Energy-Performance Tradeoffs for HPC Applications on Low Power Processors

Enrico Calore[1(✉)], Sebastiano Fabio Schifano[2], and Raffaele Tripiccione[1]

[1] Dipartimento di Fisica e Scienze Della Terra, Università di Ferrara and INFN, Ferrara, Italy
enrico.calore@fe.infn.it

[2] Dipartimento di Matematica e Informatica, Università di Ferrara and INFN, Ferrara, Italy

Abstract. Energy efficiency is becoming more and more important in the HPC field; high-end processors are quickly evolving towards more advanced power-saving and power-monitoring technologies. On the other hand, low-power processors, designed for the mobile market, attract interest in the HPC area for their increasing computing capabilities, competitive pricing and low power consumption. In this work we study energy and computing performances of a Tegra K1 mobile processor using an HPC Lattice Boltzmann application as a benchmark. We run this application on the ARM Cortex-A15 CPU and on the GK20A GPU, both available in this processor. Our analysis uses time-accurate measurements, obtained by a simple custom-developed current monitor. We discuss several energy and performance metrics, interesting *per se* and also in view of a prospective use of these processors in a HPC context.

1 Introduction

The computational performances of current HPC systems are increasingly bounded by their power consumption, and this is only expected to become worse in the foreseeable future. This is also relevant from the point of view of operating costs; indeed, large computing facilities are considering the option to charge not only running time but also energy dissipation.

In response to these problems, high-end processors are quickly introducing more advanced power-saving and power-monitoring technologies [7]. On the other hand, low-power processors, designed for the mobile market, are gaining interest as it appears that they may eventually fill (or at least reduce) their performance gap with high-end processors and still keep a competitive edge on costs, thanks to the economies of scale associated to large production volumes of mobile devices [4,13]. The power consumption problem is starting to be approached also from the software point of view, with developers focusing not only on performance, but also learning to optimize codes to achieve the most acceptable trade-off between performance and energy efficiency [5,17].

In this paper we address one facet of these issues. We analyze in details, using accurate measurements, the role played by hardware factors and by some software aspects in the energy-performance landscape of real-life HPC applications.

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 737–748, 2015.
DOI: 10.1007/978-3-319-27308-2_59

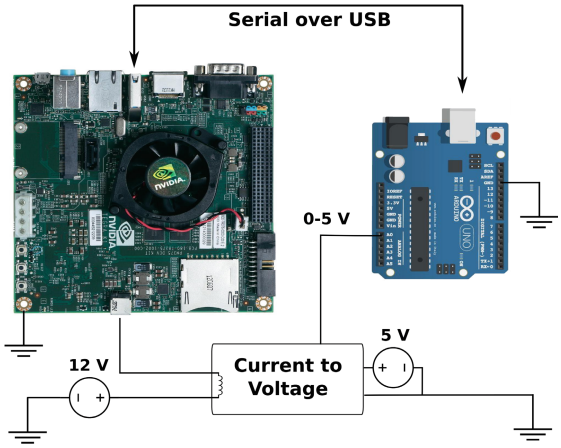


Fig. 1. Power monitoring setup. A benchtop power supply provides a constant 12 V voltage; the supply current passes through a current-to-voltage converter, whose analog output depends linearly on the current value. An Arduino UNO board digitizes and store these values and –at the end of the test– downloads them to the Jetson TK1 through a serial over USB connection.

Our application benchmark is a code based on Lattice Boltzmann methods, widely used in CFD, while our hardware testbed is a low-power Tegra K1 SoC (System on a Chip), embedding a multi-core CPU and a GPU. We use two versions of our code, optimized respectively for CPU and GPU, with different configurations and compilation options. We then measure energy consumption and performance of the computationally intensive kernels, using several clock frequency combinations, and building a large database of measured data. We then analyze these results, also guided by a simple but effective model of the energy behavior of our test system.

2 The Hardware Testbed

The hardware setup that we use is based on a NVIDIA Jetson TK1 development board, embedding a Tegra K1 SoC, and a custom current monitoring system, able to acquire and store current values and – at the end of the test – download them to the Jetson TK1. Our setup is shown in Fig. 1.

