Addressing Materials Science Challenges Using GPU-accelerated POWER8 Nodes

Paul F. Baumeister[1(✉)], Marcel Bornemann[2], Markus Bühler[3], Thorsten Hater[1], Benjamin Krill[3], Dirk Pleiter[1], and Rudolf Zeller[2]

[1] Jülich Supercomputing Centre, Forschungszentrum Jülich, Jülich 52425, Germany
p.baumeister@fz-juelich.de
[2] Institute for Advanced Simulation, Forschungszentrum Jülich, Jülich 52425, Germany
[3] IBM Germany Research and Development, Böblingen 71032, Germany

Abstract. Materials research is an area that is expected to strongly benefit from the growing performance capabilities of future supercomputers towards exascale. Density functional theory (DFT) has become one of the most important methods for numerical materials science. In this paper we present results of a performance model based analysis of a particular, scalable DFT-based application on GPU-accelerated compute nodes with POWER8 processors. These technologies are part of a future roadmap for pre-exascale architectures. With power consumption becoming a major design constraint, we also determine the energy required for executing the most performance critical kernel.

1 Introduction

Density Functional Theory (DFT) is a key method for addressing challenges in materials science that require an accurate description of the electronic properties of a material. The complexity of calculating the full wave function of the many-electron system is avoided by considering a single-particle picture with an effective potential [15], giving rise to the Kohn-Sham equation $\hat{H}\Psi = E\Psi$. The solutions can take the form of either a set of eigenstates of the Hamiltonian $\hat{H}$ as realised in wave function based implementations [3,13,21] or the Green function $\hat{G}(E) = (E - \hat{H})^{-1}$ as proposed by Korringa, Kohn and Rostoker (KKR) [5,14,16]. Here, the energy E is continued into the complex plane with a non-vanishing imaginary part in order to prevent the inversion of a singular operator. A suitable representation allows for casting the problem into a matrix inversion maintaining high accuracy via a full-potential description. Despite, the matrix dimension only grows as $16\,N_{\mathrm{atom}}$ assuming a truncation of angular momenta beyond $\ell = 3$. The screened KKR method allows for finding a short ranged formulation and, hence, the equivalent operator $\hat{G}$ becomes block-sparse [26]. In large systems, where the number of atoms $N_{\mathrm{atom}} \gg 1000$, the Green function formulation can be approximated by systematically truncating long-ranged interactions between well-separated atoms. This reduces the overall complexity of the method from cubic to linear and large systems with $100,000$ atoms

P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 77–89, 2016.
DOI: 10.1007/978-3-319-43659-3_6

and more become thus feasible. KKRnano is a DFT application implementing the original cubic method as well as the linear-scaling approach [23,27]. It has been proven to scale to massively-parallel architectures leveraging MPI and OpenMP programming models. Central to its performance is an iterative solver for the linear system and the application of the block-sparse operator.

Massively-parallel computing resources are required to facilitate high throughput for medium-sized problems as well as to address large-scale challenges. The former will, e.g., be required to scan parameter spaces and evaluate high-dimensional phase diagrams. The latter involves problems where large N_{atom} are required, e.g. when effects that occur at the length-scale of several nanometers need to be understood and investigated. Ideal atomic geometries can be analysed using a workstation to run a DFT code that exploits symmetries. In contrast, realistic samples of a material are hardly ever perfect crystals with full translational symmetry or isolated molecules in vacuum. Addressing these challenges requires dealing with broken symmetries, i.e. crystals with impurities, random alloys or amorphous materials and thus result in calculations with $N_{\text{atom}} \gg 10,000$.

Due to the end of Dennard scaling the level of parallelism in HPC systems will become even more extreme to offer an increase in the number of floating-point operations per time unit. In order to minimise power consumption, low-clocked, but highly parallel compute devices like GPUs have become increasingly popular. Operating at clock speeds below 1 GHz means that more than 10^8 floating-point operations per clock cycle are required to reach a pre-exascale performance level of about 100 PFlop/s. In case of KKRnano the exploitable parallelism scales with N_{atom}, enabling exploitation of such massively-parallel architectures.

This article makes the following contributions:

1. We present a performance analysis for highly optimised implementations of the main kernel of the application KKRnano on both, IBM POWER8 processors and NVIDIA K40 GPUs.
2. To enable analysis of performance as well as scalability properties a simple performance model is developed. We use this model to explore scalability of the application for (not yet existing) large-scale systems based on this processor and accelerator technologies.
3. Finally, we evaluate energy-to-solution of our implementation with and without GPUs based on power consumption measurements of the system.

In this section and Sect. 2 we provide background on the application domain and relevant technology. After presenting an analysis of the application's performance characteristics in Sect. 3, we outline the main features of our implementation and provide a performance analysis for the kernel on POWER8 and the GPU in Sect. 4 and 5, respectively. In Sect. 6 we present our performance model and use it to explore the scalability of the application. We continue with a power consumption analysis in Sect. 7. Before concluding in Sect. 9, we provide an overview on related work in Sect. 8.

