CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3341

SPECIAL ISSUE PAPER

Unveiling the performance-energy trade-off in iterative linear system solvers for multithreaded processors

José I. Aliaga[1,*,†], Hartwig Anzt[2], Maribel Castillo[1], Juan C. Fernández[1], Germán León[1], Joaquín Pérez[1] and Enrique S. Quintana-Ortí[1]

[1]*Departamento de Ingeniería y Ciencia de Computadores, Universitat Jaume I, 12071-Castellón, Spain*
[2]*Innovative Computing Lab (ICL), University of Tennessee, Knoxville, TN, USA*

SUMMARY

In this paper, we analyze the interactions occurring in the triangle performance-power-energy for the execution of a pivotal numerical algorithm, the iterative conjugate gradient (CG) method, on a diverse collection of parallel multithreaded architectures. This analysis is especially timely in a decade where the power wall has arisen as a major obstacle to build faster processors. Moreover, the CG method has recently been proposed as a complement to the LINPACK benchmark, as this iterative method is argued to be more archetypical of the performance of today's scientific and engineering applications. To gain insights about the benefits of hands-on optimizations we include runtime and energy efficiency results for both out-of-the-box usage relying exclusively on compiler optimizations, and implementations manually optimized for target architectures, that range from general-purpose and digital signal multicore processors to manycore graphics processing units, all representative of current multithreaded systems. Copyright © 2014 John Wiley & Sons, Ltd.

Received 30 March 2014; Accepted 5 June 2014

KEY WORDS: energy efficiency; CPUs; low-power architectures; GPUs; CG

1. INTRODUCTION

At a rough cost of $1m USD per megawatt (MW) year, current high performance computing (HPC) facilities and large-scale datacenters are painfully aware of the *power wall*. This is recognized as a crucial hurdle by the HPC community, and many ongoing developments toward the world's first exascale system are shaped by the expected power demand and the related energy costs. For instance, a simple look at the Top500 and Green500 lists [1, 2] reveals that an ExaFLOP computer built using the same technology employed in today's fastest supercomputer, would dissipate more than 520 MW, resulting in an unmistakable call for power-efficient systems and energy proportionality [3–7].

One important follow-up of the end of Dennard scaling [8] (i.e., the ability to drop the voltage and the current that transistors need to operate reliably as their size shrinks) is the surge of dark silicon [9], and consequently the development of specialized processors composed of heterogeneous core architectures with more favorable performance-power ratios. This trend is visible, for example, in the Top500 list, with the current top two systems being equipped with NVIDIA GPUs and Intel Xeon Phi accelerators. Against this background, although most manufacturers advertise the power-efficiency of their products by providing theoretical energy specifications, an equitable

*Correspondence to: José I. Aliaga, Departamento de Ingeniería y Ciencia de Computadores, Universitat Jaume I, 12071-Castellón, Spain.
†E-mail: aliaga@uji.es

comparison between different hardware architectures remains difficult. The reason is not only that distinct devices are often designed for different types of computations, but also that they are tailored for either performance or power efficiency. New energy-related metrics have been recently proposed to analyze the balance between these two key figures [10], but the situation becomes increasingly difficult once the different levels of optimization applied to an algorithm enter the picture.

In response to this situation, this paper analyzes the performance-energy trade-offs of an ample variety of multithreaded general-purpose and specialized architectures, using the CG method as a workhorse. This method is a key algorithm for the numerical solution of symmetric positive definite (s.p.d.) sparse linear systems [11]. Furthermore, because the cornerstone of this iterative method is the sparse matrix-vector product (`SPMV`), an operation that is also crucial for many other numerical methods [12], the significance of the results carries over to many other applications. In addition, the CG method has been recently proposed as a complement to the LINPACK benchmark on the basis of being more representative of the actual performance attained by current scientific and engineering codes that run on HPC platforms [13]. Concretely, the LINPACK benchmark comprises the solution of a (huge) dense system of linear equations via the LU factorization, a compute-bounded operation that is known to deliver a GFLOPS (billions of floating-point arithmetic operations, or flops, per second) rate close to that of the matrix-matrix product and, therefore, the theoretical peak performance of the underlying platform. The CG method, on the other hand, is composed of memory-bound kernels, an attribute shared by a considerably larger number of HPC applications, and offers a much lower GFLOPS throughput than the LINPACK metric.

Our analysis covers two different scenarios. We first review the study in [14], replacing the single-precision (SP) arithmetic with double-precision (DP) experiments, as this is the norm in numerical linear algebra. In [14] we refrained from applying any manual hardware-aware optimizations to the code, but instead evaluated basic SP implementations of the CG method and `SPMV`, using either the baseline CSR (compressed sparse rows) or ELLPACK sparse matrix formats [11], and relying exclusively on the optimizations applied by the compiler. That scenario thus provided insights on the resource efficiency achieved when running complex numerical codes on large HPC facilities without applying hands-on optimization. In this paper we extend the study significantly by considering an alternative scenario which aims to increase resource efficiency by replacing the standard `SPMV` with a more sophisticated implementation, and, in the presence of a hardware graphics accelerator, modifying the basic CG algorithm to reduce the overall kernel count [15]. This allows us to also assess the improvement potential of algorithmic modifications on the distinct hardware architectures.

The rest of the paper is structured as follows. In Section 2 we briefly introduce the mathematical formulation of the CG method as well as the matrix benchmarks and hardware architectures we evaluate, which include four general-purpose multicore processors (Intel Xeon E5504 and E5-2620, and AMD Opteron 6128 and 6276); a low-power multicore digital signal processor (Texas Instruments C6678); three low-power multicore processors (ARM Cortex A9, Exynos5 Octa, and Intel Atom S1260); and three GPUs with different capabilities (NVIDIA Quadro M1000, Tesla C2050, and Kepler K20). Section 3 contains the performance and energy efficiency results obtained from the basic DP implementations of the CG method, where only compiler-intrinsic optimizations are applied. These figures are contrasted next, in Section 4, against the results obtained when applying the hands-on optimizations described previously in that section. We finally conclude the paper in Section 5 with a short summary about the findings and a discussion of ideas for future research.

2. BACKGROUND AND EXPERIMENTAL SETUP

2.1. Krylov-based iterative solvers

The CG method [11] is usually the preferred Krylov subspace-based solver to tackle a linear system of the form

$$Ax = b, \tag{1}$$

DOI: 10.1002/cpe

$x_0 := 0$ // or any other initial guess
$r_0 := b - Ax_0, \quad d_0 := r_0$
$\beta_0 := r_0^T r_0, \qquad \tau_0 :=\| r_0 \|_2= \sqrt{\beta_0}, \quad k := 0$
while $(k < maxiter)$ & $(\tau_k > maxres)$
  $z_k := Ad_k$ (SpMV)
  $\rho_k := \beta_k / d_k^T z_k$ (dot)
  $x_{k+1} := x_k + \rho_k d_k$ (axpy)
  $r_{k+1} := r_k - \rho_k z_k$ (axpy)
  $\beta_{k+1} := r_{k+1}^T r_{k+1}$ (dot)
  $\alpha_k := \beta_{k+1} / \beta_k$
  $d_{k+1} := r_{k+1} + \alpha_k d_k$ (scal+axpy)
  $\tau_{k+1} :=\| r_{k+1} \|_2= \sqrt{\beta_{k+1}}$
  $k := k + 1$
end

Figure 1. Mathematical formulation of the CG method. The names inside parenthesis, except for SPMV which refers to the sparse matrix-vector kernel, identify the routine from the level-1 BLAS that offers the corresponding functionality.

```
1  while( ( k < maxiter ) && ( tau > maxres ) ){
2    SpMV ( n, Asparse, d, z );           // z := A * d
3    tmp = ddot ( n, d, 1, z, 1 );        // tmp := d' * z
4    rho = beta / tmp;                    // rho := beta / tmp
5    gamma = beta;                        // gamma := beta
6    daxpy (n,  rho, d, 1, x, 1 );        // x := x + rho * d
7    daxpy (n, -rho, z, 1, r, 1 );        // r := r - rho * z
8    beta = cublasSdot( n, r, 1, r, 1 );  // beta := r' * r
9    alpha = beta / gamma;                // alpha := beta / gamma
10   dscal (n, alpha, d, 1 );             // d := alpha * d
11   daxpy (n, one, r, 1, d, 1 );         // d := d + r
12   tau = sqrt( beta );                  // tau := sqrt(beta)
13   k++;
14 } // end-while
```

Figure 2. Simplified loop body of the basic CG implementation using double precision.

where $A \in \mathbb{R}^{n \times n}$ is sparse s.p.d., $b \in \mathbb{R}^n$ contains the independent terms, and $x \in \mathbb{R}^n$ is the sought-after solution. The method is mathematically formulated in Figure 1, where the user-defined parameters *maxres* and *maxiter* set upper bounds, respectively, on the 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 nonzero entries, this operation roughly requires $2n_z$ flops. The SPMV is ubiquitous in scientific computing, being a key operation for the iterative solution of linear systems and eigenproblems as well as the PageRank algorithm, among others [11, 12, 16]. The irregular memory access pattern of this operation, in combination with the limited memory bandwidth of current general-purpose architectures, has resulted in a considerable number of efforts that propose specialized matrix storage layouts as well as optimized implementations for a variety of architectures; see, for example, [17–19] and the references therein. Although we target only symmetric systems with our CG implementations for which storage formats and matrix-vector routines that efficiently exploit this property exist [11], we refrain from using them. We argue, that by not considering this optimization, our analysis and results can be easily extrapolated to the case of unsymmetric sparse linear system solvers.

