# INAF-OATs Technical Report 241: Performance of direct *N-body* code on ARM64 SoC

D. Goz[a], S. Bertocco[a], L. Tornatore[a], G. Taffoni[a]

[a]*INAF-Osservatorio Astronomico di Trieste, Via G. Tiepolo 11, 34131 Trieste - Italy*

**Abstract**

The ExaNeSt H2020 project will produce a prototype based on ARM64 CPUs and FPGA-based accelerators implementing a co-design approach where scientific applications requirements are driving the hardware design. We present strategies adopted in order to port our direct $N$-body code on an ARM SoC cluster, called INCAS, deployed at INAF-OATs. Each computational node is based on 64-bit ARMv8 Cortex-A72/Cortex-A53 core design, powered by the Mali-T864 GPU. We present the strategies to optimize the code and we discuss in details results of performance tests carried on ARM CPUs and GPUs, showing that embedded GPUs can be effectively used to accelerate the calculation, and that are promising also because of their energy efficiency, which is an important design in future exascale platforms.

*Keywords:* `ExaNeSt`, $N$`-body solvers`, `GPU`, `ARM`, `INCAS`

## 1. Introduction

The ExaNeSt project will produce a prototype based on low power-consumption ARM64 processors, accelerators and low-latency interconnections implementing a co-design approach where scientific applications requirements are driving the hardware design [1].

*Email addresses:* `david.goz@inaf.it, ORCID:0000-0001-9808-2283` (D. Goz), `sara.bertocco@inaf.it, ORCID:0000-0003-2386-623X` (S. Bertocco), `luca.tornatore@inaf.it, ORCID:0000-0003-1751-0130` (L. Tornatore), `giuliano.taffoni@inaf.it, ORCID:0000-0002-4211-6816` (G. Taffoni)

Graphics Processor Units (GPUs) are widely employed as accelerators in heterogeneous computing. They offer high floating-point throughput and memory bandwidth than multi-core central processing units (CPUs).

Nowadays ARM delivers technology to drive power-efficient SoC solutions combining CPU and GPU into unified compute sub-system offering double-precision floating point arithmetic, and options for high performance I/O and memory interface.

This work is devoted to measure the performance of ARM64 SoC using the ExaHiGPUs code [2, 3], which allows us to test the workload across the whole CPU+GPU system. INCAS[1] is the first power-efficient SoC-based ARM cluster built at OATs specifically designed to drive the development and re-engineering of astrophysical applications in order to exploit new Exascale computing facilities. INCAS is fully described in [4].

This technical report is organized as follows. In Section 2 we briefly describe the GPU architecture, both discrete and embedded. In Section 3 we discuss strategies to exploit Mali GPUs. In Section 4 we show our performance results.

## 2. GPU microarchitecture

Discrete GPUs are connected to their own high-speed memory system and typically offer 10× the memory bandwidth as compared to their host CPU. The drawback of this method is that input and output data must be exchanged between the host memory and device memory, which adds some overhead as compared to using the CPU only.

Embedded GPUs generally share memory with their host CPU so they are not subject to CPU-GPU communication overhead, but, at the same time, they do not take the advantage of high memory-bandwidth.

Memory hierarchy differs between CPU and GPU. CPUs derive much of their performance from their multilevel caches, which support rich feature sets (e.g

[1] Every computational node on INCAS is equipped with Six-Core ARMv8 64-bit CPU (Cortex-A72x2 and Cortex-A53x4) and ARM Mali-T864 MP4 Quad-Core GPU.

prefetching, coherency with other caches). CPUs have small general purpose register files without a large performance penalty.

On the other hand, GPUs generally have simple or no caches and large register files. Registers serve as a work-item's private memory and must have sufficient capacity to store all the registers that each work-item needs during kernel execution, multiplied by the total number of work-items assigned to the same GPU core.

For most GPUs, the unit of workload assigned to processor cores is the work-group (OpenCL terminology). In order to exploit multicore parallelism a program should instantiate as many work-groups as cores. The work-group size (i.e. the number of work-items per work-group) has some effects on the memory system, since only work-items belonging to the same work-group can share on-chip memory. Having too small work-group size may limit the number of work-items that the GPU can map on SIMD lanes or limit the pool of work-items available to be scheduled for execution in order to hide memory latency of others work-items. On the other hand, the work-group size is limited by the hardware, i.e. by the available registers and shared memory usage of the executed kernel.