2 GPU-accelerated POWER Architectures

We evaluate application performance on commercially available POWER8 824 47 L servers [8], comprising two POWER8 sockets, 256 GiByte of memory and one NVIDIA K40m GPU per socket.

The POWER8 processors in the considered system are dual-chip modules, where each module comprises 5 cores, i.e. there are 20 cores per node. Each core offers two sets of the following instruction pipelines: fixed point, floating-point, pure load (LU) and a load-store unit (LSU). Instructions are processed out-of-order to increase instruction level parallelism. The cores support 8-way Simultaneous Multithreading. For the HPC workloads, as considered here, a few details are of special interest. The floating point unit, called the Vector Scalar Unit (VSU), supports two- or four-way SIMD for single-precision and two-way SIMD for double-precision floating-point instructions. Fused multiply-add instructions are provided. In case of floating-point instructions, the operands have to be present in VSU registers; load to these are processed in the LU exclusively. Further, stores from VSU are issued both to the LSU and the VSU internally.

Per cycle up to eight double-precision floating-point operations can be performed in the form of two fused multiply-add instructions on 128 bit vector registers, providing 29 GFlop/s per core or 590 GFlop/s per node at the peak clock of 3.69 GHz. Each core has a private L1 data cache of 64 kByte, a private L2 cache of 512 kiByte and a segment of 8 MiByte associated to it in the shared L3 cache (total 80 MiByte). In concert with a set of external memory buffers – called the Centaur chip – the POWER8 CPU provides a maximum read and write bandwidth of 256 GByte/s and 128 GByte/s per socket, respectively. The memory system can provide up to two 16 Byte loads and one 16 Byte store per cycle at L1.

Each of the POWER8 sockets is connected to an NVIDIA K40m GPU via an x16 PCIe GEN3 link. The K40m is based on GK110 GPUs of the Kepler generation running at 745 MHz. With a total of 15 streaming multi-processors it has a peak performance of 1430 GFlop/s. Each GPU can either write or read to or from its 12 GiByte of GDDR5 memory with a nominal bandwidth of 288 GByte/s.

Both compute devices, the POWER8 processor as well as the K40m GPU, thus offer significantly different hardware performance capabilities. The POWER8 processor features very high memory bandwidth at moderate floating-point operations throughput and operates at relatively high clock speed. In constrast, the K40m has a much higher concurrency to provide very high floating-point operation throughput at moderate clock speed and a memory bandwidth that is relatively small compared to its compute capabilities.

3 Application Performance Characteristics

We focus on a single iteration of the KKR algorithm, comprising the solution of a linear system locally and, afterwards, setup of a new system for the next

iteration. Solving the local problem is approached using a variant of the Quasi Minimal Residual (QMR) method, an iterative solver [10]. In the case at hand, simultaneous solutions of a set of right-hand sides are sought. Given Λ and ω we have the following problem to solve

$$\Lambda\gamma = \omega \tag{1}$$

where the elements Λ_{ij} are operators describing the interaction between an atom i with its direct neighbours j. We fix the number of columns to $N_{\mathrm{cl}} = 13$ entries from here on, which corresponds to a close packed lattice structure (N_{cl}=13 for hcp or fcc, while for bcc, N_{cl}=15 is a good choice). The number of rows corresponds to the number of atoms in a truncation cluster, we primarily use $N_{\mathrm{tr}} = 1000$. The elements of Λ are small dense square matrices over $\mathbb{C}$. The size of these entries b corresponds to the order to which the expansion in the angular momentum is truncated, we pick the current default, namely $b = 16$. Since Λ is sparse, the operator is compressed in memory by dropping the zero elements in each row and carrying the appropriate index list. The runtime of the solver is dominated by the application of the operator Λ, which consumes around 90 % of the solver's runtime. `KKRnano` operates on double-precision complex numbers so 16 Byte are assumed per number and 8 Flop are required to perform a complex fused multiply-accumulate operation.

The parallelisation strategy of `KKRnano` foresees one MPI rank per atom, i.e. the number of tasks per node is given by $N_{\mathrm{atom}}/N_{\mathrm{node}}$, where N_{atom} and N_{node} are the number of atoms and nodes, respectively. Each task has to solve Eq. (1) using the iterative solver which does not require any inter-node communication. After solving the linear system, the operator Λ needs to be updated, which involves communication with N_{tr} other tasks. We analyse the relevant kernel using an information exchange approach [7,19], which models the hardware as a graph of data stores connected by edges representing communication links or processing pipelines. We choose a simple model for the processor consisting of two data stores, the external main memory and the on-chip memory, representing register file and caches.

