# Advanced performance analysis of HPC workloads on Cavium ThunderX

Enrico Calore
*University of Ferrara and INFN Ferrara*
Ferrara, Italy
enrico.calore@unife.it

Filippo Mantovani
*Barcelona Supercomputing Center*
Barcelona, Spain
filippo.mantovani@bsc.es

Daniel Ruiz
*Barcelona Supercomputing Center*
Barcelona, Spain
daniel.ruiz@bsc.es

***Abstract*—The interest towards Arm based platforms as HPC solutions increased significantly during the last 5 years. In this paper we show that, in contrast to the early days of pioneer tests, several application performance analysis techniques can now be applied also to Arm based SoCs. To show the possibilities offered by the available tools, we provide as an example, the analysis of a Lattice Boltzmann HPC production code, highly optimized for several architectures and now ported also to Armv8. We tested it on a system based on a production silicon, Cavium CN8890 SoC. In particular, as performance analysis tools we adopt Extrae and Paraver, making use of the PAPI support, initially developed by us for the ThunderX platform, and now available also upstream. The contribution of this paper is twofold: first, we demonstrate that performance analysis tools available on standard HPC platforms, independently from the CPU providers, are nowadays available also for Arm SoCs; second, we actually optimize an HPC application for this platforms, showing similarities with other architectures.**

***Index Terms*—HPC, Performance, Analysis, Arm, ThunderX, Extrae, Paraver**

## I. Introduction and related works

The HPC community is increasingly growing its attention towards the Arm architecture, considering it more and more as a viable solution for production HPC systems [1]. State-of-the-art research projects such as the European Mont-Blanc [2], the Japanese Post-K [3], and the UK's GW4 / EPSRC [4] have deployed, or are going to deploy, fairly large Arm-based HPC platforms.

Prototypes and hardware platforms are often used to evaluate new technologies and boost the software development as well as the porting process of HPC applications. While attention is primarily given to hardware platforms, the availability of a mature software ecosystem and the possibility of performing advanced performance analysis is vital in the evaluation process of a new technology, such as Arm, in HPC.

One of the most innovative Arm-based platforms deployed at Barcelona Supercomputing Center (BSC) within the second phase of the Mont-Blanc project has been a mini-cluster based on four computational nodes, each 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 [5]–[7]. This interest was further boosted after the recent Cavium's announcement about the second release of the ThunderX chip, including features targeting the datacenter and the HPC market [8].

With new complex platforms appearing at faster and faster pace, the performance analysis of HPC workloads became once again a key factor for gaining a clear understanding of how different architectural features affect the performance of an HPC application at scale. However, initially, the lack of standard software support for accessing hardware counters on Cavium new SoCs, did not allow application developers to use standard HPC performance analysis tools on those platforms. The Mont-Blanc 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.

An up-to-date review of state-of-the-art performance analysis tools is presented by B. Mohr in [9], while in [10] Stanisic et al. already provides a study of HPC benchmarks on a cluster of Arm Cortex-A9 using several tools, including Paraver. Extrae and Paraver are indeed valuable tools for bottleneck identification and performance analysis in parallel applications, as shown by G. S. Markomanolis et al. in [11].

The initial lack of support for accessing hardware counters on Cavium ThunderX, prevented the use of such analysis tools. This problem drove our efforts in the direction of supporting and extending the PAPI library – one of the most common library for accessing performance and energy counters – to support also this platform [12]. Recent works, like the one of Jagode et al. [13], seem to confirm that this is the way to go. Also, the possibility of accessing energy/power related counters as PAPI events is under discussion with Cavium and will be taken into account for future works going in a similar direction of Servat et al. in [14], but for Arm based clusters.

This paper is organized as follows: in Sec. II the Cavium platform is briefly introduced; in Sec. III the PAPI support developed for the Cavium SoC is presented; in Sec. IV we show a basic validation using micro-benchmarks; in Sec. V we introduce the profiling tools and the production grade fluid-dynamics Lattice Boltzmann application used as a case study; in Sec. VI we report the performance analysis and in Sec. VII we eventually present our conclusions.

## II. Platform description

The experiments reported in this paper have been run on a Gigabyte R270-T61 rack server, a node of the *Thunder Cluster* installed at BSC in the context of the Mont-Blanc 2 project [15]. In particular, the compute node used in this work features in a single Gigabyte MT60-SC0 motherboard: 32GB DDR4 system memory and two Cavum ThunderX SoCs. Each processor embed $48\times$ Cavium CN8890 Rev2 cores at 2.0 GHz and 16 MB of L2 cache, reaching 192 GFlops DP peak performance.

