November 1, 2017

Implementation of the K-means Algorithm on Heterogeneous Devices: a Use Case Based on an Industrial Dataset

Ying hao XU [a,b], Miquel VIDAL [a,b], Beñat AREJITA [c], Javier DIAZ [c], Carlos ALVAREZ [a,b], Daniel JIMÉNEZ-GONZÁLEZ [a,b], Xavier MARTORELL [a,b], Filippo MANTOVANI [a,1],

[a] *Barcelona Supercomputing Center (BSC)*
[b] *Universitat Politècnica de Catalunya, Barcelona Tech (UPC)*
[c] *Plethora IIoT*

Abstract. This paper presents and analyzes a heterogeneous implementation of an industrial use case based on K-means that targets symmetric multiprocessing (SMP), GPUs and FPGAs. We present how the application can be optimized from an algorithmic point of view and how this optimization performs on two heterogeneous platforms. The presented implementation relies on the OmpSs programming model, which introduces a simplified pragma-based syntax for the communication between the main processor and the accelerators. Performance improvement can be achieved by the programmer explicitly specifying the data memory accesses or copies. As expected, the newer SMP+GPU system studied is more powerful than the older SMP+FPGA system. However the latter is enough to fulfill the requirements of our use case and we show that uses less energy when considering only the active power of the execution.

Keywords. FPGA, Arm, Clustering, Heterogeneous programming, OmpSs, FPGA automatic toolchain

1. Introduction

The complexity of current commercial and enterprise applications such as e-commerce, health monitoring, industrial production and financial data analysis, rely more and more on Machine Learning techniques to achieve their objectives. As a consequence, the popularity of programmable accelerators has grown in both the industry and the research community, which can provide large gains in efficiency and performance by performing specialized tasks. The use of field-programmable gate arrays (FPGAs) and general purpose graphic processing unit (GP-GPU) as programmable accelerators can significantly increase the performance and efficiency gains while retaining some of the flexibility of general-purpose processors [9].

Accelerating machine learning algorithms with reconfigurable hardware and comparing the achieved speed up results to the ones obtained with SMP and GP-GPU is not

[1] Corresponding Author: Filippo Mantovani, Barcelona Supercomputing Center, C/ Jordi Girona, 29 - 08034 Barcelona, Spain; E-mail: filippo.mantovani@bsc.es

a new idea, in [8] for example, they compare the performance of a 16 processing node bayesian computing machine implemented with a Xilinx Virtex 5 FPGA to the performance obtained with an x86 processor and a Nvidia GeForce 9400m with CUDA cores.

In [6], the importance of organizing data is thoroughly tackled analysing the evolution of clustering algorithms since the definition of K-means. One of the most important reasons of the popularity of clustering algorithms in the scientific community is that cluster analysis and classification is prevalent in a wide range of disciplines where the analysis of multivariate data is needed.

The K-means clustering algorithm has been used in several studies to provide more insight into the performance of different platforms. The algorithm itself results to be of high interest for such comparative studies due to the characteristics it presents [11]. Firstly, the iterative nature of the algorithm implies that the current iteration results are needed in the next iteration. Secondly, calculating the centroids is a compute-intensive task. And thirdly, in order to obtain the global solution when the algorithm is parallelized, aggregation of the local results is required.

Even though clustering algorithms are commonly used as benchmarking algorithms to compare the performance of different systems, their application is also being introduced to solve computationally demanding real applications. In [7] is presented a real industrial application from data acquisition to processing and interpretation using a set of different machine learning techniques including K-means clustering to develop a knowledge discovery application for a real industrial application.

In this paper, we restrict our study to a thermal process performed by a laser over mechanical pieces supervised by a high frequency thermal camera tracking 32×32 pixel pictures every 1 ms. The stream of frames needs to be analyzed with clustering techniques within a given time window in order to find anomalies in the thermal process. The industrial equipment performing this process is equipped with computational modules that can acquire data from sensors (in this case from the camera) and will perform the clustering algorithm. The computational intensity of the algorithm is not trivial and can involve some heavy floating point computation that can be tackled using techniques derived from High Performance Computing.

