PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A

rsta.royalsocietypublishing.org

Research

Cite this article: Anzt H, Quintana-Ortí ES. 2014 Improving the energy efficiency of sparse linear system solvers on multicore and manycore systems. *Phil. Trans. R. Soc. A* **372**: 20130279.
http://dx.doi.org/10.1098/rsta.2013.0279

One contribution of 14 to a Theme Issue 'Stochastic modelling and energy-efficient computing for weather and climate prediction'.

Subject Areas:
computational physics, power and energy systems, applied mathematics, computational mathematics

Keywords:
high performance computing, energy efficiency, sparse linear systems, iterative solvers, multicore processors, graphics processing units

Author for correspondence:
H. Anzt
e-mail: hanzt@icl.utk.edu

Royal Society Publishing

Improving the energy efficiency of sparse linear system solvers on multicore and manycore systems

H. Anzt[1] and E. S. Quintana-Ortí[2]

[1]Innovative Computing Laboratory (ICL), University of Tennessee at Knoxville, Knoxville, TN 37996, USA
[2]Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain

While most recent breakthroughs in scientific research rely on complex simulations carried out in large-scale supercomputers, the power draft and energy spent for this purpose is increasingly becoming a limiting factor to this trend. In this paper, we provide an overview of the current status in energy-efficient scientific computing by reviewing different technologies used to monitor power draft as well as power- and energy-saving mechanisms available in commodity hardware. For the particular domain of sparse linear algebra, we analyse the energy efficiency of a broad collection of hardware architectures and investigate how algorithmic and implementation modifications can improve the energy performance of sparse linear system solvers, without negatively impacting their performance.

1. Introduction

Linear algebra has been at the heart of the biannual Top500 (the Top500 List, http://www.top500.org/.) and Green500 (the Green 500 List, http://www.green500.org/.) rankings of supercomputers since their respective inception in 1993 and 2005. In particular, the Top500 list classifies computers according to their raw performance (in terms of GFLOPS, or millions of floating-point arithmetic operations per second) when solving a linear system of equations via the LU factorization.[1] As a response to the increasing concern about the energy draw by high performance computing (HPC) facilities,

[1]High Performance Linpack (HPL, http://www.netlib.org/benchmark/hpl/).

the scientific computing community has welcomed the Green500 list, which puts the emphasis on power efficiency by using a performance–power ratio rather than using raw performance (concretely, the GFLOPS/W), where the performance is again based on the aforementioned HPL benchmark.

The egregious benefits that current petascale applications deliver, and those that are expected from the problems to be tackled in the upcoming exascale decade [1–4], motivate the construction of enormous and complex HPC systems. If we extrapolate the technologies underlying the current no. 1 in the Top500 and Green500 lists to build an exascale machine, the outcomes are systems that would dissipate 408 and 312 MW, respectively.

While HPL has long been criticized for not representing the actual performance of the applications currently running on HPC systems, the importance of linear algebra is far beyond just being at the base of this benchmark or modern candidates to replace it [5]. Indeed, the solution of dense and sparse linear systems is ubiquitous in scientific computing, being at the bottom of the 'food chain' for many computational applications. Therefore, any effort towards improving the performance and energy efficiency of this type of operation will surely have a strong positive impact on the countless numerical simulations that build upon them.

In this paper, we first provide a general overview of current research and developments on energy-aware HPC by reviewing software- and hardware-based power and energy evaluation methods, metrics used for quantifying the resource efficiency of numerical algorithms, and power-saving mechanisms integrated into hardware. We then specifically focus on sparse linear algebra, comparing the efficiency of different system architectures when solving sparse systems of linear equations using the conjugate gradient (CG) method, and then showing how to improve the energy efficiency of sparse linear system solvers based on this method on multicore and manycore architectures.

2. Energy-efficient computing

In everyday usage, there is often an inadequate distinction between the terms 'power' and 'energy', though the physical difference is significant. In particular, energy is the integral of power over time and, therefore, energy measures the amount of 'work' necessary to perform a task, whereas power is the rate at which work is performed and, therefore, the rate at which energy is used.

When talking about the resource demand of HPC facilities especially, the fundamental difference between power optimization and energy optimization is neglected. The term power optimization refers to adapting the power draft of the system running a specific application to some desired value, given, for example, by an upper bound or smooth transition between different values. These restrictions are in most cases motivated by the infrastructure.

When optimizing the system/application combination with respect to energy, the aim is instead to decrease the product of time and (average) power, which can be achieved by reducing execution time, diminishing average power draft or a combination of both. Thus, in some cases, it may be possible to attain a lower energy utilization at the expense of a higher power draft, or vice versa.

(a) Power metrics

As infrastructure or user demands may impose restrictions on the power draft, new metrics accounting for the key factors of runtime, power draft and total energy consumption of a system/application combination are essential for optimization. While GFLOPS/W is the de facto standard to measure the energy efficiency of a computing system (the Green 500 List, http://www.green500.org/; the Top500 List, http://www.top500.org/), recent work [6] provides strong evidence that this metric tends to promote power-hungry algorithms that deliver high

sustained performance, when, in reality, the objective should be the reduction of the total energy, preferably along with the minimization of the time-to-solution (TTS).

To avoid this effect, one can rely on the energy-delay product (EDP) [7], which combines both factors, TTS and energy, into a single figure of merit, thus promoting algorithms that are more power-friendly. The FFTSE [6] progresses further along this line by considering the product between the energy and a function f of TTS. Different cost functions for f thus lead to distinct cases: $f(t) = 1$, with $t = \text{TTS}$, renders FTTSE as simply equivalent to the energy, whereas FTTSE is analogous to EDP when $f(t) = t$. More interestingly, the linear model $f(t) = \alpha \cdot t$, with α a scalar, allows one to weight (penalize) for time increases; and the exponential case $f(t) = e^{\alpha(t-\tau)}$ exerts a stronger pressure when the TTS exceeds a certain threshold τ.

(b) Technologies for power measurement

A common approach to measuring power and energy consumption is to leverage a hardware device, in general a wattmeter, connected between the electric power outlet and the power supply unit (PSU) of the system. Alternatively, the measurement point can be inside the system itself, connecting the hardware device to the lines that run from the PSU to the different components (motherboard, disk, network card, etc.). These power measurement devices can be mainly classified according to four criteria: nature, location (internal or external, as we just observed), accuracy and sampling rate.

Nature: some devices measure the active power, that is, average power consumption obtained from the energy measured during a second, whereas other devices instead provide the instantaneous electric power that is obtained by testing the current and voltage of the machine.