The performance of the overall kernel is driven by accumulating dense matrix products when applying the operator Λ. In the following we assume that the solver always performs a fixed number of iterations N_{iter} with two applications of Λ per iteration. At node level we therefore can characterise the kernel by the following information exchange functions:

$$I_{\mathrm{fp}} = 2N_{\mathrm{iter}} \cdot \frac{N_{\mathrm{atom}}}{N_{\mathrm{node}}} \cdot N_{\mathrm{tr}} N_{\mathrm{cl}} \cdot b^3 \cdot 8\,\mathrm{Flop}, \tag{2}$$

$$I_{\mathrm{ld}} = 2N_{\mathrm{iter}} \cdot \frac{N_{\mathrm{atom}}}{N_{\mathrm{node}}} \cdot N_{\mathrm{tr}} N_{\mathrm{cl}} \cdot b^2 \cdot 16\,\mathrm{Byte}, \tag{3}$$

$$I_{\mathrm{st}} = 2N_{\mathrm{iter}} \cdot \frac{N_{\mathrm{atom}}}{N_{\mathrm{node}}} \cdot N_{\mathrm{cl}} \cdot b^2 \cdot 16\,\mathrm{Byte}, \tag{4}$$

where I_{fp} is the number of floating-point operations required to solve Eq. (1) for all atoms on one node. I_{ld} and I_{st} account for the input and output operands

that need to be loaded and stored, respectively. We furthermore assume that all other numerical subtasks - which scale as $N_{\mathrm{tr}} \cdot b^2$ - within the solver can be ignored. No assumptions are made about exploitation of data reuse outside the complex multiplications. The information exchange functions can be used to compute the arithmetic intensity

$$AI = \frac{I_{\mathrm{fp}}}{I_{\mathrm{st}} + I_{\mathrm{ld}}} \overset{N_{\mathrm{cl}} \gg 1}{\simeq} \frac{b}{4} \frac{\mathrm{Flop}}{\mathrm{Byte}} \overset{b=16}{=} 4 \frac{\mathrm{Flop}}{\mathrm{Byte}}. \tag{5}$$

Following the roofline performance model approach [25] we thus expect the maximum attainable performance of the application to be limited by the throughput of double-precision floating-point operations on the POWER8 processors, while on the K40 GPU the nominal memory bandwidth limits the attainable performance to 80 % of the nominal floating-point performance. Our previous investigations showed memory bandwidths on the CPU of more than 280 GByte/s [2], while on the K40 210 GByte/s (ECC active) were achievable, resulting in the same expected performance for the host and a reduced expectation of 840 GFlop/s on the K40m.

When the solver is executed on the GPU, additional data transfers are needed. Before launching the solver, Λ and $\boldsymbol{\omega}$ need to be transfered from host to device. After completion the result vector $\boldsymbol{\gamma}$ has to be transfered from device to host. Both vectors are stored as dense arrays of N_{tr} blocks. Thus, we write for this sub-task:

$$I_{\mathrm{acc}} = \frac{N_{\mathrm{atom}}}{N_{\mathrm{node}}} \left(2N_{\mathrm{tr}} + N_{\mathrm{cl}}\, N_{\mathrm{tr}}\right) b^2\, 16\,\mathrm{Byte}. \tag{6}$$

The full vector is required to be present on the device, if even the operator application might only utilise a subset, consequently, the full transfer is accounted for.

After solving Eq. (1), Λ is updated. In the worst case all N_{tr} pairing atoms are located on other nodes, i.e. all information needs to be communicated over the network. This information exchange is captured by by the information exchange function

$$I_{\mathrm{net}} = \frac{N_{\mathrm{atom}}}{N_{\mathrm{node}}} N_{\mathrm{cl}}\, N_{\mathrm{tr}}\, b^2\, 16\,\mathrm{Byte}. \tag{7}$$

4 Application Performance Analysis on Processor

To simplify adaption of the code, we extracted the performance critical part of the code in a benchmark, i.e. the $2N_{\mathrm{iter}}$ applications of operator Λ. While the original code is implemented in Fortran, in case of the benchmark we choose C++. The benchmark retains only the block sparse operator application from the original solver, however, this part is reproduced in full. The omission is limited to parts scaling as $b^2 N_{\mathrm{tr}}$ in arithmetic operations, compared to $b^3 N_{\mathrm{tr}} N_{\mathrm{cl}}$ for the operator. The reduction to this core can increase the effectiveness of data caches, due to the smaller working set size and higher temporal locality.

Λ is stored in compressed block sparse row format. The kernel traverses the per-row index list π to accumulate the required blocks of the result. Multiple rows

are processed in parallel using OpenMP threads. We compute the result vector in terms of its individual blocks, each corresponding to a row of the operator Λ. Each row i is processed by one thread which utilises the indices $\pi(i,j)$ to compute $\omega_i \leftarrow \Lambda_{ij}\gamma_{\pi(i,j)}$. The core of the algorithm is the dense matrix product in $\mathbb{C}^{b\times b}$.

