Enabling PAPI support for advanced performance analysis on ThunderX SoC

Daniel Ruiz[1], Enrico Calore[2], and Filippo Mantovani[1]

[1] Barcelona Supercomputing Center, Barcelona, Spain
[2] Università degli Studi di Ferrara and INFN, Ferrara, Italy

Abstract. The interest towards ARM based platforms as HPC solutions increased significantly during the last 5 years, and Cavium has been one of the first SoC provider integrating a 48 cores ARM SoC targeting the high end server market. In this paper we report about the development of the PAPI support for accessing the hardware counters of the Cavium CN8890 SoC. This contribution allows the use of advanced performance analysis tools using the underlying layer of PAPI on innovative machines such as the Cavium one. To show the possible benefits, and test our implementation, we provide as an example, the analysis of a Lattice Botzmann HPC production code, using the performance analysis tools developed within the Barcelona Supercomputing Center, Extrae and Paraver, making use of our PAPI support.

1 Introduction

Since 2011 the European projects Mont-Blanc [3] contribute in enabling ARM architecture in High Performance Computing (HPC). The project has focused on building prototype platforms based on several ARM SoC evaluating both, mobile and server technology and testing production level applications on them [21]. Beside the pioneer effort of Mont-Blanc, several data centers and system integrators have shown recently fresh and genuine interest for ARM based server technology [1, 2, 4].

One of the key steps while porting and tuning scientific parallel applications on new platforms, like the ones of Mont-Blanc, is the performance analysis phase, i.e. the possibility of understanding how different architectural features affect the performance of an application at scale. For this reason the project has always pushed the porting of a complete HPC systems software ecosystem to ARM, including debuggers and performance analysis tools. Besides other contributions, the Barcelona Supercomputing Center ported Extrae, an instrumentation library, and Paraver, an advanced trace analyzer, to ARM platforms.

One of the most innovative and unconventional platforms deployed within the second phase of the Mont-Blanc project has been a mini-cluster based on four computational nodes housing a motherboard with two Cavium ThunderX CN8890 SoCs, each of them featuring 48 ARM cores. The possibility of having 96 cache coherent cores running under the same OS instance opened of course an increasing interest by HPC application developers [13, 7]. Performance analysis became once again a key factor for gaining a clear understanding of such a new architecture. However, the lack of standard software support for accessing hardware counters on Cavium new SoC, did not allow application developers to use tools such as Extrae on those platforms.

The main contribution presented in this paper is the development of the PAPI support for Cavium ThunderX CN8890 SoC. This allows the access of the CPU hardware counters by instrumentation tools like Extrae, and therefore enables advanced performance analysis of parallel applications.

The paper is organized as follows: in Sec. 2 the Cavium platform is briefly introduced, while in Sec. 3 the PAPI support developed for the Cavium SoC is presented. In Sec. 4 we show a basic validation using micro-benchmarks while in Sec. 5 we report a more complex performance analysis of a production grade fluid-dynamics Lattice Boltzmann code.

2 Platform description

The experiments reported in this paper were run on a Gigabyte R270-T61 rack server deployed at Barcelona Supercomputing Center (BSC) within the Mont-Blanc 2 project.

2.1 Hardware Description

In particular, the used compute node features in a single motherboard with a dual socket configuration the following hardware:

- Gigabyte MT60-SC0 motherboard
- 2× Cavum ThunderX Pass2 SoC
    - 48× Cavium CN8890 Rev. 2 cores at 2.0GHz per SoC
    - 16 MB of L2 cache per SoC
    - 192.0 GFlops peak performance per SoC
- 16GB DDR4 per socket

Besides this server, a cluster composed of four other nodes featuring the first revision of the same SoC, in a slightly different node configuration, is also available for scalability tests to the partners of the Mont-Blanc project, anyhow it was not used in this work.

2.2 Software Stack

The software stack deployed on the Thunder cluster is the Mont-Blanc software stack [18]. This has been already deployed and tested on several ARM-based clusters including the Mont-Blanc prototype [3]. It is composed of a set of compilers, runtimes, scientific libraries, frameworks and developer tools.