OpenCL allows the programmer to exploit the parallelism at the finest level possible on the target device. However, general purpose programming for embedded GPUs is still relatively new, and the associated runtime libraries are often immature.

## 3. Tuning OpenCL kernel for Mali GPUs

OpenCL is a portable language but it not always performance portable, so existing OpenCL code is typically tuned for specific architecture. ARM developer guide[2] says that for best performance on Mali-T864 (Mali Midgard family) the code should be vectorized to achieve the best performance. Regardless of

[2] infocenter.arm.com/help/topic/com.arm.doc.100614_0303_00_en/arm_mali_gpu_opencl_developer_guide_100614_0303_00_en.pdf

the native width of the GPU's SIMD functional units, using wider vectors in the kernel may provide the GPU architecture more opportunity for exploiting data-level parallelism. Kernels in ExaHiGPUs have already been vectorized, since also discrete GPUs show enhanced performances exploiting vectorization.

On the Mali GPU, the global and local OpenCL address spaces are mapped to main host memory. This means that explicit data copies from global to local memory and associated barrier synchronizations are not necessary. Thus, using local memories as a cache can waste both performance and power on the Mali GPU. We implemented a specific ARM version of all kernels of ExaHiGPUs in which the local memory is not used.

## 4. Performance results

The 6th order Hermite integration scheme implemented in ExaHiGPUs relies on three different stages, described in [2]. We just focus on the *evaluation* stage, which is the most computationally demanding (with $N$ bodies the algorithm requires $O(N^2)$ computational cost). We measured the performance for both ARM CPU and GPU, testing how the running time (average of 10 runs of the kernel) changes as a function of the number of OpenMP threads in the CPU code, and of the work-group size in the GPU code. On the GPU side we have also investigated the impact of specific ARM optimizations as discussed in Section 3. Performances have been measured for both double precision (DP) and extended precision (EX) arithmetic.

In Appendix A DP kernels for discrete GPUs and embedded GPUs are shown.

### *4.1. ARM CPUs performance results*

ARM big.LITTLE processors have three main software execution models: cluster migration (a single cluster is active at a time, and migration is triggered on a given workload threshold), CPU migration (pairing every big core with a LITTLE core. Each pair of cores acts as a virtual core in which only one

actual core a is powered up and running at a time), heterogeneous multiprocessing mode (also known as Global Task Scheduling, allows using all of the cores simultaneously).