The Tegra K1 SoC, hosted on the Jetson TK1 board, has a CPU and a GPU on a single chip; the CPU is a NVIDIA "4-Plus-1", a 2.32 GHz ARM quad-core Cortex-A15 and a low-power shadow core; the GPU is a NVIDIA Kepler GK20a with 192 CUDA cores (with 3.2 compute capability). Both units access a shared DDR3L 2 GB memory bank on a 64-bit bus running at up to 933 MHz.

This system has several energy-saving features: cores in the CPU can be independently activated and the frequency of the CPU, GPU and memory system can be individually selected in a wide range (CPU: $204 \cdots 2320.5$ MHz in

20 steps; GPU: $72 \cdots 852$ MHz in 15 steps; Memory: $12.75 \cdots 924$ MHz in 12 steps). The system includes a *performance governor* that by default keeps only the low-power shadow core of the CPU active at low frequency when the processor is idle, and activates the other cores and increases their clock frequencies, as activity is detected within the system. The GPU clock frequency is also scaled in a similar fashion. For our tests, we find it more useful to disable this system and explicitly control all units and their frequency, in order to obtain accurate power consumption data for known values of frequencies and active cores.

We selected this board as our testbed for several reasons: (i) this SoC contains a multi-core CPU as well as a GPU, so we may test both architectures; (ii) this chip allows fine and independent control of the clock frequencies of CPU, GPU and memory interface; (iii) low-power systems are constantly improving their performance, so these systems may quickly become interesting building blocks for HPC platforms; (iv) the low power requirements of this system make it easy to develop an accurate power monitoring system.

```
int fd;
struct termios newtio, oldtio;

// Initialize Arduino Serial conn.
fd = init_serial(&oldtio, &newtio);

// Start arduino data acquisition
start_arduino_acq(fd);

run_my_function();

// Start arduino data read-out
start_arduino_readout(fd, filename, 900);

// Close serial connection
close_serial(fd, &oldtio);
```

Code 1.1: Example of code run on the Jetson board to trigger data acquisition and readout.

```
// This is called every ms by
// an hardware timer
ISR(TIMER0_COMPA_vect) {
 if (acquireData) {
   byte i;
   unsigned int sensorValue = 0;
   // Average over avgSamples readings
   // one read costs about 0.11ms
   for (i = 0; i < avgSamples; i++) {
     // read the input on analog pin0
     sensorValue += analogRead(A0);
   }
   isendBuffer[idx] = sensorValue;
   idx++;
 }
}
```

Code 1.2: Function executed every 1 ms on Arduino UNO board to acquire current samples.

The Jetson TK1 is powered by a single 12 V source, so its overall power consumption can be easily derived by current measurements. We have developed a simple system able to measure the current flowing into the Jetson TK1 board with very good accuracy and time resolution (≈ 1 msec) and able to correlate measurements with the execution of specific software kernels. The setup uses an analog current to voltage converter (using a LTS 25-NP current transducer) and an Arduino UNO board; the latter uses its embedded 10-bit ADC to digitize current readings and stores them in its memory. We synchronize the Arduino UNO and the Jetson TK1 through a simple serial protocol built over an USB connection. With this setup, a generic application running on the Jetson TK1 only needs to trigger the Arduino UNO to start acquisition immediately before launching the kernel function to be profiled. After the function under test completes, acquired data is downloaded from the Arduino UNO memory, so it can be stored and analyzed offline. The monitor acquires current samples with 1 ms granularity; for increased accuracy, multiple consecutive readings (e.g. 5 in our case) are performed and averaged. This setup is able to correlate current measurements with specific application events with an accuracy of a few milliseconds, minimally disrupting the execution of the kernel function to profile.

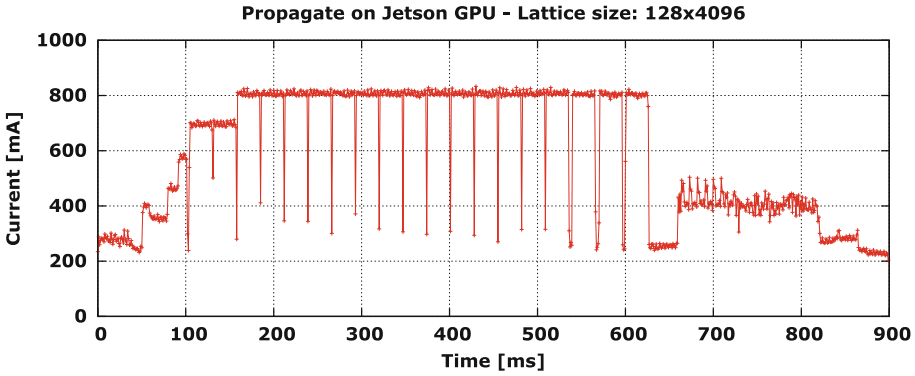


Fig. 2. Raw data collected by the current monitoring system as the Jetson GPU runs 20 iterations of a CUDA kernel. Current increases during the first iterations as the performance governor (in default mode) increases the clock frequency.