Location: the point of measurement has significant consequences. Internal supervision cards measure power consumption inside the machine (i.e. downstream from the PSU) unlike other devices which sample externally (upstream of the PSU). This implies that power measurements from such internal cards refer to direct current (DC) power and do not take into account the power wasted by the PSU. By contrast, external devices measure the overall alternating current (AC) power, and therefore include the PSU inefficiencies, which can represent a non-negligible portion of the total consumption of the machine. On the other hand, the external power meters, in general, operate at a lower sampling rate than their internal counterparts.

Accuracy and *sampling rate*: the variability of these two dimensions can be large: from devices that offer a measurement uncertainty of ± 7 W and/or a sampling rate of 1 Hz (sample per second)—in general too large to capture power variations of few watts and/or a few seconds—to extremely accurate, though in general highly expensive, elements with error margins below ± 0.1 W and sampling rates in the range 10 kHz and above.

Recent processor technologies are increasingly equipped with diverse types of counters that provide power measurements, either extracted from real data or estimated via a model. This is the case, for example, of the *running average power limit* (RAPL) model, available in the Intel Sandy Bridge microarchitecture; the *NVIDIA management library* (NVML) for graphics processors; and the *thermal power management device* (TPMD) and the *automated measurement of systems for temperature and energy reporting* (Amester) for IBM's Power 7 processors. Unfortunately, these technologies can be leveraged to measure only the power consumed by the processors, not the whole machine. Moreover, in some cases (e.g. RAPL), this is an intrusive solution as it is necessary to access the energy counters via a daemon program running on the same machine that is monitored, thus introducing unwanted noise into the measurement process.

The general conclusions from El Mehdi Diouri *et al*. [8] indicate that, thanks to their high sample rate, the internal measurement devices allow for better identification of some power fluctuations that the external wattmeters are not able to capture. However, a high sample rate is not always necessary to understand the evolution of the power consumption during the execution of a benchmark. Moreover, monitoring very large-scale distributed systems with internal wattmeters is not practicable, as the integration of those is usually not only expensive but

also very difficult. For these reasons, other sources such as RAPL, NVML or TPMD are becoming very appealing for monitoring purposes.

(c) Software for power measurement

An excellent survey on hardware and software for power profiling is given in [9]. Focusing on the software tools, PowerPack [10] is able to identify which lines feed different components such as disks, memory, network interface controllers, processors, etc. This information is then offered to the user who can gain insights on where and how applications consume power. PowerPack uses a user-friendly interface, and targets applications running on single-node platforms, though PowerPack's information can be 'manually' aggregated for parallel message passing interface (MPI) applications.

HDTrace [11] is a package to trace and simulate the power–performance footprint of MPI programs on a cluster. This software supports MPICH2 and the parallel virtual file system. Recently, it has also been extended to identify power hot spots in these applications, using information from commercial AC wattmeters.

pmlib [12] is a framework of tools used to analyse the power dissipation and the energy consumption of parallel MPI and/or multi-threaded scientific applications that comprise (i) a simple API (application programing interface) to interact with a number of power measurement devices, (ii) a library and a collection of drivers to capture the power dissipated during the execution of an application in a separate system, and (iii) an interface with integrated `Extrae` + `Paraver` packages (the Paraver Project, http://www.bsc.es/plantillaA.php?cat_id=485.) that permits an interactive analysis of a graphical trace, relating the power dissipation per node/core to information on the core activity.

(d) Energy-efficient processor technology

Current processor technology uses tools and mechanisms originally designed for embedded systems and the mobile market to use energy more efficiently. As of today, most processors adhere to the advanced configuration and power interface (ACPI) standard [13], which defines a collection of states the system can be set to depending on the workload, and thus offers a tool to tune the power usage to the current needs.

From the processor point of view, ACPI defines the so-called performance states (or P-states), which define the operating voltage–frequency pair of the processor. Vendors customize their products to feature a varying number of P-states, depending on the type of application. Additionally, in some cases, the P-state can only be set for all cores in the socket, whereas others permit each core to operate in a different P-state. In the past few years, the cost and latency of transitioning between P-states has been decreasing, and the management of the P-states has been incorporated into the operating systems (e.g. via Linux governors). State P0 corresponds to the highest voltage–frequency pair and, therefore, the most power-consuming state. Subsequent states are enumerated as P1, P2, etc.

ACPI includes an additional energy-saving mechanism in the form of the processor or CPU power states (C-states). This tool defines how and when certain processor components can be switched off to improve energy conservation. In state C0, the processor is active. In 'deeper' states (C1, C1E, C2 and so on) the clock signal is shut down to certain components (e.g. the floating-point units or certain levels in the cache hierarchy). Thus, a deeper C-state can potentially yield higher energy savings, but the latency to transition to the active C0 state will also be longer. While the P-states can be operated by the user (e.g. via utilities such as `cpufreq`), for security, the C-states are only accessible to the hardware and the operating system. In this sense, the only possibility for the programmer to interact with these states is by setting the appropriate conditions in the application so that the operating system can promote the processor, when idle, to an energy efficient C-state. Section 3 offers two relevant examples of this.

$x_0 := 0$
$r_0 := b - Ax_0$
$d_0 := r_0$
$\beta_0 := r_0^T r_0$
$\tau_0 :=\| r_0 \|_2= \sqrt{\beta_0}$
$k := 0$
while $(k < maxiter)$ & $(\tau_k > maxres)$
  $z_k := Ad_k$
  $\rho_k := \beta_k / d_k^T z_k$
  $x_{k+1} := x_k + \rho_k d_k$
  $r_{k+1} := r_k - \rho_k z_k$
  $\beta_{k+1} := r_{k+1}^T r_{k+1}$
  $\alpha_k := \beta_{k+1} / \beta_k$
  $d_{k+1} := r_{k+1} + \alpha_k d_k$
  $\tau_{k+1} :=\| r_{k+1} \|_2= \sqrt{\beta_{k+1}}$
  $k := k + 1$
end

Figure 1. Algorithmic formulation of the CG method.

3. Energy-efficient solution of sparse linear systems

(a) Preliminaries

Sparse linear algebra is in many cases characterized by the solution of linear equations with sparse coefficient matrices. For this purpose, iterative methods are often preferred to direct solvers as they usually provide high accuracy solution approximations with a significantly lower computational effort [14,15]. Although a large variety of iterative methods exist, those based on Krylov subspaces[2] have shown superior performance for many applications, therefore becoming the de facto standard in many sparse linear algebra software libraries.