Based on the analysis presented in Sect. 3 we expect the performance of the benchmark to be limited by the floating-point throughput. To maximize this throughput it is necessary to exploit 2-way SIMD. These expectations are confirmed by our observations. To enable the compiler to use SIMD instructions we changed the data layout. While the original code follows an array-of-structure design with arrays of complex numbers, the benchmark employs a structure-of-array separating real and imaginary parts numbers into different arrays.

In Table 1 we show a performance counter analysis for the full solver taken from the original code as well as our optimised benchmark. The parameters of both runs have been chosen such that the same number of inner matrix-matrix multiplications is performed. More specifically, the run was for a single atom on a single node, i.e. $N_{\text{atom}} = N_{\text{node}} = 1$. To obtain stable numbers, a single pinned core per atom was utilised and measured. Furthermore, we have set $N_{\text{tr}} = 1000$, $N_{\text{cl}} = 13$, $b = 16$ and $N_{\text{iter}} = 1000$. As not all performance counters can be measured during a single run, Table 1 combines the results obtained from multiple runs. For each performance counter we have repeated the same run 10 times and use only the minimum value for our analysis.

Table 1. Selected performance counters for the full solver mini-application and the performance optimized benchmark, which mimics the behaviour of this solver. The parameters of these runs are discussed in the text. Cycles in which the core is waiting for completion of a group of finished instructions are marked as *completing*, those in which another thread blocked the completion port are marked *thread*. Stores are counted twice by the hardware counters, as they are issued to *both* the LSU and VSU.

	Solver	Benchmark		Solver	Benchmark
Cycles (10^9)	850	188	Instructions (10^9)	1718	487
├─Running	332	116	├─Branch	11	7
├─Completing	118	28	├─Integer	30	88
├─Thread	16	4	├─Arithmetic	943	232
└─Stalled	378	38	│ ├─Scalar	656	0
│ ├─VSU	366	24	│ ├─Vector	7	217
│ └─LSU	10	14	│ └─(Stores)	280	15
Transfer (GByte)	211	213	└─Memory	1015	180

Using Eq. (2) we find $I_{\text{fp}} = 852\,\text{GFlop}$. The number of floating-point instructions would be minimized if the application could be mapped to 2-way SIMD fused multiply-add instructions, i.e. $N_{\text{vfp}} = I_{\text{fp}}/4$. In practice, we find an overhead of less than 1 % in the number of arithmetic vector instructions. For the

original code we observe no vector instructions and the number of scalar arithmetic instructions $N_{\mathrm{fp}} \gg I_{\mathrm{fp}}/2$ due to a lack of fused multiply-add operations, which is confirmed by an inspection of the assembly. Over the runtime of the benchmark a total volume of 211 GByte is loaded and stored, while the full solver transfers 213 GByte. Note that both numbers are slightly lower than the estimated value from Eqs. 3 and 4, which we attribute to the large L3 cache, which could in theory hold one full problem set. Thus, the ratio of required floating point operations to actually transferred bytes is larger than four. The two programs utilise 1.3 GByte/s and 4.6 GByte/s of memory bandwidth.

Assuming that memory instructions and arithmetic instructions can be perfectly overlapped and distributed over at least 2 pipelines, we would expect that the minimum time-to-solution in units of clock cycles is equal to $N_{\mathrm{vfp}}/2 \simeq I_{\mathrm{fp}}/8 = 106 \cdot 10^9$. In practice, we observe that due to a significant number of stall cycles the number of clock cycles spent in the solver $\Delta t_{\mathrm{solver}}$ to be almost 80 % larger. In summary, using a benchmark version of the application kernel, we are able to reach on a single core a floating-point efficiency $\epsilon_{\mathrm{fp}} = I_{\mathrm{fp}}/(8 \cdot \Delta t_{\mathrm{solver}}) = 56\,\%$.

5 Kernel Acceleration on GPU

We investigate the viability of GPU acceleration for `KKRnano` by porting the complete benchmark version of the solver. For the GPU implementation CUDA is used. The porting efforts are significantly reduced as the block sparse matrix-vector multiplication can be implemented using the cuSPARSE library.

With GPUs featuring extreme levels of parallelism, the obtained performance can in practice strongly depend on the level of parallelism of the problem solved on the GPU. Additionally, kernel launch times can have a non negligible effect. In Fig. 1 we therefore explore both kernel execution time as well as performance as a function of N_{tr} (the other parameters are the same as in the previous section). We observe that the performance saturates for $N_{\mathrm{tr}} \gtrsim 1000$. Maximum performance is obtained for $N_{\mathrm{tr}} = 3000$. From Eq. (2) we obtain $I_{\mathrm{fp}} = 2.55$ TFlop, 8 s to execute on a single K40. This corresponds to a performance of about 320 GFlop/s, which is far below the maximum attainable performance as expected from the roofline model. We analysed the resulting performance using GPU hardware counters and the NVIDIA profiling tools and observed the bandwidth to the shared memory being almost fully used. This could indicate that the bandwidth to the shared memory in the cuSPARSE implementation is the limiter and not the external memory bandwidth, as it was expected from the analysis in Sect. 3.