The software stack deployed on the Thunder cluster is the Mont-Blanc software stack [16]. This has been already deployed and tested on several Arm-based clusters including the Mont-Blanc prototype [15]. It is composed of a set of compilers, runtimes, scientific libraries, frameworks and developer tools. The developer tools include PAPI (Performance Application Programming Interface) [17]. This tool provides an interface for accessing the Performance Monitor Unit (PMU) featured on almost all the Arm SoCs. However no PAPI support was available at the moment in which we received our first ThunderX boxes.

## III. Implementation of PAPI support

When we deployed the first compute nodes described in Sec. II, none of the hardware counters included in the ThunderX SoC were accessible via standard libraries, e.g., Linux `perf` and PAPI [17]. Besides allowing an easier access to the performance counters, the PAPI library offers a common API, allowing to extend the functionalities of several performance analysis tools, also to this hardware platform. As several HPC instrumentation tools use performance counters in order to profile applications, the access to these counters was critical for performing advanced performance analysis of parallel applications within the project. To gain access to the hardware counters, two main steps had to be done: *i)* add the support for the ThunderX PMU in the installed Linux Kernel 4.2.6 and *ii)* extend PAPI event definitions in order to support the ThunderX SoC and its hardware counters.

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 common and implementation defined PMUv3 events [12]. 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[1].

## IV. Validation using micro-benchmarks

After enabling the readout of performance counters through the use of the PAPI interface, we performed a test set of 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 a *for* loop containing 32 scalar single (or double) floating-point FMADD (Fused Multiply-Add) operations, using as operands only data already present in CPU 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* vector register.

TABLE I
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.

| | *fpu_uKernel* | |
|---|---|---|
| PAPI Event Name | PAPI Value | Theoretical |
| PAPI_TOT_CYC | $17.773\times10^9$ | $> 16 \times 10^9$ |
| PAPI_TOT_INS | $17.009\times10^9$ | $> 16 \times 10^9$ |
| PAPI_FP_INS | $16.000\times10^9$ | $= 16 \times 10^9$ |
| PAPI_VEC_INS | 0 | 0 |
| | *simd_uKernel* | |
| PAPI Event Name | PAPI Value | Theoretical |
| PAPI_TOT_CYC | $32.037\times10^9$ | $> 32 \times 10^9$ |
| PAPI_TOT_INS | $17.017\times10^9$ | $> 16 \times 10^9$ |
| PAPI_FP_INS | $16.000\times10^9$ | 0 |
| PAPI_VEC_INS | 0 | $= 16 \times 10^9$ |

In Tab. I 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 one Cavium ThunderX. In particular we highlight here one counter (i.e. PAPI_VEC_INS) which we found to be not consistent with the Armv8 specification.

Concerning the upper part of Tab. I, acquired counters are 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 one cycle each for the ThunderX [18]. We expect therefore PAPI_TOT_CYC to be equal or higher than PAPI_TOT_INS and PAPI_TOT_INS to be higher (due to integer operations for cycle handling) than PAPI_FP_INS, which on its turn should equal to $16 \times 10^9$.

In the lower part of Tab. I 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 [18]. 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. However, this issue seems not to be related to our PAPI support and is still under investigation with Cavium.

[1]Since version 4.4 of the Linux kernel the support for the PMU has become part of the official release. The same holds true for our patch to *libpfm4* and PAPI which is now implemented upstream.

## V. Application and profiling tools description

To demonstrate the functionality of the described implementation and the obtainable benefits, we used the Extrae tool (introduced in the following Sec. V-A), to analyze an actual HPC application (introduced in the following Sec. V-B), while running on a machine equipped with two production ThunderX SoC. The application was chosen in order to be an actual HPC application, already studied in detail on other platforms, whose performance and behavior are already well known on more traditional HPC architectures. This gave the possibility to compare the previously known information with data acquired by Extrae, from the PAPI counters of the ThuderX SoC. This allows to demonstrate the strength of a possible advanced analysis, enabled by the availability of these metrics, as shown in the following Sec. VI-B.

### A. 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 [19] tracefiles. 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 Extrae 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 ecosystem, used to extract various kind of information from traces, as shown in the following Sec. VI-B.

### B. Lattice Boltzmann Application

Lattice Boltzmann methods (LB) are widely used in computational fluid dynamics, to describe flows in two and three dimensions. LB methods [20] – 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 ($p = \rho T$) [21], [22].

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: i) the propagate function, which moves populations across lattice sites collecting at each site all populations that will interact at the next phase; ii) the collide function 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 execution 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 (on most architectures), heavily using the floating-point units of the processor.