The CG method, introduced by Hestenes & Stiefel in 1952 [15], is among the most effective Krylov subspace solvers for symmetric positive definite (SPD) systems. As all experiments in this section will be based on this method, we shortly recall the algorithm and its main characteristics next. For a linear system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is an SPD coefficient matrix, $b \in \mathbb{R}^n$ contains the independent terms and $x \in \mathbb{R}^n$ is the solution, the method is algorithmically formulated in figure 1, where the user-defined parameters *maxres* and *maxiter* set upper bounds, respectively, on the relative residual for the computed approximation to the solution x_k and the maximum number of iterations.

In practical applications, the computational cost of the CG method is dominated by the matrix–vector multiplication $z_k := Ad_k$. Given a sparse matrix A with n_z non-zero entries, this operation roughly requires $2n_z$ floating-point arithmetic operations (flops). Additionally, the loop body contains several vector operations (for the updates of x_{k+1}, r_{k+1}, d_{k+1} and the computation of ρ_k and β_{k+1}) that cost $O(n)$ flops each. A detailed derivation of the CG method can be found in [16], including a broad convergence analysis.

In the following sections, we apply this algorithm to generate a solution approximation to different linear systems where the SPD coefficient matrices are either chosen from the University of Florida Matrix Collection (UFMC),[3] corresponding to finite-element discretizations of several

[2]While, in principle, Krylov subspace methods belong to the class of direct solvers, as they approximate the solution in a subspace expanded in every iteration by an additional search direction until the complete space is generated, they are almost always used as iterative solvers providing a very good approximation of the solution after a few iterations.

[3]See http://www.cise.ufl.edu/research/sparse/matrices/.

Table 1. Description and properties of the test matrices.

source	acronym	matrix	no. non-zeros (n_z)	size (n)	n_z/n
UFMC	AUDI	AUDIKW_1	77 651 847	943 645	82.28
	BMW	BMWCRA1	10 641 602	148 770	71.53
	CRANK	CRANKSEG_2	14 148 858	63 838	221.63
	F1	F1	26 837 113	343 791	78.06
	INLINE	INLINE_1	38 816 170	503 712	77.06
	LDOOR	LDOOR	42 493 817	952 203	44.62
Laplace	A100	A100	6 940 000	1 000 000	6.94
	A126	A126	13 907 370	2 000 376	6.94
	A159	A159	27 986 067	4 019 679	6.94
	A200	A200	55 760 000	8 000 000	6.94
	A252	A252	111 640 032	16 003 001	6.94

structural problems arising in mechanics, or derived from a finite difference discretization of the three-dimensional Laplace problem. A few key parameters from these matrices are collected in table 1 while sparsity plots can be found online.[2] In all cases, the vector of independent terms b is initialized so that all the entries of the solution vector x equal 1, and the CG iteration is started with the initial guess $x_0 \equiv 0$.

All experiments (except for those in §3*c*) were performed using IEEE single-precision (SP) arithmetic. While the use of double-precision (DP) arithmetic is in general mandatory for the solution of sparse linear systems, in [17] it is shown how the use of mixed SP–DP arithmetic, in combination with iterative refinement, leads to improved execution time and energy consumption for GPU-accelerated solver-platforms. The underlying idea of a hybrid SP–DP solver, with iterative refinement, is visualized in figure 2 as an iterative process where the DP solution approximation is updated by the solution of an error correction equation based on the residual and solved in SP. Note that a major conclusion from [17] is that, with this kind of solver, the impact on the global execution time and energy consumption by the parts that are performed in DP arithmetic is negligible. Therefore, it is natural to study the cost of the SP solver. Unfortunately, there exists only a DP version of the solver evaluated in §3*c* so that, for those experiments, we had to rely on the more conventional DP arithmetic.

(b) Time–power–energy trade-offs

(i) General overview

Sparse linear algebra in general, and sparse linear system solvers in particular, are characterized by a high number of memory accesses compared with a low number of flops. Therefore, for many computing systems, the actual sustained performance peak that can be attained in a given platform is dictated by the bandwidth between the computational units and the level of the memory where the data resides. Hardware designs often try to account for this issue by either providing sophisticated memory hierarchies consisting of multiple cache levels, and/or data streaming mechanisms, such as those in GPUs [21].

Aiming for higher energy efficiency, many multicore systems allow for modification of the core clock rate (via the P-states). Although decreasing operating frequency for memory-bound operations may, in principle, yield lower energy usage while preserving performance, this is not true for all architectures, as processor frequency and memory throughput rate are sometimes

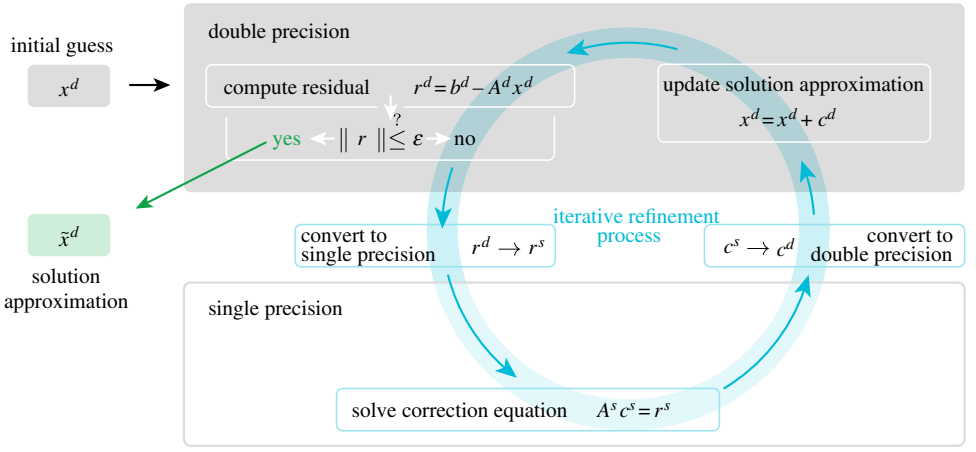

Figure 2. Mixed precision iterative refinement process [18–20] using an error correction solver in single precision ($.^s$) to generate solution updates for a system of linear equations $Ax = b$ in double precision ($.^d$). (Online version in colour.)

coupled. Decreasing the number of working cores on multicore platforms (thus enabling the idle hardware to transition to a sleep C-state) may have a similar effect: if the memory bandwidth is independent of the number of active cores, for this type of computations, we may be able to decrease the power draft without impacting performance by deactivating a subset of the cores.

In summary, the key factor that determines the energy efficiency for sparse linear algebra applications is the underlying paradigm concerning the hardware complexity. An architecture composed of a few complex cores, featuring high clock frequencies and deep cache hierarchies, provides high floating-point performance and greatly benefits from data locality, at the price of a high power draft. On the other hand, a low-power processor consisting of simple cores and an unsophisticated memory hierarchy, operating at moderate frequencies, aims for a lower power draft at the expense of extended execution time. Both paradigms, though, have to compete with the data-streaming approach taken by the manycore GPUs.