In order to improve the resource utilisation on the GPU, we investigated how performance changes when multiple tasks running on the CPU use the GPU simultaneous for solving Eq. (1). This is possible using the multi-process service `mps`. The performance gain can be quantified by a weak-scaling efficiency $\epsilon_{par}(n) = n\Delta t_s/\Delta t_p(n)$, where Δt_s is the serial solver execution time for a single solver instance without `mps` and $\Delta t_p(n)$ is the time required for n concurrent calls of the solver. The results for $1 \leq n \leq 10$ are shown in Fig. 1. The upper

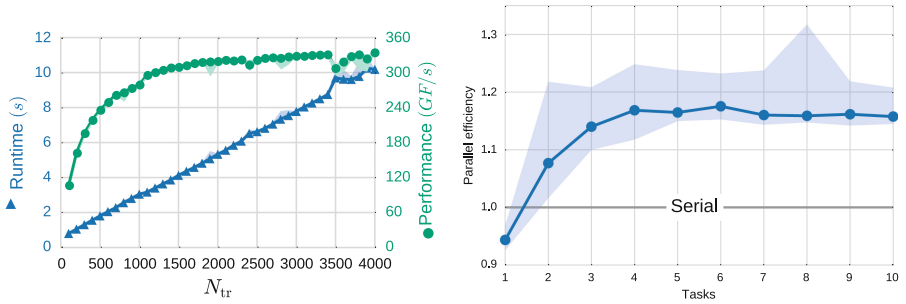


Fig. 1. Benchmark performance results obtained using $N_{\mathrm{iter}} = 1000$ using a single (left) and multiple tasks for $N_{\mathrm{tr}} = 1000$ (right) on a single K40m.

limit corresponds to one task per core of the processor, to which the GPU is attached. A gain of 17 % in efficiency is observed.

6 Performance Model Analysis

To enable an assessment of the performance of `KKRnano` on not yet existing larger systems based on GPU-accelerated nodes with POWER8 processors, we employ a performance modeling approach used in [4], which combines the information exchange analysis with semi-empirical performance analysis [12]. For this we assume that time-to-solution depends linearly on the information exchange. Furthermore, we assume that arithmetic operations and memory transfers can be perfectly overlapped. In case the solver is executed on the POWER8 processor, the performance can be expected to be limited by the floating-point operation throughput and we thus make the following ansatz:

$$\Delta t_{\mathrm{solver}}^{\mathrm{CPU}} = a_0^{\mathrm{CPU}} + a_{1,\mathrm{fp}}^{\mathrm{CPU}} I_{\mathrm{fp}}, \tag{8}$$

where I_{fp} is defined in Eq. (2). The coefficients a_0^{CPU}, $a_{1,\mathrm{fp}}^{\mathrm{CPU}}$ are determined by fitting Eq. (8) to timing measurements for different application parameters.

If the solver is executed on the GPU, we assume performance to be limited by memory bandwidth. Additionally we have to take the time into account that is required to data transfer from host to device and vice versa. This results in a slightly more complex ansatz using I_{ld}, I_{st} and I_{acc} from Eq. (3), (4) and (6), respectively:

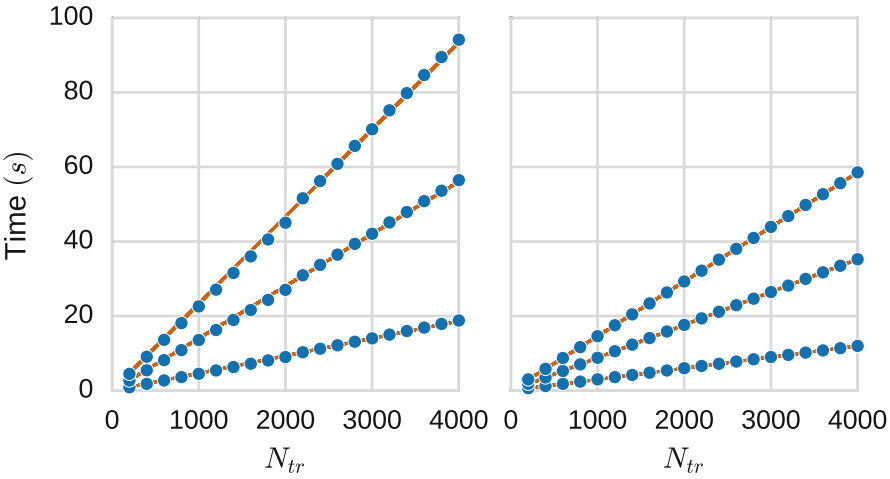
$$\Delta t_{\mathrm{solver}}^{\mathrm{GPU}} = a_0^{\mathrm{GPU}} + a_{1,\mathrm{mem}}^{\mathrm{GPU}}(I_{\mathrm{ld}} + I_{\mathrm{st}}) + a_{1,\mathrm{acc}}^{\mathrm{GPU}} I_{\mathrm{acc}}. \tag{9}$$