The main contributions of our paper are *i)* the implementation of a complex industrial use case with near-real time constraints and near-HPC computational requirements on two different heterogeneous platforms *ii)* the handling of the heterogeneity and the parallelism using OmpSs [5,4], a task based programming model that allows us to keep almost the same source code running on the different heterogeneous platforms, and *iii)* the study of instantaneous power consumption and total energy spent to reach the solution of the problem.

The paper is structured as follows: in Section 2, we briefly introduce the platforms used for evaluating our tests. Section 3 includes the details of the implementation of the K-means algorithm in matrix form. Sections 4 and 5 present different implementations of the problem on the considered heterogeneous platforms. We present our results in Section 6 and explain our conclusions in Section 7.

2. Platform description

In this section, we introduce the two heterogeneous platforms used for the experiments of this paper. Both of them have a multi-core CPU based on Arm technology plus an

embedded accelerator part. Table 1 shows the detailed architectural features of the two platforms.

	Zynq 7020		**Jetson TX1**	
	CPU	FPGA accel.	CPU	GPU accel.
Compute resources	2× Cortex-A9	106.4k FFs, 53.2k LUTs, 220 DSPs	4× Cortex-A53	256 Maxwell CUDA cores
Frequency [MHz]	667	200	up to 1730	up to 998
Memory [MB]	1024	4.9 + CPU mem	4096	shared with CPU
Interconnection	1 GbE (native)	–	1 GbE (USB3 bridge)	–

Table 1. Technical specifications of Xilinx Zynq 7020 and Nvidia Jetson TX1 platforms

Both platforms are operated using a standard software stack for scientific computing, including Linux OS (Ubuntu), network file system (NFS), GNU compiler suite together with linear algebra and communication libraries. We operated the compute nodes as nodes of a cluster and we took advantage of the Mont-Blanc system software stack already deployed on Arm-based clusters [10]. A key part of the software stack installed in these machines is the OmpSs programming model [5,4], composed of the source-to-source Mercurium compiler and the Nanos++ runtime library. OmpSs is a task-based programming model with explicit inter-task dataflow that allows the runtime system to orchestrate out-of-order execution of the tasks, selectively off-loading of a task to the GPU/FPGA when possible. OmpSs is developed at Barcelona Supercomputing Center and it has been used in this paper to maintain a single portable and scalable code that can be executed on parallel heterogeneous devices only by changing a few pragmas.

3. Algorithm implementation and analysis

3.1. Algorithmic optimization

The K-means problem that we consider in this paper consists of classifying a set of N points of D dimension in K different groups, called *clusters*. The criteria for classifying the points involve to minimization of the intra-class variance, e.g. minimizing the sum of squared distances from each point to the cluster point. It is a well known clusterization technique already applied in similar cases [3].

In our experiments the K-means++ algorithm [2] specifies a procedure to initialize the cluster centers (centroids) before proceeding with the standard K-means. This algorithm helps avoiding poor clusterings found by the standard K-means algorithm and to converge to the desired solution faster.

As part of the K-means problem consists in computing distances between two points, the binomial theorem can be applied. The computation of the distance between two points is mathematically defined as $d^2 = \sum_{i=1}^{D}(c_i - p_i)^2$ where c_i is the coordinate of the centroids and p_i the coordinate of each of the point to clusterize. In our case $D = 1024$, as we are working with 32×32 pixels images. The binomial theorem defines the equivalence $(c_i - p_i)^2 = c_i^2 + p_i^2 - 2c_i p_i$. Applying it to the original distance formula, we obtain that the distance can be expressed as $d^2 = \sum_{i=1}^{D}(c_i^2 + p_i^2 - 2c_i p_i)$.

We can also note that, as the points do not change their position during the clustering process, any operation that only implies the p_i can be precomputed (memoization) and reused each time is needed. In our case therefore all the p_i^2 are computed at the beginning as a simple dot product before starting the K-means algorithm. Following the same idea, the c_i^2 operations are computed as a dot product, but this time during each iteration, as the centroids change their position during the clustering iterations.