Code. 1.1 shows how to instrument a test program, while Code 1.2 shows the routine running on Arduino UNO every millisecond to acquire a current sample. Figure 2 shows a snapshot of raw current measurements; the plot refers to a CUDA kernel running 20 times consecutively on the Jetson's GPU, highlighting the good time resolution and accuracy. In this case, the configuration of the performance governor was the default one, so power consumption changes during the first iterations, reflecting automatic frequency scaling.

Once N current samples $i[n]$ are available for the time interval T_S corresponding to the execution of a given kernel, different power metrics can be computed. The instantaneous power can be computed as $p[n] = V \times i[n]$ and the average power as $P_{\text{avg}} = \frac{1}{N} \sum_{n=0}^{N-1} p[n]$. Another popular metric, the so-called *energy-to-solution* is defined as $E_{\text{S}} = T_S \times P_{avg}$.

3 The Application Benchmark

Lattice Boltzmann methods (LB) are widely used in computational fluid dynamics, to describe flows in two and three dimensions. LB methods [16] – 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 hop 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 $DnQp$; 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 ($p = \rho T$) [14,15]; this model has been extensively used for large scale simulations of convective turbulence (see e.g., [1,2]). In the algorithm, a set of populations ($f_l(\boldsymbol{x}, t)$ $l = 1 \cdots 37$), defined at the points of a discrete and regular lattice and each having a given lattice velocity $\boldsymbol{c}_l$, evolve in (discrete) time according to the following equation:

$$f_l(\boldsymbol{x}, t + \Delta t) = f_l(\boldsymbol{x} - \boldsymbol{c}_l \Delta t, t) - \frac{\Delta t}{\tau}\left(f_l(\boldsymbol{x} - \boldsymbol{c}_l \Delta t, t) - f_l^{(eq)}\right) \quad (1)$$

The macroscopic variables, density ρ, velocity $\boldsymbol{u}$ and temperature T are defined in terms of the $f_l(x,t)$ and of the $\boldsymbol{c}_l$s (D is the number of space dimensions):

$$\rho = \sum_l f_l, \qquad \rho \boldsymbol{u} = \sum_l \boldsymbol{c}_l f_l, \qquad D\rho T = \sum_l |\boldsymbol{c}_l - \boldsymbol{u}|^2 f_l, \quad (2)$$

the equilibrium distributions ($f_l^{(eq)}$) are themselves a function of these macroscopic quantities [16]. In words, populations drift from different lattice sites (*propagation*), according to the value of their velocities and, on arrival at point $\boldsymbol{x}$, they change their values according to Eq. 1 (*collision*). One can show that, in suitable limiting cases, the evolution of the macroscopic variables obeys the thermo-hydrodynamical equations of motion of the fluid. Close inspection of Eq. 1 also shows that the algorithm offers a huge degree of easily identifiable parallelism; this makes LB algorithms popular HPC massively-parallel applications.

An LB simulation starts with an initial assignment of the populations, in accordance with a given initial condition at $t = 0$ on some spatial domain, and iterates Eq. 1 for each point in the domain and for as many time-steps as needed. At each iteration two critical kernel functions are executed: (i) propagate moves populations across lattice sites collecting at each site all populations that will interact at the next phase (collide). Consequently, propagate moves blocks of memory locations allocated at sparse addresses, corresponding to populations of neighbor cells; (ii) collide performs all the mathematical steps associated to Eq. 1 and needed to compute the 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 are the most time consuming parts of any LB simulation. It is very helpful for our purposes that propagate involves a large number of sparse memory accesses, so it is strongly memory-bound. collide, on the other hand, is strongly compute-bound, heavily using the floating-point unit of either processor, and the performance of the floating-point unit is the ultimate bottleneck.