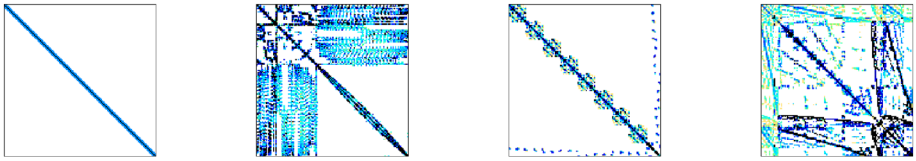
In addition to the matrix-vector multiplication, the loop body of the CG method contains several vector operations, with a cost of $O(n)$ flops each, for the updates of x_{k+1}, r_{k+1}, d_{k+1}, and the computation of ρ_k and β_{k+1}. These operations are supported as part of the level-1 BLAS [20]; see Figure 2.

DOI: 10.1002/cpe

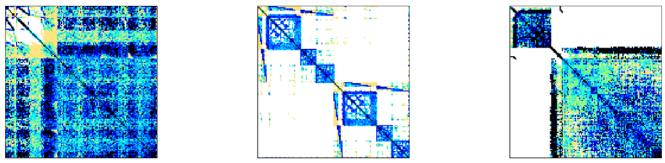
Table I. Description and properties of the test matrices (top) and the corresponding sparsity plots (bottom).

Source	Acronym	Matrix	#nonzeros (n_z)	Size (n)	n_z/n
Laplace	A159	A159	27,986,067	4,019,679	6.96
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

(a) A159 (b) AUDIKW_1 (c) BMWCRA_1 (d) CRANKSEG_2



(e) F1 (f) INLINE_1 (g) LDOOR



UFMC, University of Florida Matrix Collection.

2.2. *Matrix benchmarks*

For our experiments, we selected six s.p.d. matrices from the University of Florida Matrix Collection[‡], corresponding to finite element discretizations of several structural problems arising in mechanics, and an additional case derived from a finite difference discretization of the 3D Laplace problem; see Table I. For these linear systems, the right-hand side vector b was initialized to be consistent with the solution $x \equiv 1$, while the CG iteration was started with the initial guess $x_0 \equiv 0$. Except where stated otherwise, all tests were conducted with DP arithmetic. (We note that the experimentation in [14] was performed using SP arithmetic only.)

2.3. *Hardware setup and compilers*

Table II lists the main features of the hardware systems and compilers utilized in the experiments. For each multicore processor we also report the different processor frequencies that were evaluated. Aggressive optimization was applied to all implementations through flag `-O3`. In order to measure power, we leveraged a WATTSUP?PRO wattmeter, connected to the line from the electrical socket to the power supply unit, with an accuracy of $\pm 1.5\%$ and a rate of 1 sample/sec. These results were collected on a separate server to avoid interfering with the application performance and consumption. For all test matrices, we based the analysis on the average power draft when executing 1000 CG iterations after a warm up period of 5 minutes using the same test. Because the platforms where the processors are embedded contain other devices—for example, disks, network interface cards, and fans—on each platform we measured the average idle power and then subtracted the corresponding value (Table II) from all the samples obtained from the wattmeter. We believe this setup enables a relevant comparison between the energy efficiencies of the different architectures, as proceeding

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

DOI: 10.1002/cpe

Table II. Multithreaded architectures. For the GPU systems (FER, KEP, and QDR), the idle power includes the consumption of both the server and the accelerator.

Acron	Architecture	Total #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	64 GB, DDR3 1.3 GHz	`icc` 12.1.3
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	`icc` 12.1.3
IAT	Intel Atom S1260	2	0.6–41.94, 0.90–41.93, 1.30–41.97, 1.70–41.95 2.0–42.01	8 GB, DDR3 1.3 GHz	`icc` 12.1.3
INH	Intel Xeon E5504 (Nehalem)	8	1.60–33.43, 1,73–33.43, 1.87–33.43, 2.00–33.43	32 GB, DDR3 800 MHz	`icc` 12.1.3
ISB	Intel E5-2620 (Sandy-Bridge)	6	1.2–113.00, 1.4–112.96, 1.6–112.77, 1.8–112.87, 2.0–112.85	32 GB, DDR3 1.3 GHz	`icc` 12.1.3
A9	ARM Cortex A9	4	0.76–10.0, 1.3–10.1	2 GB, DDR3L	`gcc` 4.6.3
A15	Exynos5 Octa (ARM Cortex A15 + A7)	4 + 4	0.25–2.2, 1.6–2.4	2 GB, LPDDR3	`gcc` 4.7
FER	Intel Xeon E5520 NVIDIA Tesla C2050 (Fermi)	8 448	1.6–222.0, 2.27–226.0 1.15	24 GB, 3 GB, GDDR5	`gcc` 4.4.6 `nvcc` 5.5
KEP	Intel Xeon i7-3930K NVIDIA Tesla K20 (Kepler)	6 2496	1.2–106.30, 3.2–106.50 0.7	24 GB, 5 GB, GDDR5	`gcc` 4.4.6 `nvcc` 5.5
QDR	ARM Cortex A9 NVIDIA Quadro 1000M	4 96	0.120–11.2, 1.3–12.2 1.4	2 GB, DDR3L 2 GB, DDR3	`gcc` 4.6.3 `nvcc` 5.5
TIC	Texas Instruments C6678	8	1.0–18.0	512 MB, DDR3	`cl6x` 7.4.1

in this manner we only take into account the (net) energy that is drawn for the actual work, but eliminate power sinks such as the inefficiencies of the power supply unit, the network interface, or the disk.

3. BASIC CG METHOD RELYING ON COMPILER OPTIMIZATION

In this section, we analyze a straight-forward implementation of the CG method, with the only optimizations being applied automatically by the compiler, when configuring the hardware for either performance or energy efficiency through adjusting the processor operation frequency and the number of active cores. The reference code for all of these baseline implementations is given in Figure 2.

3.1. Basic implementation of the CG method

In our basic implementations of the `SPMV` for the CG method, matrix A is stored in the CSR format for multicore architectures, or the ELLPACK format for the GPUs. Both schemes aim to reduce the memory footprint by explicitly storing only the nonzero elements, though the ELLPACK format may store some zero elements for padding all rows to the same length; see Figure 3 and Table III

DOI: 10.1002/cpe

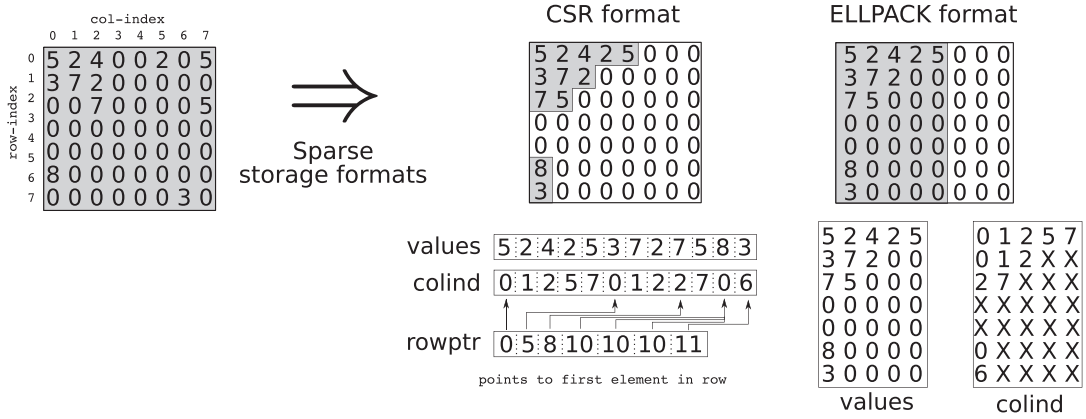


Figure 3. Basic dense and sparse matrix storage formats. The memory demand corresponds to the gray areas.

Table III. Storage overhead of the test matrices when using ELLPACK or SELL-P format.

					ELLPACK		SELL-P	
Matrix	#nonzeros (n_z)	Size (n)	n_z/n	n_z^{row}	$n_z^{ELLPACK}$	Overhead (%)	n_z^{SELL-P}	Overhead (%)
A159	27,986,067	4,019,679	6.96	7	28,137,753	0.54	32,157,440	12.97
AUDI	77,651,847	943,645	82.28	345	325,574,775	76.15	95,556,416	18.74
BMW	10,641,602	148,770	71.53	351	52,218,270	79.62	12,232,960	13.01
CRANK	14,148,858	63,838	221.63	3423	218,517,474	93.53	15,991,232	11.52
F1	26,837,113	343,791	78.06	435	149,549,085	82.05	33,286,592	19.38
INLINE	38,816,170	503,712	77.06	843	424,629,216	91.33	45,603,264	19.27
LDOOR	42,493,817	952,203	44.62	77	73,319,631	42.04	52,696,384	19.36

n_z^{FORMAT} refers to the explicitly stored elements (n_z nonzero elements and the explicitly stored zeros for padding).

for the evoked overhead. While in general this incurs some additional storage cost, the aligned structure allows for more efficient hardware use when targeting streaming processors such as the GPUs [21, 22].

In Figure 4, we sketch the `SPMV` kernels for the CSR and ELLPACK formats. In both routines, `n` refers to the matrix size; the matrix is stowed using arrays `values`, `colind` and, in the case of routine `SpMV_csr`, also array `rowptr` (Figure 3); the input and output vectors of the product $y := Ax$ are, respectively, `x` and `y`; finally, `nzr` refers to the number of entries per row in the ELLPACK-based routine.

In the basic implementation for multicore processors (`SpMV_csr`), concurrency is exploited via the OpenMP application programming interface, with the matrix-vector operation partitioned by rows, and each OpenMP thread being responsible for all the arithmetic operations necessary to update a certain number of entries of `y`. In these architectures, we employed the legacy implementation of BLAS from *netlib*[§] for the level-1 (vector) operations (kernels `sdot`, `daxpy`, `scal` in Figure 2). No attempt was made to extract parallelism from these BLAS kernels as, because of their low computational cost, a parallel execution on a conventional multithreaded architecture rarely offers any benefits.

For the GPUs, concurrency is exploited in `SpMV_ell` by rows, with one CUDA thread being responsible for the computation of one element of `y`. On these streaming architectures we used the multithreaded implementation in NVIDIA's CUBLAS, replacing the invocations to `ddot`, `daxpy`, and `dscal` in Figure 2 with calls to `cublasDdot`, `cublasDaxpy`, and `cublasDcal`,

[§]http://www.netlib.org.