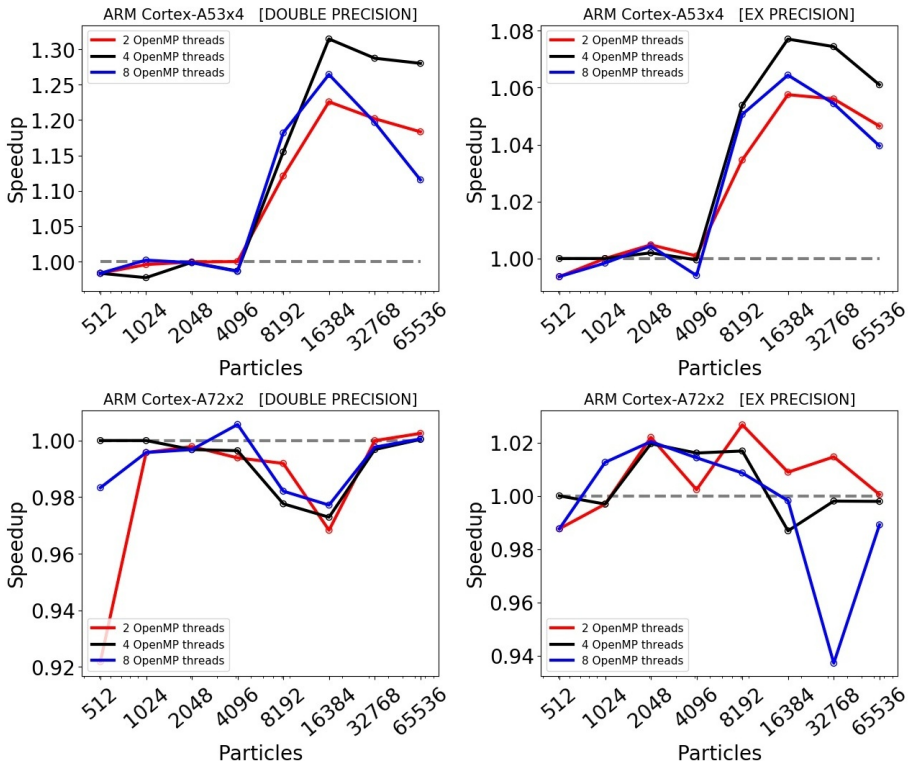


Figure 1: Host speedup for DP-arithmetic (left panels) and EX-arithmetic (right panels) varying OpenMP threads as a function of the number of particles. Top panels for ARM Cortex-A53x4 CPU and bottom panels for ARM Cortex-A72x2 CPU.

We studied the CPUs speedup, i.e. the ratio of the serial execution time to the parallel execution time utilizing multiple cores by means of OpenMP threads. Kernel execution time on both ARM Cortex-A53x4 and Cortex-A72x2 CPUs have been obtained setting explicit CPU affinity and using the Linux system function `getrusage`, getting the total amount of time spent executing in user mode.

Figure 1 shows the speedup for both ARM Cortex-A53x4 and Cortex-A72x2

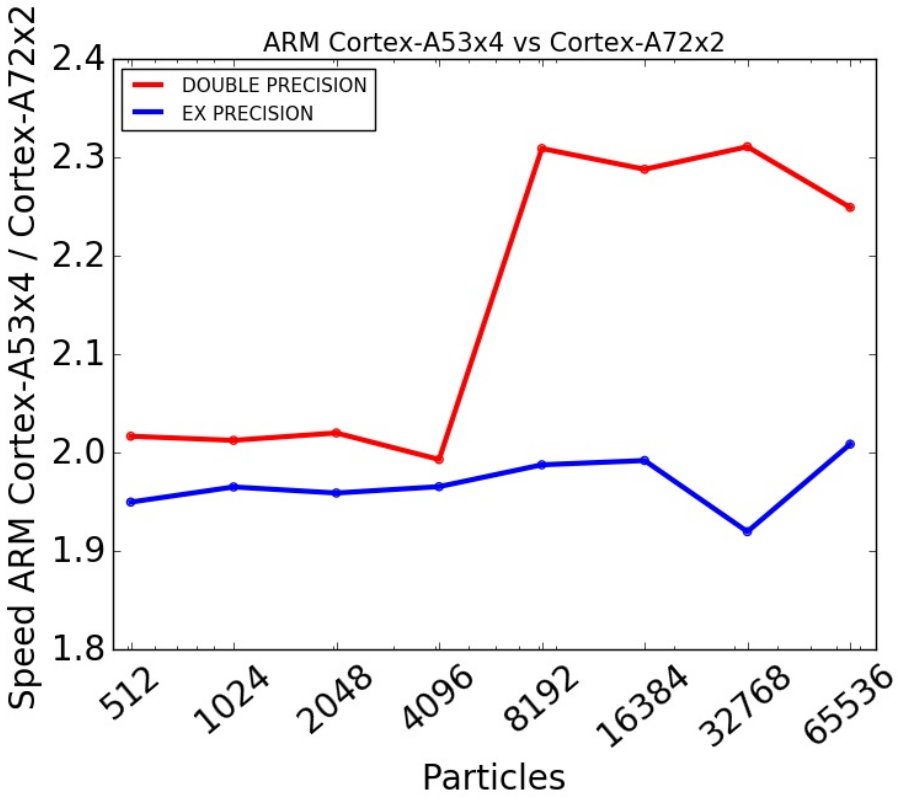


Figure 2: Speed comparison between ARM Cortex-A53x4 and Cortex-A72x2 CPUs for both DP-arithmetic (red line) and EX-arithmetic (blue line).

CPUs varying the number of OpenMP threads as a function of the number of particles. On the ARM Cortex-A53x4, for both DP-arithmetic and EX-arithmetic, some speedup is obtained only when the number of particles exceeds 4096 in number. As expected, the best performance is achieved with four OpenMP threads, where most likely there is one thread per available core. On the ARM Cortex-A72x2 one thread is always faster then multiple threads for DP-arithmetic and only a minor speedup is achieved with two threads adopting EX-arithmetic.