4 Measurements

Our benchmark is based on codes implementing the LB algorithm described in the previous section and exploiting to a large extent the available parallelism. On the GPU we run an optimized CUDA code, developed for large scale CFD simulations on large HPC systems [3,9]. On the CPU we run a plain C version [6,11] using NEON SIMD *intrinsics* exploiting the vector unit of the ARM Cortex-A15 cores. We also use OpenMP for multi-threading within the 4 cores and OpenMPI for future testing purposes on multiple boards.

We have instrumented both kernels as described in Sec. 2, and performed a large number of test runs, monitoring the current profile at all times during the

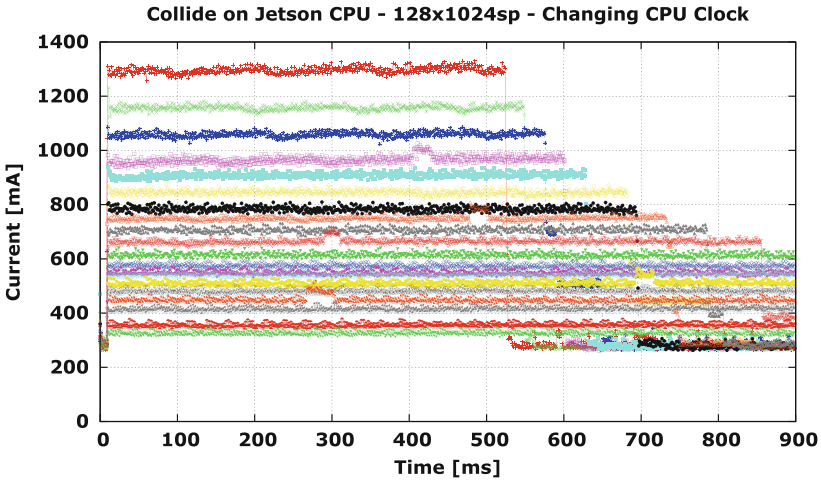


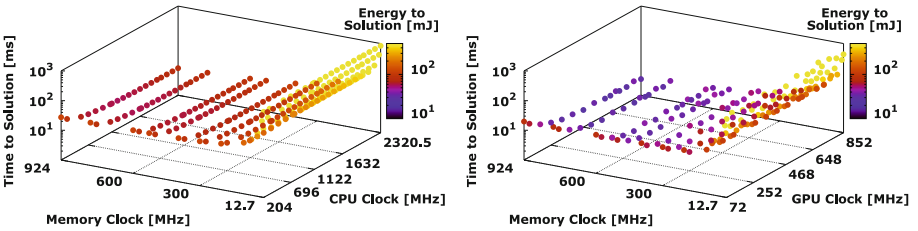
Fig. 3. Current measurements while running the collide kernel on the CPU with 4 OpenMP threads for a 128×1024 points lattice. Each run (plot line) is performed at a different CPU clock frequency, spanning between 204MHz (lowest green line) and 2.3GHz (highest red line). The Memory clock is set at its maximum value (Color figure online).

tests, accumulating a large data-base of measured data. On the software side, we have included runs with different numbers of OpenMP threads (for CPU) and CUDA block sizes (for the GPU); on the hardware side we have logged data for most combinations of the adjustable clock frequencies, disabling automatic frequency scaling. The C code using NEON *intrinsics*, was run on the CPU manually forcing the use of the G cluster (i.e. the high performance quad-core). When running on the GPU, the CPU was forced to use the LP cluster (i.e. the low performance shadow core). For the sake of a fair comparison, all tests adopt single precision for floating point data, since the Cortex-A15 is a 32-bit CPU and double precision vector instructions are not available.

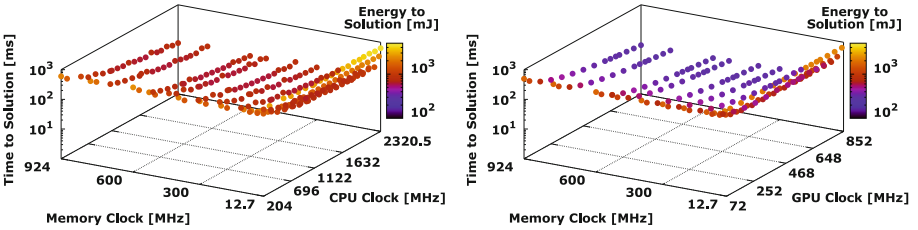
Figure 3 shows current data for collide given different settings of the CPU clock (GPU and memory clocks are fixed); similar results are available for the propagate kernel, for most of the clock combinations and for both GPU and CPU.