Developer tools include PAPI (Performance Application Programming Interface) [12]. This tool provides an interface for accessing the PMU (Performance Monitor Unit) featured on almost all the ARM SoCs. However no PAPI support was available at the moment in which we received our first ThunderX boxes.

3 Implementation of PAPI support

When we deployed the first nodes of the Thunder cluster (described in Sec. 2) none of the hardware counters included in the SoC were accessible via standard libraries, e.g., Linux `perf` and PAPI. As most of the instrumentation tools (e.g., Extrae) in the Mont-Blanc software stack use performance counters in order to gather information regarding the application performance, the access to these counters was critical for performing advanced performance analysis of parallel applications within the project.

The steps performed to gain access ot the hardware counters have been: *i)* enabling the support for the ThunderX PMU in the installed Linux Kernel 4.2.6 and *ii)* extend PAPI event definition in order to support the ThunderX SoC and its hardware counters.

3.1 Enabling PMU support in the kernel

The ARMv8 architecture foresees that performance counters can be accessed via a PMU. However, even if strongly recommended, this hardware module is optional on ARMv8 architecture definition. In the case of the Cavium ThunderX, a PMU has been implemented and documented on the processor's hardware technical reference. In order to access the PMU at user level, a proper definition of the PMU itself has to be placed at the DTB (Device Tree Binary) and the Linux kernel needs to support the access to the PMU. Due to the fact that PMU is optional and tight to the implementation of the SoC, the result of this work is SoC dependent.

The kernel running on the ThunderX node was the Linux kernel v4.2.6 with several modifications made by Cavium. We modified the DTS (Device Tree Source) file of the ThunderX SoC to support its PMU. We also modified the Linux Kernel to support the ThunderX PMU. After applying these modifications, we recompiled both DTS and kernel and gained access to hardware counters via the Linux `perf` command. Please note that starting from version 4.4 of the Linux kernel our patch is not needed anymore, as the support for the PMU has become part of the standard kernel release.

3.2 Extending PAPI to support Cavium ThunderX

After making the ThunderX PMU accessible through kernel tools such as Linux `perf`, we extended the PAPI library in order to access the ThunderX hardware counters through it. Besides allowing an easier access to the performance counters, this library offers a common API, allowing to extend the functionalities of several performance analysis tools, within the Mont-Blanc software stack, also to this hardware platform. In general, PAPI detects and then access the performance counters of a given CPU as follows:

1. Identify the CPU by reading `/proc/cpuinfo` file.

2. Identify which performance counters are available by looking at CPU definition files included in the PAPI distribution.
3. Access the performance counter registers by using the *Perfmon* library: `libpfm`.

PAPI library also splits the different performance counters into two categories:

PAPI preset events is a set of 108 events that the PAPI library considers as standard counters. These events have a known alias that can be used directly with PAPI. These events are mapped to native events that may or may not be available depending on the performance counters available in the target CPU.

PAPI native events is a set of a variable number of events that the PAPI library can access. It is left to the user to know to which registers they are mapped. These events are CPU dependent.

We have implemented the support for the Cavium ThunderX CPU on both the `PAPI` and the `libpfm` libraries by providing all the functions and definitions needed to access ThunderX's PMU, including preset and native events.

4 Validation using micro-benchmarks

After enabling the readout of performance counters through the use of the PAPI interface, we performed some tests with custom micro-benchmarks in order to validate readings. As micro-benchmarks we used two custom applications named *fpu_uKernel* and *simd_uKernel*. The first, *fpu_uKernel*, performs i iterations of 32 scalar single (or double) floating-point FMADD (Fused Multiply-Add) operations, using as operands only data already present in registers. Thus no cache or memory accesses are performed. The latter, *simd_uKernel*, based on the same principle, do the same, but executing 32 single (or double) floating-point FMLA (Floating-point fused multiply-accumulate) on 4-float, or 2-double *simd* vectors.