DOI: 10.1002/cpe

```
1   void SpMV_csr( int n,
2                  int * rowptr, int * colind, double * values,
3                  double * x, double * y ) {
4     int i, j;    double tmp;
5
6     #pragma omp parallel for private ( j, tmp )
7     for ( i = 0; i < n; i++ ) {
8       tmp = 0.0;
9       for ( j = rowptr [ i ]; j < rowptr [ i+1 ]; j++ )
10        tmp +=  values [ j ] * x [ colind[ j ] ];
11      y[i] = tmp;
12    }
13  }
```

```
1   __global__
2   void SpMV_ell( int n, int nzr,
3                  int * colind, double * values,
4                  double * x,   double * y ) {
5     int i, j, k; double tmp;
6
7     i = blockDim.x * blockIdx.x + threadIdx.x;
8     if ( i < n ){
9       tmp = 0.0;
10      for ( j = 0; j < nzr; j++ ) {
11        k = n * j + i;
12        if ( values[ k ] != 0 )
13          tmp += values [ k ] * x [ colind [ k ] ];
14      }
15      y[i] = tmp;
16    }
17  }
```

Figure 4. Sparse matrix-vector product using the compressed sparse rows and ELLPACK formats (`SpMV_csr` and `SpMV_ell`, respectively).

respectively. (For the resulting code, see also the left-hand side of Figure 7.) We consider this a fair comparison as in the following: (1) In general, the time cost of the vector operations is significantly lower than that of the `SPMV`; (2) Because of their reduced cost, there is little opportunity to benefit from a concurrent execution of the vector operations on a multicore processor.

3.2. Search for the optimal configuration

Tables IV and V and Figure 5 report the performance and net energy (i.e., energy after subtracting the cost of idle power) per iteration, for the different platforms and benchmark cases, obtained with the basic DP implementations. (For an analogous study with SP arithmetic, refer to [14].) For each architecture and matrix, we report the combination of core number and frequency that yields the shortest execution time per iteration, together with the corresponding net energy ('optimized w.r.t. time'), as well as the configuration that provides the most energy-efficient result ('optimized w.r.t. energy'). As the tables and figure convey a considerable amount of information, we limit the following analysis to some central aspects.

3.2.1. Optimization with respect to time. If aiming for performance using the basic CG implementation, it is not possible to identify an overall winner, as the performance turns out to be highly problem dependent (see the top-left graph in Figure 5). For example, the Tesla K20 (KEP) achieves almost 12 GFLOPS for the A159 matrix, a number that is most closely followed by the Tesla C2050 GPU's 7.5 GFLOPS (FER), and about nine times faster than the best of all the CPUs for that particular case (1.3 GFLOPS attained with the Intel E5-2620, ISB). On the other hand, for the more involved sparsity structure of the CRANK problem, KEP only delivers 1.2 GFLOPS, while the CPU implementations, using the less problem-dependent CSR format instead of ELLPACK, achieve between 2 and 4 GFLOPS on recent hardware (AIL and ISB, respectively). A closer inspection reveals that the GPU's performance is directly related to the overhead induced by the usage of the ELLPACK format (Table III). Also, the performance of the older CPU generations (AMC and INH) is less problem-dependent, whereas the other two GPU architectures are similarly sensitive to the matrix structure.

DOI: 10.1002/cpe

Table IV. Optimal hardware parameter configuration when optimizing the general-purpose architectures for runtime or energy efficiency using the basic implementations.

		Optimized w.r.t. time				Optimized w.r.t. net energy			
	Matrix	c	f	T	E_{net}	c	f	T	E_{net}
AIL	A159	8	2300	1.00E − 01	1.72E + 01	4	2100	1.14E − 01	1.53E + 01
	AUDI	8	2300	1.07E − 01	1.70E + 01	8	2300	1.07E − 01	1.70E + 01
	BMW	8	2300	1.13E − 02	1.89E + 00	8	2100	1.15E − 02	1.82E + 00
	CRANK	8	2300	1.42E − 02	2.46E + 00	8	2100	1.46E − 02	2.34E + 00
	F1	8	2300	3.82E − 02	6.15E + 00	8	2300	3.82E − 02	6.15E + 00
	INLINE	8	2300	4.48E − 02	7.77E + 00	4	2100	5.24E − 02	7.26E + 00
	LDOOR	8	2300	6.17E − 02	1.09E + 01	8	2100	6.25E − 02	9.76E + 00
AMC	A159	8	2000	2.11E − 01	1.94E + 01	8	2000	2.11E − 01	1.94E + 01
	AUDI	8	2000	2.12E − 01	2.06E + 01	8	2000	2.12E − 01	2.06E + 01
	BMW	8	2000	1.83E − 02	2.01E + 00	8	2000	1.83E − 02	2.01E + 00
	CRANK	8	2000	2.57E − 02	2.82E + 00	8	2000	2.57E − 02	2.82E + 00
	F1	8	2000	7.40E − 02	7.59E + 00	8	2000	7.40E − 02	7.59E + 00
	INLINE	8	2000	8.15E − 02	8.55E + 00	8	2000	8.15E − 02	8.55E + 00
	LDOOR	8	2000	1.02E − 01	1.08E + 01	8	2000	1.02E − 01	1.08E + 01
IAT	A159	2	2000	3.01E − 01	1.27E + 00	1	600	1.25E + 00	8.53E − 01
	AUDI	2	2000	3.79E − 01	1.48E + 00	1	600	1.84E + 00	9.60E − 01
	BMW	2	2000	4.35E − 02	1.95E − 01	1	600	2.43E − 01	1.37E − 01
	CRANK	2	2000	5.82E − 02	2.43E − 01	1	600	3.07E − 01	1.43E − 01
	F1	2	2000	1.78E − 01	6.12E − 01	1	600	8.15E − 01	3.94E − 01
	INLINE	2	2000	1.74E − 01	7.23E − 01	1	600	9.09E − 01	4.24E − 01
	LDOOR	2	2000	2.38E − 01	9.61E − 01	1	600	1.19E + 00	6.60E − 01
INH	A159	4	1870	1.01E − 01	1.01E + 01	2	1870	1.03E − 01	9.30E + 00
	AUDI	8	2000	1.00E − 01	1.35E + 01	4	1730	1.34E − 01	1.28E + 01
	BMW	4	1870	1.29E − 02	1.37E + 00	4	1600	1.33E − 02	1.29E + 00
	CRANK	4	1870	1.65E − 02	1.75E + 00	4	1600	1.72E − 02	1.64E + 00
	F1	8	1730	3.81E − 02	4.97E + 00	4	1600	5.01E − 02	4.72E + 00
	INLINE	8	1600	5.00E − 02	5.97E + 00	4	1600	5.27E − 02	5.04E + 00
	LDOOR	8	1870	6.84E − 02	9.66E + 00	4	1600	7.12E − 02	6.91E + 00
ISB	A159	6	2000	5.56E − 02	2.78E + 00	2	1200	1.08E − 01	1.86E + 00
	AUDI	6	2000	6.12E − 02	3.10E + 00	2	1400	1.30E − 01	2.46E + 00
	BMW	6	2000	5.51E − 03	3.24E − 01	6	1200	8.17E − 03	2.63E − 01
	CRANK	6	2000	7.22E − 03	4.56E − 01	6	1200	1.11E − 02	3.45E − 01
	F1	6	2000	1.98E − 02	1.04E + 00	6	1200	2.92E − 02	8.51E − 01
	INLINE	6	2000	2.31E − 02	1.27E + 00	4	1400	3.53E − 02	1.05E + 00
	LDOOR	6	2000	3.06E − 02	1.75E + 00	6	1200	4.54E − 02	1.45E + 00

In the labels, c denotes the number of cores, f the frequency (in MHz), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules).

Finally, while the low-power architectures—IAT, A9, A15, and TIC—provide the lowest GFLOPS rate for most problems, in some cases the usage of the ELLPACK format on the GPUs may incur significant overhead, which their computing power cannot compensate for, such that even the low-power processors may become competitive in those cases (compare, e.g., the performance of TIC and QDR for the BMW case).

The performance sensitivity of the architectures to the problem characteristics and the format-dependent overhead also translate into the corresponding energy efficiency, so that the GPUs are only superior to the CPUs for the structured problems A159 and LDOOR (see Figure 5 right-top). While the C66780 from Texas Instruments (TIC) also 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 remaining low-power processors, A9, A15, and IAT, achieve higher GFLOPS/W rates than the conventional general-purpose processors AIL, AMC, and

DOI: 10.1002/cpe

Table V. Optimal hardware parameter configuration when optimizing the specialized architectures for runtime or energy efficiency using the basic implementations.