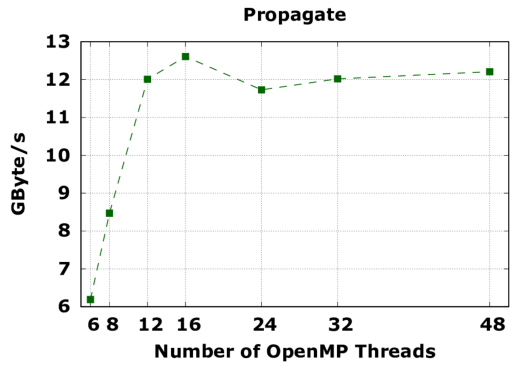
In the last years several implementations of this model were developed, which were used both for convective turbulence studies [23], [24], and as benchmarking applications for programming models and HPC hardware architectures [25]–[27]. In this work we utilize an implementation initially developed for Intel CPUs [28], later ported also to the Armv7 architecture [29] 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 boundaries.

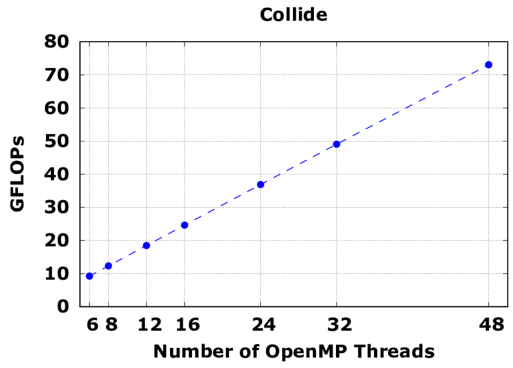
Each sub-lattice handled by each process includes also three left and three right halo-columns. At the beginning of each iteration, processes exchanges the three leftmost and rightmost columns from their sub-lattice, with the previous and next process in the logical ring, saving the received columns 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. 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. Waiving these two threads from most of the propagate duties while performing MPI transfers, allow to overlap MPI communications and computations.

### C. Performance results

In the compute node described in Sec. II, two ThunderX SoCs are available, having 48-cores each, 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 overall simulation performance. Despite of this, a simple single socket performance analysis




(a) Propagate




(b) Collide

Fig. 1. Bandwidth and GFLOPs (in double precision) respectively for the propagate (left) and collide (right) functions, for a growing number of OpenMP threads, on one ThunderX SoC. Lattice size: $1536 \times 1024$ sites.

of the two main functions of this code, reported in Fig. 1, shows that changing the number of OpenMP threads executing on one ThunderX SoC, the maximum bandwidth and FLOPs of these functions are reached at different threads numbers. More precisely, the propagate function reaches its maximum bandwidth of 12.6 GB/s when using 16 threads, while collide reaches its maximum performance of 73 GFLOPs (in double precision) when using 48 threads. The maximum bandwidth reachable by the STREAM benchmark [30] on this system is 39.6 GB/s [6], while the theoretical peak performance equals to 192 GFLOPs (computed as: 48 cores $\times$ 2 GHz $\times$ 0.5 instructions per cycle $\times$ 2 *simd* vector length $\times$ 2 floating point operations per FMLA instruction). Therefore, collide reaches $\sim 38\%$ of the theoretical peak performance, while propagate reaches $\sim 32\%$ of the available bandwidth.

On other architectures collide reaches similar fractions of the peak performance e.g., 36% for Intel E5-2630v3 and 30% for Intel Xeon-Phi 7120X [27]. On the other hand, propagate may also reach higher fractions of bandwidth e.g., 75% for Intel E5-2630v3, although just 28% for Intel Xeon-Phi 7120X [27].

## VI. Application Performance Analysis

The preliminary performance results described in Sec. V highlight an almost perfect scaling for the collide function, which is also able to reach a fair fraction of the peak performance, but on the other hand, show a possible optimization space for the propagate function, which seems to saturate the memory subsystem resources before reaching the maximum available bandwidth.

### A. Basic analysis

Using Extrae and Paraver tools [19] and thanks to the PAPI support to read performance counters, we are able to analyze the runtime execution of the LB application introduced in Sec. V-B. As an example, we show in Fig. 2 a Paraver view of the traces acquired while running the LB application on one ThunderX SoC, using one OpenMP thread per core (i.e., 48 threads). The simulation has run for 100 iterations over a lattice of $1536 \times 1024$ sites, taking on average $\sim 249ms$ per iteration, of which $75.5ms$ for the propagate and $163.6ms$ for the collide. In the upper part of Fig. 2 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. 2 we report an enlargement of just $250ms$ of the whole simulation, highlighting all the components of a single iteration.