We also directly compared the performance of ARM CPUs. Figure 2 shows the ratio of the best running time achieved by the CPUs as a function of the

number of particles for both arithmetic. ARM Cortex-A72x2 is faster than Cortex-A53x4 by approximately a factor of two.

### *4.2. ARM GPU performance results*

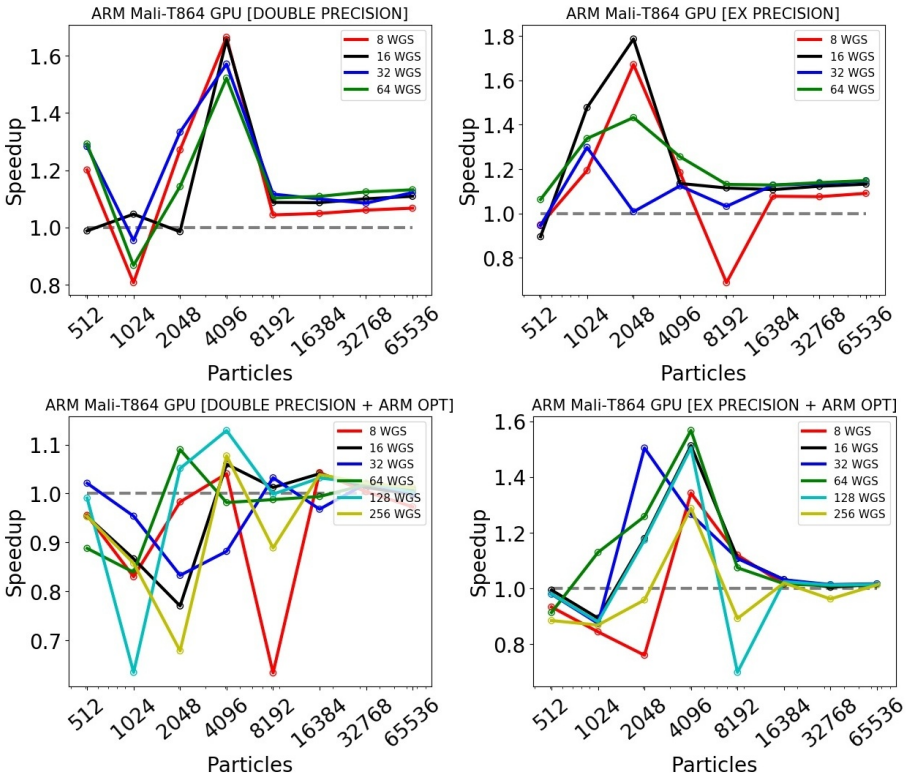


Figure 3: GPU speedup for DP-arithmetic (left panels) and EX-arithmetic (right-panels) varying the OpenCL work-group size as a function of the number of particles. Speedup is normalized by the time to solution with work-group of size 4. Top panels for GPGPU kernel code and bottom panels for ARM-optimized kernel code.

We studied how the work-group size affects the ARM Mali-T864 GPU performance. Figure 3 shows the speedup achieved varying the OpenCL work-group size for GPGPU kernel code (top panels) and embedded-ARM-optimized kernel code (bottom panels) as a function of the number of particles. The speedup is normalized by the time to solution obtained with work-group size of four. Kernel execution times on the GPU have been obtained by means of OpenCL's built-in