To determine the model parameters we have performed multiple runs with fixed $N_{\mathrm{atom}} = 20$, $N_{\mathrm{node}} = 1$, $N_{\mathrm{cl}} = 13$, $b = 16$, and different N_{iter} as well as N_{tr}. The runs are repeated multiple times for the same parameter setting and the minimal value is used. Error bounds are established by *k-fold* cross-validation with $k = 100$. Due to the size of the problem, the constant terms turned out to be insignificant and have been ignored.

The final contribution to our model is the update of the operator Λ, which requires a local computation of one row (neglected) and assembling the remote rows into the full operator. Applying the same approach as before we have

$$\Delta t_{\text{upd}} = c_{0,\text{net}} + c_{1,\text{net}} I_{\text{net}}, \tag{10}$$

where I_{net} is defined in Eq. (7). To determine the coefficients $c_{0,\text{net}}$ and $c_{1,\text{net}}$ we used the OSU micro-benchmarks [1] to measure the bandwidth between two POWER8 systems interconnected via a Mellanox EDR Infiniband network. Since for realistic parameter settings the effect of the constants a_0^{CPU}, a_0^{GPU} and c_0 is negligible, we focus on the linear term only. In Fig. 2 we show the inverse values for the coefficients of the linear terms to facilitate comparison with the bandwidth and throughput parameters of the hardware.



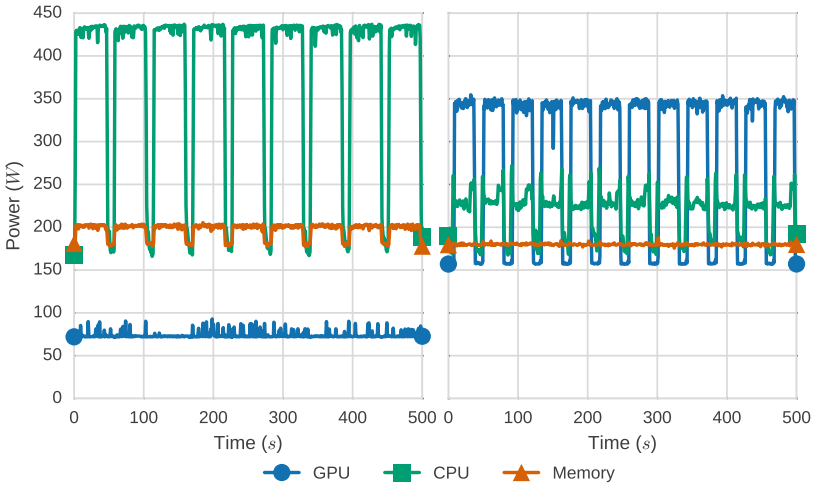
$1/a_1^{\text{CPU}}$	(364.8 ± 0.06) GFlop/s	$1/a_1^{\text{GPU}}$	(152.1 ± 0.002) GByte/s
$1/b_1^{\text{acc}}$	(16.2 ± 0.02) GByte/s	$1/c_{1,\text{net}}$	(22.7 ± 0.1) GByte/s

Fig. 2. Data points used to determine the model parameters for CPU (left) and GPU (right) and predictions for $N_{\text{iter}} = 200, 600, 1000$. The parameters are tabulated below with their respective errors.

The model allows us to assess whether `KKRnano`, which scales with good efficiency on a 28-rack Blue Gene/Q system, could scale on a hypothetical system comprising nodes that have a similar architecture as the one considered in this paper. We would need at least 2100 nodes to reach a similar peak performance. For an efficient utilization of the resources of a single node, we assume $N_{\text{atom}}/N_{\text{node}} \geq 20$, i.e. $N_{\text{atom}} \geq 42000$ for $N_{\text{node}} = 2100$. This matches the target problem size of this application area. From the performance model we find that $\Delta t_{\text{upd}} \ll \Delta t_{\text{solver}}$, even were we to assume much smaller values of $1/c_{1,\text{net}}$ due to network congestion.