5 Results and Discussion

We consider *energy-to-solution* (E_S) and *time-to-solution* (T_S) – and the correlations thereof – as relevant and interesting parameters when looking for tradeoffs between time and energy conflicting requirements. Figure 4 shows measured values of T_S (vertical axis) and E_S (coded by colors) for propagate and collide kernels and for several clock frequency settings, when running on both CPU and GPU processors. From these plot, we see, for instance, that energy consumption is dominated by the collide kernel (notice the different color scales). propagate



(a) propagate function on the CPU (b) propagate function on the GPU



(c) collide function on the CPU (d) collide function on the GPU

Fig. 4. Time- and energy-to-solution of the profiled kernels while running on the CPU and GPU with different CPU, GPU and memory clock frequencies. Lattice size is 128x1024 points.

is equally performing and power-greedy on both processors; this was expected, since on this system the CPU and GPU share the same memory.

To better highlight the time/energy tradeoff, we plot E_S as a function of T_S, on either processors and for both kernels, see Fig. 5. Interestingly enough, E_S scales approximately linearly with T_S, but large fluctuations are present. A crude way to understand this behavior is as follows: as the processor executes a kernel, it consumes power in two ways: (i) the power associated to the (constant in time) background current (including the leakage current of the processor and the current drawn by ancillary circuits on the board) and (ii) the power associated to the switching activity of all gates of the processor as it transitions across different states while executing the program. The first term implies, in our crude model, a constant power rate (P_0), while the second term implies an average energy dissipation CV^2 every time a bit in the state of the processor toggles during execution (V is the processor power supply, while C is an average value of the capacitance associated to the output of each gate); this model derives directly from early power analyses found in classic books in VLSI design [12], and recently discussed in [8]. While we are fully aware that the actual situation is more complex, we let this simple model guide our further analysis. We fit E_S as a function of T_S as follows:

$$E_S = E_0 + P_0 \times T_S \tag{3}$$

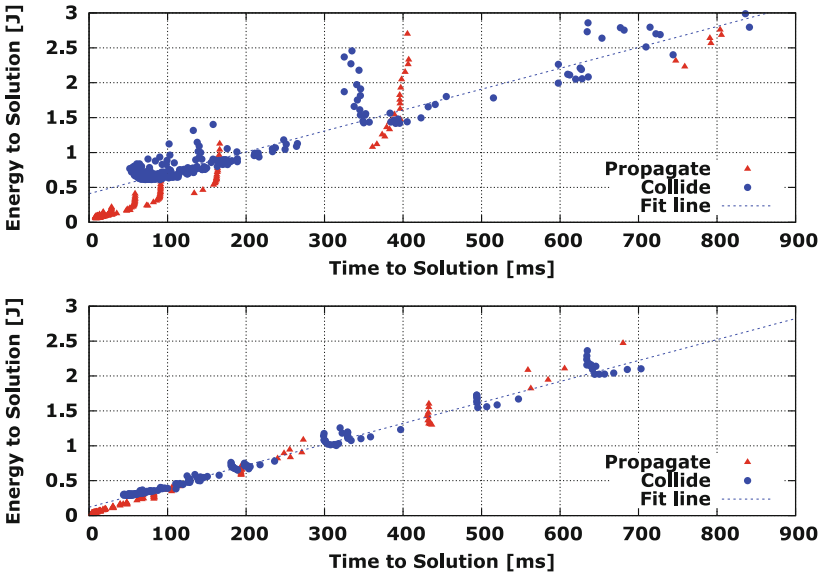


Fig. 5. Measured values of E_S vs. T_S on the CPU (top) and GPU (bottom), for the collide (blu) and propagate (red) kernels corresponding to several values of the clock frequencies. Results of a fit of collide data to Eq. 3 of is also shown (Color figure online).

P_0 should be independent of the program under test, while E_0 should depend on the kernel and the processor executing it, as – to first approximation – it counts the number of state transitions that the processor has to go to execute the code, *irrespective* of the frequency at which they happen.

Possible sanity checks for our models are: (i) P_0 is the same for all measured kernels and processors, and (ii) its value is close to the one derived from current measurements as the system rests in the idle state.