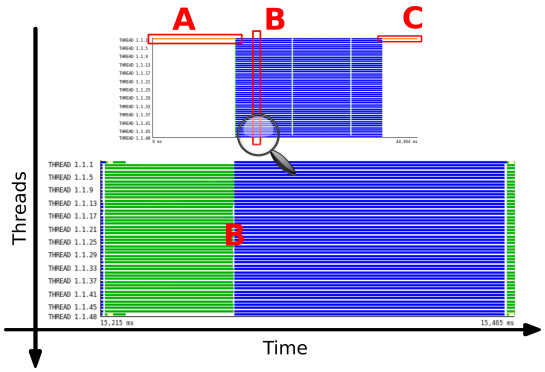



Fig. 2. Traces acquired by Extrae end visualized by Paraver. In the upper part: traces of the full execution. In the lower part: detail of $250ms$ (i.e. approximately one iteration) showing the PAPI_TOT_INS value, while inside an OpenMP region, represented in color scale, per thread. In the run, 1 MPI process was executing bound to one ThunderX SoC, sub-dividing its lattice across 48 OpenMP threads (i.e., one per core). The green color accounts for $\sim$ 1.8M instructions, while the blue color for $\sim$ 160M instructions.

In Fig. 2 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. 2 for threads 1 and 48.

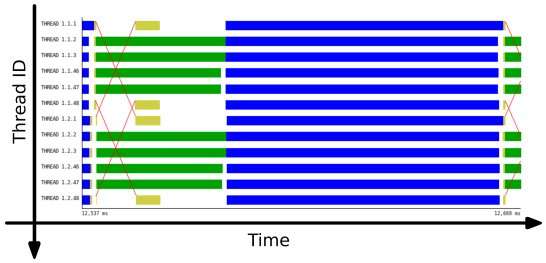



Fig. 3. Traces acquired by Extrae end visualized by Paraver. Detail of $130ms$ (i.e. approximately one iteration) showing the PAPI_TOT_INS value while inside an OpenMP region, represented in color scale, per thread. In the run, 2 MPI processes were executing bound to each of the two ThunderX SoC embedded 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 $\sim$ 0.9M instructions, while the blue color for $\sim$ 80M instructions. Red lines represent MPI communications.

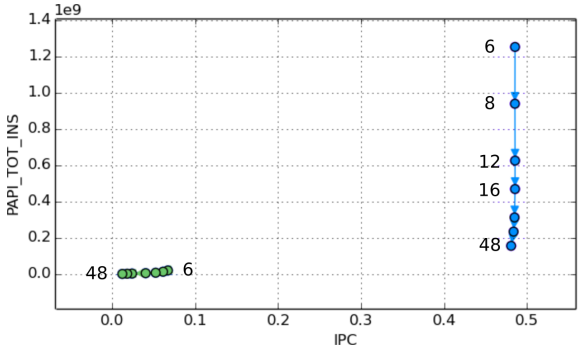



Fig. 4. Tracking of the clusters' centroids. Each point is the centroid of a cluster and the arrows represent its movement in the IPC / Instructions-completed space while increasing the number of OpenMP threads, reported as labels. Green points correspond to the propagate function, while blue ones to the collide.

Knowing the execution length and the number of completed instruction, within Paraver, the IPC (Instruction Per Cycle) metric can be computed, 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. I, we showed that on this SoC, each instruction of this kind needs two cycles to be completed, thus an IPC of 0.48, out of a maximum of 0.5, tells us that the collide phase of our code can take advantage of 96% of the instruction throughput of the cores in the SoC under analysis. This holds true despite the fact that it reaches $\sim$ 38% of the theoretical floating-point peak performance. This may seems a modest fraction, but the theoretical peak is computed taking into account the execution of just fused multiply-add instructions (accounting for 2 FLOP each), which are used by our code, but are obviously not the only instructions executed, as is commonly true for any real world application. 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 highly memory-bound.

To show that similar analysis can be easily performed also running several MPI processes, in Fig. 3 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 bound 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. 3, a perfect overlap between computations and communications can be appreciated.

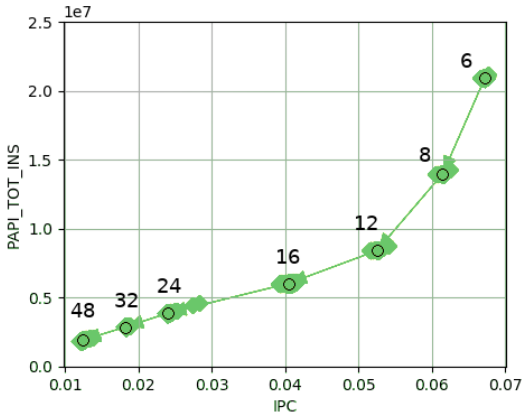
Anyhow in the rest of this work we will continue the analysis running a single process on a single ThunderX SoC.

