GPU Accelerated Sparse Matrix Matrix Multiplication for Linear Scaling Density Functional Theory

Ole Schütt,[†] Peter Messmer,[‡,¶] Jürg Hutter,[§] and Joost VandeVondele[*,†]

Nanoscale Simulations, Department of Materials, ETH Zürich, Wolfgang-Pauli-Str. 27, CH-8093 Zürich, Switzerland, NVIDIA Switzerland, Technoparkstr 1, CH-8005 Zürich, NVIDIA Co-design lab for hybrid multicore computing, Wolfgang-Pauli-Str. 27, CH-8093 Zürich, Switzerland, and Institute of Physical Chemistry, University of Zürich, Winterthurerstrasse 190, CH-8057 Zürich, Switzerland

E-mail: Joost.VandeVondele@mat.ethz.ch

[*]To whom correspondence should be addressed

[†]Nanoscale Simulations, Department of Materials, ETH Zürich, Wolfgang-Pauli-Str. 27, CH-8093 Zürich, Switzerland

[‡]NVIDIA Switzerland, Technoparkstr 1, CH-8005 Zürich

[¶]NVIDIA Co-design lab for hybrid multicore computing, Wolfgang-Pauli-Str. 27, CH-8093 Zürich, Switzerland

[§]Institute of Physical Chemistry, University of Zürich, Winterthurerstrasse 190, CH-8057 Zürich, Switzerland

Introduction

Linear Scaling SCF

With the steady increase in computer power available, larger and larger systems are simulated using electronic structure methods. These large simulations are exposing the asymptotic scaling of the traditional algorithms used in electronic structure calculations. Many of the traditional algorithms have a cubic or higher scaling with system size, effectively blocking the path to very large scale simulations. However, it is known that effective interactions are of a short-ranged nature for many systems, which can be exploited in linear scaling methods [1,2]. Consequently, a large effort has been spent in developing algorithms with a computational cost that scales linearly with system size. Originally, most applications and benchmarks of these methods were restricted to systems with a quasi one-dimensional structure, were the prefactor of linear scaling methods is very favorable. Now, the huge increase in computational power and the refinement of the algorithms, has made it possible to study scientifically relevant three-dimensional systems. Basis sets of good quality and tight numerical thresholds can be employed, essentially allowing for an accuracy that is identical to that of the cubically scaling methods.

For the important class of mean field methods, e.g. Hartree–Fock and Kohn–Sham (KS) density functional theory (DFT), linear scaling methods have to address the build-up of the Hamiltonian matrix (KS matrix) and the solution of the self-consistent field (SCF) equations (see Fig. 1). Many different algorithms for these two tasks have been proposed and detailed discussions can be found in a recent review.[2] Here, we concentrate on methods for the solution of the KS equation, i.e. replacements for the KS matrix diagonalization, that directly calculate the one-particle density matrix and are implemented in the CP2K simulation package.[3,4] The impact of such methods on the computational cost of three-dimensional systems can be inferred from the curves in Fig. 2 where the simulation time for the calculation of the electronic structure of bulk liquid water for conventional, cubically scaling, and linear scaling methods is compared.

The most basic algorithm investigated is based on the matrix sign function that can be defined

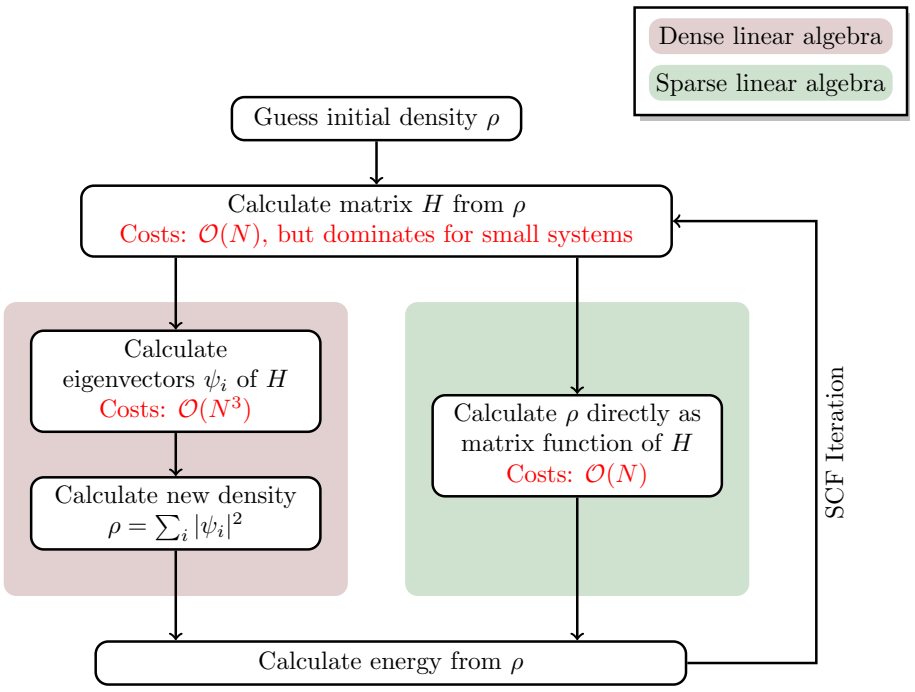

Figure 1: Workflow for a self-consistent electronic structure calculation, illustrating both the use of traditional $O(N^3)$ as well as $O(N)$ methods.

as

$$\text{sign}(A) = A(A^2)^{-\frac{1}{2}}. \tag{1}$$

For diagonalizable A, eigenvectors of A are eigenvectors of $\text{sign}(A)$, with the eigenvalues of $\text{sign}(A)$ being -1 or 1 for negative or positive eigenvalues of A respectively. Various simple iterative algorithms are available to compute the matrix sign function[5] and these approaches have found early application.[6,7] These algorithms converge super-linearly and are numerically stable. The simplest form, which only requires two matrix multiplies per iteration, is (I is the identity matrix)

$$X_{n+1} = \frac{1}{2}X_n(3I - X_n^2). \tag{2}$$

For $X_0 = cA$ and $c < ||A||^{-1}$ this iteration converges quadratically to $X_\infty = \text{sign}(A)$. The convergence criterion employed terminates the iteration at X_{n+1} if $||I - X_n^2||_F < \sqrt{\varepsilon_{filter}}||X_n^2||_F$ where $||.||_F$ is the Frobenius norm. Since the algorithm is quadratically convergent, near convergence, each iteration will approximately double the number of correct digits in the solution.