(ii) Experimental analysis

Next, we summarize the key results and observations of our energy efficiency evaluation of the CG method for the solution of sparse linear equation systems on several architectures representative of today's HPC landscape [22]. Following a recent approach that aims at designing an exascale architecture from low-power and embedded components (the Mont Blanc Project, http://montblanc-project.eu), we included several low-power processors in the comparison. One crucial challenge in conducting a fair comparison is choosing an appropriate level of hardware-dependent optimizations to the code as, for example, different systems usually benefit from distinct layouts of the data structures. In particular, data-streaming hardware usually exhibits significant performance increases for coalesced memory access patterns. To account for this issue, we adopt the ELLPACK sparse matrix storage format for GPU systems, while we rely on the standard compressed sparse row (CSR) format for the multicore architectures [23,24]. More details on the implementation of the sparse matrix–vector (SpMV) kernels underlying the CG method, for the CSR and ELLPACKT formats, can be found in [22].

In the following experiments, we execute 1000 iterations of the CG solver on the systems listed along with their characteristics in table 2, for the subset of the UFMC test matrices introduced in table 1, which fit in the main memory of all architectures (BMW, CRANK and F1), as well as the Laplace A159 matrix. All power measurements are collected using a WATTSUP?PRO wattmeter. In figure 3, we report the performance (figure 3*a*,*c*) and energy efficiency (figure 3*b*,*d*) for the

Table 2. Architectures with their corresponding idle power characteristics. For the GPU systems (FER, KEP and QDR), the idle power includes the accelerator.

acronym	architecture	total no. cores	frequency (GHz)–idle power (W)	RAM size, type	compiler
AIL	AMD Opteron 6276 (Interlagos)	8	1.4–167.29, 1.6–167.66, 1.8–167.31, 2.1–167.17, 2.3–168.90	66 GB, DDR3 1.3 GHz	gcc 4.4.6
AMC	AMD Opteron 6128 (Magny-Cours)	8	0.8–107.48, 1.0–109.75, 1.2–114.27, 1.5–121.15, 2.0–130.07	48 GB, DDR3 1.3 GHz	gcc 4.4.6
IAT	Intel Atom D510	2	0.8–11.82, 1.06–11.59, 1.33-11.51, 1.6–11.64	1 GB, DDR2 533 MHz	gcc 4.5.2
INH	Intel Xeon E5504 (Nehalem)	8	2.0–280.6, 2.33–281.48, 2.83–282.17	32 GB, DDR3 800 MHz	gcc 4.1.2
ISB	Intel E5-2620 (Sandy-Bridge)	6	1.2–93.35, 1.4–93.51, 1.6–93.69, 1.8–93.72, 2.0–93.5	32 GB, DDR3 1.3 GHz	gcc 4.1.2
ARM	ARM Cortex A9	4	0.62–11.7, 1.3–12.2	2 GB, DDR3L	gcc 4.5.2
FER	Intel Xeon E5520	8	1.6–222.0, 2.27–226.0	24 GB,	gcc 4.4.6
	NVIDIA Tesla C2050 (Fermi)	448	1.15	3 GB, GDDR5	nvcc 4.2
KEP	Intel Xeon i7-3930K	6	1.2–106.30, 3.2–106.50	24 GB,	gcc 4.4.6
	NVIDIA Tesla K20 (Kepler)	2496	0.7	5 GB, GDDR5	nvcc 4.2
QDR	ARM Cortex A9	4	0.120–11.2, 1.3–12.2	2 GB, DDR3L	gcc 4.5.2
	NVIDIA Quadro 1000M	96	1.4	2 GB, DDR3	nvcc 4.2
TIC	Texas Instruments C6678	8	1.0–18.0	512 MB, DDR3	cl6x 7.4.1

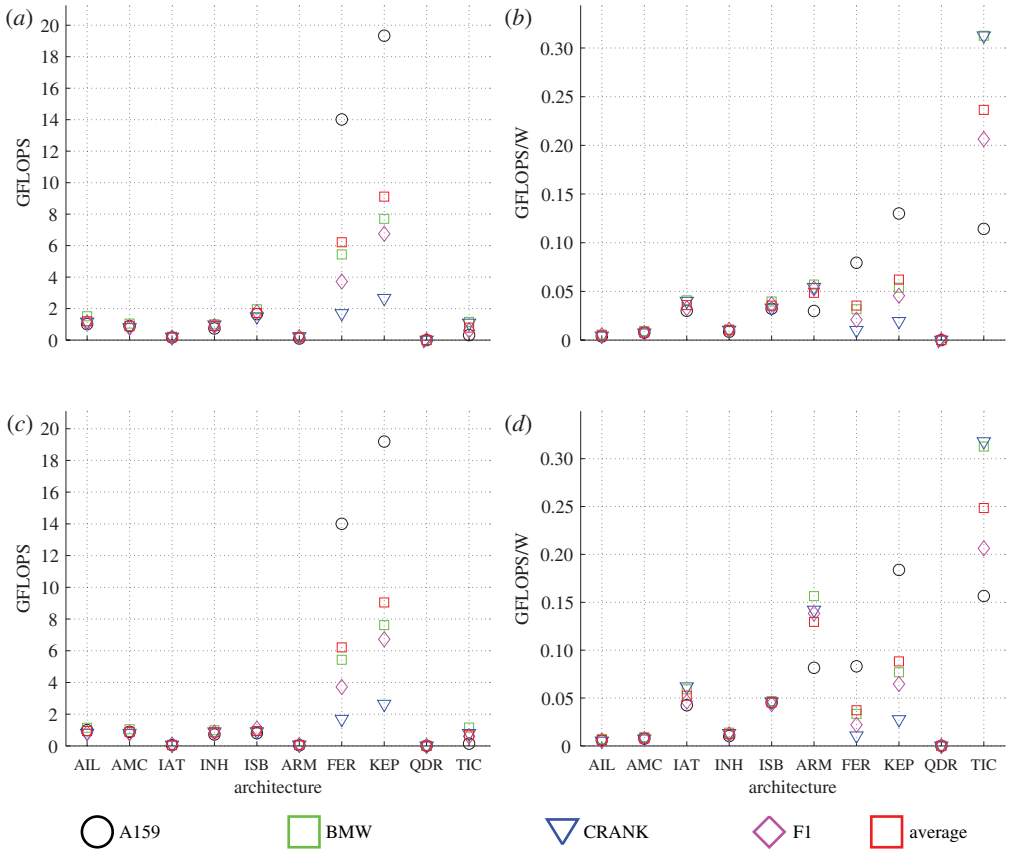

Figure 3. Comparison of performance (*a*,*c*) and performance/watt (*b*,*d*) when optimizing with respect to run time (*a*,*b*) or net energy (*c*,*d*), measured, respectively, in terms of GFLOPS and GFLOPS/W (1 GFLOPS $= 10^9$ flops s^{-1}). (Online version in colour.)

different test platforms when optimizing the operating frequency for either runtime performance (figure 3*a*,*b*) or energy efficiency (figure 3*c*,*d*).