### B. Advanced analysis

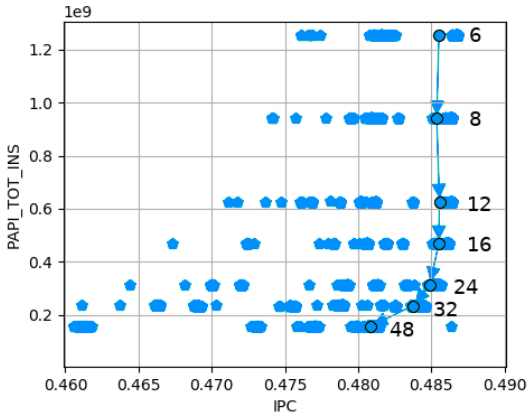
The acquired metrics, such as the ones plotted in Fig. 2, apart from being visualized can also be used to analyze the application performance and behavior changes across different executions. Given the performance results described at the end of Sec. V-B, one may want to analyze the reasons causing the sub-optimal exploitation of the available memory bandwidth by the propagate, or better understand the reasons for the peculiar pattern of its bandwidth, shown in Fig. 1, highlighting a decrease for more than 16 threads. Therefore, in this section we study the evolution of various metrics, while changing the number of threads per SoC, studying in particular the behavior of the propagate function, as an example of an "Advanced Analysis", possible thanks to the PAPI events being readable by tools such as Extrae. To accomplish this task, we use two other tools from the Extrae ecosystem, to perform Clustering [31], [32] and Tracking [33].

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. 2 and Fig. 3). 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 [31], 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 [33] the movement of the centroids of such clusters in the IPC / Instructions-completed space, as shown in Fig. 4.

The different runs were performed over the same lattice size with a varying number of threads, i.e., 6, 8, 12, 16, 24, 32, 48. This gives 7 cluster centroid points for the propagate, on the lower left of Fig 4, and 7 cluster centroid points for the collide, on the right of Fig 4. Increasing the number of threads, the centroids for the propagate move horizontally, from right to left, while for the collide they move along




(a) Propagate




(b) Collide

Fig. 5. Enlargement of the same data of Fig. 4 over the propagate (5a) and collide (5b) clusters. Here we plot every point, corresponding to each function iteration, plus the cluster centroid, highlighting the variability between different iterations.

an almost perfectly vertical line from top to bottom. This kind of analysis tell us that the collide function (in blue) 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, i.e. the less instruction each of them will execute. On the other hand, for the propagate function (in green), IPC clearly decreases, highlighting the fact that increasing the number of threads, they compete for the same resource and thus they start to hamper each other work.

A closer inspection of the propagate and collide clusters, reported in Fig. 5 plotting a point for each iteration and the cluster centroid, show us a slightly more complex dynamic. Here we see that also for the collide there is a moderated ($\sim 1\%$) IPC decrease, associated with an increased variability in the IPC of each iteration, for growing number of threads. For the propagate, as already mentioned, the IPC decreases drastically ($\sim 86\%$) and one may desire to further investigate

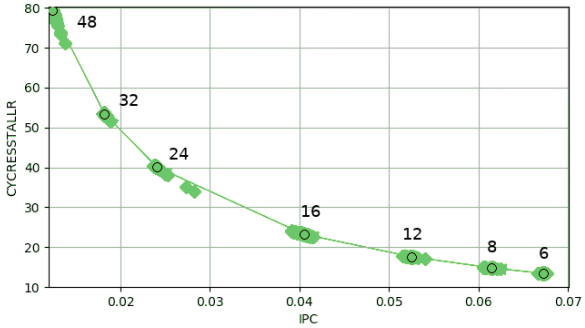


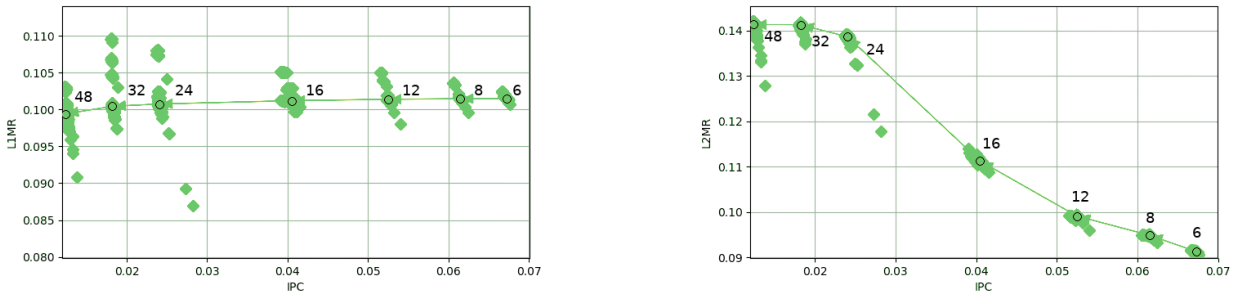

Fig. 6. Tracking of the clusters' movement in the Cycle-wasted-in-resource-stall-ratio / IPC space, while increasing the number of OpenMP threads for the propagate function.

a more specific reason for this effect. To do so, we may look for other performance metrics, looking for the ones correlated to the IPC decrease.