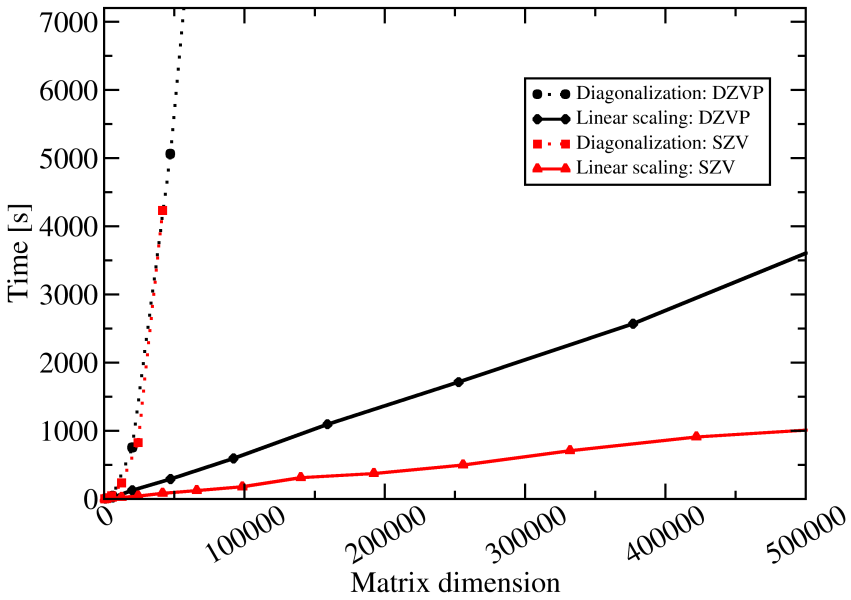

Figure 2: Direct comparison of the time needed for calculations on bulk liquid water using linear scaling and diagonalization based SCF procedures. For matrices larger than 20000 basis functions a speedup is observed (filtering threshold 10^{-5}). Note that for linear scaling approaches, the time needed does not only depend on the matrix size, but also on the sparsity pattern, and hence better quality basis sets typically have a larger relative cost.

Linear scaling results from the fact that all matrix operations are performed on sparse matrices, which have a number of non-zero entries per row that is independent of system size. In order to retain sparsity during the iterations a threshold (ε_{filter}) is employed to set small entries to zero after multiplication, thereby reducing the data volume and speeding up the following multiplies.

The density matrix P corresponding to a given Hamiltonian matrix H, overlap matrix S and chemical potential μ can be computed as

$$P = \frac{1}{2}(I - \mathrm{sign}(S^{-1}H - \mu I))S^{-1}. \tag{3}$$

The important idempotency ($PSPS = PS$) and commutativity ($SPH - HPS = 0$) conditions, equivalent to wavefunction orthonormality, are automatically satisfied. The number of electrons N_{el} is determined by the chemical potential μ, and can be obtained from $N_{el} = \mathrm{trace}(PS)$. S^{-1} is com-

puted conveniently using $S^{-1} = S^{-\frac{1}{2}} S^{-\frac{1}{2}}$ where the square root and inverse square root can be obtained from

$$\text{sign}\left(\begin{bmatrix} 0 & \text{A} \\ I & 0 \end{bmatrix}\right) = \left(\begin{bmatrix} 0 & A^{\frac{1}{2}} \\ A^{-\frac{1}{2}} & 0 \end{bmatrix}\right). \tag{4}$$

A stationary solution of the self-consistent equations can be obtained by a simple mixing approach:

$$\begin{aligned} P_{n+1} &= \frac{1}{2}(I - \text{sign}(S^{-1}\hat{H}_n - \mu_n I))S^{-1} \\ \hat{H}_{n+1} &= (1-\alpha)\hat{H}_n + \alpha H_{n+1} \end{aligned}$$

α is a mixing parameter between zero and one, and $\hat{H}_n$ an auxiliary matrix. The fixed point implies that $\hat{H}_n = H_n$ and thus $SP_nH_n - H_nP_nS = 0$. For each iteration, the total electronic energy (E_n) and Hamiltonian matrix (H_n) are computed from the density matrix P_n. The value of the chemical potential μ_n is determined by bisecting a suitable interval until $|\text{trace}(P_{n+1}S) - N_{el}| < \frac{1}{2}$, for a given N_{el}. Note that the $\text{trace}(P_{n+1}S)$ is integer-valued unless finite accuracy is employed in the calculation of the sign function. For a given SCF threshold (ε_{SCF}), the convergence criterion employed is $E_n - E_{n-1} < \varepsilon_{SCF} N_{el}$.

More advanced algorithms that still use fix point iterations exist. They include the optimization of the chemical potential[8] as part of the density matrix computation, and achieve faster convergence by relaxing absolute trace conservation[9]. These methods represent a significant advantage over the sign matrix iteration, if the chemical potential is not known in advance, as the cumbersome bisection can be omitted. Also trace resetting (TRS) purification starts from a normalized Hamiltonian matrix (X_0, eigenvalues in the interval $[0,1]$). The algorithm then calls for iterations where the update depends on the value of the quantity $\gamma_n = (N - \text{trace}(\mathscr{F}(X_n)))/\text{trace}(\mathscr{G}(X_n))$.

γ_n	update
$\gamma_n > \gamma_{max}$	$X_{n+1} = 2X_n - X_n^2$
$\gamma_n < \gamma_{min}$	$X_{n+1} = X_n^2$
$\gamma_n \in [\gamma_{min}, \gamma_{max}]$	$X_{n+1} = \mathscr{F}(X_n) + \gamma_n \mathscr{G}(X_n)$

The choice of the polynomial functions $\mathscr{F}$ and $\mathscr{G}$ is not unique, but an efficient algorithm

(TRS4) is achieved by using

$$\mathscr{F}(x) = x^2(4x - 3x^2) \tag{5}$$

$$\mathscr{G}(x) = x^2(1-x)^2 \quad . \tag{6}$$

For this choice of polynomials the values for γ_{min} and γ_{max} are 0 and 6, respectively.

Another class of algorithms aims at a direct minimization of the energy functional, avoiding the self-consistent mixing, and thus adding robustness. To achieve this, the constraints on the density matrix have to be included into the algorithm. In the work of Li, Nunes, and Vanderbilt[10] this was achieved by using an extended energy functional. Helgaker et al.[11] proposed a parameterization of the density matrix, that conserves idempotency. Within this curvy-step method[12,13], starting from an idempotent P_0, as obtained for example from the TRS method, one performs updates of the form

$$P_{n+1} = e^{-\Delta S} P_n e^{\Delta S} \quad , \tag{7}$$

where Δ is an anti-Hermitian matrix, $\Delta^\dagger = -\Delta$. This unitary transformation is evaluated using the Baker-Campbell-Hausdorff expansion

$$P_{n+1} = P_n + [P_n, \Delta]_S + \frac{1}{2}[[P_n, \Delta]_S, \Delta]_S + \frac{1}{6}[[[P_n, \Delta]_S, \Delta]_S, \Delta]_S + \cdots \quad , \tag{8}$$

where the commutator within a nonorthogonal basis is

$$[X, \Delta]_S = XS\Delta - \Delta SX \quad . \tag{9}$$

In the minimization, the matrix elements of the curvy-step matrix Δ are the free variables and are calculated from the energy gradient

$$\frac{\partial E}{\partial \Delta} = [H, P_n]_S \tag{10}$$

using for example a steepest descent, conjugate gradient, or a Newton-Raphson method.

All of the above algorithms have in common that matrix multiplication is the dominant operation. The performance of the underlying sparse matrix multiplication routines is of paramount importance for the overall computational efficiency.