In Tab. 1 we show some significant performance counters acquired using PAPI while running the *fpu_uKernel* and *simd_uKernel* micro-benchmarks for 500 millions of iterations on the Cavium ThunderX. As can be seen on the left part of Tab. 1, acquired data is consistent to what theoretically expected. In fact, the *fpu* micro-benchmark is performing 32 scalar floating-point FMADD, thus $32 \times 500 \times 10^6 = 16 \times 10^9$ operations, which take 1 cycle each for the ThunderX [15].

In the right part of Tab. 1 we do the same for the *simd_uKernel* micro-benchmark. Also in this case the expectations were confirmed for PAPI_TOT_CYC and PAPI_TOT_INS, since in the ThunderX a *simd* instruction can be issued every two cycle on just one pipeline [15]. On the other hand we found a discrepancy between PAPI_FP_INS and PAPI_VEC_INS counters: as all the operations are now vectorial, they should be counted as PAPI_VEC_INS. However on ThunderX they still get counted as scalar floating-point instructions as PAPI_FP_INS. On other ARMv8 SoCs the same benchmark produce the expected result in the

	fpu_uKernel		*simd_uKernel*	
PAPI Event Name	PAPI Value	Theoretical	PAPI Value	Theoretical
PAPI_TOT_CYC	17773056543	16×10^9	32037793993	32×10^9
PAPI_TOT_INS	17009690527	16×10^9	17017190608	16×10^9
PAPI_FP_INS	16000016858	16×10^9	16000020004	0
PAPI_VEC_INS	0	0	0	16×10^9

Table 1: Performance counters recorded while executing $500M$ iterations of the *fpu* and *simd* micro-benchmarks on double precision floating-point data stored in the CPU registers. Read values and theoretical expectations.

PAPI readings. However, this issue seems not to be related to our PAPI support and is still under investigation with Cavium.

5 Enabling advanced performance analysis

To demonstrate the functionality of the described implementation and the obtainable benefits, we used the Extrae tool, introduced in Sec. 5.1, to analyze an actual HPC application, introduced in Sec. 5.2, while running on a machine equipped with two Pass2 ThunderX SoC.

The application was chosen in order to be a real HPC application, already studied in detail, whose performance and behavior are already well known. This gave the possibilities to compare the previously known information about the application with data acquired by Extrae, from the PAPI counters. Moreover, this choice gave the possibility to demonstrate the strength of a possible advanced analysis, enabled by the availability of these metrics, as shown in Sec. 5.4.

5.1 Extrae and Paraver

Extrae is a tool which uses different interposition mechanisms to inject probes into a generic target application in order to collect performance metrics at known applications points to eventually provide the performance analyst a correlation between performance and the application execution. This tool make extensive use of the PAPI interface to collect information regarding the microprocessor performance, allowing to capture such information at the parallel programming calls, but also at the entry and exit points of instrumented user routines.

Extrae is the package devoted to generate Paraver [20] trace-files. Paraver, on the other side, is a visualization tool allowing to have a qualitative global perception of the behavior of an application previously run acquiring Extra traces. The same traces, extracted by Extrae, apart from being visualized by Paraver, could also be fed to a variety of tools, developed within the Extrae eco-system, used to extract various kind of information from traces, as shown in Sec. 5.4.

5.2 Lattice Boltzmann Application

Lattice Boltzmann methods (LB) are widely used in computational fluid dynamics, to describe flows in two and three dimensions. LB methods [24] – discrete in position and momentum spaces – are based on the synthetic dynamics of *populations* sitting at the sites of a discrete lattice. At each time step, populations *propagate* from lattice-site to lattice-site and then incoming populations *collide* among one another, that is, they mix and their values change accordingly. LB models in n dimensions with p populations are labeled as $DnQp$ and in this work we consider a state-of-the-art $D2Q37$ model that correctly reproduces the thermo-hydrodynamical evolution of a fluid in two dimensions, and enforces the equation of state of a perfect gas ($p = \rho T$) [22, 23]; this model has been extensively used for large scale simulations of convective turbulence (e.g., [6, 5]).