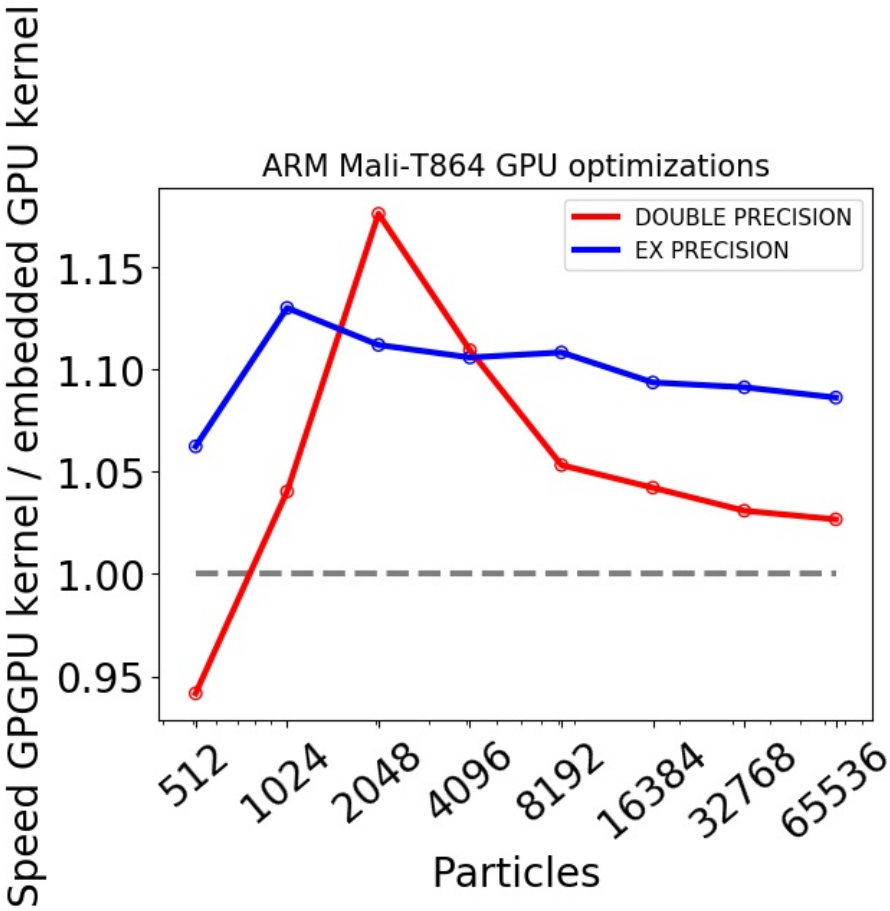


Figure 4: Impact of ARM-optimizations on time to solution for Mali-T864 GPU as a function of the number of particles. Red line for DP-arithmetic and blue-line for EX-arithmetic.

profiling functionality, which allows the host to collect runtime information. It is worth noting that work-group sizes of 128 and 256 cause a failure to execute the GPGPU kernel (top panels of Fig. 3) because of insufficient local memory on the GPU. Only ARM-optimized version of the kernel, which avoids the usage of local memory, can be run with those work-group sizes (the maximum possible work-group size on ARM Mali-T864 is 256). Despite ARM recommends for best performance using a work-group size that is between 4 and 64 inclusive, our results show that speedup is not driven by any specific work-group size, regardless

the usage of local memory. For our kernel we suggest to let the driver to pick the work-group size it thinks as best (the driver usually selects the work-group size as 64).

We also quantified the impact of ARM-optimizations. Figure 4 shows the ratio of the best time to solution achieved by GPGPU kernel code and ARM-optimized kernel code for both arithmetic. In the case of EX-arithmetic the speedup is approximately 10%, while adopting DP-arithmetic the speedup is nearing 5% increasing the number of particles. Our experiments reveal that adopting the same optimization strategies as those used for high-performance GPGPU computing might lead to worse performance on embedded GPUs. This is in agreement with what was found by [5], when they tested some non-graphic benchmarks on embedded GPUs.

### *4.3. ARM CPU-GPU comparison*

It is widely accepted that high-end GPGPUs can greatly speedup the solution of the direct $N$-body problem. However in this work we want also to evaluate the performance of low-power embedded GPUs for our kernel. We studied the best running time on ARM Cortex-A72x2 as the ratio over the best execution time taken by our ARM-optimized GPU implementation, as shown in Figure 5. The ARM-optimized implementation is as fast as the dual-core implementation on the ARM Cortex-A72x2 using DP-arithmetic, as long as the GPU is kept fed with enough particles, while is almost three times faster using EX-precision.