The $c_i p_i$ operation can be computed as a matrix multiplication of the matrices P, storing the D coordinates of the N points to clusterize, and the transposed of the matrix C storing the D coordinates of the K centroids.

$$P = \begin{bmatrix} p_1^1 & p_2^1 & \cdots & p_D^1 \\ p_1^2 & p_2^2 & \cdots & p_D^2 \\ \cdots & & & \\ p_1^N & p_2^N & \cdots & p_D^N \end{bmatrix} \qquad C^T = \begin{bmatrix} c_1^1 & c_1^2 & \cdots & c_1^K \\ c_2^1 & c_2^2 & \cdots & c_2^K \\ \cdots & & & \\ c_D^1 & c_D^2 & \cdots & c_D^K \end{bmatrix}$$

Each cell of the final matrix is then multiplied by the constant -2. The resulting matrix will contain for each cell, the $-2c_i p_i$ operation that is part of the original expression of d^2.

3.2. Computational cost of the matrix implementation

As result of applying the optimizations explained in Section 3.1, the number of floating point operations have been reduced as some of the operations are precomputed once or within an iteration. Table 2 shows the computational cost of the two implementations for the size evaluated. Note that, following the original K-means algorithm, for each iteration, the distance from the points to the centroids is calculated. The distance formula is computed K times for each point and, as the centroids do not change their position within an iteration, the operation c_i^2 can be precomputed at the beginning of each iteration.

	Precomputed (once)	**Per iteration**
Original	-	$7DNK$
Optimized	$2DN$	$(2DK)+(2+2D)NK$

Table 2. Computational cost comparison: dimension (D), num. elements (N), num. centroids (K).

3.3. Cache locality

In Table 3, we present the data locality improvement achieved after applying the matrix implementation explained in Section 3.1. This comparison shows the total number of L1 data cache misses of a K-means execution for both platforms. The improvement is noticeable, as the number of data cache misses in the optimized version decreases $1.56\times$ in Jetson TX1 and $2.49\times$ in Zynq 7020. The used input parameters are $N = 21500$ points and $K = 8$ centroids, both of dimension $D = 1024$.

3.4. SIMD operations

The new implementation strategy ensures not only a smaller number of floating point operations to be carried out, but also data locality of the coordinates of the points. SIMD instructions can therefore be exploited and automatically inserted by the compiler. Besides the L1 data cache misses, Table 3 shows the floating point and vector operations executed in a K-means execution. In the Jetson TX1 case, the vector operations represents the 99% of the total floating point operations, meaning that a huge level of data parallelism has been exploited. On the other hand, due to the fact that Armv7 NEON hardware does not fully implement the IEEE 754 standard for floating-point arithmetic [1] (e.g. the direct comparison of single-precision values, used by our algorithm), no SIMD operations have been used for the Zynq 7020 execution.

	Jetson TX1			**Zynq 7020**		
	L1 DCM	FP	VEC	L1 DCM	FP	VEC
Original	$4.3 \cdot 10^8$	$1.5 \cdot 10^{10}$	~ 0	$8.9 \cdot 10^8$	$3.2 \cdot 10^{10}$	~ 0
Optimized	$2.7 \cdot 10^8$	$3.9 \cdot 10^7$	$5.7 \cdot 10^9$	$3.6 \cdot 10^8$	$1.3 \cdot 10^{10}$	~ 0

Table 3. Comparison of figures of merit related to the reference implementation and the one based on matrix operations. Parameters considered are L1 Data Cache Misses (DCM), Floating Point operations (FP) and Vectorial operations (VEC).

4. FPGA implementation

The FPGA device is used to accelerate the matrix multiply PC^T introduced in Section 3.1, which is the most time consuming part of our K-means implementation. P and C matrices do not fit inside the BRAM/Distributed RAM of the FPGA, and therefore, a blocking algorithm is necessary to split them during the process and reduce resource usage. The blocking strategy is implemented directly inside the FPGA to decouple the SMP work and the FPGA communication and computation. Our OmpSs@FPGA ecosystem (compiler and runtime) makes this implementation easy to program.