A Lattice Boltzmann simulation starts with an initial assignment of the populations and then iterates for each point in the domain, and for as many time-steps as needed, two critical kernel functions:

– propagate which moves populations across lattice sites collecting at each site all populations that will interact at the next phase (collide). Consequently, propagate moves blocks of memory locations allocated at sparse addresses, corresponding to populations of neighbor cells;
– collide which performs all the mathematical steps associated to the computation of new population values at each lattice site at the new time step. Input data for this phase are the populations gathered by the previous propagate phase. This step is the floating point intensive step of the code.

These two kernels take most of the time of any LB simulation. In particular, it has to be noticed that propagate just move data values and it involves a large number of sparse memory accesses, so it is strongly memory-bound. collide, on the other hand, is strongly compute-bound, heavily using the floating-point units of the processor, thus the performance of the floating-point unit is the ultimate bottleneck.

In the last years were developed several implementations of this model, which were used both for convective turbulence studies [6, 5], and as benchmarking applications for programming models and HPC hardware architectures [10, 9, 8]. In this work we utilize an implementation initially developed for Intel CPUs [17], later ported also to the ARMv7 architecture [11] and recently to ARMv8. To fully exploit the high level of parallelism made available by the model, this implementation exploits MPI (Message Passing Interface) to divide computations across several processes, OpenMP to further divide them across threads, and NEON *intrinsics* to exploit vector units where available.

In particular, for all the tests presented in this work, we simulate a 2-dimensional fluid described by a lattice of $L_x \times L_y$ sites. Each of the N_p MPI processes handle a partition of the lattice of size $L_x/N_p \times L_y$ and further divides it across N_t OpenMP threads, which therefore on their turn will handle a sub-lattice of size $\frac{L_x/N_p}{N_t} \times L_y$. MPI processes are logically arranged in a ring, thus simulating a 2-dimensional fluid shaped as the surface of a cylinder. This organization was chosen to avoid the need of boundary conditions on the left and right side of the lattice, implementing them only for the upper and lower parts.

Each sub-lattice handled by each process includes also three left and three right halo-columns. At the beginning of each iteration, processes exchanges the three leftmost and rightmost columns from their sub-lattice, with the previous and next process in the logical ring, saving the received columns in their halo-columns. The algorithm requires a halo thickness of just 3 points, since populations move up to three sites at each time step. As already mentioned, the sub-lattice handled by each process is further divided along the x-dimension across the spawned N_t OpenMP threads. The two threads taking care of the leftmost and rightmost part of the sub-lattice (i.e., the first and the last) for each process, initiate the MPI communications with the left and right neighbors. Moreover, relieving these two threads from part of the propagate computation duties while performing MPI transfers, allow to overlap MPI communications and computations.

In the compute node described in Sec. 2, are available two 48-cores ThunderX SoCs, thus an handy configuration to deploy the LB implementation, hereby introduced, would be to run 48 OpenMP threads for each MPI process and one MPI process per socket. This is indeed the configuration giving the best performances.

5.3 Basic analysis

Using Extrae and Paraver tools [20] and thanks to the PAPI support to read performance counters, we were able to analyze the runtime execution of the LB application introduced in Sec. 5.2. As an example, we show in Fig. 1 a Paraver view of the traces acquired while running the LB application on one ThunderX SoC, using one OpenMP thread per core. The simulation was run for 100 iterations over a lattice of 1536×1024 sites, taking on average $\approx 249ms$ per iteration, of which $75.5ms$ for the propagate and $163.6ms$ for the collide. In the upper part of Fig. 1 we show traces of the full simulation highlighting the lattice initialization phase (A), the compute iterations over time step (B) and the computation of the final mass used to check the consistency of the simulation result (C). In the lower part of Fig. 1 we report an enlargement of just $250ms$ of the whole simulation, highlighting all the components of a single iteration.