7 Energy Efficiency Analysis

Let us finally consider the energy-to-solution for a single execution of the solver on the considered architecture. The POWER8 processor provides an on-chip controller (OCC) to measure a set of sensors in the hardware. The data is available out-of-band via a service processor and can be read out by the AMESTER tool [9,17]. The measurement granularity depends on the number of sensors, each requires an additional latency of typically 200 ms. The data is, therefore, gathered in irregular intervals. We resample it to a set of regular 1 s measurement points. The incoming data represents the current power consumption of the component corresponding to the sensor. To calculate the overall energy consumption, we use thresholding of the data to detect active phases, sum the power consumption measurements P_i and scale with the measurement interval Δt and the number of detected solver executions. We do not report all available measurements, only the total of memory, CPU and GPU values are provided. The sensor for the 12 V domain includes different I/O devices, including part of the power consumed by the GPUs. We attribute the values of these sensors fully to the GPU's power consumption, which leads to a slight overestimate of the actual value. The power consumed by the cooling fans shows significant variation and no distinguishable correlation with the workload. The signal was replaced by its average. We utilise a setup close to the



Energy (kJ)	Disk	IO	Memory	GPU	Fan	CPU	Total
Power8	1.78	2.90	11.07	1.25	4.09	21.61	42.70
K40	1.33	4.79	7.52	7.19	3.11	9.2	33.13

Fig. 3. Power consumption of the linear solver, CPU (left) and GPU (right). Below, we report the averaged total energy to solution for the corresponding benchmarks. (Color figure online)

configuration used in production runs of `KKRnano`, that is $N_{\mathrm{cl}} = 13$, $N_{\mathrm{tr}} = 1000$, $b = 16$ and $N_{\mathrm{iter}} = 1000$ iterations inside the solver. The number of iteration is chosen as the maximum number allowed in `KKRnano`, on average numbers are typically $\mathcal{O}(100)$. Per core one instance of the problem is solved, for a total of 20; from prior analysis we have the requirement of 17 TFlop. The power consumption over multiple invocations of the CPU and GPU implementations of solver is shown in Fig. 3. Only about 20 W of additional power is utilised by the memory system, as the solver is not very memory intensive. We report energy metrics of the full node in Fig. 3 for the solution of one instance of the problem per socket or GPU respectively. Power consumption is averaged over multiple invocations of the solver. Since power required for an idle system is quite high, much of the total energy required for the solution is explained by the base cost. Thus, this metric likely favors fast implementations of the solver.

8 Related Work

Recent efforts to accelerate DFT methods leveraging GPU-based systems can be found in literature. GPU acceleration has been achieved for wave function based DFT methods, e.g. plane wave methods, wavelets, grid and local orbitals [11,20,22,24]. Closer to this work are projects in the class of linear-scaling methods like SIESTA or CP2K [6,21]. As node architectures based on POWER8 processors are relatively new and, in particular, the GPU-accelerated versions not yet widely available, only few performance investigations related to scientific applications have been published. Applications based on the Lattice Boltzmann method, a brain simulator as well as an application based on the Finite Difference Time Domain method are considered [2,4]. The authors of [18] focus on server workloads as well as big data, analytics, and cloud workloads.

9 Conclusions and Future Work

In this paper we presented results for a highly scalable materials science application based on the Density Functional Theory (DFT) method. Typically, most of the computational resources are spent in an iterative solver. We could demonstrate that for this kernel a high or at least good floating-point efficiency could be obtained on the POWER8 processors and K40m GPUs, respectively.

To explore the scalability properties of this application on future systems based on GPU-accelerated compute nodes with POWER processors, which could provide a performance of $\mathcal{O}(10)$ PFlop/s, we designed a simple performance model. From this we could conclude that assuming a network technology that is state-of-the-art as of today a good scalability is achievable. An analysis of the energy-to-solution for the relevant kernel revealed that, although much higher floating-point operation efficiency can be obtained on the POWER8 processors, the energy-to-solution is significantly smaller when using GPUs.

This work leaves multiple opportunities for future work. First, the model analysis suggests that a specialised implementation of the sparse operator application GPU kernel could outperform the cuSPARSE library for this concrete problem. Second, with the upcoming availability of large scale POWER8 based systems employing a high performance interconnect, we will investigate the validity of the developed models. Finally, the application might benefit from a flexible distribution of work among processor and accelerator, as the application kernel runs efficiently on both.

Acknowledgements. This work was done in the framework of the POWER Acceleration and Design Center. We acknowledge the support from Charles Lefurgy (IBM) and Willi Homberg (JSC) on performing power consumption measurements using Amester. Furthermore, we thank Jiri Kraus (NVIDIA) for many helpful discussions.