## 5. Conclusions

We have shown that SoC boards can be successfully used to execute our $N$-body code. Our experience reveals that adopting the same optimization strategies as those employed for high-end GPUs might not be the best approach on embedded low-power GPUs, because of restricted hardware features. Secondly, in light of our findings, embedded GPUs appear to be attractive from

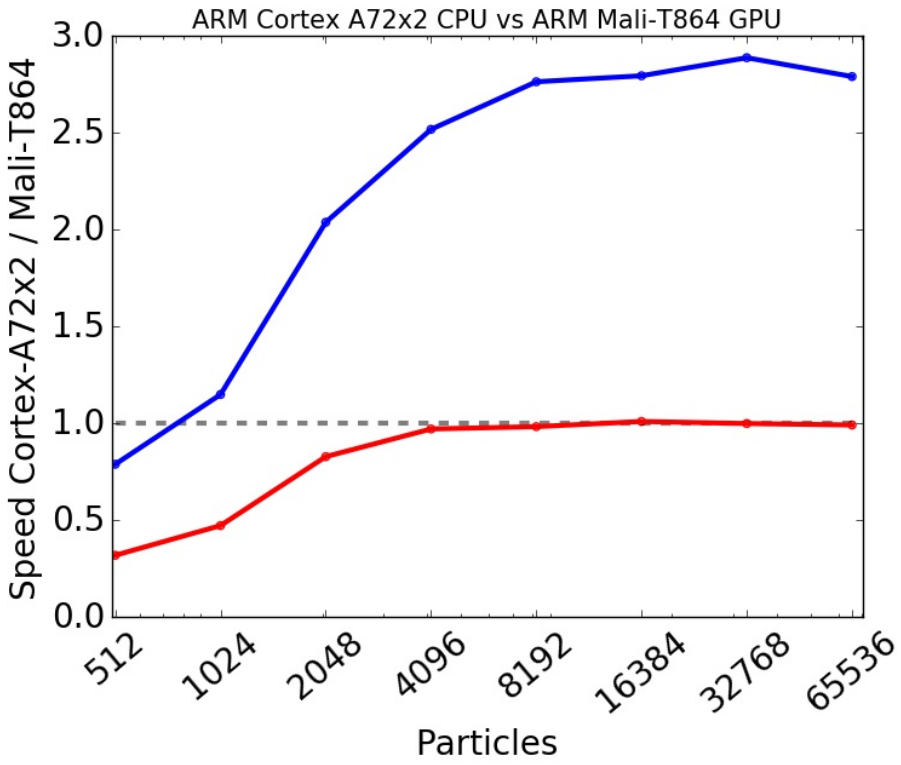


Figure 5: Comparison of the time to solution between ARM Cortex-A72x2 CPU and Mali-T864 GPU for both DP-arithmetic (red line) and DX-arithmetic (blue line) as a function of the number of particles.

a performance perspective as soon as their DP compute capability increases. However, the EX precision approach can be a solution to supply enough power to execute scientific computation and benefit at maximum of the SoC devices.

In the next technical report, we will quantitatively measure the impact of our algorithms on power or energy consumption on SoC, possibly shedding some light on their suitability for exascale applications.

## 6. Acknowledgments

This work was carried out within the ExaNeSt (FET-HPC) project (grant no. 671553) and the ASTERICS project (grant no. 653477), funded by the

European Unions Horizon 2020 research and innovation programme.

This research has been made use of IPython [6], Scipy [7], Numpy [8] and MatPlotLib [9].