In Fig. 1 can be noticed the 48 threads running on a single SoC, and looking at the colors, representing the number of useful completed instructions (PAPI_TOT_INS while inside an OpenMP region), two different color regions can be easily spotted. The green region correspond to threads executing the propagate function, while the blue one correspond to the collide. The first and last thread, as already mentioned, are waived of most of the computation duties for the propagate in order to perform MPI communications instead. This can clearly be noticed in Fig. 1 for threads 1 and 48, although running with just one MPI process is a trivial case from the communications point of view.

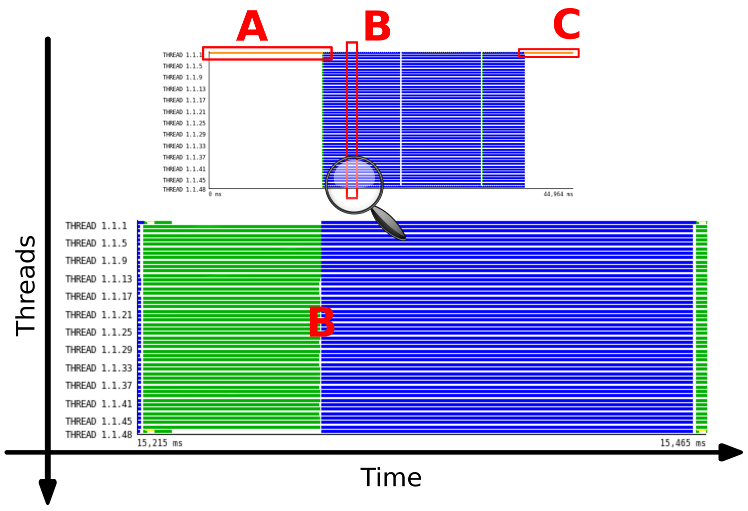

Fig. 1: Traces acquired by Extrae end visualized by Paraver. In the upper part: traces of the full execution.In the lower part: detail of $250ms$ (i.e. $\approx$ one iteration) showing the PAPI_TOT_INS value, while inside an OpenMP region, represented in color scale, per thread. In the run, 1 MPI process was executing binded to one TnhuderX SoC, sub-dividing its lattice across 48 OpenMP threads (i.e., one per core). The green color accounts for $\approx$ 1.8M instructions, while the blue color for $\approx$ 160M instructions.

In Fig. 2 we show traces of the same lattice computed by the two ThunderX SoCs attached to the same motherboard, in this case we run 2 MPI processes binded respectively to the two SoCs, spawning 48 OpenMP threads each for a total of 96 threads. To minimize the figure size we show only the first 3 and last 3 threads for each process. As can be seen in Fig. 2, a perfect overlap between computations and communications can be appreciated.

Knowing the execution length and the number of completed instruction, within Paraver, it can be computed the IPC (Instruction Per Cycle) metric, resulting to an average IPC of 0.01 for the propagate and 0.48 for the collide. Interestingly, we know that the collide function is implemented using NEON *intrinsics* operating on *simd* vectors of 2 doubles and in Tab. 1, we showed that on this SoC, each instruction of this kind needs two cycles to be completed, thus an IPC of 0.48 for the collide tells us that it is almost perfectly optimized. On the other side, for the propagate, such a low IPC just confirms that this function is not limited by the instruction throughput, and in fact we know it to be completely memory-bound.

5.4 Advanced analysis

The acquired metrics, such as the ones plotted in Fig. 1, apart from being visualized can also be used to analyze the application performance and behavior