References

1. OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/
2. Adinetz, A.V., Baumeister, P.F., Böttiger, H., Hater, T., Maurer, T., Pleiter, D., Schenck, W., Schifano, S.F.: Performance evaluation of scientific applications on POWER8. In: Jarvis, S.A., Wright, S.A., Hammond, S.D. (eds.) PMBS 2014. LNCS, vol. 8966, pp. 24–45. Springer, Heidelberg (2015)
3. Baumeister, P.F.: Real-Space Finite-Difference PAW Method for Large-Scale Applications on Massively Parallel Computers. Ph.D. thesis, RWTH Aachen (2012)
4. Baumeister, P.F., Hater, T., Kraus, J., Pleiter, D., Wahl, P.: A performance model for GPU-accelerated FDTD applications. In: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), pp. 185–193 (2015)
5. Beeby, J.: The density of electrons in a perfect or imperfect lattice. In: Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, vol. 302. The Royal Society (1967)
6. Ben, M.D., Hutter, J., VandeVondele, J.: Second-order Møller-Plesset perturbation theory in the condensed phase. J. Chem. Theory. Comput. **8**(11), 4177–4188 (2012)
7. Bilardi, G., Pietracaprina, A., Pucci, G., Schifano, F., Tripiccione, R.: The potential of on-chip multiprocessing for QCD machines. In: Bader, D.A., Parashar, M., Sridhar, V., Prasanna, V.K. (eds.) HiPC 2005. LNCS, vol. 3769, pp. 386–397. Springer, Heidelberg (2005)
8. Caldeira, A.B., et al.: IBM Power System S824L technical overview and introduction (2014). redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/redp5139.html
9. Floyd, M., et al.: Introducing the adaptive energy management features of the POWER7 chip. IEEE Micro **31**(2), 60–75 (2011)
10. Freund, R.W., Nachtigal, N.: QMR: a quasi-minimal residual method for non-Hermitian linear systems. Numer. Math. **60**(1), 315–339 (1991)
11. Hakala, S., Havu, V., Enkovaara, J., Nieminen, R.: Parallel electronic structure calculations using multiple graphics processing units (GPUs). In: Manninen, P., Öster, P. (eds.) PARA. LNCS, vol. 7782, pp. 63–76. Springer, Heidelberg (2013)
12. Hoefler, T., Gropp, W., Kramer, W., Snir, M.: Performance modeling for systematic performance tuning. In: State of the Practice Reports. SC 2011. ACM (2011)
13. Hutter, J., Iannuzzi, M., Schiffmann, F., VandeVondele, J.: CP2K: atomistic simulations of condensed matter systems. Comp. Mol. Sci. **4**(1), 15–25 (2014)

14. Kohn, W., Rostoker, N.: Solution of the Schrödinger equation in periodic lattices with an application to metallic Lithium. Phys. Rev. **94**, 1111–1120 (1954)
15. Kohn, W., Sham, L.J.: Self-consistent equations including exchange and correlation effects. Phys. Rev. **140**, A1133–A1138 (1965)
16. Korringa, J.: On the calculation of the energy of a Bloch wave in a metal. Physica **13**(6), 392–400 (1947)
17. Lefurgy, C., Wang, X., Ware, M.: Server-level power control. In: Fourth International Conference on Autonomic Computing, 2007. ICAC 2007, pp. 4–4, June 2007
18. Mericas, A.E.A.: IBM POWER8 performance features and evaluation. IBM J. Res. Dev. **59**(1), 6:1–6:10 (2015)
19. Pleiter, D.: Parallel computer architectures. In: 45th IFF Spring School 2014 "Computing Solids Models, ab-initio Methods and Supercomputing", Schriften des Forschungszentrums Jülich, Reihe Schlüsseltechnologien, vol. 74 (2014)
20. Solcà, R., Kozhevnikov, A., et al.: Efficient implementation of quantum materials simulations on distributed CPU-GPU systems. In: SC 2015 Conference on Proceed, pp. 10:1 (2015)
21. Soler, J.M., et al.: The SIESTA method for ab initio order-N materials simulation. J. Phys.: Condens. Matter **14**(11), 2745 (2002)
22. Spiga, F., Girotto, I.: phiGEMM: a CPU-GPU library for porting Quantum ESPRESSO on hybrid systems. In: 2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2012, pp. 368–375, February 2012
23. Thiess, A., et al.: Massively parallel density functional calculations for thousands of atoms: KKRnano. Phys. Rev. B **85**, 235103 (2012)
24. Videau, B., Marangozova-Martin, V., Genovese, L., Deutsch, T.: Optimizing 3D convolutions for wavelet transforms on CPUs with SSE units and GPUs. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 826–837. Springer, Heidelberg (2013)
25. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)
26. Zeller, R., et al.: Theory and convergence properties of the screened Korringa-Kohn-Rostoker method. Phys. Rev. B **52**, 8807–8812 (1995)
27. Zeller, R.: Towards a linear-scaling algorithm for electronic structure calculations with the tight-binding Korringa-Kohn-Rostoker Green function method. J. Phys.: Condens. Matter **20**(29), 294215 (2008)