We fitted the two parameters from data for the collide kernel, obtaining $E_0^{CPU} = 410$ mJ, $P_0^{CPU} = 2.99$ W, $E_0^{GPU} = 120$ mJ, $P_0^{GPU} = 3.00$ W; as expected the two values for P_0 are almost equal and consistent with the value, ≈ 3 W, obtained from current readings (≈ 250 mA) when the system is idle.

It is interesting to subtract from E_S the contribution associated to the background current, as computed from Eq. 3, and define $E_0^K = E_S - P_0 \times T_S$, that should depend only on the profiled kernel (labelled by superscript K) and on the processing unit. Figure 6 plots E_0^K as function of T_S; this shows a clean lower-bound constant envelope of all data points – as expected from our model – with large fluctuations, clustering around some values of T_S. The insets in Fig. 6 provide a closer look at two such clusters, showing for each point a label with GPU (left) and memory (right) frequency. This clarifies the origin of these clusters: their points correspond to cases in which one of the two subsystems (either memory or compute-unit) [10] has become the performance bottleneck;

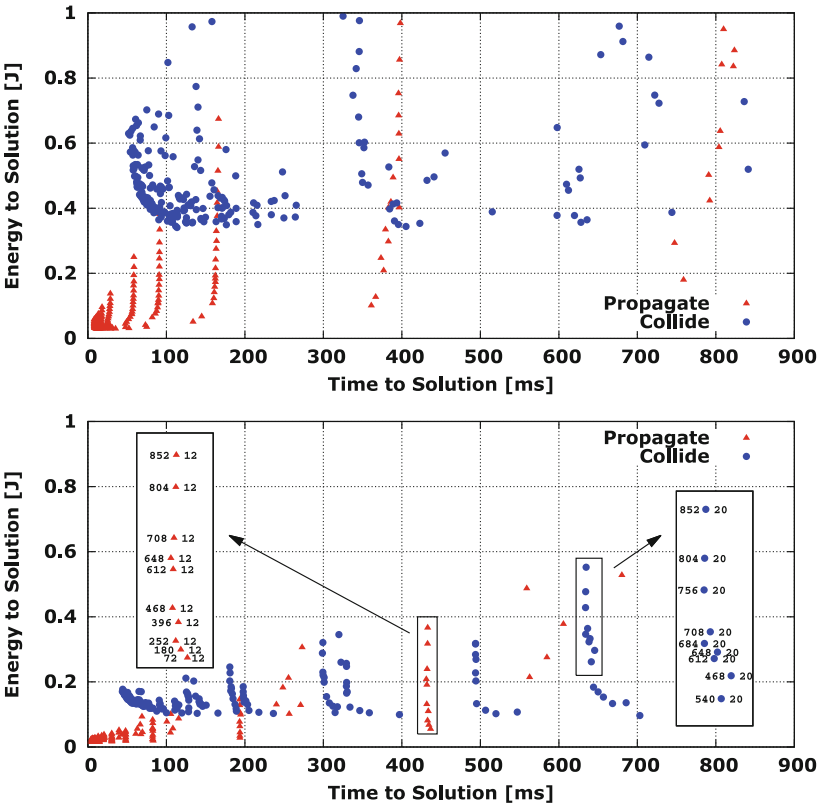


Fig. 6. Measured values of E_0^K, defined in the text, as a function of T_S, for the collide (blue) and propagate (red) kernels on CPU (top) and GPU (bottom). The insets zoom onto two data clusters, showing for each point the GPU (left) and memory (right) clock frequency in MHz (Color figure online).

increasing the frequency of the other subsystem means that the latter has to go through system states (stalled states) that do not advance the computation, so performance remains approximately constant, while energy dissipation increases.

This analysis helps identify best values of processor/memory clock pairs for each kernel, given a target T_S or an assigned energy budget. This is particularly relevant of course when T_S is close to its lower possible value: indeed, this is the only area (enlarged in Fig. 7 for the collide kernel on GPU) where one can look for an optimal energy/time trade-off; in fact, the plots make it evident that an accurate matching of clock frequencies is an effective way to reduce E_0^K; on the other hand, accepting lower performance – that is settling for T_S significantly longer than the best possible value – does not reduce E_0^K but rather increases the energy burden associated to the $P_0 \times T_S$ contribution, and consequently total energy dissipation. We also note that different kernels may have different ideal clocks pairs, so, after finding a satisfactory tradeoff for each of them, it helps if