DBCSR: a Sparse Matrix Library

The linear scaling SCF implementation in CP2K is centered around sparse matrix matrix multiplication.[3,4] This choice is motivated by the fact that matrix multiplication is a basic primitive that is suitable for a high performance parallel implementation. Furthermore, this operation can be used to compute matrix functionals, such as for example inv, sqrt, sign, and exponential. Surprisingly, no established software library is available that performs a parallel sparse matrix matrix multiplication. Such a library should, in the context of quantum chemistry, exploit the concept of sub-matrix blocks, rather than individual elements for a description of the sparsity pattern. These sub-matrix blocks, also named atomic blocks as they correspond to basis functions of an atom, are small (typical numbers are 5, 13, 23), and exploiting them is key to achieve good performance. Furthermore, as most calculations are currently performed near the cross-over regime between dense and sparse, the library must be highly efficient for relatively high occupations (for example 50% non-zero elements), and 10000s of non-zeros per row, while optimal performance for very sparse matrices (less than 1-5% non-zero elements) will become more important in the future. In order to address these needs, a general purpose sparse matrix library has been developed.[14] This library is currently distributed as part of the CP2K package, but it is our aim to provide a general purpose sparse matrix library that can ultimately be made available as a fully independent tool. The name of the library is DBCSR, which is an abbreviation for Distributed Blocked Compressed Sparse Row, or Distributed Blocked Cannon Sparse Recursive. The full names emphasize the storage format or the multiplication algorithm respectively. Data is stored distributed over all processes, using a blocked variant of the compressed sparse row storage format. The parallel algorithm to perform the matrix matrix multiplication is based on the Cannon scheme[15], which is optimal in the dense case, if memory is a limited resource. In particular, it guarantees that communication per process decreases with

increasing process count, and is free from all-to-all communication. These properties guarantee strong scaling of the algorithm, and good performance in the dense limit. Nevertheless, sparse matrix multiplication has $O(N)$ flops and $O(N)$ data, and reaching peak performance is thus difficult. The ratio of flops to data does not depend on system size, but rather on the number of non-zeros per row. The later depends typically on the accuracy of the calculation, such as tighter filtering thresholds and larger basis sets in our quantum chemical applications. We refer to Ref.[14] for an in-depth discussion. In the following, we focus on those aspects that are important in the context of GPU-acceleration, and on the recent developments that have enabled a significant increase in accelerated performance.

Software Architecture for GPU-acceleration

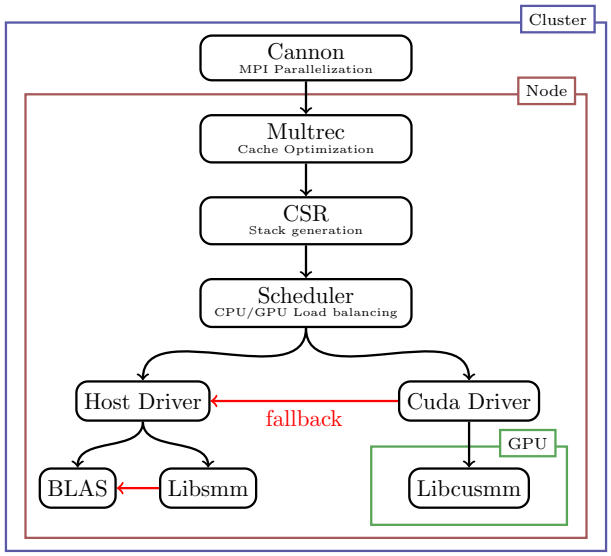

Figure 3: Shown is a schematic representation of the software architecture employed in the GPU accelerated DBCSR library. The various layers correspond to key steps in the matrix multiplication algorithm. While the Cannon layer is essential for the parallelism between processes or on the cluster level, the lower layers deal with parallelism and acceleration on the node level.

In this section, we outline the various layers of the DBCSR matrix multiplication architecture. It has been designed to decouple the various steps of the calculation, and is schematically shown in Fig. 3. As we go down the layers, the granularity of the data becomes smaller and the compu-

tational workload increases. However, the higher level layers manage data transfers, optimize data access, and enable asynchronous progress. These steps are essential to fully benefit from the high performance that modern CPUs and GPUs offer.

Cannon Layer

The top-most layer deals with the parallelization of the matrix multiplication over the nodes of a cluster, and enables good parallel performance by managing the message passing between MPI processes. One MPI process can consist of several CPU threads, based on OpenMP, and can off-load to a dedicated or shared GPU. For the MPI-parallelization, the sparse matrices are divided into large sparse sub-matrices named panels. These panels are regular in shape, and by a suitable row and column permutation, the sparsity pattern has been homogenized so that all panels contain approximately the same amount of data, which is favorable for load-balancing the calculations. These panels are distributed over a regular 2D grid of processes and *Cannon's algorithm*[15] is used to communicate these panels between processes in a regular and ordered fashion, according to 'ticks' in the Cannon metronome. In each tick of Cannon's algorithm, each process sends and receives two panels, multiplies the two panels that are available, and accumulate the results locally. As discussed below, messages can be processed asynchronously, i.e. be in transit over the network, while computation takes place. The panel data are also uploaded to the GPU in this layer. Our approach to enable the asynchronous message passing and uploads will be discussed in more detail below. The following, lower layers deal with the process or node local multiplication of panels.

Multrec Layer

The *multrec* layer, is a high level process-local layer that aims at optimizing memory access, and in particular at exploiting the deep cache hierarchy of modern processors. Indeed, even for a standard dense matrix multiplication, optimal data reuse is essential to reach good performance. Usually, detailed knowledge of the architecture can be combined with the well known data-access

pattern of a dense matrix multiplication to optimally block matrices and to guarantee best cache reuse. However, due to the unknown sparsity pattern and relatively complicated data structures, this approach is not general enough in the sparse case. An alternative technique, also derived in the context of dense matrix multiplication is therefore employed, which instead uses a recursive approach to matrix multiplication.[16] Matrices are multiplied by recursively dividing the longest dimension of the matrix in two, until sufficiently small matrix dimensions have been obtained and all the data fits fully into a low-level cache. This *cache-oblivious algorithm* results in a near-optimal data access pattern for dense matrices, without explicit knowledge of the cache hierarchy, and is easily adapted to the sparse case.

CSR Layer

The compressed sparse row layer, or CSR layer, determines from the CSR data which blocks have to be multiplied. It is important to emphasize that the sparsity pattern of the result matrix is not fixed or known a priori, so that this processes is driven by the right-hand-side of the equation ($C = AB$). In DBCSR, a two step approach, well suitable for GPUs, has been adopted. It separates performing the actual floating point operations from the indexing and book-keeping. The CSR layer performs the latter, on the host, deferring flops to lower layers. During the indexing, lists of needed block-multiplications, named 'stacks' are generated, and passed on to the lower scheduling layers, i.e. are flushed, as soon as the limited space of a stack is exhausted, or the end of a Cannon tick is reached. In order to allow for efficient processing, so-called homogeneous stacks are employed for the most common block sizes, these contain entries that have all the same block-dimensions, while a default stack contains the remaining cases. An important optimization has been introduced in this layer namely on-the-fly filtering. This optimization employs precomputed matrix block norms to decide if a given block product contributes to the final result significantly in comparison to the sparsity threshold, and skips negligible multiplications. In actual applications, even for matrices that are dense in data, this optimization can reduce the number of needed flops by a factor two to four. The relative computational cost of the indexing operations depends

strongly on the size of the basic blocks employed in the application calling the DBCSR library. It is significant if blocks are as small as 5x5, while it is clearly negligible if blocks are of size 23x23 or larger.