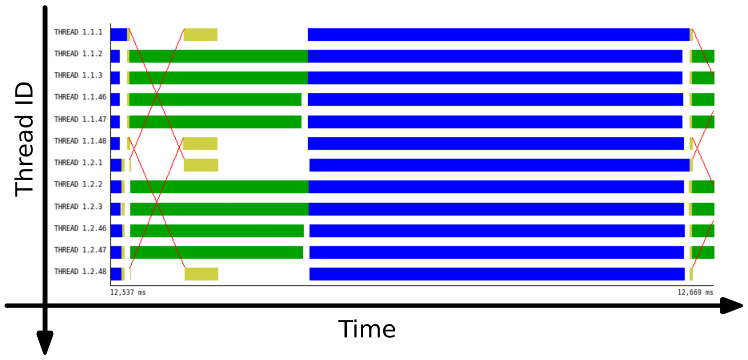

Fig. 2: Traces acquired by Extrae end visualized by Paraver. Detail of $130ms$ (i.e. $\approx$ one iteration) showing the PAPI_TOT_INS value while inside an OpenMP region, represented in color scale, per thread. In the run, 2 MPI processes were executing binded to each of the two TnhuderX SoC embedded in a compute node, sub-dividing its lattice across 96 OpenMP threads (i.e., one per core). To minimize the figure size we show only the first 3 and last 3 threads for each process. The green color accounts for $\approx$ 0.9M instructions, while the blue color for $\approx$ 80M instructions. Red lines represent MPI communications.

changes across different executions. As an example of an "Advanced Analysis", possible thanks to the PAPI events being readable by tools such as Extrae, in this section we will analyze the IPC change, of the already introduced LB code functions, changing the number of threads per SoC. To do this, we will use two other tools from the Extra eco-system, to perform Clustering [14] and Tracking [16].

Clustering or cluster analysis, is a common data mining technique used for classification of data. Data is partitioned into groups called clusters which represent collections of elements that are "close" to each other, based on a distance or similarity function. In this work, we search the trends of the different "CPU bursts" of our application, which are the regions between calls to the OpenMP runtime (i.e., the colored bars in Fig. 1 and Fig. 2). To describe each of the CPU bursts of a parallel application, any of the acquired PAPI events could be taken into account to apply a clustering algorithm. In particular we apply DBSCAN (Density-Based Spatial Clustering of Applications with Noise) clustering algorithm [14], selecting as interesting metrics the instruction completed and IPC. As a result, we obtain different groups of bursts according to these performance counters, from which can be easily distinguished two main clusters, one related to the `propagate` function and one to the `collide`. Once obtained the clusters for different runs of our application changing the number of threads, running on a single SoC, we were able to track [16] the movement of the centroids of such clusters in the IPC / Instructions-completed space, as shown in Fig. 3.

The different runs were performed over the same lattice size with a varying number of threads: 6, 8, 12, 16, 24, 32, 48. This gives 7 cluster centroid points for the `propagate`, on the lower left of Fig 3, and 7 cluster centroid points for the `collide`, on the right of Fig 3. Increasing the number of threads, the centroids for

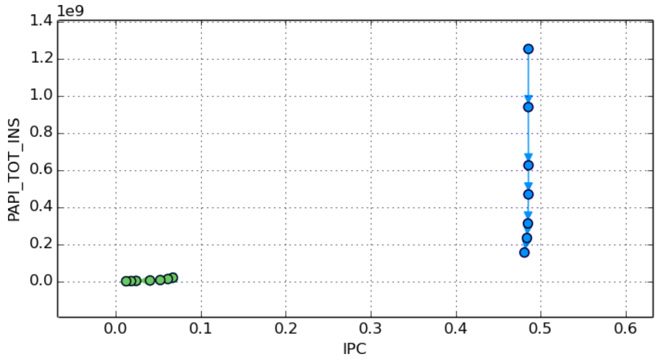

Fig. 3: Tracking of the cluster centroids while increasing the number of OpenMP threads. Each point is the centroid of a cluster and the arrows represent its movement in the IPC / Instructions-completed space, while increasing the number of threads. Green points correspond to the propagate function, while blue ones to the collide.

the propagate move horizontally, from right to left, while for the collide they move along an almost perfectly vertical line from top to bottom. This kind of analysis tell us that the collide function is scaling almost perfectly up to 48 threads since the number of instructions completed by each thread keeps decreasing while IPC of each of them remains constant. This translate to the fact that the more threads working, the less work each of them will have to do and the less CPU cycles each of them will need. On the other hand, for the propagate function, IPC clearly decreases, highlighting the fact that increasing the number of threads they compete for the same resource and thus they start to hamper their own work. The propagate is in fact completely memory-bound.