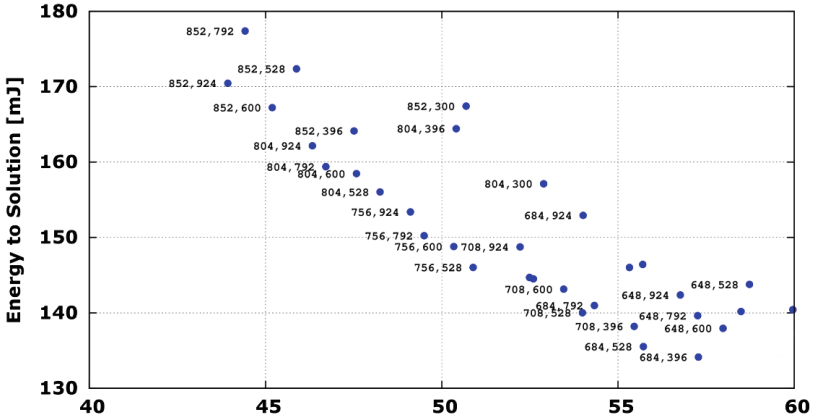


Fig. 7. Close-in view of the E_0^K vs. T_S graph, in a region corresponding to the best possible values for T_S. For each data point, we list the GPU (left) and memory (right) clock frequencies (MHz).

they can be dynamically adjusted before each kernel starts. Clock adjustments, in our test case, may improve E_0^K by large factors; however what is really relevant is the total energy dissipation, E_S, for which the background contribution is not small and linearly dependent on time; so the bottom line is that: (i) careful energy optimization may reduce E_S by $\approx 10 \cdots 20\,\%$ at best, and (ii) contrary to intuition, using very low clock frequencies is an ineffective way to reduce dissipation.

6 Conclusions and Future Works

The analysis of the previous section suggests the following remarks:

- to first approximation, the best energy saving option for this class of processors, correspond to running the system at a frequency close to the highest possible value, as determined from data shown in Figs. 5, 6 and 7, and then (if possible) remove power from the (sub-)system.
- as a corollary, options to run codes at very low frequencies are almost useless; it is probably more useful to add more flexible (and fast) options to remove power from parts of the processor; efficient ways to save the state of the subsystem before shut down would be most welcome.
- limited but not negligible power optimization is possible by adjusting clocks on a kernel-by-kernel basis; this is best done via direct energy profiling of the actual codes; it is then important that low-power system are able to measure their consumption with minimal disruption to the running code and make results easily available.
- reducing the latency time associated to clock changes is also important to make the selection of the best clock values possible even for short kernels, without significant performance loss.

– a more radical way to address the problem is to reduce V, as one reduces clock frequencies; in fact, to first approximation, V scales linearly with frequency, while power is proportional to V^2. We are not sure to which extent this strategy is already carried out for the Jetson TK1; we stress that this should be done consistently across the whole system.

In this paper, we have described a number of power benchmarks of a Jetson TK1 processor, made possible by a power monitoring system with good time resolution and accurate time correlation with the execution of the kernels under test. We have been able to disentangle the energy dissipation associated to the actual computation and the one associated to background currents. We have shown that – at least for this low power system – background dissipation has a significant impact on overall energy dissipation; in spite of that, limited but not negligible optimization is possible by carefully matching the values of all clock frequencies on the system; we have finally discussed possible new features that low-power systems should have to improve energy tuning.

We plan to continue along this line of analysis in several ways: (i) improving the current monitoring system, to have more information available and to measure how the various parts of the system contribute to energy dissipation; (ii) extend the analysis to more advanced low-power systems supporting double precision floating point maths, such as the forthcoming Jetson X1 board, and to high-end HPC accelerators, such as NVIDIA K40 or K80 GPUs; (iii) consider not only hardware-based tuning, but also software options toward energy saving: work is in progress along these directions.

Acknowledgements. This work has been done in the framework of the COSA, COKA and Suma projects, supported by INFN. The Jetson TK1 development board used as a testbed was awarded to E.C. for the best paper at UCHPC 2014.