Scheduler and Driver Layers

The scheduler layer receives filled stacks and arranges for their processing by handing them off to one of the drivers. The host-driver is employed for CPU-processing and the cuda-driver for GPU-processing. Both the host and device drivers are built on top of libraries that efficiently perform small matrix multiplications, libsmm and libcusmm for host and device respectively. libsmm has been described in Ref.[14], while libcusmm is described in detail in a following section. Both libraries are significantly more efficient than standard matrix multiplication libraries for the small matrix sizes that are relevant for quantum chemical applications. The scheduler decides where the stack will be processed, and is currently basd on a very simple scheme. The GPU is queried using the event based mechanism described below, and if buffer space is available on the GPU the stack will be handed over to the cuda-driver, otherwise the host-driver processes the stack. The amount of buffer space made available on the device is thus a mechanism to tune the host-device load balancing. Following this, the cuda driver will check if highly tuned kernels are available in the libcusmm library for the particular matrix sizes in the homogeneous stack. If so, the stack is shipped to the GPU for processing, and otherwise is send to the host driver. The latter can deal with small matrix multiplications of all sizes, ultimately falling back to an optimized BLAS library calling DGEMM.

Maximizing Asynchronous Progress

Part of the challenge in writing efficient GPU-accelerated code, is to exploit the asynchronous task based programming model. Whereas on a homogeneous system typically all processors execute the same program on different parts of the data in a lock-step fashion, on a hybrid system the CPU

and GPU complement each other, and are partially independent. In order to fully utilize such a system, different programs need to be executed on the CPU and the GPU. Typically, the host-CPU drives the GPU-device by handing over tasks, and while the GPU is executing these tasks, the CPU can perform other tasks on its own. In the following subsections, our approach to enable this asynchronous processing is explained.

Cuda Streams and Events

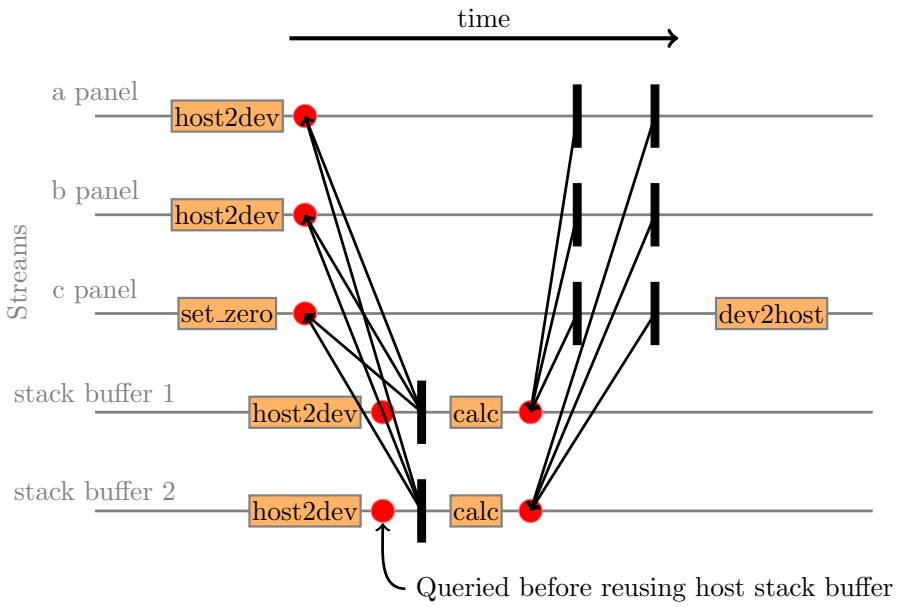

Figure 4: Enabling concurrency and enforcing dependencies in DBCSR. Multiple streams are used to transfer data from the host to the device, and to process independent stacks. Dependencies between the streams, for example a panel upload and stack calculations, and between host and device, for example device buffer reuse, are enforced using events.

Once the host has submitted a task to the device, the CPU loses control over it the GPU has significant freedom to schedule the task execution. However, dependencies between tasks might be present. For example, a task processing some data might depend on the completion of a prior task that copies this data from the host to the device. These dependencies have to be made explicit by the programmer. The Cuda programming environment provides two powerful mechanisms to enable further concurrency and to enforce dependencies: *streams* and *events*. Streams are a simple mechanism to establish dependencies and to enable concurrency. Tasks submitted to a given stream

are processed in the order in which they are submitted, while tasks from different streams can be processed in any order or concurrently. Using multiple streams is essential to overlap computation with host-to-device or device-to-host transfers, and to enable concurrent task execution. Events can be used to express more general dependencies. Just like a task, an event can be created and submitted to a stream, and is processed after the previous task submitted to the same stream is completed. However, tasks can be submitted that wait for the completion of events in other streams. These "waiting-tasks" will block a stream until the referred event has occurred, and by submitting waiting-tasks prior to an actual task on the same stream, multiple cross-stream dependencies can be enforced.

In Fig. 4, the scheme that is employed in the DBCSR library is illustrated. Stack buffers are transferred and processed in a number of independent streams, so that the stack buffer transfers can overlap with computations in other streams, and that concurrent stack processing is possible. The GPU can only process stacks if the panel and stack data is present, so that for each kernel dependencies on the completed transfer of the A, B, and *C* panels, taking place in different streams, and completion of the stack buffer transfer, in the same stream, must be present. Retaining the A, B, and *C* panels on the device while stack buffer processing is in progress is enforced with additional events. Notice that explicit synchronization between host and device is rarely needed, the host can query events to make sure that, for example, stack buffers have been uploaded before they are overwritten with new data. The host only has to wait for the device when the previous panel is still in use, and at the very end when the final results are downloaded from the device.

The cuda API allows for an unlimited number of stream, but these are mapped to a limited number of *hardware queues*. Both the number or hardware queues and the mapping scheme are likely to change depending on the hardware and cuda version. Unfortunately, mapping of otherwise independent streams to the same hardware queue can lead to unwanted serialization. Therefore, only a limited number of streams is created in DBCSR, specifically, two streams are exclusively for host-to-device transfers, one for odd Cannon ticks and one for even ticks, while a configurable but small (typically 2-4) number of streams is used for stack transfers and kernel launches. Fi-

nally, 'priority streams' are a recent cuda feature that introduces some way for the programmer to influence scheduling of kernels. In DBCSR this feature is used to load balance between host threads. In addition to generating stacks, occasionally a host threads will also process a stack. This happens when a host thread has no more free stack buffers available, i.e. when the device is busy. In order to avoid that the device works on buffers of a thread that has finished its work already, and a busy thread looses time processing stacks, stack buffers come in two flavors: priority buffers and posterior buffers. A limited number of priority buffers is assigned to each thread, and mapped to a stream with high priority, while the posterior buffers are mapped to streams with lower priority. The effect of this is that the device will focus on doing the work for those threads that are actively generating stacks, i.e. writing them to the priority buffers, while the posterior buffers are handled later. These buffers, as discussed below, are useful to overlap computation and communication during message passing or host to device transfers. Good performance requires that the number of priority buffers is tuned such that device never idles if all threads are active and using priority buffers only.