		Optimized w.r.t time				Optimized w.r.t net energy			
	Matrix	c	f	T	E_{net}	c	f	T	E_{net}
A9	A159	4	760	7.15E − 01	1.04E + 00	4	760	7.15E − 01	1.04E + 00
	AUDI	4	1300	9.06E − 01	2.89E + 00	2	760	1.30E + 00	1.76E + 00
	BMW	4	1300	1.45E − 01	5.35E − 01	2	760	1.83E − 01	2.74E − 01
	CRANK	4	1300	1.68E − 01	5.15E − 01	2	760	2.26E − 01	3.05E − 01
	F1	4	1300	3.57E − 01	1.22E + 00	2	760	5.03E − 01	6.74E − 01
	INLINE	4	1300	4.88E − 01	1.72E + 00	2	760	6.59E − 01	9.88E − 01
	LDOOR	4	1300	6.16E − 01	2.10E + 00	2	760	8.45E − 01	1.17E + 00
A15	A159	4	1600	1.70E − 01	1.12E + 00	4	250	8.08E − 01	2.02E − 01
	AUDI	4	1600	2.11E − 01	1.64E + 00	4	250	7.87E − 01	2.68E − 01
	BMW	4	1600	3.12E − 02	2.76E − 01	4	250	1.10E − 01	4.62E − 02
	CRANK	4	1600	3.61E − 02	3.17E − 01	4	250	1.34E − 01	5.88E − 02
	F1	4	1600	8.44E − 02	6.65E − 01	4	250	3.11E − 01	1.15E − 01
	INLINE	4	1600	1.11E − 01	9.50E − 01	4	250	3.89E − 01	1.75E − 01
	LDOOR	4	1600	1.44E − 01	1.23E + 00	4	250	5.21E − 01	1.98E − 01
FER	A159	1	1600	1.01E − 02	1.77E + 00	1	1600	1.01E − 02	1.77E + 00
	BMW	1	1600	1.02E − 02	1.69E + 00	1	1600	1.02E − 02	1.69E + 00
	CRANK	1	1600	4.58E − 02	7.20E + 00	1	1600	4.58E−02	7.20E+00
	F1	1	1600	3.36E − 02	5.56E + 00	1	1600	3.36E − 02	5.56E + 00
	LDOOR	1	1600	1.41E − 02	2.58E + 00	1	1600	1.41E − 02	2.58E + 00
KEP	A159	1	1200	6.80E − 03	6.73E − 01	1	1200	6.80E − 03	6.73E − 01
	AUDI	1	1200	3.53E − 02	3.51E + 00	1	1200	3.53E − 02	3.51E + 00
	BMW	1	1200	5.57E − 03	5.37E − 01	1	1200	5.57E − 03	5.37E − 01
	CRANK	1	1200	2.26E − 02	2.07E + 00	1	1200	2.26E − 02	2.07E + 00
	F1	1	1200	1.68E − 02	1.64E + 00	1	1200	1.68E − 02	1.64E + 00
	LDOOR	1	1200	8.57E − 03	9.12E − 01	1	1200	8.57E − 03	9.12E − 01
QDR	A159	1	1300	4.54E − 02	1.15E + 00	1	51	6.06E − 02	8.15E − 01
	BMW	1	1300	4.56E − 02	1.07E + 00	1	51	7.36E − 02	9.78E − 01
TIC	BMW	8	1000	3.06E − 02	8.83E − 02	4	1000	4.37E − 02	1.098E − 01
	CRANK	8	1000	4.18E − 02	1.10E − 01	8	1000	4.18E − 02	1.10E − 01
	F1	8	1000	1.27E − 01	2.69E − 01	8	1000	1.27E − 01	2.69E − 01

In the labels, c denotes the number of cores, f the frequency (in MHz), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules).

INH, but are only more efficient than GPUs for certain matrix cases. With respect to energy, ISB is at least competitive with the older ARM Cortex A9.

3.2.2. Optimization with respect to net energy. The first general observation is that, in many cases, reducing the CPU operating frequency (and voltage) pays off. The FER and KEP GPU-accelerated platforms are two notable exceptions, as rescaling the frequency operation of the host CPU of these systems has negligible impact on the overall performance and energy efficiency of the solver. The behavior is very different for the low-power processors and QDR, where rescaling the CPU frequency can improve the energy efficiency by a wide margin. For instance, reducing the operating frequency of the recent ARM Cortex (A15) from 1600 to 250 MHz improves the efficiency ratio by a factor of 5.5 (see the results for A159 in Table V: from 1.12 to 2.02E − 01 GFLOPS/W; also compare the top and bottom plots on the right of Figure 5). The results for IAT and A9 (Tables IV and V) also show significant improvements, which reveal that these architectures provide not only a more favorable baseline energy efficiency, but also higher optimization potential compared with the conventional general-purpose CPUs or GPUs. While the TIC provided the highest performance/watt

DOI: 10.1002/cpe

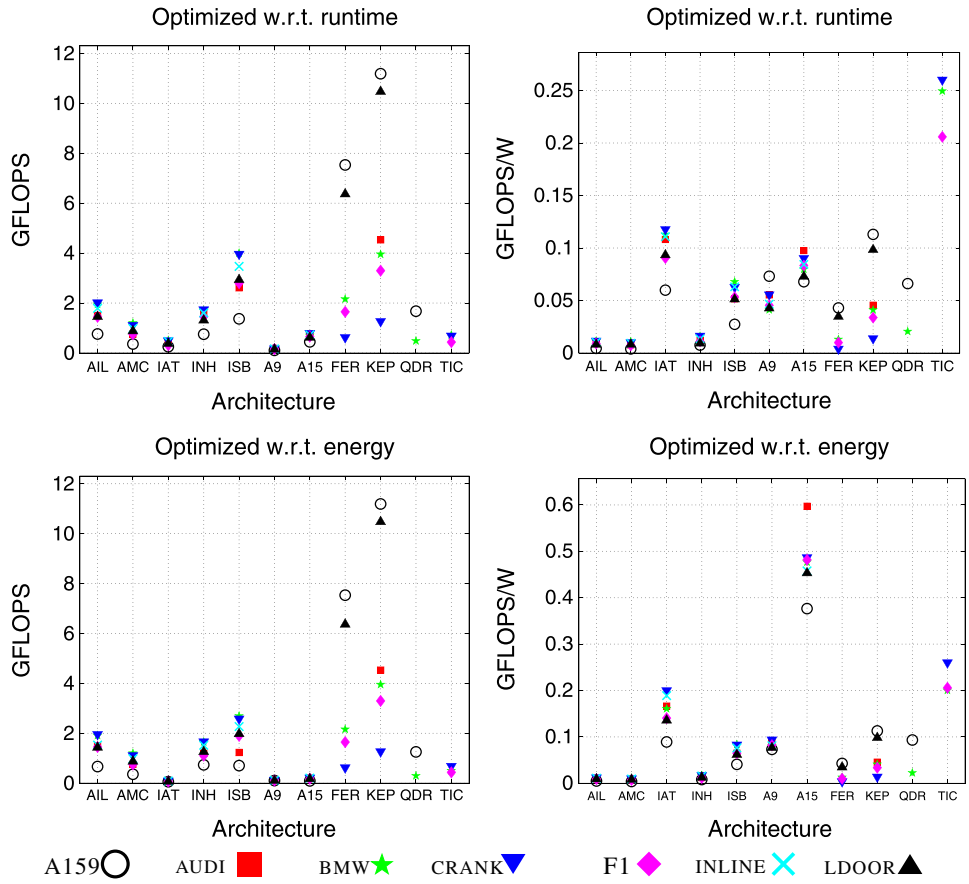


Figure 5. Comparison of performance (left) and energy efficiency (right), measured, respectively, in terms of GFLOPS and GFLOPS/W, when optimizing the basic CG implementations with respect to run time (top) or net energy (bottom).

ratio when optimizing for performance, A15 now becomes the overall winner, followed by TIC, IAT, and the older generation Cortex (A9). As the energy efficiency of the graphics accelerators KEP, FER, and QDR is again very problem-dependent, it is difficult to compare them against ISB, but they still deliver higher GFLOPS/W ratios than the older CPU architectures AIL, AMC, and INH. Unfortunately, the factors gained when improving energy efficiency translate into the related execution time, as the reduced Joule-per-iteration values for TIC, A9, A15, and IAT come at the price of significantly higher execution times. However, as these provide the lowest performance in any case, and the GPUs do not allow for too much hardware reconfiguration, the left-top and left-bottom graphs in Figure 5 look very similar. Optimizing ISB for either performance or energy efficiency renders improvement factors around 2, significantly higher than those observed for other CPU architectures.

Finally, we mention that optimizing with respect to net energy is not necessarily equivalent to optimizing for the total energy consumption. It may happen that the net energy consumption is reduced by decreasing the CPU frequency, at the cost of an increase of the total energy, as we did not consider the system's baseline power draft (idle power) in this analysis.

4. HARDWARE-AWARE OPTIMIZATION OF THE CG METHOD

While the results in the previous section offer insights on the computational and energy efficiencies of the target hardware architectures when running 'unoptimized' code, we now address the question of how much improvement can be achieved by tailoring the codes to specific hardware. For this purpose we modify the basic implementations of the CG solver in two ways:

DOI: 10.1002/cpe

1. We replace the basic sparse matrix formats and the associated baseline matrix-vector product codes (Figures 3 and 4) with more sophisticated solutions which can be expected to render higher performance, depending on the target hardware and the matrix benchmark.
2. On the streaming processors, the performance of memory-bound algorithms suffers from the read/write operations from/to global memory that are required when launching a kernel. For this reason we leverage a variant of the CG algorithm where the `SPMV` and vector operations are reorganized and merged into several kernels [15], reducing the volume of memory access and the kernel launch overhead.

In the remainder of this section we expose the modifications introduced in the CG method, in the same order as enumerated earlier, and experimentally evaluate the effect they exert on the performance and energy efficiency of the solver.

4.1. Optimizing the matrix layout for `SPMV`

4.1.1. Multicore processors. CSR has become the *de facto* standard format for sparse matrices thanks to its flexibility and low memory requirements [23]. In particular, in exchange for keeping only the nonzero entries of the matrix, CSR introduces a storage overhead of only $n + n_z$ integer entries to represent the indices or pointers that determine the positions of these elements.

Block CSR (BCSR) is a blocked variant of CSR that aims to improve register reuse for `SPMV` [24], by dividing the matrix into small dense blocks that are kept in consecutive locations in memory, as a sparse collection of dense blocks; see Figure 6. While BCSR introduces some storage overhead by padding the dense blocks with zeros, this is potentially compensated for by having to maintain a single index per block only, and by the performance advantage of improved register reuse that serves as motivation for the format. A detailed description of BCSR and a basic implementation of the `SPMV` operation can be found in [24]. In our tailored version of the CG method, we leveraged two multithreaded routines of the `SPMV` routine in Intel MKL (version 10.3 update 9), based on the CSR and BCSR formats: `mkl_cspblas_dcsrgemv` and `mkl_cspblas_dbsrgemv`, respectively [25]. For the latter code, our tests considered values for the block size (parameter `lb`) equal to 2 and 3, but we only report the best result.