Optimization with respect to time. In terms of runtime performance, the Tesla K20 (KEP) and the Tesla C2050 (FER) are superior to all other architectures by almost by an order of magnitude (see figure 3*a*). However, the performance of these two GPU-based systems strongly depends on the matrix sparsity pattern, a sensitivity that is also shared by the Quadro 1000M (QDR), which outperforms the C6678 DSP (TIC) only for those cases where the use of the ELLPACK format does not incur significant overhead on the GPUs. While TIC shows some variation on performance as well as the performance-per-watt ratio depending on the matrix structure, its energy efficiency is unmatched by any other architecture. The two other low-power processors, ARM and Atom (IAT), show better ratios than the conventional general-purpose processors (Interlagos AIL, Magny-Cours AMC, Nehalem INH, and Sandy-Bridge ISB), but only for some matrix cases do they show higher energy efficiency than the GPUs. For FER and QDR, it is interesting to note that, although they yield very different performance, their performance-per-watt ratios are almost equal [22].

Optimization with respect to the net energy efficiency. If we optimize the configurations with respect to energy consumption, it is, in many cases, beneficial to reduce the operating voltage and frequency. Particularly, significant improvements can be observed for the low-power processors as these architectures provide not only higher optimization potential, but also exhibit superior energy efficiency compared with the conventional general-purpose CPUs or GPUs [22]. Again, the TIC exhibits the best energy efficiency, followed by ARM, with the latter still superior to the GPUs or the general-purpose CPUs (see figure 3*d*). IAT is, in terms of energy efficiency, competitive with the FER and QDR GPU-based systems. Unfortunately, the decreased voltage and frequency, which

results in the superior energy efficiency, also incurs an extended run time as the reduced watt-per-iteration values for TIC, ARM and IAT come at the price of significantly higher execution times; see the performance decrease from figure 3*a*,*b* to figure 3*c*,*d*.

(c) Leveraging the CPU states on manycore systems

The results in §3*b* illustrate that GPUs are among the most energy-efficient hardware architectures for sparse linear algebra. Despite the fact that these hardware accelerators rarely offer any user-controlled power-saving mechanism, their vast number of floating-point arithmetic units, in combination with an outstanding bandwidth to memory and a data-streaming architecture, results in highly favourable performance/watt ratios. However, as GPUs cannot be used as stand-alone systems, one also has to take the power draft of the host into account. In the end, the energy efficiency depends on the performance/watt ratio of the GPU and the question becomes how to save power for the host.

In this section, we show how to leverage the CPU states in order to improve the energy balance of a GPU-based iterative solver. For this purpose, we evaluate different implementations of the CG method on a platform equipped with both a multicore processor and a GPU. First, we consider a straightforward CG-CUBLAS implementation that consists of individual routines that are either taken from the CUBLAS library or self-written kernels, reflecting specific numerical operations of the method (compare the left-hand side implementation in figure 5 with algorithm 1). This natural approach has not only the advantage of the clear representation of the mathematical description of the method, but also provides the possibility of replacing individual kernels for alternative codes, and, for these reasons, will later be used as a reference implementation.

One major disadvantage of this initial implementation becomes obvious when monitoring the power draft of the CPU, which is in charge of launching the GPU kernels for the vector- and matrix–vector operations. Counter to what one might expect, the CPU does not enter a power-efficient state during idle periods, but instead remains in a power-oblivious polling loop waiting for the GPU kernel to finish. In past work, we leveraged an idle-wait approach [25] to avoid this behaviour, though the same effect can be achieved by running the code in the CUDA blocking mode, which suspends the CPU thread when waiting for the GPU to finish a task. While the suspended threads of the CPU can now be promoted by the hardware into an inactive power-friendly C-state, e.g. C1 (halt) or deeper [13], the reactivation cost may cause some runtime overhead; see figure 4, which illustrates the power benefits and runtime overhead of the blocking mode for the CG-CUBLAS implementation applied to the AUDI test matrix. Hence, the generally superior power efficiency of the CUDA blocking mode comes along with a certain price in terms of performance, which may negatively affect energy consumption. As the runtime is, in general, regarded as the crucial figure, the challenge is thus to recover the performance of the polling mode while leveraging the power-saving mechanisms embedded in the hardware by operating in blocking mode. The key to this lies in developing a new implementation of the CG method that breaks down the rigid structure of the original code, so that several kernels can be gathered and merged together, with the result offering a higher computational intensity.

Let us reconsider the loop of the original CG-CUBLAS implementation, in the left-hand side of figure 5, corresponding to algorithm 1. We can observe that the loop body comprises a number of calls to CUBLAS kernels inside a loop that can be potentially invoked from hundreds to thousands of times. This large number of kernel calls and the frequent data transfers between GPU and main memory are major sources of overhead. Reducing the number of calls by merging kernels and avoiding synchronization owing to data transfers are both crucial to alleviating the time overhead, but can also render a lower number of activations of the CPU, yielding a more efficient usage of the C-states and, consequently, improved energy efficiency. (We note that this technique can be applied not only to the CG method but, in general, the same approach can be applied in many other codes that off-load most of the computations to the accelerator as a sequence of short GPU kernels.)

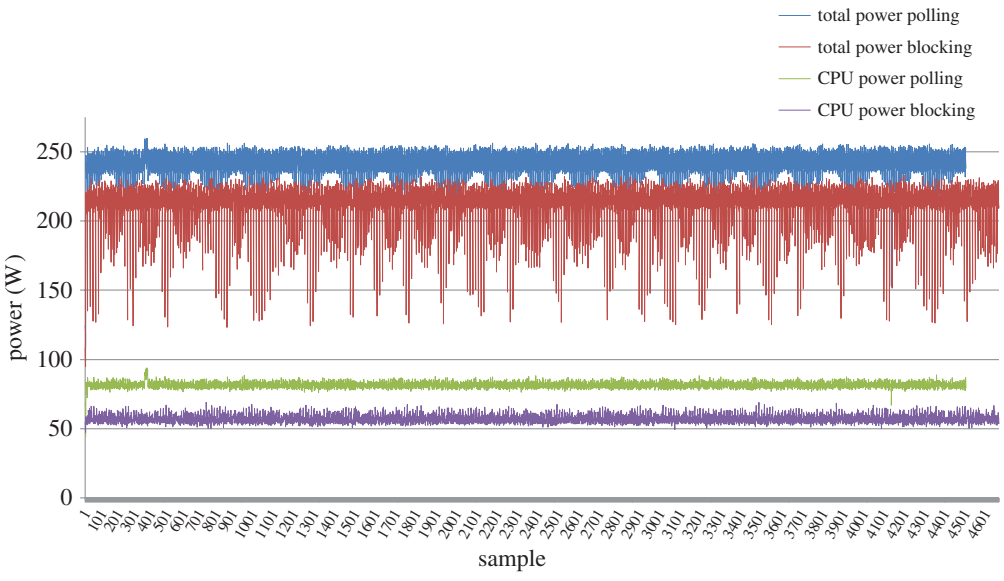

Figure 4. Power (W) dissipated by the CG-CUBLAS implementation, using the polling and blocking modes, for the AUDIKW_1 matrix. (Online version in colour.)