Figure 1 on the left shows the accelerator Blocking (Tiled) Matrix Multiply code. This optimized blocking version requires matrix block rows to be continuous in memory and thus, a pre and post memory processing may be necessary in the SMP code. Pre- and post-processings can be annotated with OmpSs tasks to exploit parallelism and overlap it with other computation and communication in the SMP and FPGA. Figure 1 on the bottom-right shows the `MxM_TILE` function with Vivado HLS pragmas. Those pragmas are introduced by the programmer to specify data array partitioning and pipeline optimizations, achieving full parallelism of the innermost loop of the matrix multiply (multiply and add operations), and a initiation interval of 1, which significantly reduces the computation time. The blocking factors BI, BJ and BK have been set to 1024, 8, and 32, respectively, to balance the communication and computation time, and limit the amount of resources used by the accelerator. The percentage of hardware resources reported is: 32% BRAMs, 80% DSPs, 26% FFs and 62% LUTs.

At compile time, this code is automatically completed by our infrastructure inserting Vivado HLS pragmas to specify bus/port interfaces (Figure 1 on the top-right (a)) for each of the arguments of the task and the hardware management of the communi-

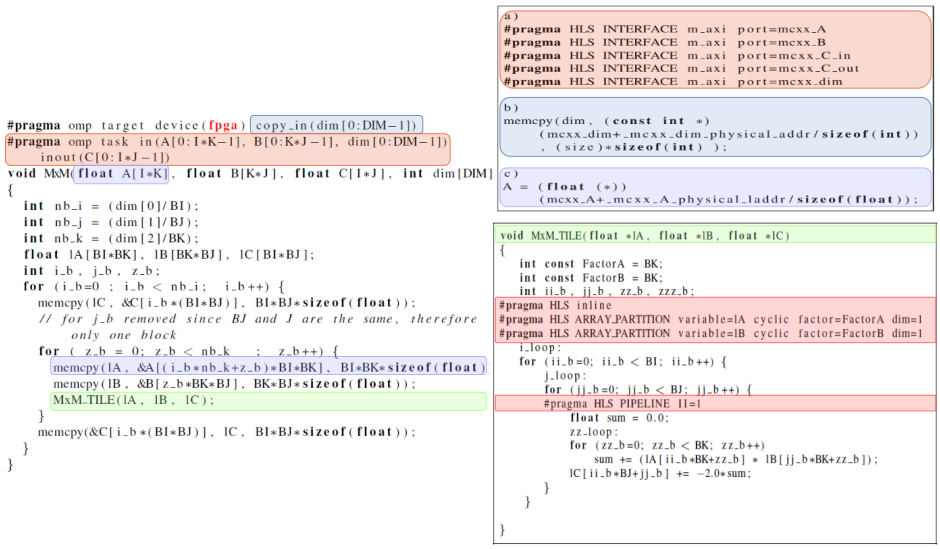

Figure 1. Code of Blocking Matrix Multiply (left) and insights of OmpSs@FPGA ecosystem (right)

cation from/to the main memory. Our current argument interface uses both AXI Stream (`axis` directive – not shown) and Master AXI (`m_axi` directive) bus protocols, which allow data transfers between SMP memory and FPGA device from the hardware accelerator. Indeed, this decouples the SMP computation of other tasks (runtime or programmer tasks) and the communication (data transfers) and computation of the accelerator. Physical memory addresses of the task arguments (`A`, `B`, `C` and `dim` arguments in the example) are passed by the OmpSs@FPGA runtime to the accelerator using an AXI stream (`axis`) port. Those physical memory addresses are used by the hardware accelerator to perform data transfers through the `m_axi` ports using simple memory accesses or `memcpy` call operations in the accelerator code.