The alternative compressed sparse blocks (CSB) format [26] maintains the matrix as a dense collection of sparse blocks, with the blocks themselves being stored in Z-morton order [27]; see Figure 6 for a sketch of this layout. This format exhibits negligible storage overhead compared with CSR, and, more importantly, allows for an efficient parallelization of `SPMV` on current multicore architectures. We employ the multithreaded routine `bicsb_gespmv` from the CSB library [28] for

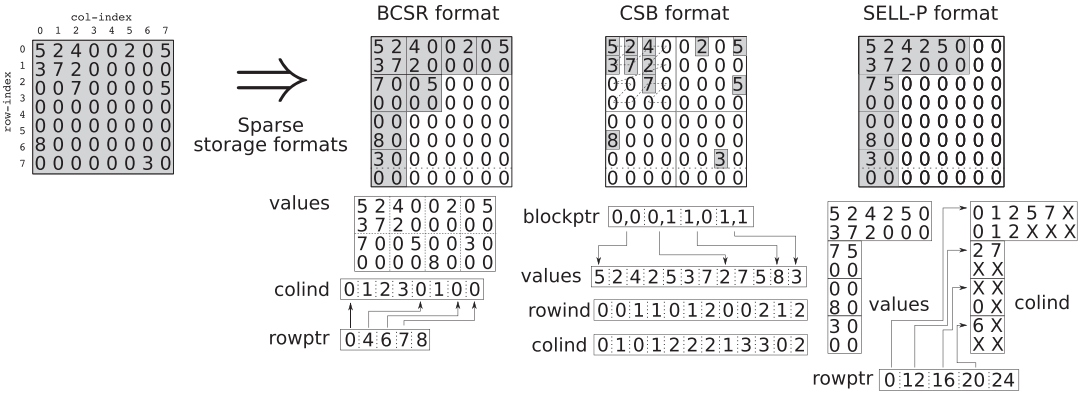


Figure 6. Visualizing the block compressed sparse rows (BCSR), compressed sparse blocks (CSB), and SELL-P formats. Note that choosing the block size 2 for BCSR and SELL-P, as well as the block size 4 for CSB, requires adding a zero row to the original matrix. Furthermore, padding the SELL-P format to a row length divisible by 2 requires explicit storage of a few additional zeros.

DOI: 10.1002/cpe

the SPMV routine, linked to release 1.1 of Intel Cilk Plus and release 4.0 of Intel Thread Building Blocks.

The optimized implementations of the CG method for the multicore platforms are analogous to the code in Figure 2, with the SPMV routine replaced by the appropriate implementation of the matrix-vector product (either one of the MKL routines or the CSB library code), and linked in all cases with the MKL implementation of BLAS [25] for routines ddot, daxpy, and dscal.

4.1.2. Graphics accelerators. The ELLPACK format incurs a storage overhead, for the general case, which grows with the differences in the number of nonzero elements per row (Table III). In those cases, the associated memory and computational overheads may result in poor performance, despite that coalesced memory access is highly beneficial for streaming processors such as the GPUs. ELLR-T [22, 29] is a subtle variant of ELLPACK that addresses this problem while maintaining the coalesced memory access pattern of the original layout. In particular, it reduces useless computations with zeros and improves thread load balancing, often resulting in superior performance.

An alternative workaround to reduce memory and computational overhead is to split the original matrix into row blocks before converting these into the ELLPACK format. In the resulting sliced ELLPACK format (SELL or SELL-C where C denotes the size of the row blocks [30, 31]), the overhead is no longer determined by the matrix row containing the largest number of nonzeros, but by the row with the largest number of nonzero elements in the respective block.

In [30], the SELL-C layout is enhanced with an *a-priori* sorting step, which aims to gather rows with a similar number of nonzeros into the same blocks. While this may improve the performance of SPMV, the impact on complex algorithms is still an open research topic. Another optimization, specific to streaming processors such as GPUs, enforces a memory alignment within the SELL-P format that allows for applying the sophisticated matrix-vector kernel proposed in [32]. Although the padding that comes along with this format introduces some zero fill-in, a comparison between the SELL-P format visualized in Figure 6 and the plain ELLPACK in Figure 3 reveals that the blocking strategy may still render significant memory savings (see also Table III), which directly translate into reduced computational cost and improved runtime performance. Previous experiments with this SELL-P format have shown high performance without impacting the algorithm's stability by applying row-sorting, so we will include it as an option for the GPU implementations. In particular, we apply a default configuration for SELL-P with the blocksize 8 and the row length padded to a multiple of 8.

Compared with the baseline (unoptimized) implementation of the CG solver, the optimized GPU codes consider the simple ELLPACK kernel for SPMV as well, but also include implementations for this operation based on the alternative formats ELLR-T and SELL-P. Furthermore, the optimized implementations do not invoke routines from CUBLAS, but instead reorganize and merge the vector operations in CG as described next.

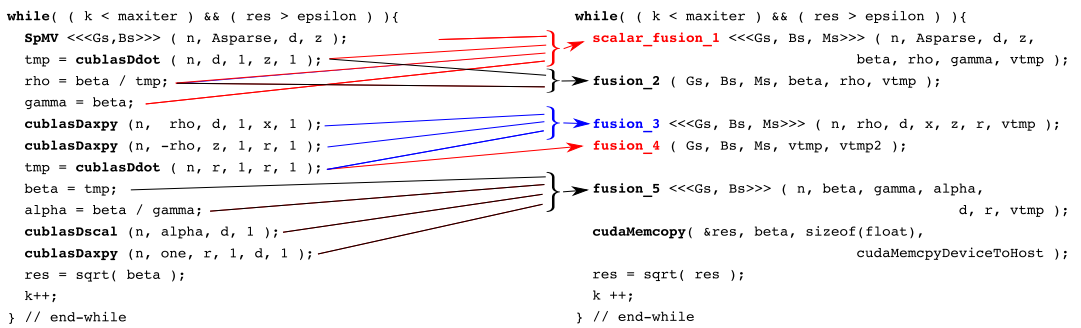


Figure 7. Aggregation of kernels to transform the basic implementation into a merged implementation. Here, the invocation to the sparse matrix-vector kernel SPMV has to be replaced by the appropriate graphics processing unit routine, depending on the sparse matrix format.

DOI: 10.1002/cpe

4.2. Reformulating the CG algorithm for graphics processing units

The performance of memory-bound algorithms, such as the CG solver, is determined by the volume and pattern of the memory accesses. On streaming processors, the number reads/writes to global memory usually correlates to the number of kernels launched over the runtime. While this is motivation enough to reduce the number of kernel calls, additional performance benefits may come through reduced CPU/GPU communication and a lower number of kernel launches. Here, we follow the ideas presented in [15] to adopt merged versions of the basic CG method. The resulting algorithm is illustrated in Figure 7, where `SPMV` denotes a generic implementation of the sparse matrix-vector kernel, which is replaced in the actual codes with one of the optimized cases described in the previous subsection.

Table VI. Optimal hardware parameter configuration when optimizing the general-purpose architectures for runtime or energy efficiency using the tuned implementations.

		Optimized w.r.t time					Optimized w.r.t net energy				
	Matrix	c	f	s	T	E_{net}	c	f	s	T	E_{net}
AIL	A159	8	2300	2	8.89E − 02	1.48E + 01	8	2100	2	9.46E − 02	1.27E + 01
	AUDI	8	2300	1(3)	7.90E − 02	1.44E + 01	8	2100	1(3)	8.08E − 02	1.26E + 01
	BMW	8	2300	1(3)	8.46E − 03	1.50E + 00	8	2100	1(3)	8.58E − 03	1.34E + 00
	CRANK	8	2300	2	1.29E − 02	2.05E + 00	8	2300	2	1.29E − 02	2.05E + 00
	F1	8	2300	1(3)	2.76E − 02	4.46E + 00	8	2100	1(3)	2.84E − 02	4.42E + 00
	INLINE	8	2300	1(3)	3.35E − 02	5.73E + 00	8	2100	1(3)	3.43E − 02	5.37E + 00
	LDOOR	8	2300	1(2)	5.40E − 02	8.76E + 00	8	2100	1(2)	5.49E − 02	8.63E + 00
AMC	A159	8	1500	2	1.69E − 01	1.65E + 01	8	1500	2	1.69E − 01	1.65E + 01
	AUDI	8	2000	1(3)	1.67E − 01	1.63E + 01	8	2000	1(3)	1.67E − 01	1.63E + 01
	BMW	8	2000	1(3)	1.44E − 02	1.52E + 00	8	2000	1(3)	1.44E − 02	1.52E + 00
	CRANK	8	2000	0	2.35E − 02	2.56E + 00	8	2000	0	2.35E − 02	2.56E + 00
	F1	8	2000	1(3)	5.59E − 02	5.58E + 00	8	2000	1(3)	5.59E − 02	5.58E + 00
	INLINE	8	2000	1(3)	6.48E − 02	6.61E + 00	8	2000	1(3)	6.48E − 02	6.61E + 00
	LDOOR	8	2000	1(2)	9.59E − 02	9.92E + 00	8	2000	1(2)	9.59E − 02	9.92E + 00
IAT	A159	2	2000	0	2.88E − 01	1.33E + 00	1	600	2	1.56E + 00	7.71E − 01
	AUDI	2	2000	0	3.20E − 01	1.41E + 00	1	600	1(3)	1.64E + 00	7.65E − 01
	BMW	2	2000	0	3.65E − 02	1.83E − 01	1	600	1(3)	2.16E − 01	1.14E − 01
	CRANK	2	2000	1(2)	4.70E − 02	2.16E − 01	1	600	2	5.40E − 01	1.02E − 01
	F1	2	2000	1(3)	1.55E − 01	5.27E − 01	1	600	1(3)	7.17E − 01	3.28E − 01
	INLINE	2	2000	0	1.50E − 01	6.90E − 01	1	600	2	1.48E + 00	3.73E − 01
	LDOOR	2	2000	1(2)	1.92E − 01	8.54E − 01	1	600	1(2)	8.93E − 01	6.18E − 01
INH	A159	8	1870	0	7.04E − 02	1.03E + 01	4	1600	0	8.71E − 02	8.38E + 00
	AUDI	8	2000	1(3)	6.86E − 02	9.82E + 00	8	1600	1(3)	6.91E − 02	8.57E + 00
	BMW	4	2000	1(3)	8.88E − 03	9.48E − 01	4	1600	1(3)	9.14E − 03	8.88E − 01
	CRANK	4	1870	1(2)	1.53E − 02	1.63E + 00	4	1600	1(2)	1.59E − 02	1.54E + 00
	F1	8	1870	1(3)	2.53E − 02	3.62E + 00	8	1600	1(3)	2.57E − 02	3.30E + 00
	INLINE	8	2000	1(3)	3.05E − 02	4.35E + 00	4	1600	1(3)	3.72E − 02	3.61E + 00
	LDOOR	8	1870	1(2)	5.46E − 02	7.74E + 00	4	1600	0	6.56E − 02	6.37E + 00
ISB	A159	6	2000	0	3.65E − 02	2.15E + 00	4	1200	0	5.69E − 02	1.68E + 00
	AUDI	6	2000	1(3)	4.04E − 02	2.08E + 00	2	1200	1(3)	9.65E − 02	1.62E + 00
	BMW	6	2000	1(3)	3.57E − 03	2.13E − 01	6	1200	1(3)	5.15E − 03	1.71E − 01
	CRANK	6	2000	0	6.45E − 03	4.23E − 01	6	1200	1(2)	9.67E − 03	3.11E − 01
	F1	6	2000	1(3)	1.35E − 02	6.92E − 01	6	1200	1(3)	1.97E − 02	5.68E − 01
	INLINE	6	2000	1(3)	1.54E − 02	8.51E − 01	4	1200	1(3)	2.58E − 02	6.96E − 01
	LDOOR	6	2000	1(2)	2.38E − 02	1.40E + 00	6	1200	1(2)	3.40E − 02	1.16E + 00