6 Conclusions

In this paper we reported the work performed within the Barcelona Supercomputing Center for enabling the support of the Cavium ThunderX CN8890 SoC in the PAPI library. This contribution allowed us to access hardware performance counters to enable advanced performance analysis of applications, using the tools developed at BSC. In this paper we also demonstrated its usefulness analyzing with such tools a well known Lattice Boltzmann application, while running on this SoC. The patch is available on-line[3] and we plan to release it for being included in the main distribution of PAPI.

As the ThunderX also embeds registers storing power related information, we plan to extend this work, including the possibility to read power figures from the SoC using PAPI [25], thus enabling fine grained energy analyses [11, 19] without the need for external power-meters.

[3] `https://goo.gl/MXn2h2`

Acknowledgements The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] and Horizon 2020 under the Mont-Blanc projects [3], grant agreements n. 288777, 610402 and 671697. E.C. was partially founded by "Contributo 5 per mille assegnato all'Università degli Studi di Ferrara - dichiarazione dei redditi dell'anno 2014". Cavium Inc. has kindly supported this research providing access to documentation and platforms.

References

1. GW4 Tier 2 HPC: Isambard | GW4, `http://gw4.ac.uk/isambard/`
2. ISC16 Recap Fujitsu Takes the Stage - Processors blog - Processors - ARM Community, `https://community.arm.com/processors/b/blog/posts/isc16-recap-fujitsu-takes-the-stage`
3. Mont-Blanc Project, `http://www.montblanc-project.eu/`
4. ARM Waving: Attention, Deployments, and Development (Jan 2017), `https://www.hpcwire.com/2017/01/18/arm-waving-gathering-attention/`
5. Biferale, L., Mantovani, F., Sbragaglia, M., Scagliarini, A., Toschi, F., Tripiccione, R.: Reactive Rayleigh-Taylor systems: Front propagation and non-stationarity. EPL 94(5), 54004 (2011), `doi:10.1209/0295-5075/94/54004`
6. Biferale, L., Mantovani, F., Sbragaglia, M., Scagliarini, A., Toschi, F., Tripiccione, R.: Second-order closure in stratified turbulence: Simulations and modeling of bulk and entrainment regions. Physical Review E 84(1), 016305 (2011), `doi:10.1103/PhysRevE.84.016305`
7. Bortolotti, D., Tinti, S., Alto, P., Bartolini, A.: User-space APIs for dynamic power management in many-core ARMv8 computing nodes. In: 2016 International Conference on High Performance Computing Simulation (HPCS). pp. 675–681 (Jul 2016), `doi:10.1109/HPCSim.2016.7568400`
8. Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S.F., Tripiccione, R.: Massively parallel latticeboltzmann codes on large GPU clusters. Parallel Computing 58, 1 – 24 (2016), `doi:10.1016/j.parco.2016.08.005`
9. Calore, E., Gabbana, A., Kraus, J., Schifano, S.F., Tripiccione, R.: Performance and portability of accelerated lattice Boltzmann applications with OpenACC. Concurrency and Computation: Practice and Experience 28(12), 3485–3502 (2016), `doi:10.1002/cpe.3862`
10. Calore, E., Schifano, S.F., Tripiccione, R.: On portability, performance and scalability of an MPI OpenCL lattice boltzmann code. In: Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II, pp. 438–449. LNCS, Springer (2014), `doi:10.1007/978-3-319-14313-2_37`
11. Calore, E., Schifano, S.F., Tripiccione, R.: Energy-performance tradeoffs for HPC applications on low power processors. In: Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers. pp. 737–748. LNCS, Springer (2015), `doi:10.1007/978-3-319-27308-2_59`
12. Dongarra, J., London, K., Moore, S., Mucci, P., Terpstra, D.: Using PAPI for hardware performance monitoring on linux systems. In: Conference on Linux Clusters: The HPC Revolution. vol. 5. Linux Clusters Institute (2001)