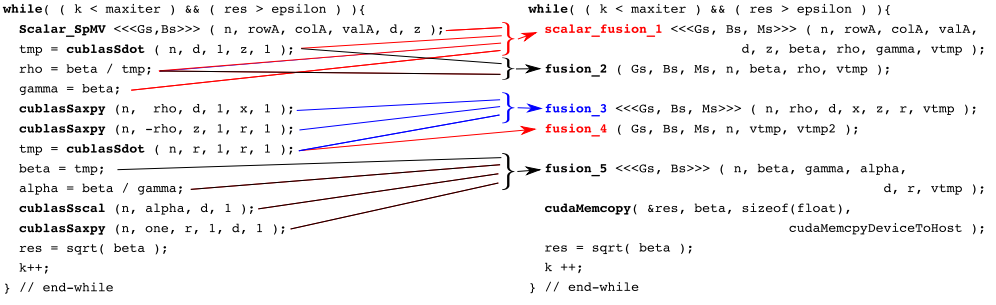

Figure 5. Aggregation of kernels to transform the CG-CUBLAS implementation into the CG-merged implementation. (Online version in colour.)

In [26], we describe how to decrease the kernel count by gathering multiple operations into a single GPU kernel, and how to reduce the host-device communication by keeping the outcome from reduction operations on the GPU. The resulting implementation, hereafter referred to as 'merge', is shown in the right-hand side of figure 5. An enhanced version, 'merge-s', that checks the convergence criterion not every single but every s iterations of the CG loop, is also proposed in [26]. While, on average, this technique incurs $s/2$ additional iterations of the solver, it will also decrease the transfers from GPU to main memory as $(11/s)$ of the transfers/synchronization are economized.

The benefits of the modifications visualized in figure 5 are summarized in figure 6 for all test matrices listed in table 1, showing the average variation of runtime (figure 6*a*) and CPU energy (figure 6*b*) in polling mode (left) and blocking mode (right), with respect to the CG-CUBLAS reference implementation running in polling mode. The target platform for this experiment was equipped with an Intel Core i7-3770K (four cores at 3.5 GHz) and 16 GB of RAM, connected to an NVIDIA GeForce GTX480 'Fermi' GPU. Power was measured using an internal DC wattmeter connected to the 12 V lines that run from the PSU to the motherboard. For further details about the experiment set-up and the results, see [26].

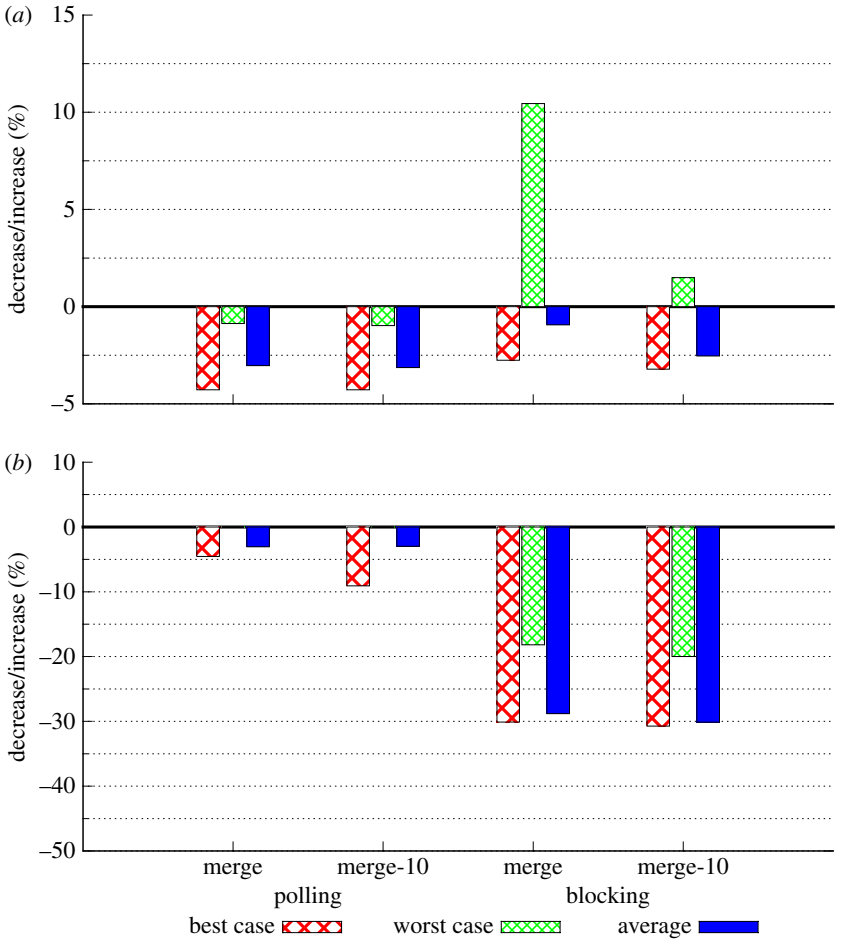

Figure 6. Comparison of time (*a*) and CPU energy (*b*) of the merged implementations, in polling (left) and blocking (right) modes, using the polling-CUBLAS code as reference. (Online version in colour.)

Concerning the performance objective, we deduce from the left-hand side of figure 6*a* that the merged implementations running in polling mode outperform the reference polling-CUBLAS implementation. The blocking mode may still have a negative impact on the execution time, and in the worst case we observe an increase of about 10%. However, this only occurs for very specific cases where the time for the matrix and vector operations is short compared with the cost of reactivating CPU cores, whereas, on average, the performance of the reference CUBLAS implementation can be matched. When the blocking-merge implementation is slower than the polling-CUBLAS code, shifting to blocking-merge-10 compensates for this, making the difference negligible. In all cases, the blocking-merge-10 implementation exhibits higher performance gains with respect to the reference than the blocking-merge code. In summary, figure 6*a* demonstrates that the performance goal is fulfilled as the merge-10 implementation in blocking mode matches or even outperforms CG-CUBLAS in polling mode.