In the labels, c denotes the number of cores, f the frequency (in MHz), s the sparse matrix layout/`SPMV` implementation (0 for CSR + MKL, 1(2) for BCSR + MKL with block size `lb` = 2, 1(3) for BCSR + MKL with block size `lb` = 3, and 2 for CSB), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules).

DOI: 10.1002/cpe

All the tuned implementations of CG that we evaluate next are part of a pre-release extension of the MAGMA library [33] composed of iterative algorithms for sparse linear algebra. These codes block the CPU threads while the computation is proceeding in the GPU, so as to avoid a power-hungry polling during the execution. This is easily achieved by setting parameter `cudaDeviceBlockingSync` of the CUDA routine `cudaSetDeviceFlags`.

4.3. Search for the optimal configuration

Similarly to Section 3, we next analyze the runtime and energy efficiency of the algorithmically-optimized implementations of the CG method when configuring the hardware for either performance or energy efficiency by varying the CPU operation frequency and number of active cores. For completeness, we again provide the detailed information of the optimal configuration for each architecture and matrix combination (Tables VI and VII), but in the following analysis, we prefer referring to the graphical results. In particular, Figure 8 compares, analogously to Figure 5, the performance and energy efficiency of the architectures when configuring the hardware execution parameters either for runtime performance or energy efficiency.

At this point, we note that the algorithmic optimizations intrinsic to the data layouts BCSR and CSB, and the tuned implementations of `SPMV` embedded in the Intel MKL and CSB libraries, could not be leveraged on A9, A15, and TIC, because these are not `x86`-based architectures, and, therefore, it was not possible to use Intel's MKL, Cilk, or Thread Building Blocks. Hence, in Figure 5, we reproduce the data obtained with the basic implementations for these three architectures.

4.3.1. Optimization with respect to time. When comparing the runtime performance of the optimized implementations on the different architectures, the high-end GPUs KEP and FER are the overall winners, independently of whether the hardware execution parameters are tuned for performance or energy efficiency (see the left top and bottom plots in Figure 8, respectively). For example, the recent CPU generations—AIL and ISB—are outperformed by NVIDIA Tesla K20 (KEP) by

Table VII. Optimal hardware parameter configuration when optimizing the specialized architectures for runtime or energy efficiency using the tuned implementations.

		Optimized w.r.t time					Optimized w.r.t net energy				
	Matrix	c	f	s	T	E_{net}	c	f	s	T	E_{net}
FER	A159	1	2270	1	8.22E − 03	1.57E + 00	1	1600	1	8.24E − 03	1.49E + 00
	AUDI	1	1600	3	1.58E − 02	2.95E + 00	1	1600	3	1.58E − 02	2.95E + 00
	BMW	1	1600	3	2.06E − 03	3.73E − 01	1	1600	3	2.06E − 03	3.73E − 01
	CRANK	1	1600	3	2.43E − 03	4.50E − 01	1	1600	3	2.43E − 03	4.50E − 01
	F1	1	1600	3	6.21E − 03	1.16E + 00	1	1600	3	6.21E − 03	1.16E + 00
	LDOOR	1	1600	3	9.55E − 03	1.75E + 00	1	1600	3	9.55E − 03	1.75E + 00
KEP	A159	1	3200	1	5.29E − 03	7.45E − 01	1	1200	1	5.30E − 03	5.89E − 01
	AUDI	1	3200	3	9.37E − 03	1.39E + 00	1	1200	3	9.38E − 03	1.13E + 00
	BMW	1	3200	3	1.28E − 03	1.82E − 01	1	1200	3	1.29E − 03	1.47E − 01
	CRANK	1	3200	3	1.52E − 03	2.18E − 01	1	1200	3	1.53E − 03	1.76E − 01
	F1	1	3200	3	3.63E − 03	5.32E − 01	1	1200	3	3.64E − 03	4.31E − 01
	INLINE	1	3200	3	4.69E − 03	6.58E − 01	1	1200	3	4.71E − 03	5.55E − 01
	LDOOR	1	3200	3	5.61E − 03	8.21E − 01	1	1200	3	5.62E − 03	6.64E − 01
QDR	A159	1	1300	1	3.60E − 02	1.01E + 00	1	51	1	3.92E − 02	5.96E − 01
	BMW	1	1300	3	1.03E − 02	2.76E − 01	1	1300	3	1.03E − 02	2.76E − 01
	CRANK	1	1300	3	1.18E − 02	3.31E − 01	1	1300	3	1.18E − 02	3.31E − 01
	F1	1	1300	3	3.05E − 02	8.59E − 01	1	51	3	4.05E − 02	6.77E − 01
	INLINE	1	1300	0	3.42E − 01	9.59E + 00	1	51	0	3.46E − 01	7.73E + 00

In the labels, c denotes the number of cores, f the frequency (in MHz), s the sparse matrix layout/`SPMV` implementation (0 for CSR, 1 for ELLPACK, 2 for ELLR-T, and 3 for SELL-P), T the time per iteration (in seconds), and E_{net} the net energy per iteration (in Joules).

DOI: 10.1002/cpe

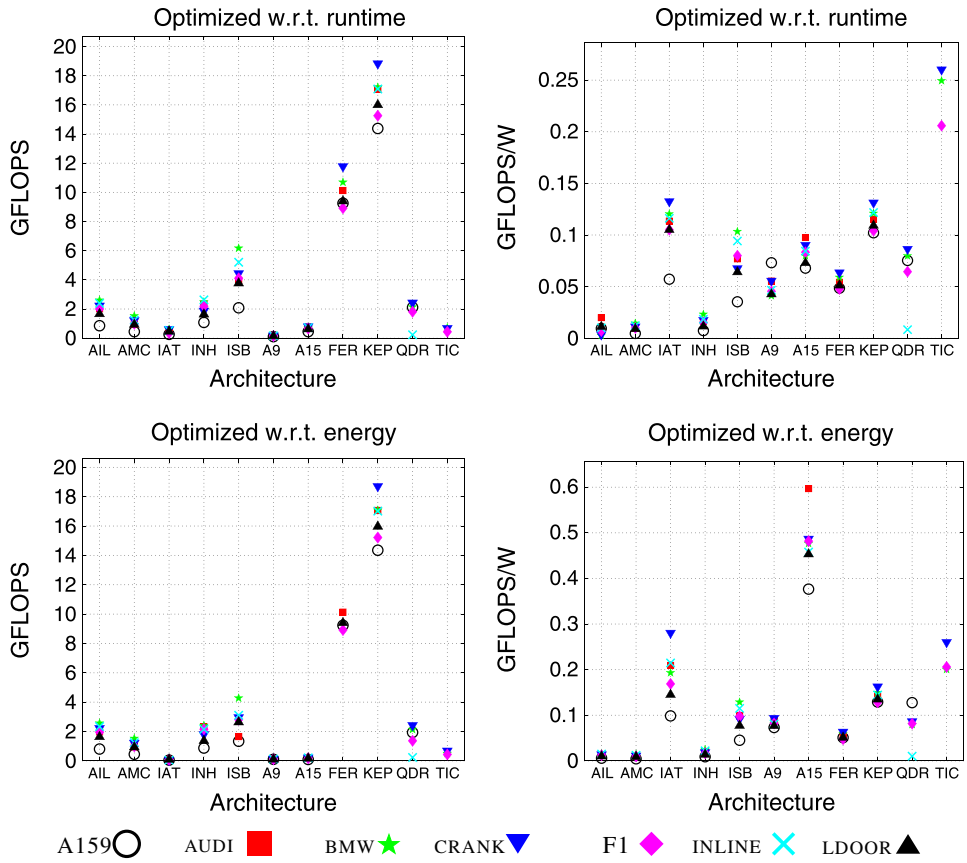


Figure 8. Comparison of performance (left) and energy efficiency (right), measured, respectively, in terms of GFLOPS and GFLOPS/W, when optimizing the tuned CG implementations with respect to run time (top) or net energy (bottom).

factors up to 10, and the reduced runtime of the GPU implementations directly translates into outstanding energy efficiency. From the energy perspective, all GPU architectures are competitive with the low-power processors IAT, A9, A15, and the Intel Sandy Bridge CPU ISB (see right-top graph in Figure 8). Only the TIC achieves about twice the energy efficiency. The most inefficient architectures from this point of view are, such as in the non-optimized case, the older general-purpose CPUs AIL, AMC, and INH.