In particular we have found a very strong correlation with the number of cycles wasted on a resource stall, as shown in Fig. 6, where we report the ratio of this metric per instruction retired. The correlation is clear, but this metric is quite generic, thus we may look to more specific ones, such as the miss ratios in the different cache levels. In Fig. 7a we show the L1 cache miss ratio, while in Fig. 7b the L2 cache miss ratio, where we can clearly see that the L1 miss ratio is pretty constant, while the L2 miss ratio is increasing rapidly up to 24 threads and then it reaches a plateau. We remark that on this architecture the L2 cache is shared between all the 48 cores [6] and it is the LLC (Last Level Cache), thus a miss in the L2 causes a DRAM access. In particular this architecture does not seem to support multiple outstanding L1 misses, thus a second cache miss needs to wait until the first one gets handled, reflecting on all the subsequent levels, giving latencies in the order of 40/80 cycles for an L2 hit and 103/206 ns for a DRAM access [6].

This partially explains the increase of cycles wasted on a resource stall, but for that metric we couldn't see a plateau over 24 threads, thus there should be at least another cause behind the higher stalls while running with 32 and 48 threads. Since the main suspect is the memory subsystem, we have further investigated in this direction. Plotting the TLB miss ratio, as shown in Fig. 8, we can appreciate that it is still increasing significantly when running with more than 24 threads, being strongly correlated with the IPC decrease indeed.

The above analysis shows that, for this architecture, the memory data layout in conjunction with a possible sub-optimal division of data domain among the threads may lead to a high-level of data and TLB caches trashing. In the implementation discussed in this work we have adopted the AoS (Array of Structure) data layout, coding with explicit vectorization using *intrinsics* instructions. However, as studied in detail in [34], on CPU architectures, more complex data organizations could be used for LBM, such as CSoA (Cluster Structure of Array) or CAoSoA (Clustered Array of Structure of Array). These




(a) L1 Miss Ratio vs IPC

(b) L2 Miss Ratio vs IPC

Fig. 7. Tracking of the clusters' movement in the miss-ratio / IPC space, while increasing the number of OpenMP threads for the propagate function.

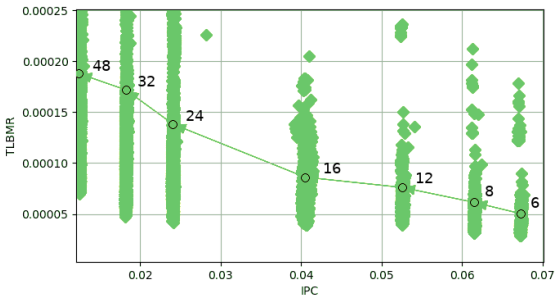



Fig. 8. Tracking of the clusters' movement in the TLB-miss-ratio / IPC space, while increasing the number of OpenMP threads for the propagate function.

aim to reduce the L2 and TLB misses, increasing data locality, avoiding misaligned memory accesses, and enabling automatic compiler vectorization. For the same algorithm described in Sec. V-B, using the CAoSoA data layout, the propagate function could reach $\sim 88\%$ of the peak bandwidth on an Intel Xeon Phi 7230 using its high bandwidth MCDRAM memory [35].

In fact, adopting this implementation, exploiting the CAoSoA data layout – with just minor modifications allowing to compile it for the Arm architecture – we have succeeded to increase the propagate bandwidth on the ThunderX, reaching 20.5 GB/s, which is $\sim 52\%$ of the bandwidth measured with STREAM on this SoC.

## VII. Conclusions

In this paper we have summarized the work which has been performed at the Barcelona Supercomputing Center to access the hardware performance counters of the Cavium ThunderX CN8890 SoC, with the aim to exploit these counters in standard HPC performance analysis tools on this platform. All the patches produced are now available in the respective projects' upstream releases.

Thanks to simple micro-benchmarks we have been able to identify counters which do not conform with other Arm platforms, as reported in Tab. I. In particular the PAPI_VEC_INS counter should not be considered trustworthy on this architecture for the moment.

Working on a Cavium ThunderX node, we have demonstrated that it is possible also on Arm SoCs to perform advanced performance analyses leveraging tools already developed and used on other architectures, such as Extrae and Paraver, which exploit the PAPI library to read hardware performance counters. In particular, we have shown how these tools can be utilized, analyzing an actual HPC application implementing a Lattice Boltzmann model developed at INFN & University of Ferrara, running on the Cavium ThunderX.

Thanks to this analysis we have been able to identify inefficiencies which lead to an increase of the performances of this application on the Cavium ThunderX architecture (i.e., $62\%$ increase in the propagate bandwidth). Further investigations may lead to other optimizations towards even better results.