Figure 6*b* illustrates the CPU energy consumption of the merged implementations, compared against those of the CG-CUBLAS code. While the merged variants executed in polling mode (left-hand side) may, on average, render some appealing savings owing to the reduced execution time, the clear winners are the same variants operating in blocking mode (right-hand side), which yield a reduction of the CPU energy draft of about 25–30%. Hence, the reformulated CG meets the twofold goal of rendering a significant reduction of the total energy consumption while retaining the runtime performance.

(d) Leveraging the CPU states on multicore systems

Here, we revisit the solution of sparse linear systems via ILUPACK (incomplete LU package[4]), a numerical software based on Krylov iterative methods. The package implements multilevel ILU preconditioners for general, symmetric indefinite and Hermitian positive definite systems, in combination with inverse-based ILUs and Krylov subspace solvers.

The parallelization of ILUPACK is analysed in [27], where the task concurrency intrinsic to the preconditioned iterative solution of the sparse linear systems is efficiently exploited on multicore architectures. Specifically, in this approach, parallelism is exposed by means of nested dissection applied to the adjacency graph representing the non-zero connectivity of the sparse coefficient matrix of the linear system. By recursively splitting this graph, the result is a hierarchy of independent subgraphs that is highly amenable to parallelization.

This type of concurrency can be expressed as a (directed) task dependency tree, where nodes represent concurrent tasks and arcs specify dependencies among them. The parallel execution of this tree on multicore processors is then orchestrated by a runtime which dynamically maps tasks to threads (cores) in order to improve load balance requirements during the computation of the ILU preconditioner and the subsequent solution of the triangular linear systems.

From the implementation point of view, the runtime keeps a shared queue of ready tasks (i.e. tasks with their dependencies fulfilled), which is initialized with the tasks corresponding to the independent subgraphs (leaves of the tree). Processor threads update this queue as they complete the execution of tasks allocated to them. The access to shared data structures is carefully controlled via low-cost synchronization primitives.

In the following, we evaluate the exploitation of the C-states made by two implementations of the runtime underlying ILUPACK. For this purpose, we use a scalable SPD sparse linear system of dimension $n = N^3$, resulting from the finite difference discretization of the partial differential equation $-\Delta u = f$ in a three-dimensional unit cube $\Omega = [0,1]^3$, with Dirichlet boundary conditions $u = g$ on $\partial\Omega$, using a uniform mesh of size $h = 1/(N+1)$. We set $N = 252$, which yields a linear system with about 16×10^6 unknowns and 111×10^6 non-zero entries in the coefficient matrix. The tests reported next were performed using IEEE DP arithmetic. The target server contains a single AMD Opteron 6128 processor (eight cores) and 24 GB of RAM. This processor features three operating or C-states (C0, C1 and C1E). Samples were collected in this case using an internal wattmeter. Unfortunately, there exists no version of ILUPACK using SP computations, so we have to use DP for the experiments in this section.

In the previous paragraphs, we exposed how the task-parallel calculation of the preconditioner in ILUPACK is organized as a binary tree and bottom-up dependencies. The subsequent iterative process basically requires the solution of two triangular linear systems per iteration, with tasks that are also organized as binary task trees, but with bottom-up (lower triangular system) or top-down (upper triangular system) dependencies. When these tasks are dynamically mapped to a multicore platform by the runtime, the execution can be expected to exhibit a number of time periods during which certain cores are idle, depending on the number of tasks of the tree, their computational complexity, the number of cores of the system, etc. As in the previous case, where we outsourced computations to the accelerator (see §3*c*), it is basically these idle periods that we could expect the operating system to leverage, by promoting the corresponding cores into a power-saving C-state (sleep mode).

In the original runtime underlying ILUPACK, upon encountering no tasks ready to be executed, 'idle' threads simply perform a 'busy-wait' (polling) on a condition variable, until a new task is available. This strategy prevents the operating system from promoting the associated cores into a power-saving C-state, because the threads are not idle but doing useless work by actively waiting for tasks.

As an alternative to the power-hungry strategy, a power-aware version of the runtime underlying ILUPACK can apply an 'idle-wait' (blocking) whenever a thread does not encounter a task ready for execution and, thus, becomes inactive. Note that, as previously mentioned in §2*d*,

[4]http://ilupack.tu-bs.de.

Table 3. Execution time (s), average power (W) and energy (J) of the power-oblivious and power-aware implementations of the runtime.

	preconditioner			iterative solver			total	
runtime	time	power	energy	time	power	energy	time	energy
oblivious	66.11	240.17	15 878.82	220.17	269.27	59 284.92	286.28	75 163.74
aware	66.52	227.16	15 112.06	221.39	237.80	52 646.41	287.91	67 758.84

setting the necessary conditions for the operating system to promote the cores into a power-saving C-state is all we can do, as we cannot explicitly enforce the transition from the user side.

The question is whether the use of the power-aware runtime comes with a performance penalty which masks the energy benefits. Table 3 compares the two runtimes, showing that this is not the case for both the computation of the preconditioner and iterative solution. Indeed, for a negligible increase in the total execution time, from 286.28 to 287.91 s, we observe reductions in the (average) power from 240.17 to 227.16 W for the preconditioner; and from 269.27 to 230.80 W for the solver. The outcome is a more than 10% decrease of the total energy from 75 163.74 to 67 758.84 J.

4. Concluding remarks

In this paper, we have reviewed the current status in energy-efficient HPC, focusing the analysis in the particular domain of sparse linear algebra. Specifically, we have experimentally compared different architectures using the solution of sparse linear systems via the CG method as the key benchmark. Furthermore, we have identified current challenges that stand in the path to effectively lower energy utilization, and we have also assessed the potential of energy-aware implementations. One key observation of our studies is that the energy efficiency of linear system solvers can be improved, without impacting runtime performance, by carefully adapting the algorithm's implementation to the target hardware. Doing so allows for better exploitation of the hardware-featured power-saving techniques. As a more general conclusion of this work, we believe that, in the context of the energy challenge of exascale computing, hand-in-hand development of computing architectures and simulation algorithms is crucial for achieving efficient resource utilization.

Acknowledgements. Enrique S. Quintana-Ortí was supported by project CICYT TIN2011-23283 of the Ministerio de Ciencia e Innovación and FEDER and the EU Project FP7 318793 'EXA2Green'. This work summarizes some key insights gained from a collaboration during the last few years. We thank the long list of colleagues that were involved in this cooperation.

