}}$. This is expected as reaching peak floating-point performance is challenging. On average, ThunderX2 achieves an efficiency of 0.20 GFlops/W. HPCG present the lowest power efficiency, around 0.003 GFlops/W for GCC and 0.007 for Arm HPC Compiler.

We have performed the same experiments on the Skylake processor. Figure 10 shows the performance per Watt comparison between ThunderX2 (labeled as TX2) and Skylake (labeled as SKX) processors. As in previous figures, for Skylake we have SCALAR an AVX512 versions using GCC, and AVX512 versions using ICC.

On average, Skylake is around 7% more efficient when comparing SCALAR versions, and 50% when comparing SIMD versions. There are several factors that affect performance-power trade-offs and explain these differences. One factor is the fact that Skylake has two execution units more than ThunderX2, these can improve performance when turned on, and are easy to switch off when unused. Additionally, Skylake uses a decoded instruction cache, which allows turning off the decode and fetch stages frequently. Finally, Skylake's vector length is four times that of ThunderX2. Wider vector lengths present good power efficiency when there is sufficient vector utilization, as can be seen in high OI benchmarks.

The power efficiency of both processors is similar for low OI benchmarks, as performance is memory-bound, which hides microarchitectural differences. On the other hand, in high OI benchmarks, Skylake is clearly more efficient in most benchmarks, as can be seen in JACOBI_1D, JACOBI_2D, HACCK FIR, and LTIMES benchmarks; nonetheless ThunderX2 is better in EOS and VOL3D kernels. Overall, we conclude

that for SCALAR versions both processors have similar performance-power metrics, and the main difference resides in Skylake's wider vectors, which lead to better efficiency when their utilization is high. As mentioned above, technologies like SVE, that will enable wider vectors on Arm platforms, can close this performance-power gap and make future Arm-based SoC more competitive.

## 9 Conclusions

This article characterizes in detail the Marvell ThunderX2 processor running HPC workloads. We have seen how even with 170 GB/s of bandwidth, performance of 8 out of 14 HPC benchmarks is memory bandwidth limited. Memory bandwidth precludes benefits coming from the utilization of SIMD extensions. When bandwidth is not the limiting factor, NEON vectorization improves performance, ranging between $1.32\times$ and $2.0\times$ for JACOBI_2D and VOL3D benchmarks, respectively. However, in some cases, suboptimal vector code generation leads to processor performance that is far from the memory bandwidth and computational ceilings.

We have compared the performance of two different compilers, GCC and Arm HPC Compiler, in order to test their maturity and expose differences in code generation. We can conclude that the Arm HPC Compiler is already a solid tool able to yield better performance than GCC on average. Furthermore, to understand performance differences across the two compilers, we have dug into the details of the generated assembly code to identify the different optimizations applied. In general, simple optimizations are effective in order to speed-up the workloads, which is noticeable in single-thread executions. These optimizations try to reduce the amount of total instructions executed, by replacing multiple consecutive memory accesses by pair memory accesses, by unifying the control loop counters with the array indices, or by keeping constants in registers. However, these optimizations do not make a substantial difference on multi-threaded executions due to the memory bandwidth bottleneck.

We have also presented a comparison between ThunderX2 and a Skylake-based processor. In scalar single-thread performance Skylake, is $1.93\times$ faster than ThunderX2. However, multi-threaded executions in ThunderX2 nearly match Skylake's performance, mainly due to higher memory bandwidth available. In addition, we have also compared the power efficiency of both processors and found that the Skylake architecture is ahead in terms of performance per power unit (GFlops/W). While this difference is not large in scalar executions, i.e., around 7%, it is quite noticeable on SIMD runs, where Skylake is about 50% more efficient due to its AVX512 extension.

In the future, we intend to research the effects of Arm SVE on performance and power efficiency and include additional compilers, in order to have a broader view of the Arm HPC ecosystem.

## References

1. Adrià Armejach, Helena Caminal, Juan M. Cebrian, Rekai González-Alberquilla, Chris Adeniyi-Jones, Mateo Valero, Marc Casas, and Miquel Moretó. Stencil codes on a vector length agnostic

architecture. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT*, pages 13:1–13:12. ACM, 2018.