The programmer should code almost nothing to carry out the aforementioned memory operations from inside the hardware accelerator. The automatic compilation generates code to perform data memory transfers for each argument with `copy_in/out`. For instance, for the dimensions (`dim`) variable, defined as `copy_in`, a Vivado HLS `memcpy` call (Figure 1 on the top-right (b)) is introduced in the code just before starting the actual task acceleration execution. Otherwise, for those arguments that are not defined as `copy_in/out`, our infrastructure maps those to `m_axi` ports to allow the programmer to easily perform memory accesses or copies of the blocks from/to shared memory by using Vivado HLS `memcpy` calls. For instance, Figure 1 on the top-right (c) shows part of the automatic generated Vivado HLS code to map `A` argument variable to `m_axi` port `mcxx_A`. That allows to perform the `A` copy from SMP memory to `lA` local variable (FPGA BRAM) using a `memcpy` operation on the blocking code. Once the code is completed with the Vivado HLS interface and the hardware management for data memory transfers, this is automatically synthesized using Vivado HLS, exported and integrated into a Vivado project, to generate a bitstream. To speedup the computation time and the communication time even more, the frequency of the accelerator has been set to 200 MHz.

At execution time, OmpSs@FPGA runtime will take care of the synchronization and memory copies defined in the `copy_in/out` and provides mechanisms to use continuous pinned memory, necessary for the memory transfers between the accelerator and the main memory.

5. Other implementations

All algorithms have been implemented in C and parallelized with the OmpSs programming model. Each of them used a serial K-means code with the optimizations presented in Section 3.1 as baseline.

As mentioned before, the OmpSs programming model has been used not only for parallelizing the code but also to handle the heterogeneous devices in a transparent way for the user. Figure 2 shows in a schematic way how, by just adding few pragmas within the almost same source code, OmpSs is capable of managing three different devices (SMP, FPGA, GPU). Our heterogeneous K-means implementations follow this approach and leverages in OmpSs to exploit hardware accelerators.

```
A = dot_product(DATA)
CENTROIDS = kpp(DATA)
do {
  B = dot_product(CENTROIDS)
  for each block i {
    #pragma omp target device(fpga,smp,cuda) copy_deps
    #pragma omp task in(DATA,CENTROIDS) out(Ci)
    Ci = MxM(DATA,CENTROIDS)
    #pragma omp task in(A,B,Ci) out(CENTROIDS,LABELS)
      <CENTROIDS,LABELS,error> = compute_centroids( A, B, Ci)
      #pragma omp atomic
      total_error+=error;
  }
  #pragma omp taskwait
} while (total_error > tolerance);
```

Figure 2. K-means pseudocode with OmpSs pragmas with support for executing in a SMP, FPGA and GPU

The **OmpSs** version is the parallelized version of the sequential code. In this case, the parallel granularity is expressed at point level, as the tasks compute the dot product and the matrix multiply of a continuous subset of points.

The **OmpSs+BLAS** version is identical to the OmpSs version but adapting the dot product and matrix multiply operations to be done using a BLAS library.

The **OmpSs@CUDA+BLAS** version uses as baseline the OmpSs+BLAS version but combines the power of an NVIDIA GPU accelerator. The OmpSs@CUDA ecosystem manages all the data transfers between the GPU and the host in a transparent way. The OmpSs runtime scheduler will optimize the data transfers by analyzing the data dependencies and moving data only when it is necessary.

6. Evaluation

For the evaluation of both platforms (Jetson TX1 and Zynq 7020) the following input parameters has been used: dimension $D = 1024$, number of element to clusterize $N = 21500$, number of clusters $K = 8$, 1 iteration and 10^{-4} error tolerance.

The used input set is a real industrial dataset. This dataset represents a short video of 21.5 seconds at 1000 FPS ($N = 21500$). Each frame has a resolution of 32×32 pixels, giving a total of $D = 1024$ pixels per frame. The initial centroids are chosen using K-means++ algorithm which helps avoiding to fall into local optimums and get poor clusterings.