References

1. Ashby S *et al*. 2010 The opportunities and challenges of exascale computing. Summary report of the Advanced Scientific Computing Advisory Committee (ASCAC) subcommittee.
2. Kogge P *et al*. 2008 ExaScale computing study: technology challenges in achieving exascale systems. Technical report TR-2008-13. Notre Dame, IN: University of Notre Dame. See http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.165.6676.
3. Dongarra J *et al.* 2011 The international ExaScale software project roadmap. *Int. J. High Perf. Comput. Appl.* **25**, 3–60. (doi:10.1177/1094342010391989)
4. Duranton M, Black-Schaffer D, De Bosschere K, Maebe J. 2013 The HiPEAC vision for advanced computing in horizon 2020. HiPEAC High-Performance Embedded Architecture and Compilation. See http://hdl.handle.net/1854/LU-3234355.
5. Dongarra J, Heroux MA. 2013 Toward a new metric for ranking high performance computing systems. Sandia report SAND2013-4744, Sandia National Laboratories.
6. Bekas C, Curioni A. 2010 A new energy aware performance metric. *Comput. Sci. Res. Dev.* **25**, 187–195. (doi:10.1007/s00450-010-0119-z)
7. Gonzalez R, Gordon BM, Horowitz MA. 1997 Supply and threshold voltage scaling for low power CMOS. *IEEE J. Solid-State Circuits* **32**, 1210–1216. (doi:10.1109/4.604077)

8. El Mehdi Diouri M, Dolz MF, Glück O, Lefèvre L, Alonso P, Catalán S, Mayo R, Quintana-Ortí ES. 2013 Solving some mysteries in power monitoring of servers: take care of your wattmeters! In *Energy efficiency in large scale distributed systems* (eds J-M Pierson, G Da Costa, L Dittmann). Lecture Notes in Computer Science, vol. 8046, pp. 3–18. Berlin, Germany: Springer. (doi:10.1007/978-3-642-40517-4_1)
9. Wang S, Chen H, Shi W. 2011 SPAN: a software power analyzer for multicore computer systems. *Sustain. Comput. Inform. Syst.* **1**, 23–34.
10. Ge R, Song S, Chang HC, Li D, Feng X, Cameron KW. 2010 PowerPack: energy profiling and analysis of high-performance systems and applications. *IEEE Trans. Parallel Distrib. Syst.* **21**, 658–671. (doi:10.1109/TPDS.2009.76)
11. Kunkel J. 2011 HDTrace: a tracing and simulation environment of application and system interaction. Technical report 2, Department of Informatics, Scientific Computing, Universität Hamburg.
12. Alonso P, Badia RM, Labarta J, Barreda M, Dolz MF, Mayo R, Quintana-Ortí ES, Reyes R. 2012 Tools for power-energy modelling and analysis of parallel scientific applications. In *Proc. 41st Int. Conf. on Parallel Processing* (*ICPP*), pp. 420–429. (doi:10.1109/ICPP.2012.57)
13. HP Corp., Intel Corp., Microsoft Corp., Phoenix Tech. Ltd, Toshiba Corp. 2011 *Advanced configuration and power interface specification, revision 5.0*.
14. George A. 1973 Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.* **10**, 345–363. (doi:10.1137/0710032)
15. Saad Y. 2003 *Iterative methods for sparse linear systems*. Philadelphia, PA: SIAM.
16. Shewchuk JR. 1994 An introduction to the conjugate gradient method without the agonizing pain. Technical report. Pittsburgh, PA: Carnegie Mellon University. See http://www.citeulike.org/user/asterix77/article/264342.
17. Anzt H, Castillo M, Fernández J, Heuveline V, Igual F, Mayo R, Quintana-Ortí E. 2012 Optimization of power consumption in the iterative solution of sparse linear systems on graphics processors. *Comput. Sci. Res. Dev.* **27**, 299–307. (doi:10.1007/s00450-011-0195-8)
18. Anzt H, Heuveline V, Rocker B. 2012 Mixed precision iterative refinement methods for linear systems: convergence analysis based on Krylov subspace methods. In *Applied parallel and scientific computing* (ed. K Jónasson). Lecture Notes in Computer Science, vol. 7134, pp. 237–247. Berlin, Germany: Springer. (doi:10.1007/978-3-642-28145-7_24)
19. Buttari A, Dongarra JJ, Langou J, Langou J, Luszczek P, Kurzak J. 2007 Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perf. Comput. Appl.* **21**, 457–486. (doi:10.1177/1094342007084026)
20. Baboulin M, Buttari A, Dongarra JJ, Langou J, Langou J, Luszczek P, Kurzak J, Tomov S. 2009 Accelerating scientific computations with mixed precision algorithms. *Comput. Phys. Commun.* **180**, 2526–2533. (doi:10.1016/j.cpc.2008.11.005)
21. Hennessy JL, Patterson DA. 2011 *Computer architecture: a quantitative approach*, 5th edn. San Francisco, CA: Morgan Kaufmann.
22. Aliaga J, Anzt H, Castillo M, Fernández JC, León G, Pérez J, Quintana-Ortí ES. In press. Performance and energy analysis of the iterative solution of sparse linear systems on multicore and manycore architectures. In *Parallel processing and applied mathematics*. Lecture Notes in Computer Science, vol. 8384. Berlin, Germany: Springer.
23. Barrett R *et al.* 1994 *Templates for the solution of linear systems: building blocks for iterative methods*, 2nd edn. Philadelphia, PA: SIAM.
24. Bell N, Garland M. 2008 Efficient sparse matrix-vector multiplication on CUDA. NVIDIA technical report NVR-2008-004. See http://www.nvidia.com/object/nvidia_research_pub_001.html.
25. Anzt H, Heuveline V, Aliaga JI, Castillo M, Fernández JC, Mayo R, Quintana-Ortí ES. 2011 Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms. In *2011 Int. Green Computing Conf. and Workshops* (*IGCC*), pp. 1–6. (doi:10.1109/IGCC.2011.6008594)
26. Aliaga J, Anzt H, Pérez J, Quintana-Ortí ES. 2013 Reformulated conjugate gradient for the energy-aware solution of linear systems on GPUs. In *Proc. 42nd Int. Conf. on Parallel Processing* (*ICPP*), pp. 320–329. (doi:10.1109/ICPP.2013.41)
27. Aliaga JI, Bollhöfer M, Martín AF, Quintana-Ortí ES. 2011 Exploiting thread-level parallelism in the iterative solution of sparse linear systems. *Parallel Comput.* **37**, 183–202. (doi:10.1016/j.parco.2010.11.002)