Eventually we can state that on this platform is now possible to perform most of the performance analyses available on standard HPC machines. We have also shown that, for the fluid dynamics application considered in this paper, optimizations introduced for other many-core architectures (e.g., the Intel KNL), can be beneficial also on an Arm SoC, such as the Cavium ThunderX.

We plan to extend this work also to the second release of the ThunderX chip which is expected to be widely adopted in the HPC market [8] and moreover we expect to be able to access also non-standard power related registers embedded in this SoC [7], as done for other architectures [36]. The possibility to read power figures using PAPI [37], would enable also fine grained energy analyses [29], [38] without the need for external power-meters [39].

*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 [15], 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] "ARM Waving: Attention, Deployments, and Development," Jan. 2017. [Online]. Available: https://www.hpcwire.com/2017/01/18/arm-waving-gathering-attention/

[2] N. Rajovic *et al.*, "The Mont-blanc Prototype: An Alternative Approach for HPC Systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 38:1–38:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014955

[3] "ISC16 Recap – Fujitsu Takes the Stage - Processors blog - Processors - ARM Community." [Online]. Available: https://community.arm.com/processors/b/blog/posts/isc16-recap-fujitsu-takes-the-stage

[4] "GW4 Tier 2 HPC: Isambard | GW4." [Online]. Available: http://gw4.ac.uk/isambard/

[5] D. Bortolotti, S. Tinti, P. Altoé, and A. Bartolini, "User-space APIs for dynamic power management in many-core ARMv8 computing nodes," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, Jul. 2016, pp. 675–681, doi: 10.1109/HPCSim.2016.7568400.

[6] J. D. Gelas, "Investigating Cavium's ThunderX: The First ARM Server SoC With Ambition." [Online]. Available: http://www.anandtech.com/show/10353/investigating-cavium-thunderx-48-arm-cores

[7] M. Puzović, S. Manne, S. GalOn, and M. Ono, "Quantifying energy use in dense shared memory HPC node," in *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing*, ser. E2SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 16–23.

[8] "Cavium Expands the ThunderX2 Server Ecosystem for Cloud and HPC Applications." [Online]. Available: https://www.cavium.com/product-thunderx2-arm-processors.html

[9] B. Mohr, "Scalable parallel performance measurement and analysis tools - state-of-the-art and future challenges," *Supercomputing Frontiers and Innovations*, vol. 1, no. 2, pp. 108–123, Sep. 2014, doi: 10.14529/jsfi140207.

[10] L. Stanisic, B. Videau, J. Cronsioe, A. Degomme, V. Marangozova-Martin, A. Legrand, and J.-F. Méhaut, "Performance Analysis of HPC Applications on Low-power Embedded Platforms," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 475–480.

[11] G. S. Markomanolis, O. Jorba, and J. M. Baldasano, "Performance analysis of an online atmospheric-chemistry global model with Paraver: Identification of scaling limitations," in *2014 International Conference on High Performance Computing Simulation (HPCS)*, Jul. 2014, pp. 738–745, doi: 10.1109/HPCSim.2014.6903763.

[12] D. Ruiz, F. Mantovani, and E. Calore, "Enabling PAPI support for advanced performance analysis on ThunderX SoC," Tech. Rep., 2017. [Online]. Available: http://hdl.handle.net/2117/107063

[13] H. Jagode, A. YarKhan, A. Danalis, and J. Dongarra, "Power Management and Event Verification in PAPI," in *Tools for High Performance Computing 2015*. Springer, Cham, 2016, pp. 41–51, doi: 10.1007/978-3-319-39589-0_4.

[14] H. Servat, G. Llort, J. Giménez, and J. Labarta, "Detailed and simultaneous power and performance analysis," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 2, pp. 252–273, Feb. 2016, doi: 10.1002/cpe.3188.

[15] "Mont-Blanc Project." [Online]. Available: http://www.montblanc-project.eu/

[16] F. Mantovani *et al.*, "D5.11 - final report on porting and tuning the system software to ARM architecture," Tech. Rep., 2015. [Online]. Available: http://www.montblanc-project.eu/sites/default/files/D5.11%20Final%20report%20on%20porting%20and%20tuning.%20V1.0.pdf

[17] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, "Using PAPI for hardware performance monitoring on linux systems," in *Conference on Linux Clusters: The HPC Revolution*, vol. 5. Linux Clusters Institute, 2001.

[18] L. Gwennap, "ThunderX rattles server market," *Microprocessor Report*, vol. 29, no. 6, pp. 1–4, 2014.

[19] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, no. 1, 1995, pp. 17–31.

[20] S. Succi, *The Lattice-Boltzmann Equation*. Oxford university press, Oxford, 2001.