4.3.2. Optimization with respect to net energy. As already observed for the basic implementation, modifying the hardware execution parameters for the GPU systems has only negligible impact on performance (compare left-top and bottom plots in Figure 8). This is different for ISB which outperforms QDR only when aiming for performance. Like in the non-optimized case, improving the energy demand of the CPUs and the low-power architectures comes at the price of a certain level of performance loss. As we could not apply algorithmic optimizations for the latter architectures (A9, A15, TIC), it is not surprising that they again deliver the lowest GFLOPS rates; despite that, A15 and TIC are the overall winners when aiming for low energy demands. Even the algorithmically-optimized CG implementation on the top-end GPUs achieves energy efficiency rates far from those of A15. The IAT and TIC are competitive, while the ISB achieves ratios similar to those of the older A9 and QDR. KEP has a slightly higher, and FER a slightly lower, energy efficiency than Intel's latest CPU generation (ISB), while the AMD CPUs (AIL and AMC) stay, such as in the performance-oriented configuration, far behind (see right-bottom plot in Figure 8). In contrast to the GPUs, the outstanding energy efficiency of the low-power architectures again comes at the cost of increased runtime, such that KEP outperforms A15 by almost two orders of magnitude.

DOI: 10.1002/cpe

Table VIII. Average GFLOPS and GFLOPS/W rates of the double-precision tuned implementations normalized with respect to the 'fastest' (highest GFLOPS rate when optimizing w.r.t. time) and 'most energy-efficient' (highest GFLOPS/W rate when optimizing w.r.t. energy) architectures, KEP and A15, respectively.

	Optimized w.r.t time		Optimized w.r.t net energy	
	Normal average GFLOPS	Normal average GFLOPS/W	Normal average GFLOPS	Normal average GFLOPS/W
AIL	1.18E − 01	7.95E − 02	1.06E + 01	2.61E − 02
AMC	6.31E − 02	8.84E − 02	5.77E + 00	2.13E − 02
IAT	2.87E − 02	9.31E − 01	4.20E − 01	3.93E − 01
INH	1.22E − 01	1.36E − 01	1.03E + 01	3.73E − 02
ISB	2.56E − 01	6.48E − 01	1.48E + 01	1.96E − 01
A9	9.24E − 03	4.47E − 01	6.42E − 01	1.73E − 01
A15	4.06E − 02	7.15E − 01	1.00E + 00	**1.00E + 00**
FER	6.05E − 01	4.70E − 01	3.46E + 01	1.14E − 01
KEP	**1.00E + 00**	1.00E + 00	9.10E + 01	2.97E − 01
QDR	1.05E − 01	5.46E − 01	8.94E + 00	1.62E − 01
TIC	3.70E − 02	2.07E + 00	2.99E + 00	4.67E − 01

To summarize this initial study (search for the optimal configuration), next we rank the systems according to their performance and energy efficiency. In order to do this, for each architecture we employ the GFLOPS and GFLOPS/W rates, averaged for all matrix benchmarks, when optimizing for time and for energy efficiency. The purpose and effect of using average values is to collapse the results for all the matrix cases into a single figure-of-merit, which allows an easier (albeit less precise) comparison of the architectures. Table VIII shows the results of these metrics, normalized with respect to the fastest architecture when optimizing for time (KEP), and with respect to the most energy-efficient one when optimizing for energy (A15). The results in the second and third columns of the table show that, when optimizing for performance, on average, KEP is roughly half an order of magnitude faster than FER (6.04E − 01), around one order of magnitude faster than QDR (1.05E − 01), and slightly more than two orders of magnitude faster than A9 (9.24E − 03). If the goal is runtime performance, KEP consumes less energy than all the other architectures except TIC. On the other hand, the last two columns of the table compare the energy efficiency of A15 with that of its competitors. Here, the differences (obviously in favor of A15 in all cases) are smaller, and range between close to half an order of magnitude for TIC (4.67E−01) and more than two orders of magnitude for AMC (2.13E−02).

4.4. Complementary analyses

In the remainder of this section, we elaborate on two additional studies of the impact of algorithmic optimization and the advantages of SP arithmetic, on IAT, ISB, and KEP. We selected these three systems because they all correspond to recent processor architectures and each one represents a different class of the systems included in this study (low-power architectures, server-oriented general-purpose multicore processors, and manycore GPUs, respectively). Finally, we chose IAT instead of the more appealing TIC and A15 because, as noted earlier, we could not apply algorithmic optimizations on the latter two.

4.4.1. Algorithmic optimization potential. Table IX assesses the impact of the algorithmic optimizations described at the beginning of this section. A first observation from this table is that significant performance improvements can be achieved through algorithmic optimizations of the implementations. This is especially true for KEP, where we replaced the standard ELLPACK-based implementation of SPMV with a format tailored to each matrix case, and substituted the CUBLAS-based implementation of the CG method with a version using algorithm-specific kernels. At the same time, the improvement potential on GPUs is again very dependent on the matrix characteristics, as we reduce runtime by 'only' 28% for the A159 case, but in a factor larger than 14× for

DOI: 10.1002/cpe

Table IX. Speed-ups resulting from the algorithmic optimizations.

		Optimized w.r.t. time		Optimized w.r.t. net energy	
	Matrix	Speed-up in T	Speed-up in E_{net}	Speed-up in T	Speed-up in E_{net}
IAT	A159	1.04	0.95	0.80	1.10
	AUDI	1.18	1.04	1.12	1.25
	BMW	1.19	1.06	1.12	1.20
	CRANK	1.23	1.12	0.56	1.40
	F1	1.14	1.16	1.13	1.20
	INLINE	1.16	1.04	0.61	1.13
	LDOOR	1.23	1.12	1.33	1.06
	AVERAGE	1.17	1.07	0.95	1.19
ISB	A159	1.52	1.29	1.89	1.10
	AUDI	1.51	1.49	1.34	1.51
	BMW	1.54	1.52	1.58	1.53
	CRANK	1.11	1.07	1.14	1.10
	F1	1.46	1.50	1.48	1.49
	INLINE	1.50	1.49	1.36	1.50
	LDOOR	1.28	1.25	1.33	1.25
	AVERAGE	1.42	1.37	1.45	1.36
KEP	A159	1.28	0.90	1.28	1.14
	AUDI	3.76	2.52	3.76	3.10
	BMW	4.35	2.95	4.31	3.65
	CRANK	14.86	9.49	14.77	11.76
	F1	4.62	3.08	4.61	3.80
	LDOOR	1.52	1.11	1.52	1.37
	AVERAGE	5.07	3.34	5.04	4.14

CRANK. An explanation for this effect can be found in Table III, showing a drastically reduced storage (and computational) overhead when we replace the ELLPACK with the SELLP kernel for the CRANK test case, while the improvement for the A159, where we keep the ELLPACK format (TableVII), comes exclusively from enhancing the implementation using algorithm-specific kernels. On average, Table IX reveals that replacing the basic CG implementation with an optimized variant improves performance on KEP by a factor of 5.07×. For the CPU-based architectures, the reported speed-ups are lower on average, but are, on the other hand, more consistent. On ISB, for example, the optimization techniques render an average performance improvement of 1.42×, with 1.11×, and 1.54× as upper and lower bounds, respectively; and similar variations when optimizing for energy efficiency. On the low-power architecture IAT, the improvements are slightly lower.

4.5. Single-precision performance through iterative refinement

It is well-known that iterative refinement, in combination with mixed precision, can render the benefits of a faster SP arithmetic while still delivering DP accuracy for some applications [34]. In particular, while the use of DP arithmetic is in general mandatory for the solution of sparse linear systems, in [35], for instance, it is shown how the use of mixed SP–DP arithmetic and iterative refinement leads to improved execution time and energy consumption for GPU-accelerated solver-platforms. Our last experiment thus aims to evaluate the performance and energy efficiency variations that can be expected when embedding a CG solver, which performs its arithmetic in SP instead of DP, into the mixed precision iterative refinement framework.

Table X reports the results of this test, comparing the performance and energy efficiency of SP and DP implementations of the optimized CG solvers in terms of their respective speed-ups. In this case, we do not include results for IAT, as the CSB library does not provide the necessary SP implementation of SPMV and, therefore, the collection of matrix cases that could be reported for IAT is limited (on this platform, the CSB solution is optimal for A159, CRANK, and INLINE; see

DOI: 10.1002/cpe

Table X. Speed-ups resulting from the use of the SP tuned implementations instead of the double-precision versions.

		Optimized w.r.t time		Optimized w.r.t net energy	
		Speed-up in T	Speed-up in E_{net}	Speed-up in T	Speed-up in E_{net}
ISB	A159	1.72	1.54	1.81	1.54
	AUDI	1.53	1.40	2.34	1.38
	BMW	1.64	1.53	1.51	1.54
	CRANK	1.47	1.54	1.41	1.43
	F1	1.59	1.42	1.48	1.50
	INLINE	1.64	1.48	1.75	1.54
	LDOOR	1.60	1.47	1.47	1.53
	AVERAGE	1.60	1.48	1.68	1.49
KEP	A159	1.52	1.70	1.52	1.72
	AUDI	1.34	1.47	1.34	1.49
	BMW	1.30	1.45	1.29	1.47
	CRANK	1.39	1.51	1.39	1.51
	F1	1.31	1.45	1.31	1.48
	INLINE	1.32	1.40	1.33	1.48
	LDOOR	1.28	1.43	1.28	1.44
	AVERAGE	1.35	1.49	1.35	1.51

Table VI). Furthermore, we found out that, for this particular architecture, in many cases, the optimal configuration when operating with DP data differed from that identified for SP arithmetic. On average, the variations are quite independent of the target platform, revealing acceleration factors on performance, when optimizing w.r.t. time, that vary between 1.35× (KEP) and 1.60× (ISB). Similar numbers are observed when the optimization is w.r.t. energy. It is, however, interesting to notice that the usage of SP arithmetic is more beneficial to performance on the CPU-based systems, while it renders larger improvement to the energy efficiency on the KEP GPU. If the hardware execution parameters are configured for energy efficiency, the same effect occurs at a larger scale. This can be explained by the fact that the algorithm is memory-bound, and decreasing frequency (and voltage) has, in general, more effect on the core clock than on the memory clock rate.