13. Gelas, J.D.: Investigating Cavium's ThunderX: The First ARM Server SoC With Ambition, `http://www.anandtech.com/show/10353/investigating-cavium-thunderx-48-arm-cores`
14. Gonzalez, J., Gimenez, J., Labarta, J.: Automatic detection of parallel applications computation phases. In: 2009 IEEE International Symposium on Parallel Distributed Processing. pp. 1–11 (May 2009), `doi:10.1109/IPDPS.2009.5161027`
15. Gwennap, L.: Thunderx rattles server market. Microprocessor Report 29(6), 1–4 (2014)
16. Llort, G., Servat, H., Gonzlez, J., Gimnez, J., Labarta, J.: On the usefulness of object tracking techniques in performance analysis. In: 2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC). pp. 1–11 (Nov 2013), `doi:10.1145/2503210.2503267`
17. Mantovani, F., Pivanti, M., Schifano, S.F., Tripiccione, R.: Performance issues on many-core processors: A D2Q37 lattice boltzmann scheme as a test-case. Computers & Fluids 88, 743 – 752 (2013), `doi:10.1016/j.compfluid.2013.05.014`
18. Mantovani, F., Ruiz, D., Vilarrubi, O., Auweter, A., Tafani, D., Adeniyi-Jones, C., Gloaguen, H., Iglesias, G.U.: D5.11 - final report on porting and tuning the system software to ARM architecture. Tech. rep. (2015), `http://www.montblanc-project.eu/sites/default/files/D5.11%20Final%20report%20on%20porting%20and%20tuning.%20V1.0.pdf`
19. Nikolskiy, V.P., Stegailov, V.V., Vecher, V.S.: Efficiency of the Tegra K1 and X1 systems-on-chip for classical molecular dynamics. In: 2016 International Conference on High Performance Computing Simulation (HPCS). pp. 682–689 (July 2016), `doi:10.1109/HPCSim.2016.7568401`
20. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: transputer and occam developments. vol. 44, pp. 17–31 (1995)
21. Rajovic, N., Rico, A., Mantovani, F., Ruiz, D., Vilarrubi, J.O., Gomez, C., Backes, L., Nieto, D., Servat, H., Martorell, X., Labarta, J., Ayguade, E., Adeniyi-Jones, C., Derradji, S., Gloaguen, H., Lanucara, P., Sanna, N., Mehaut, J.F., Pouget, K., Videau, B., Boyer, E., Allalen, M., Auweter, A., Brayford, D., Tafani, D., Weinberg, V., Brmmel, D., Halver, R., Meinke, J.H., Beivide, R., Benito, M., Vallejo, E., Valero, M., Ramirez, A.: The Mont-blanc Prototype: An Alternative Approach for HPC Systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 38:1–38:12. SC '16, IEEE Press, Piscataway, NJ, USA (2016), `http://dl.acm.org/citation.cfm?id=3014904.3014955`
22. Sbragaglia, M., Benzi, R., Biferale, L., Chen, H., Shan, X., Succi, S.: Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria. Journal of Fluid Mechanics 628, 299–309 (2009), `doi:10.1017/S002211200900665X`
23. Scagliarini, A., Biferale, L., Sbragaglia, M., Sugiyama, K., Toschi, F.: Lattice Boltzmann methods for thermal flows: Continuum limit and applications to compressible Rayleigh–Taylor systems. Physics of Fluids (1994-present) 22(5), 055101 (2010), `doi:10.1063/1.3392774`
24. Succi, S.: The Lattice-Boltzmann Equation. Oxford university press, Oxford (2001)
25. Weaver, V., Johnson, M., Kasichayanula, K., Ralph, J., Luszczek, P., Terpstra, D., Moore, S.: Measuring energy and power with PAPI. In: Parallel Processing Workshops (ICPPW), 2012 41st International Conference on. pp. 262–268 (2012), `doi:10.1109/ICPPW.2012.39`