Double Buffered Cannon on Host and Device

In a sparse matrix multiplication algorithm, both data movement and floating point operations can contribute significantly to the total runtime. Maximum performance can only be achieved when the corresponding resources are utilized in parallel. To accomplish this, a double buffered scheme has been employed for both host and device. As shown in Fig. 5, these two panel buffers are used in a complementary fashion, while one buffer is used for the computation, the other buffer is overwritten as part of a data transfer operation. Data transfer happens between MPI processes and between host and device, and thus double buffering is required for both operations. The host buffers are alternately used for MPI-send and MPI-receive. Once a panel has been received on the host it is copied to the corresponding device buffer, using the asynchronous *host-to-device* copy operation. At the same time the CPU threads start to generate and fill stacks buffers. Stack buffers are transferred and processed by the device as soon as the host-to-device panel copy has finished.

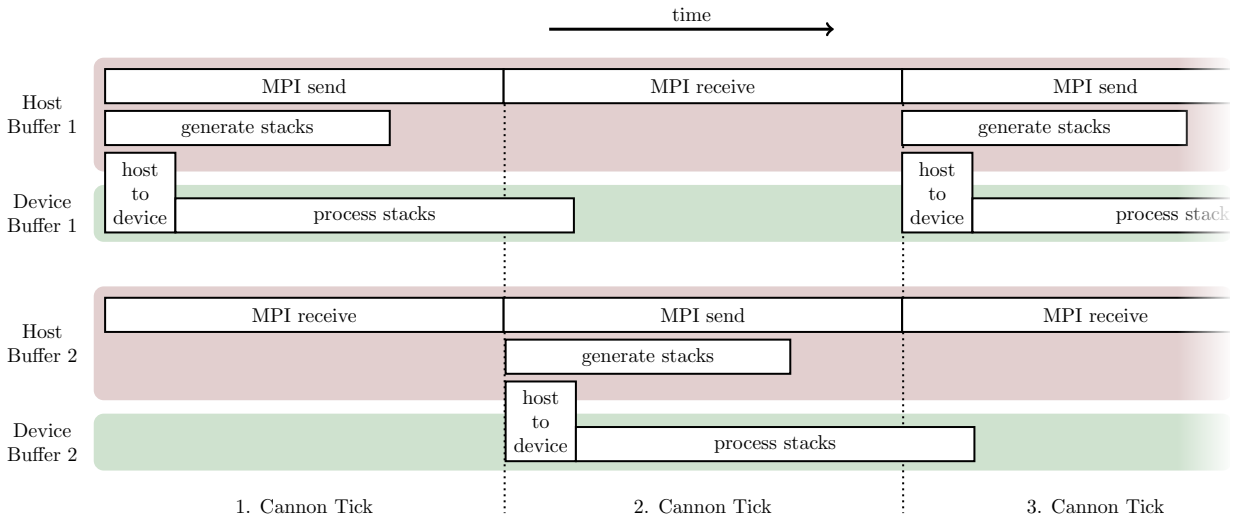

Figure 5: Schematic representation of the double buffered Cannon algorithm, which illustrates how the use of two host and two device buffers for the A and B panels enables overlapping of message passing, host to device memory copies and computations. The ratio of the time needed for the important steps of the algorithm, depends on the hardware and on the science problem at hand.

Typically, the CPU threads can generate the stacks faster than the GPU can process them, and a large number of stack buffer can be outstanding. These outstanding stack buffers can be processed by the device while the MPI transfer and the host to device copy of the next panel to the second buffer is taking place. Good performance requires that the number of posterior buffers is tuned such that device never idles during these transfers. A too large number of posterior buffers might lead to host threads waiting for the previous device panel buffer to finished. In the last Cannon tick, posterior buffers are not employed, as threads and device should finish roughly at the same time.

Fast device-to-host transfers require host-pinned memory. Since allocating host-pinned memory and cuda device memory are slow operations, and memory usage is hard to predict in the case of varying sparsity patterns, memory-pools have been introduced that are persistent across sparse multiplications and only allowed to grow. In our application, the gain in performance outweighs the additional complexity and the fact that less memory is available for the rest of the application in between matrix multiplications.

Finally, whereas the MPI standard specifies non-blocking versions of send and receive (`isent`

/ `irecv`), actual implementations often perform the complete transfer in the corresponding `wait` statements. We have found that this is in particular the case for multi-megabyte messages, as is required for the panel transfers. To nevertheless overlap computation and communication, a 'communication thread' has been introduced in the OpenMP parallel version of the DBCSR library. The master thread, which is responsible for all communication, is underloaded compared to the other threads, and will, given the barrier free nature of the implementation, enter early in a polling loop based on `test_any` to progress outstanding MPI communication. Tuning the load of the master thread, message passing can be effectively overlapped with computation performed by the other threads and the device.

Libcusmm: GPU Accelerated Small Matrix Multiplications

The core computational kernel in DBCSR is the computation of stacks of small matrix multiplications. The result block matrix $C^{u,v}$ is computed as the product of the block matrices $A^{u,w}$ and $B^{w,v}$ according to

$$C^{u,v}_{i,j} = C^{u,v}_{i,j} + \sum_{w,k} A^{u,w}_{i,k} B^{w,v}_{k,j} \tag{11}$$

using superscripts to indicate the matrix block indices and subscripts to denote the matrix elements in each of the block matrices. The sum over w takes the sparsity pattern of A and B into account, i.e. the product will be omitted whenever either $A^{u,w}$ or $B^{w,v}$ is absent, or their norms are small. Furthermore, w can only refer to those parts of A and B that are part of the panels of A and B that are local to the node for a given tick of Cannon's metronome. In a single stack, anywhere between one and a few tens of products will be present for a given block $C^{u,v}$. Note that this operation resembles the batched DGEMM operation in CUBLAS, but that this library expects all C matrices in a single batch to be different, and can thus not be used. In the following, the steps necessary to optimize these products on GPUs are described.