6.1. Performance

The Figure 3 shows the achieved speedup the execution of the different implementations on one node of both platforms, Jetson TX1 and Zynq 7020, taking as reference a CPU-only serial implementation of the problem. Overall, by just annotating the code with few OmpSs pragmas and execute it in parallel, the performance achieved in both platforms is quite good (3.13× and 1.58× respectively). The modified version implementing the mathematical approaches explained in the Section 3.1 with an optimized linear algebra library such as ATLAS (BLAS), achieves a performance boost up to 7.39× and 6.20× respectively. Combining the CPU plus the accelerator embedded in the SoC (GPU or FPGA), the performance in the Jetson TX1 (GPU) is worse than only using the CPU. This is directly related to the fact that the dimension of the used input set is too small to benefit from the embedded accelerator. However, the performance in the Zynq 7020 (FPGA) increased up to 7.76× compared to the serial version. This shows that for certain real case scenarios (as this industrial one), using a GPU is not always the best way to achieve performance.

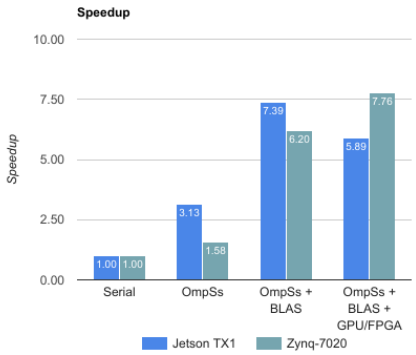

Figure 3. Speedup comparison between different implementations on Jetson TX1 and Zynq 7020

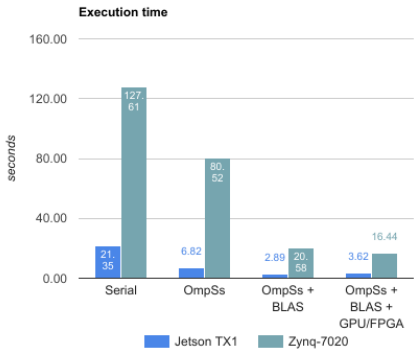

Figure 4. Execution time comparison between different implementations on Jetson TX1 and Zynq 7020

The Figure 4 shows the same results as the Figure 3 but quantifying the performance in seconds instead of the achieved speedup between implementations. As expected, the Zynq 7020 performs worse than the Jetson TX1. In the serial execution, the Jetson TX1

is 6× faster than Zynq 7020 while in the execution using OmpSs plus BLAS combined with the accelerator (GPU or FPGA) is only 4.5× faster. Even if the speedup is greater on the Zynq 7020, as expected, the raw performance is much better in the Jetson TX1, as the Jetson TX1 features the double of cores clocked to a higher frequency and a much more refined architecture (Armv8 vs Armv7). As the industrial constraints forced us to work with one computational node, we intentionally left out the evaluation on the problem on a multi-node parallel environment (cluster) that can be explained in a future work.

6.2. Energy to solution

Power consumption is one of the main constraints when designing an embedded system. This section will discuss the power profile and the total energy to reach the solution of the problem (called *Energy to Solution*) on both aforementioned platforms. The power data has been collected using a Yokogawa power meter [12]. The energy to solution has been computed as the sum over execution time of the instantaneous power. The evaluated implementations are the ones using an accelerator (GPU / FPGA).

Both platforms have been tested using the input set mentioned at the beginning of the section but computing 10 iterations instead of only one, for power sampling reasons. The Jetson TX1 board consumed 307.54 J in total. 181.43 J of them are given by the contribution of active power, i.e. ignoring static idle power. The Zynq 7020 consumed 1009.72 J in total and only 157.99 J was the contribution of the active part.

The total energy of Jetson TX1 is 3× lower than the Zynq 7020. This is mostly due to the fact that the first one has an idle power of 3.96 W while in the second platform is 5.23 W. The lower idle power combined with the fact that the execution time is 4.5× faster in the Jetson TX1, makes the total energy consumed by this platform significantly lower. However, considering only the active energy, the Zynq 7020 consumes 15% less energy than Jetson TX1 even if the lithography process in Zynq 7020 is 28 nm and in Jetson TX1 is 20 nm.