5. SUMMARY AND FUTURE WORK

Current processor architectures take different roads toward delivering raw performance and energy efficiency: in the form of low-power, general-purpose multicore designs and digital signal processors (DSPs) for mobile and embedded appliances; power-hungrier, but more versatile, general-purpose multicore architectures for servers; and hardware accelerators such as manycore graphics processors (GPUs) or the Intel Xeon Phi. In this paper, we have provided a broad overview about the potential of representative samples of these three different approaches, assessing their performance and energy efficiency using basic and advanced implementations of a crucial numerical algorithm: the iterative CG method. We observed that the flops-per-watt rate of manycore systems, such as the GPUs from NVIDIA, can be matched by low-power devices such as the Intel Atom, the ARM A9, or a DSP from Texas Instruments; and clearly outperformed by the more recent ARM A15. While GPUs traditionally deliver high energy efficiency with outstanding performance, the less energy hungry architectures provide it less cores, a lower power dissipation and/or smaller memories. This reduces the suitability of these inexpensive architectures for general-purpose computing, but makes them appealing candidates for mobile and embedded scenarios (their original target) as well as specific applications. Despite the fact that conventional general-purpose processors attained neither the performance of the GPUs nor the performance-per-watt rates of the low-power alternatives, they provide an interesting balance between these two extremes.

DOI: 10.1002/cpe

Future research will increase the scope of the study by adding problems not directly related to sparse linear algebra, for example, the seven dwarfs [36], as well as more complex scientific applications.

ACKNOWLEDGEMENTS

The authors from Universitat Jaume I were supported by the CICYT project TIN2011-23283 and FEDER, and by the EU FET FP7 project 318793 'EXA2GREEN'.

We thank Francisco D. Igual, from *Universidad Complutense de Madrid*, for his help with the low-power architectures evaluated in this work.

REFERENCES

1. The green500 list, June 2013. (Available from: http://www.green500.org).
2. The top500 list, June 2013. (Available from: http://www.top500.org).
3. Ashby S, Beckman P, Chen J, Colella P, Collins B, Crawford D, Dongarra J, Kothe D, Lusk R, Messina P, Mezzacappa T, Moin P, Norman M, Rosner R, Sarkar V, Siegel A, Streitz F, White A, Wright M. The opportunities and challenges of exascale computing. Summary report of the Advanced Scientific Computing Advisory Committee (ASCAC) subcommittee, November 2010.
4. Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hill K, Hiller J, Karp S, Keckler S, Klein D, Lucas R, Richards M, Scarpelli A, Scott S, Snavely A, Sterling T, Williams RS, Yelick K. Exascale computing study: technology challenges in achieving exascale systems. DARPA IPTO ExaScale Computing Study, 2008.
5. Dongarra J, Beckman P, Moore T, Aerts P, Aloisio G, Andre JC, Barkai D, Berthou JY, Boku T, Braunschweig B, Cappello F, Chapman B, Chi X, Choudhary A, Dosanjh S, Dunning T, Fiore S, Geist A, Gropp B, Harrison R, Hereld M, Heroux M, Hoisie A, Hotta K, Ishikawa Y, Jin Z, Johnson F, Kale S, Kenway R, Keyes D, Kramer B, Labarta J, Lichnewsky A, Lippert T, Lucas B, Maccabe B, Matsuoka S, Messina P, Michielse P, Mohr B, Mueller M, Nagel W, Nakashima H, Papka ME, Reed D, Sato M, Seidel E, Shalf J, Skinner D, Snir M, Sterling T, Stevens R, Streitz F, Sugar B, Sumimoto S, Tang W, Taylor J, Thakur R, Trefethen A, Valero M, van der Steen A, Vetter J, Williams P, Wisniewski R, Yelick K. The international exaScale software project roadmap. *International Journal of High Performance Computing Applications* 2011; **25**(1):3–60.
6. Duranton M, Black-Schaffer D, De Bosschere K, Maebe J. The HiPEAC vision for advanced computing in horizon 2020, 2013.
7. Fuller SH, Millett LI. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press: Washington, D.C., USA, 2011.
8. Dennard RH, Gaensslen FH, Rideout VL, Bassous E, LeBlanc AR. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 1974; **9**(5):256–268.
9. Esmaeilzadeh H, Blem E, Amant RS, Sankaralingam K, Burger D. Dark silicon and the end of multicore scaling. *Proceedings 38th Annual International Symposium on Computer Architecture, ISCA '11*, San Jose, California, USA, 2011; 365–376.
10. Bekas C, Curioni A. A new energy aware performance metric. *Computer Science - Research and Development* 2010; **25**:187–195. 10.1007/s00450-010-0119-z.
11. Saad Y. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2003.
12. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA. The landscape of parallel computing research: a view from Berkeley. *Technical Report UCB/EECS-2006-183*, University of California at Berkeley, Electrical Engineering and Computer Sciences, 2006.
13. Dongarra J, Heroux MA. Toward a new metric for ranking high performance computing systems. *Technical Report Sandia Report SAND2013-4744*, Sandia National Laboratories, 2013.
14. Aliaga J, Anzt H, Castillo M, Fernández J, Léon G, Pérez J, Quintana-Ortí ES. Performance and energy analysis of the iterative solution of sparse linear systems on multicore and manycore architectures. In *Lecture Notes in Computer Science, 10th International Conference on Parallel Processing and Applied Mathematics – PPAM 2013*, Warsaw, Poland, 2014.
15. Aliaga JI, Perez J, Quintana-Orti ES, Anzt H. Reformulated conjugate gradient for the energy-aware solution of linear systems on GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, October 2013; 320–329.
16. Langville AN, Meyer CD. *Google's Pagerank and Beyond: The Science of Search Engine Rankings*. Princeton University Press: Princeton, NJ, USA, 2009.
17. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, der Vorst HV. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (2nd edn). SIAM: Philadelphia, PA, USA, 1994.
18. Buluç A, Williams S, Oliker L, Demmel J. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proceedings IPDPS*, Anchorage, Alaska, USA, 2011; 721–733.

DOI: 10.1002/cpe

19. Williams S, Bell N, Choi J, Garland M, Oliker L, Vuduc R. Sparse matrix vector multiplication on multicore and accelerator systems. In *Scientific Computing with Multicore Processors and Accelerators*, Kurzak J, Bader DA, Dongarra J (eds). CRC Press: Boca Raton, FL, USA, 2010; 83–109.
20. Lawson CL, Hanson RJ, Kincaid DR, Krogh FT. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software* 1979; **5**(3):308–323.
21. Bell N, Garland M. Efficient sparse matrix-vector multiplication on CUDA. *Technical Report NVIDIA, NVR-2008-004*, NVIDIA Corporation, December 2008.
22. Vázquez F, Fernández JJ, Garzón EM. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 2011; **23**(8):815–826.
23. Vuduc R. Automatic performance tuning of sparse matrix kernels. *Ph.D. dissertation*, January 2004.
24. Im E-J, Yelick K, Vuduc R. Sparsity: optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications* 2004; **18**(1):135–158.
25. Intel. Intel math kernel library reference manual (release 10.3), 2014. Document Number: 630813-053US.
26. Buluç A, Fineman JT, Frigo M, Gilbert JR, Leiserson CE. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures. SPAA '09*. ACM: New York, NY, USA, 2009; 233–244.
27. Morton. A computer oriented geodetic data base and a new technique in file sequencing: *Technical Report* Ottawa, Ontario, Canada, 1966.
28. Buluç A, Aktulga HM, Demmel J, Fineman J, Frigo M, Gilbert J, Leiserson C, Oliker L, Williams S. Compressed sparse block library: Combinatorial Scientific Computing Lab at the University of California: Santa Barbara, 2014.
29. Vázquez F, Fernández JJ, Garzón EM. Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach. *Parallel Computing* 2012; **38**(8):408–420.
30. Kreutzer M, Hager G, Wellein G, Fehske H, Bishop AR. A unified sparse matrix data format for modern processors with wide SIMD units. *Computing Research Repository* 2013:1–23. abs/1307.6209.
31. Monakov A, Lokhmotov A, Avetisyan A. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC'10*. Springer-Verlag: Berlin, Heidelberg, 2010; 111–125.
32. Anzt H, Tomov S, Dongarra J. Implementing a sparse matrix vector product for the sell-C/sELL-C-σ formats on NVIDIA GPUs. *Technical Report ut-eecs-14-727*, University of Tennessee, March 2014.
33. MAGMA project home page, June 2014. (Available from: http://icl.cs.utk.edu/magma/).
34. Higham NJ. *Accuracy and Stability of Numerical Algorithms: Second Edition*. Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2002.
35. Anzt H, Heuveline V, Aliaga JI, Castillo M, Fernández JC, Mayo R, Quintana-Ortí E S. Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms. *Green Computing Conference and Workshops (IGCC), 2011*, Orlando, Florida, USA, July 2011; 1–6.
36. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA. The landscape of parallel computing research: a view from Berkeley. *Technical Report UCB/EECS-2006-183*, EECS Department, University of California: Berkeley, December 2006.

DOI: 10.1002/cpe