Small Matrix Multiplication Performance Model

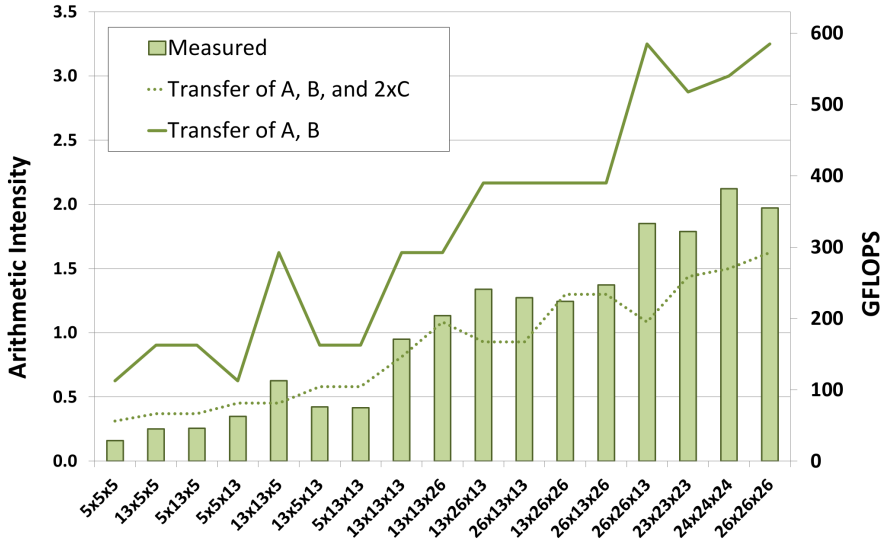

Figure 6: Minimum (dotted line) and maximum (solid line) arithmetic intensity for different matrix sizes commonly employed in CP2K simulations, and the corresponding maximum possible flop rate. The performance as obtained from individual kernel launches in a mini-app is shown as green bars.

At first sight, matrix multiplication seems dominated by floating point operations, while memory transfer is less important. This certainly is the case for large matrices, but not quite for the small matrices required in the current context. It is therefore useful to look at the arithmetic intensity, which we define to be the ratio of number of floating point operations vs. number of bytes transferred between memory and processing units. In order to perform the matrix multiplication, A and B will need to be loaded from the device memory to the streaming multiprocessor (SM), while C might be assumed present on the SM (favorable limit of a large number of contributions from the summation over w), or might need to be loaded and stored as well. For the multiplication of an $m \times k$ by a $k \times n$ matrices, the intensity is thus between $\frac{2mnk}{8(mk+kn)}$ and $\frac{2mnk}{8(mk+kn+2mn)}$. In order to reach the favorable limit, the DBCSR library might sort the stacks, such that C matrix access occurs in order, prior to handing them to the GPU. Given a K20X GPU with 1.3TFlops peak double precision performance and 250GB/s peak bandwidth, an arithmetic intensity of at least 5.2 is needed to achieve peak performance. With ECC turned on, a bandwidth of 180BG/s is more realistically

achievable for a kernel of this complexity, so an arithmetic intensity of at least 7.2 is needed to reach peak performance. Multiplications of matrices smaller than 60x60 are thus necessarily limited by the memory bandwidth, and this remains an important factor, even for significantly larger matrices. This clearly implies that the optimization should focus on reaching optimal memory bandwidth usage. For selected sizes of the small matrices encountered in CP2K applications, the arithmetic intensity, the reachable flop rate, and the actually achieved performance are shown in Fig. 6.

Matrix-Product Algorithm Choice

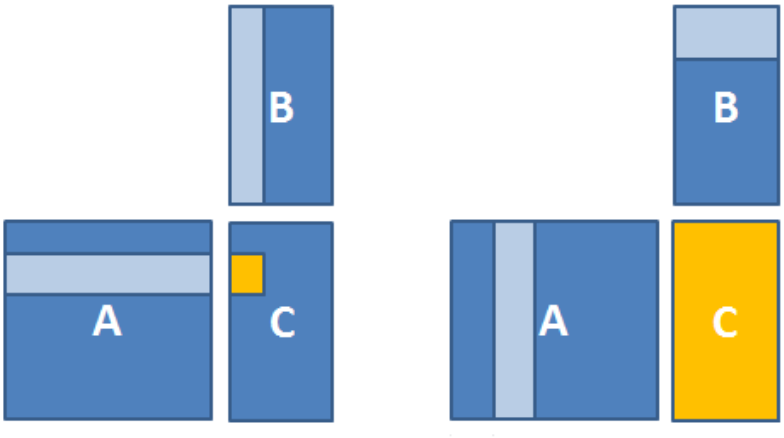

Figure 7: Inner-product (left) and outer-product (right) form of matrix multiplication. The yellow areas indicate elements that can be computed independently by accessing the highlighted areas of A and B.

The first step in implementing the small matrix products is to pick the most appropriate algorithm. Fig. 7 shows two possible algorithms for computing the matrix product ($C = C + AB$). In the canonical form, the result elements in C are computed using the inner product of rows of A and columns of B, while an alternative algorithm is based on an outer product of columns of A and rows of B. These two algorithms result in the same number of floating point operations, but the latter option exhibits significantly more parallelism in that it allows for computing an update for all elements of C using a single column of A and a single row of B. An additional benefit of using

outer products is data locality, the outer product algorithm touches elements of A and B only once, while for the inner products, when computing one row of C, one row of A and the entire matrix B needs to be accessed. Based on the model developed in the previous section it is known that the kernel's performance for problem sizes of interest to CP2K will be limited by memory bandwidth. The outer products algorithm is therefore preferred.

GPU Implementation: Generic Algorithm

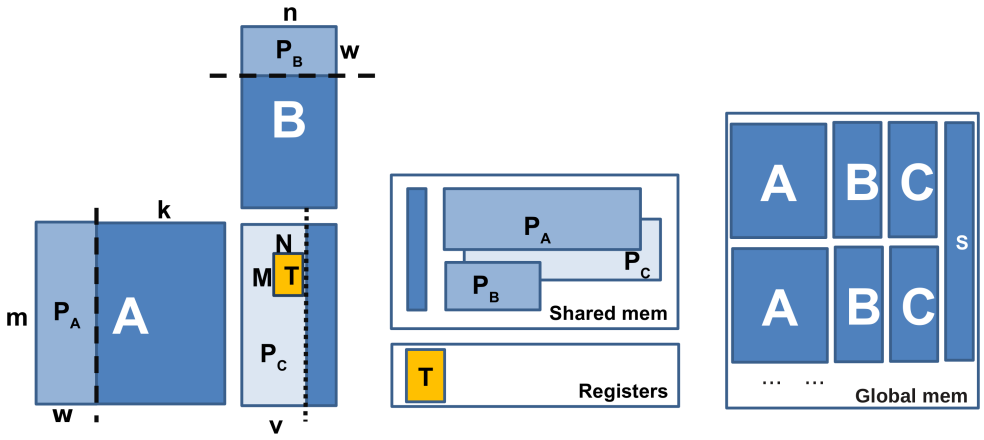

Figure 8: Left: Parameterization of the m×n×k-matrix product $C = C + AB$. Each thread computes an M×N tile (T) of the result matrix C. In order to accommodate matrix sizes larger than the available shared memory, matrices are processed in slabs (P_A, P_B), with an input slab width w. In order to optimize the data output, the matrices (P_C) are written back using the output slab width v. Middle: close to the SM, registers are used to store the C matrix tile, while slabs of A, B, and C are stored in shared memory Right: GPU memory stores all panel data, including the various blocks of A, B, C and the stack buffers S.