## Appendix A. Evaluation kernel

Evaluation kernel optimized for discrete GPUs. This kernel exploits the local memory of the GPU.

```
__kernel void DP_Evaluation(__private const unsigned int      ParOffset,
                            __global  const double4 *restrict PosPred,
                            __global  const double4 *restrict VelPred,
                            __global  const double4 *restrict AccPred,
                            __global        double4 *restrict aC,
                            __global        double4 *restrict aC1,
                            __global        double4 *restrict aC2)
{
  /* work-item gets its own global id */
  const unsigned int globalID = get_global_id(0);
  /* work-item gets its own local id */
  const unsigned int localID = get_local_id(0);

  /* position stored in local memory */
  __local double4 shPos[LOCAL_WORK_SIZE];
  /* velocity stored in local memory */
  __local double4 shVel[LOCAL_WORK_SIZE];
  /* acceleration stored in local memory */
  __local double4 shAcc[LOCAL_WORK_SIZE];

  /* Loop over all particles */
  for (unsigned int Part=globalID ; Part<PARTICLES ; Part+=GLOBAL_WORK_SIZE)
    {
```

```
/* STORE IN PRIVATE MEMORY ACC, JRK AND SNP */
DOUBLE4 ACC = 0.0;
DOUBLE4 JRK = 0.0;
DOUBLE4 SNP = 0.0;

/* LOAD PARTICLE POSITION IN PRIVATE MEMORY */
DOUBLE4 MYPOS = POSPRED[PART];
/* LOAD PARTICLE VELOCITY IN PRIVATE MEMORY */
DOUBLE4 MYVEL = VELPRED[PART];
/* LOAD PARTICLE ACCELERATION IN PRIVATE MEMORY */
DOUBLE4 MYACC = ACCPRED[PART];

/* LOOP OVER PPDEV PARTICLES (DIVIDED INTO WORK-GROUPS) */
FOR (UNSIGNED INT WORKGROUP=0 ; WORKGROUP<PPDEV ; WORKGROUP+=LOCAL_WORK_SIZE)
  {
    UNSIGNED INT LOCAL_ADDRESS = WORKGROUP + LOCALID + PAROFFSET;

    /* LOAD POSITIONS, VELOCITIES AND ACCELERATIONS IN LOCAL MEMORY */
    SHPOS[LOCALID] = POSPRED[LOCAL_ADDRESS];
    SHVEL[LOCALID] = VELPRED[LOCAL_ADDRESS];
    SHACC[LOCALID] = ACCPRED[LOCAL_ADDRESS];

    /* WORK-ITEMS SYNCHRONIZATION */
    BARRIER(CLK_LOCAL_MEM_FENCE);

    /* LOOP OVER PARTICLES WITHIN THE WORK-GROUP */
    FOR (UNSIGNED INT J=0 ; J<LOCAL_WORK_SIZE ; J++)
      {
        DOUBLE4 R           = SHPOS[J] - MYPOS; R.W = 0.0;
        DOUBLE  R2          = DOT(R, R) + EPS2;
        DOUBLE  MASS_INV_R3 = SHPOS[J].W * NATIVE_RSQRT(POWN(R2,3));
```

```
            DOUBLE  INV_R2      = NATIVE_RECIP(R2);

            DOUBLE4 V = SHVEL[J] - MYVEL; V.W = 0.0;
            DOUBLE4 A = SHACC[J] - MYACC; A.W = 0.0;

            DOUBLE ALPHA = (DOT(R, V) * INV_R2);
            DOUBLE BETA  = ((DOT(V, V) + DOT(R, A)) * INV_R2) + (ALPHA * ALPHA);

            DOUBLE4 A1    = (MASS_INV_R3 * R);
            DOUBLE4 A1DOT = (MASS_INV_R3 * V) - (3.0 * ALPHA * A1);
            DOUBLE4 A2DOT = (MASS_INV_R3 * A) - (6.0 * ALPHA * A1DOT) - (3.0 * BETA * A1);

            IF ((J + WORKGROUP + PAROFFSET) != PART)
              {
                ACC += A1;
                JRK += A1DOT;
                SNP += A2DOT;
              } /* AVOID  SELF-INTERACTION */
           } /* LOOP OVER WORK-ITEMS WITHIN THE WORK-GROUP */

         /* WORK-ITEMS SYNCHRONIZATION */
         BARRIER(CLK_LOCAL_MEM_FENCE);
       } /* LOOP OVER WORK-GROUPS */

     /* STORE RESULTS IN GLOBAL MEMORY */
     AC[PART]  = ACC;
     AC1[PART] = JRK;
     AC2[PART] = SNP;
   } /* LOOP OVER PARTICLES */

RETURN;
```

```
}
```