[21] M. Sbragaglia, R. Benzi, L. Biferale, H. Chen, X. Shan, and S. Succi, "Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria," *Journal of Fluid Mechanics*, vol. 628, pp. 299–309, 2009.

[22] A. Scagliarini, L. Biferale, M. Sbragaglia, K. Sugiyama, and F. Toschi, "Lattice Boltzmann methods for thermal flows: Continuum limit and applications to compressible Rayleigh–Taylor systems," *Physics of Fluids (1994-present)*, vol. 22, no. 5, p. 055101, 2010.

[23] L. Biferale, F. Mantovani, M. Sbragaglia, A. Scagliarini, F. Toschi, and R. Tripiccione, "Second-order closure in stratified turbulence: Simulations and modeling of bulk and entrainment regions," *Physical Review E*, vol. 84, no. 1, p. 016305, 2011, doi: 10.1103/PhysRevE.84.016305.

[24] ——, "Reactive Rayleigh-Taylor systems: Front propagation and non-stationarity," *EPL*, vol. 94, no. 5, 2011, doi: 10.1209/0295-5075/94/54004.

[25] E. Calore, S. F. Schifano, and R. Tripiccione, "On portability, performance and scalability of an MPI OpenCL lattice boltzmann code," in *Euro-Par 2014: Parallel Processing Workshops, Porto, Portugal*, ser. LNCS, 2014, pp. 438–449, doi: 10.1007/978-3-319-14313-2_37.

[26] E. Calore, A. Gabbana, J. Kraus, S. F. Schifano, and R. Tripiccione, "Performance and portability of accelerated lattice Boltzmann applications with OpenACC," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 12, pp. 3485–3502, 2016, doi: 10.1002/cpe.3862.

[27] E. Calore, A. Gabbana, J. Kraus, E. Pellegrini, S. F. Schifano, and R. Tripiccione, "Massively parallel lattice-Boltzmann codes on large GPU clusters," *Parallel Computing*, vol. 58, pp. 1 – 24, 2016, doi: 10.1016/j.parco.2016.08.005.

[28] F. Mantovani, M. Pivanti, S. F. Schifano, and R. Tripiccione, "Performance issues on many-core processors: A D2Q37 lattice boltzmann scheme as a test-case," *Computers & Fluids*, vol. 88, pp. 743 – 752, 2013, doi: 10.1016/j.compfluid.2013.05.014.

[29] E. Calore, S. F. Schifano, and R. Tripiccione, "Energy-performance tradeoffs for HPC applications on low power processors," in *Euro-Par 2015: Parallel Processing Workshops, Vienna, Austria*, ser. LNCS, 2015, pp. 737–748, doi: 10.1007/978-3-319-27308-2_59.

[30] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[31] J. Gonzalez, J. Gimenez, and J. Labarta, "Automatic detection of parallel applications computation phases," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–11, doi: 10.1109/IPDPS.2009.5161027.

[32] ——, "Performance Analytics: Understanding Parallel Applications Using Cluster and Sequence Analysis," in *Tools for High Performance Computing 2013*, 2014, pp. 1–17, doi: 10.1007/978-3-319-08144-1_1.

[33] G. Llort, H. Servat, J. González, J. Giménez, and J. Labarta, "On the usefulness of object tracking techniques in performance analysis," in *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013, pp. 1–11, doi: 10.1145/2503210.2503267.

[34] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Optimization of lattice boltzmann simulations on heterogeneous computers," *The International Journal of High Performance Computing Applications*, 2017, doi: 10.1177/1094342017703771.

[35] ——, "Early experience on using knights landing processors for lattice boltzmann applications," in *Parallel Processing and Applied Mathematics: 12th International Conference, PPAM 2017, Lublin, Poland*, ser. LNCS, vol. 1077, 2018, pp. 1–12, doi: 10.1007/978-3-319-78024-5_45.

[36] F. Mantovani and E. Calore, "Multi-node advanced performance and power analysis with paraver," in *Parallel Computing is Everywhere*, ser. Advances in Parallel Computing, vol. 32, 2018, pp. 723–732, doi: 10.3233/978-1-61499-843-3-723.

[37] V. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with PAPI," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, 2012, pp. 262–268.

[38] V. P. Nikolskiy, V. V. Stegailov, and V. S. Vecher, "Efficiency of the Tegra K1 and X1 systems-on-chip for classical molecular dynamics," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, July 2016, pp. 682–689, doi: 10.1109/HPCSim.2016.7568401.

[39] F. Mantovani and E. Calore, "Performance and power analysis of HPC workloads on heterogeneous multi-node clusters," *Journal of Low Power Electronics and Applications*, vol. 8, no. 2, 2018, doi: 10.3390/jlpea8020013.