For stressing the fact that the FPGA approach is rewarding, the FPGA power measurement has been repeated but this time varying the set frequency of the FPGA (the previous frequency was 200 MHz). A slower frequency will slightly decrease the performance, but at the same time will save energy. For 100 MHz, the total energy consumed is 1233.37 J and 147.82 J of active part. For 50 MHz, the total energy consumed is 1743.63 J and only 140.95 J due to the active part, 1.12× lower than when running at 200 MHz. We observe that the majority of the active power is currently used by the Arm cores: the impact of the power consumption of the FPGA is below 1 W, but its performance benefits are notable. From all these observations, for the case analyzed in this paper, we foresee a greater benefit in having a platform with more programmable logic resources to be used as accelerator, than a GPU.

7. Conclusions

This paper presents a complete analysis of the implementation of an industrial application of the K-means algorithm in different heterogeneous devices. First, the algorithm has been analyzed and optimized from an algorithmic point of view. Afterwards, it has been adapted to be executed in three different computing resources: SMP, GPU and FPGA.

The results show that although the system composed by SMP and GPU is newer and more powerful, the one composed by SMP and FPGA uses less active energy to reach the solution while fulfilling the requirements of the problem.

In order to tackle the adaptation of the application to different devices this paper uses the OmpSs environment. It takes care of connecting the SMP and the accelerators in the system allowing the programmer to focus on optimizing the code without caring about the (usually) cumbersome details of accelerator programming.

Finally, this paper also shows the importance of data transfers when dealing with accelerators. While the GPU implementation is not able to improve the best SMP version due to the characteristics of the problem, the FPGA one is by carefully managing data movements and decoupling accelerator communication and computation, and SMP computation.

Acknowledgments

This work is partially supported by the European Union H2020 project *AXIOM* (grant agreement n. 645496), *HiPEAC* (grant agreement n. 687698), and *Mont-Blanc* (grant agreements n. 288777, 610402 and 671697), the Spanish Government *Programa Severo Ochoa* (SEV-2015-0493), the Spanish Ministry of Science and Technology (TIN2015-65316-P) and the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project *MPEXPAR: Models de Programació i Entorns d'Execució Paral·lels* (2014-SGR-1051).

References

[1] ARM Information Center: Cortex-A9 Reference Manual, `https://goo.gl/FNwdaK`

[2] Arthur, D., Vassilvitskii, S.: K-means++: The advantages of careful seeding. In: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1027–1035. SODA '07 (2007)

[3] Bekkerman, R., Bilenko, M., Langford, J.: Scaling up machine learning: Parallel and distributed approaches. Cambridge University Press (2011)

[4] Bueno, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguad, E., Labarta, J.: Productive Programming of GPU Clusters with OmpSs. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium. pp. 557–568 (2012)

[5] Duran, A., Ayguad, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters 21(02), 173–193 (2011)

[6] Jain, A.K.: Data clustering: 50 years beyond k-means. Pattern recognition letters 31(8), 651–666 (2010)

[7] Javier Diaz-Rozo, Concha Bielza, P.L.: Machine learning-based cps for clustering high throughput machining cycle conditions. Procedia Technology (2017)

[8] Lin, M., Lebedev, I., Wawrzynek, J.: High-throughput bayesian computing machine with reconfigurable hardware. In: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays. pp. 73–82. ACM (2010)

[9] Mahajan, D., Park, J., Amaro, E., Sharma, H., Yazdanbakhsh, A., Kim, J.K., Esmaeilzadeh, H.: Tabla: A unified template-based framework for accelerating statistical machine learning. In: High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on. pp. 14–26. IEEE (2016)

[10] Rajovic, N., 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. pp. 38:1–38:12. SC '16, IEEE Press (2016)

[11] Singh, D., Reddy, C.K.: A survey on platforms for big data analytics. Journal of Big Data 2(1), 8 (2015)

[12] Yokogawa Power Meter Specifications, `https://goo.gl/TNouXQ`