The next step in the design of a kernel for small matrix products is to consider data locality. Initially, the A, B, and C matrices, as well as the product descriptors, the so-called stacks, are all located in global memory on the GPU. Each entry in the stack describes one matrix-matrix product, thus containing three pointer to the blocks in the A, B, and C panels. After the kernel has read a stack entry, it fetches the matrices A and B, and updates the C matrix with the product of A and B. The matrix sizes of interest correspond to typically 10 – 1000 elements per result matrix C, limiting the degree of parallelism to a similar order. An appropriate choice is therefore to process a

matrix product using a single thread block. While this allows for efficient synchronization between the threads processing one product, it requires appropriate safe-guards to avoid data races between multiple updates of the same C matrix block. Multiple consecutive products updating the same result matrix C can be processed by the same thread block, requiring fewer reads and writes of C from global memory. In addition to reducing the number transfers of C between global memory and the SM, this also reduces the probability of collisions that happen when multiple thread blocks update the same C matrix block at the same time. On Kepler-Generation GPUs, atomic memory operations are efficient enough and are hence used to prevent data races. In this context, the overhead of using atomics instead of regular memory updates is on the order of 5%.

How a single thread block deals with the data is illustrated in Fig. 8 and explained in the following. First, given that the elements of the result matrix C do not need to be shared between threads, the ideal location to store C is registers. In order to increase instruction level parallelism per thread, and given the large number of registers available per thread, a small tile (T) rather than a single result matrix element is processed per thread. The optimal choice of tile dimensions (MxN) is determined via auto-tuning as described later. Next, the elements of matrix A and B need to be accessed by multiple threads, thus making them ideal candidates to be stored in shared memory. In order to avoid that shared memory utilization limits the number of concurrent thread-blocks (occupancy), a maximum of 3kB per thread-block can be used. For larger blocks A and B, it is thus desirable to read only parts into shared memory, we name these parts slabs. Using the outer product algorithm, these matrix slabs only need to be read once per product. The optimal width (w) of the slabs is also determined by auto-tuning. Given that matrices are stored in column major format, reading a slab of A still leads to perfectly coalesced memory access. However, reading only a slab of matrix B can lead to both significant memory access penalties and complex address computations. In order to avoid these penalties it is therefore desirable to compute the product $C = A(B^T)^T$ instead, resulting in perfectly coalesced memory accesses with simple address computations for both A and B. Given that typically each B matrix is used many times per Cannon tick, the cost of transposing the full panel of B's once after the upload in a separate kernel is negligible

compared to the time savings due to more efficient memory access and simplified address calculation. Finally, once the entire product is computed, and only when the next stack entry refers to a different C block, the results are added to the corresponding block in global memory. In order to ensure coalesced writes, an intermediate step is employed, in which slabs of C (of width v) are first put in shared memory, and only then added using an atomic compare-and-swap operation.

Auto-tuning and Performance

The generic algorithm outlined above requires several parameters (M, N, w, and v), and finding an optimal set of values is not always intuitive. Fetching larger panels of A and B tends to improve performance, but at the same time will also increase the shared memory footprint and limit occupancy, thus potentially limiting the amount of latency hiding. A similar effect occurs for the number of result elements processed per thread: increasing this parameter improves the instruction level parallelism, but at the same time this limits the number of thread blocks resident on each SM. Additionally, some matrix sizes allow for significantly simplified versions of the general kernel and separate implementations were developed. In order to hide details of the tool chain, such as register allocation, that are unknown or subject to change, an autotuning framework based on a small standalone benchmark application is used to find optimal parameters and implementations for each given set of block dimensions m,n,k. It has been verified that the kernel performance in the small standalone benchmark application is very similar to the one observed in full CP2K simulations.

Fig. 6 shows the performance obtained in the mini-app for relevant blocks sizes and optimal parameters. The performance is close to that estimated from the model based solely on memory bandwidth considerations. For very small matrices, the measured performance starts to deviate from the theoretically expected performance. We currently attribute this to the warp granularity of the execution on the SM, but have not further optimized for these sizes as we expect that small matrix sizes can just be handled on the CPU side if needed. Finally, for comparison, batched DGEMM in cuBLAS (version 5.0) for a 23x23x23 problem runs at 132 GFLOPS on a K20X,

while the current implementation in libcusmm achieves about 322 GFLOPS. For most of the small matrix sizes of interest, a speedup in the range of 2-4x has been measured. This demonstrates the quality of the generated kernels, and the appropriate choice of optimization techniques.

Benchmarks and Conclusions

In this final section, we illustrate the performance of the linear scaling GPU based implementation. In doing so, we attempt to cover synthetic benchmarks, current application style simulations, as well as very large scale simulations. Given the computational demands of these simulations, the focus is on parallel application of CP2K. The latter calculations have been performed on a recent hybrid architecture, a Cray XC30, which was installed in the fall of 2013 at CSCS, Switzerland. This machine is named Piz Daint, and is currently the leading European computer in the Top500 list. It features 5272 hybrid compute nodes, with one Intel Xeon E5-2670 processor (8 core, Sandy Bridge), and one NVIDIA K20X per node. The nodes are connected with an aries network based on a dragonfly topology.

As a first demonstration of performance, we focus in two synthetic benchmarks on the matrix-matrix multiplication of nearly dense matrices (occupied 50% or more), with favorable block sizes (23x23). A single node CPU-GPU comparison is shown in Fig. 9. In this case, the potential of the GPU is demonstrated, as it outperforms 12 Sandy Bridge cores by a significant factor. Furthermore, increasing the number of CPU cores, the hybrid implementation displays improving performance, showing that host-device sharing is effective. The CPU-only curve demonstrates good parallel efficiency of the OpenMP code. Taking this benchmark to the scale of 5184 hybrid nodes, a matrix of size 536544 x 536544, with 50% occupation, and 23x23 subblocks, can be multiplied in approx 36 seconds with a sustained machine performance in excess of 2 Pflops (nearly 400 Gflops per node). Thus, exploiting the fact that the matrices are 50% occupied, already brings a speedup over a dense matrix multiplication. Indeed, assuming a dense parallel matrix multiplication to run at 6.2 Pflops (the Linpack number for Piz Daint), such a calculation would require 50s. This performance