Evaluation kernel optimized for embedded GPUs. This kernel does not exploit the local memory of the GPU.

```
__kernel void ARM_DP_Evaluation(__private const unsigned int      ParOffset,
                                __global  const double4 *restrict PosPred,
                                __global  const double4 *restrict VelPred,
                                __global  const double4 *restrict AccPred,
                                __global        double4 *restrict aC,
                                __global        double4 *restrict aC1,
                                __global        double4 *restrict aC2)
{
  /* work-item gets its own global id */
  const unsigned int globalID = get_global_id(0);

  /* Loop over all particles */
  for (unsigned int Part=globalID ; Part<PARTICLES ; Part+=GLOBAL_WORK_SIZE)
    {
      /* store in private memory acc, jrk and snp */
      double4 acc = 0.0;
      double4 jrk = 0.0;
      double4 snp = 0.0;

      /* load particle position in private memory */
      double4 myPos = PosPred[Part];
      /* load particle velocity in private memory */
      double4 myVel = VelPred[Part];
      /* load particle acceleration in private memory */
      double4 myAcc = AccPred[Part];

      /* Loop over PPDEV particles */
```

```
FOR (UNSIGNED INT MYPART=0 ; MYPART<PPDEV ; MYPART++)
  {
    UNSIGNED INT ADDRESS = MYPART + PAROFFSET;

    /* AVOID SELF-INTERACTION */
    IF (ADDRESS != PART)
      {
        DOUBLE4 R           = POSPRED[ADDRESS] - MYPOS; R.W = 0.0;
        DOUBLE  R2          = DOT(R, R) + EPS2;
        DOUBLE  MASS_INV_R3 = POSPRED[ADDRESS].W * NATIVE_RSQRT(POWN(R2,3));
        DOUBLE  INV_R2      = NATIVE_RECIP(R2);

        DOUBLE4 V = VELPRED[ADDRESS] - MYVEL; V.W = 0.0;
        DOUBLE4 A = ACCPRED[ADDRESS] - MYACC; A.W = 0.0;

        DOUBLE ALPHA = (DOT(R, V) * INV_R2);
        DOUBLE BETA  = ((DOT(V, V) + DOT(R, A)) * INV_R2) + (ALPHA * ALPHA);

        DOUBLE4 A1    = (MASS_INV_R3 * R);
        DOUBLE4 A1DOT = (MASS_INV_R3 * V) - (3.0 * ALPHA * A1);
        DOUBLE4 A2DOT = (MASS_INV_R3 * A) - (6.0 * ALPHA * A1DOT) - (3.0 * BETA * A1);

        ACC += A1;
        JRK += A1DOT;
        SNP += A2DOT;
      } /* AVOID SELF-INTERACTION */
  } /* LOOP OVER PPDEV PARTICLES */

/* STORE RESULTS IN GLOBAL MEMORY */
AC[PART]  = ACC;
AC1[PART] = JRK;
```

```
      aC2[Part] = snp;
    } /* Loop over particles */

  return;
}
```

## References


[1] G. Taffoni, G. Murante, L. Tornatore, D. Goz, S. Borgani, M. Katevenis, N. Chrysos, M. Marazakis, Shall numerical astrophysics step into the era of Exascale computing?, in: Astronomical Data Analysis Software an Systems XXVI (ADASS XXVI), Astronomical Society of the Pacific Conference Series, 2016.

[2] D. Goz, L. Tornatore, S. Bertocco, G. Taffoni, Direct N-body code designed for heterogeneous platforms, in: INAF-OATs technical report, 223, July 2018. doi:10.20371/INAF/PUB/2018_00002.

[3] D. Goz, L. Tornatore, S. Bertocco, G. Taffoni, Direct N-body code designed for a cluster based on heterogeneous computational nodes, in: INAF-OATs technical report, 224, August 2018. doi:10.20371/INAF/PUB/2018_00005.

[4] S. Bertocco, D. Goz, L. Tornatore, G. Taffoni, INCAS: INtensive Clustered ARM SoC - Cluster Deployment, in: INAF-OATs technical report, 222, August 2018. doi:10.20371/INAF/PUB/2018_00004.

[5] A. Maghazeh, U. D. Bordoloi, P. Eles, Z. Peng, General purpose computing on low-power embedded gpus: Has it come of age?, in: 2013 International Conference on Embedded Computer Systems:

Architectures, Modeling, and Simulation (SAMOS), 2013, pp. 1–10. doi:10.1109/SAMOS.2013.6621099.

[6] F. Perez, B. Granger, Ipython: A system for interactive scientific computing, Computing in Science Engineering 9 (3) (2007) 21–29. doi:10.1109/MCSE.2007.53.

[7] E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, [Online; accessed 2015-09-13] (2001–).
URL http://www.scipy.org/

[8] S. van der Walt, S. Colbert, G. Varoquaux, The numpy array: A structure for efficient numerical computation, Computing in Science Engineering 13 (2) (2011) 22–30. doi:10.1109/MCSE.2011.37.

[9] J. Hunter, Matplotlib: A 2d graphics environment, Computing in Science Engineering 9 (3) (2007) 90–95. doi:10.1109/MCSE.2007.55.