References

1. Biferale, L., Mantovani, F., Sbragaglia, M., Scagliarini, A., Toschi, F., Tripiccione, R.: Reactive Rayleigh-Taylor systems: front propagation and non-stationarity. EPL **94**(5), 54004 (2011). doi:10.1209/0295-5075/94/54004
2. Biferale, L., Mantovani, F., Sbragaglia, M., Scagliarini, A., Toschi, F., Tripiccione, R.: Second-order closure in stratified turbulence: simulations and modeling of bulk and entrainment regions. Phys. Rev. E **84**(1), 016305 (2011). doi:10.1103/PhysRevE.84.016305
3. Calore, E., Schifano, S.F., Tripiccione, R.: On portability, performance and scalability of an MPI OpenCL lattice boltzmann code. In: Lopes, L., et al. (eds.) Euro-Par 2014, Part II. LNCS, vol. 8806, pp. 438–449. Springer, Heidelberg (2014)
4. Choi, J., Dukhan, M., Liu, X., Vuduc, R.: Algorithmic time, energy, and power on candidate HPC compute building blocks. In: IEEE 28th International Parallel and Distributed Processing Symposium, pp. 447–457 (2014). doi:10.1109/IPDPS.2014.54

5. Coplin, J., Burtscher, M.: Effects of source-code optimizations on GPU performance and energy consumption. In: Proceedings of the 8th Workshop on General Purpose Processing Using GPUs, GPGPU 2015, pp. 48–58 (2015). doi:10.1145/2716282.2716292
6. Crimi, G., Mantovani, F., Pivanti, M., Schifano, S.F., Tripiccione, R.: Early experience on porting and running a lattice boltzmann code on the xeon-phi co-processor. Procedia Comput. Sci. **18**, 551–560 (2013). doi:10.1016/j.procs.2013.05.219
7. Hackenberg, D., Ilsche, T., Schone, R., Molka, D., Schmidt, M., Nagel, W.: Power measurement techniques on standard compute nodes: a quantitative comparison. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 194–204 (2013). doi:10.1109/ISPASS.2013.6557170
8. Kim, N., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J., Irwin, M., Kandemir, M., Narayanan, V.: Leakage current: moore's law meets static power. Computer **36**(12), 68–75 (2003). doi:10.1109/MC.2003.1250885
9. Kraus, J., Pivanti, M., Schifano, S.F., Tripiccione, R., Zanella, M.: Benchmarking GPUs with a parallel lattice-boltzmann code. In: 25th Int. Symposiumon Computer Architecture and High Performance Computing (SBAC-PAD), pp. 160–167. IEEE (2013). doi:10.1109/SBAC-PAD.2013.37
10. Laurenzano, M.A., Tiwari, A., Jundt, A., Peraza, J., Ward Jr, W.A., Campbell, R., Carrington, L.: Characterizing the performance-energy tradeoff of small ARM cores in HPC computation. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014 Parallel Processing. LNCS, vol. 8632, pp. 124–137. Springer, Heidelberg (2014)
11. Mantovani, F., Pivanti, M., Schifano, S.F., Tripiccione, R.: Performance issues on many-core processors: a D2Q37 lattice boltzmann scheme as a test-case. Comput. Fluids **88**, 743–752 (2013). doi:10.1016/j.compfluid.2013.05.014
12. Mead, C., Conway, L.: Introduction to VLSI systems, vol. 802. Addison-Wesley, Reading (1980)
13. Rajovic, N., Rico, A., Puzovic, N., Adeniyi-Jones, C., Ramirez, A.: Tibidabo: making the case for an ARM-based HPC system. Future Generation Computer Systems **36**, 322–334 (2014)
14. Sbragaglia, M., Benzi, R., Biferale, L., Chen, H., Shan, X., Succi, S.: Lattice boltzmann method with self-consistent thermo-hydrodynamic equilibria. J. Fluid Mech. **628**, 299–309 (2009). doi:10.1017/S002211200900665X
15. Scagliarini, A., Biferale, L., Sbragaglia, M., Sugiyama, K., Toschi, F.: Lattice boltzmann methods for thermal flows: continuum limit and applications to compressible Rayleigh-Taylor systems. Phys. Fluids (1994-present) **22**(5), 055101 (2010). doi:10.1063/1.3392774
16. Succi, S.: The Lattice-Boltzmann Equation. Oxford University Press, Oxford (2001)
17. Wittmann, M., Hager, G., Zeiser, T., Treibig, J., Wellein, G.: Chip-level and multi-node analysis of energy-optimized lattice Boltzmann CFD simulations. Concurr. Comput. Pract. Exp. (2015). doi:10.1002/cpe.3489. ISSN: 1532-0634