2. Adrià Armejach, Helena Caminal, Juan M. Cebrian, Rubén Langarita, Rekai González-Alberquilla, Chris Adeniyi-Jones, Mateo Valero, Marc Casas, and Miquel Moretó. Using arm's scalable vector extension on stencil codes. *J. Supercomput.*, 76(3):2039–2062, 2020.
3. Adrià Armejach, Marc Casas, and Miquel Moretó. Design trade-offs for emerging HPC processors based on mobile market technology. *J. Supercomput.*, 75(9):5717–5740, 2019.
4. F.F Banchelli-Gracia, D. Ruiz, Y. Hao-Xu-Lin, and F. Mantovani. Is Arm software ecosystem ready for HPC? In *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.
5. E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2013.
6. BSC. Extrae: Paraver trace-files generator. `https://tools.bsc.es/extrae`, 2019. [Online; accessed 21-July-2019].
7. Inc. Free Software Foundation. GCC 8.2.0. `http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/`, 2019. [Online; accessed 21-July-2019].
8. M. Garcia-Gasulla, F. Mantovani, M. Josep-Fabrego, B. Eguzkitza, and G. Houzeaux. Runtime mechanisms to survive new hpc architectures: A use case in human respiratory simulations. *The International Journal of High Performance Computing Applications*, page 1094342019842919.
9. D. Hackenberg, T. Ilsche, J. Schuchart, R. Schöne, W. E. Nagel, M. Simon, and Y. Georgiou. HDEEM: High definition energy efficiency monitoring. In *Proceedings of the 2Nd International Workshop on Energy Efficient Supercomputing*, E2SC '14, pages 1–10, 2014.
10. Arm Holdings. Arm HPC compiler 19.0. `https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/arm-allinea-studio/download`, 2019. [Online; accessed 21-July-2019].
11. A. Jackson, A. Turner, M. Weiland, N. Johnson, O. Perks, and M. Parsons. Evaluating the arm ecosystem for high performance computing. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '19, New York, NY, USA, 2019. Association for Computing Machinery.
12. A. Jundt, A. Cauble-Chantrenne, A. Tiwari, J. Peraza, M. A. Laurenzano, and L. Carrington. Compute bottlenecks on the new 64-bit arm. In *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing*, E2SC '15, pages 6:1–6:7, 2015.
13. J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. Dip: A parallel program development environment. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, pages 665–674, 1996.
14. Sandia National Laboratories. HPCG benchmark. `https://github.com/hpcg-benchmark/hpcg/`, 2018. [Online; accessed 21-July-2019].
15. Argonne National Laboratory. HACCKernels benchmark. `https://xgitlab.cels.anl.gov/hacc/HACCKernels`, 2018. [Online; accessed 21-July-2019].
16. Lawrence Livermore National Laboratory. RAJAPerf. `https://xgitlab.cels.anl.gov/hacc/HACCKernels`, 2018. [Online; accessed 21-July-2019].
17. M. A. Laurenzano, A. Tiwari, A. Jundt, J. Peraza, W. A. Ward, R. Campbell, and L. Carrington. Characterizing the performance-energy tradeoff of small Arm cores in HPC computation. In Fernando Silva, Inês Dutra, and Vítor Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, pages 124–137, 2014.
18. Kevin T. Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant D. Patel, Trevor N. Mudge, and Steven K. Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pages 315–326. IEEE Computer Society, 2008.
19. Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Yusuf Onur Koçberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Özer, and Babak Falsafi. Scale-out processors. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pages 500–511. IEEE Computer Society, 2012.
20. S. McIntosh-Smith, J. Price, T. Deakin, and A. Poenaru. Comparative benchmarking of the first generation of HPC-optimised Arm processors on Isambard. In *Cray User Group*, 5 2018.
21. F. Petrogalli and P. Walker. LLVM and the automatic vectorization of loops invoking math routines: -FSIMDMATH. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 30–38, Nov 2018.

22. N. Rajovic, A. Rico, F. Mantovani, D. Ruiz, J. O. Vilarrubi, C. Gomez, L. Backes, D. Nieto, H. Servat, X. Martorell, J. Labarta, E. Ayguade, C. Adeniyi-Jones, S. Derradji, H. Gloaguen, P. Lanucara, N. Sanna, J. Mehaut, K. Pouget, B. Videau, E. Boyer, M. Allalen, A. Auweter, D. Brayford, D. Tafani, V. Weinberg, D. Brömmel, R. Halver, J. H. Meinke, R. Beivide, M. Benito, E. Vallejo, M. Valero, and A. Ramirez. The mont-blanc prototype: An alternative approach for HPC systems. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 444–455, Nov 2016.
23. G. Ramirez-Gargallo, M. Garcia-Gasulla, and F. Mantovani. Tensorflow on state-of-the-art HPC clusters: A machine learning use case. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 526–533, May 2019.
24. A. Rico, J. A. Joao, C. Adeniyi-Jones, and R. V. Hensbergen. Arm HPC ecosystem and the reemergence of vectors: Invited paper. In *Proceedings of the Computing Frontiers Conference*, pages 329–334, 2017.
25. 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*, 37(2):26–39, March 2017.
26. Wikichip. Wikichip: Vulcan microarchitecture. `https://en.wikichip.org/wiki/cavium/microarchitectures/vulcan`, 2019. [Online; accessed 21-July-2019].
27. S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.
28. D. Yokoyama, B. Schulze, F. Borges, and G. Mc Evoy. The survey on arm processors for hpc. *The Journal of Supercomputing*, 2019.
29. T. Yoshida. Fujitsu high performance CPU for the post-k computer. In *Hot Chips 30 Symposium (HCS), Series Hot Chips*, volume 18, 2018.