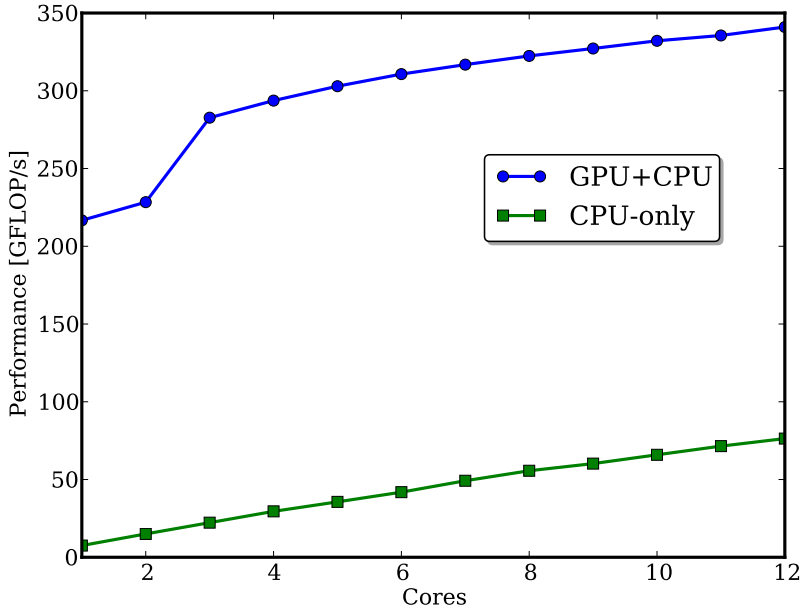

Figure 9: Performance comparison of the multi-threaded DBCSR library based on 23x23 matrix blocks, and was not using the MPI capabilities. The benchmark was run on a dual Sandy Bridge (E5-2620, 2.0GHz, 6 cores) machine, equipped with one NVIDIA Tesla K20 card.

illustrates the quality of the parallel implementation of the sparse matrix code.

More important is application level performance for realistic simulation setups. In order to assess this, we employ three benchmarks that are also part of the CP2K distribution, named amorph, H2O and TiO2. These describe an amorphous organic hole conducting material, bulk liquid water and Titanium dioxide nanoparticles respectively. Geometries are realistic, disordered, three dimensional, and with periodic boundary conditions. Basis sets are of double zeta quality (DZVP-MOLOPT-SR-GTH) and include diffuse primitives, contraction based on molecular optimization makes them at the same time accurate and suitable for linear scaling calculations in the condensed phase.[17] Since these benchmarks are designed to run quickly on a relatively small number of compute nodes they have reduced SCF counts. Key quantities and results are provided in Table 1. First, for the given basis sets and thresholds, it shows that systems of approximately 10000 atoms can be computed in minutes using 169 nodes (only 4% of the national supercomputer). This opens the way for scientific applications based on models of this size, including geometry optimization

Table 1: Key quantities of three linear scaling benchmarks that are distributed with CP2K. The run time is provided for two setups, one in which 2 Sandy Bridge (SB) CPUs are present per node, and a hybrid architecture in which 1 SB and 1 K20X GPU is present per node. Performance ratio compares the run time between these setups, GPU flop % gives the percentage of flops that is executed on the GPU in the hybrid setup.

	Amorph	H_2O	TiO_2
number of atoms	13846	20736	9786
number of basis functions	133214	158976	169624
block sizes	5, 13	23	13, 26
number of SCF steps in benchmark	2	2	1
filtering threshold	10^{-6}	10^{-6}	10^{-5}
typical matrix occupation %	16	11	12
run time on 169 x 2 SB [s]	372	275	446
run time on 169 x 1 SB + 1 K20X [s]	272	187	263
performance ratio on 169 nodes	1.4	1.5	1.7
GPU flop %	92	99	88

and molecular dynamics based relaxation. Second, this comparison at 169 nodes shows a speedup of 1.37–1.70 going from a traditional homogeneous node to a hybrid node. In these cases, the GPU is processing most of the flops. More detailed analysis shows that for the amorph benchmark the small block sizes limit the speedup, while the H_2O testcase is already limited by MPI communication.

The largest system computed so far on the hybrid system Piz Daint is shown in Fig. 10. It is consists of aggregated nanoparticles of TiO2 in an explicit acetonitrile solvent, as found in dye sensitized solar cells, and requires 77538 atoms and 772868 basis functions. For a filtering threshold of 10^{-6} a matrix occupation of 4% is found. Running on 5184 nodes, a single SCF step takes approximately 122s. Performance is roughly 30 Gflops per node, as the calculation is strongly dominated by MPI communication. The GPUs perform 99.4% of the flops.

To conclude, we have shown that linear scaling SCF calculations with good quality on large three dimensional systems have become possible with good time to solution. As such, linear scaling approaches on large models have become one of the many tools that atomistic simulation offers to investigate an ever increasing range of systems. The progress can be attributed to an evolution and interplay between hardware, algorithms and implementations. The GPU work presented here

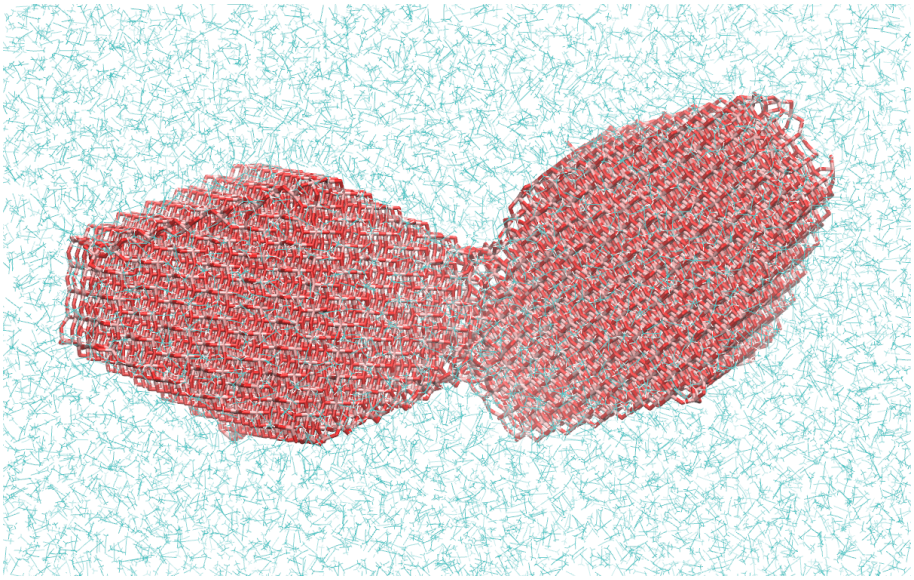

Figure 10: Aggregated nanoparticles in explicit solution (77538 atoms) can be run on the Piz Daint computer (5272 hybrid compute nodes) at approx. 122s per SCF step.

is a prime example. To harvest the raw power of GPUs, suitable algorithms had to be adopted and a very careful implementation was needed. In particular, the asynchronous nature of the device had to be taken into account at a sufficiently high level, and a library of highly optimized kernels had to be created. This required a detailed understanding of the GPU device and the application programming interface. Finally, good performance has been demonstrated on one of the largest GPU based supercomputers worldwide. To further benefit from the GPU compute power, new message passing algorithms are being developed.

Acknowledgements

The authors acknowledge Urban Borštnik, Florian Schiffmann, Florian Thöle, Jinwoong Cha, Valéry Weber, Christiane Pousa Ribeiro, Iain Bethune (EPCC), Chris Mundy (PNNL), Nikolay Markovskiy (NVIDIA), Neil Stringfellow (CSCS), Gilles Fourestey (CSCS), Alfio Lazzaro (Cray) and Roberto Ansaloni (Cray) for their help with the implementation, discussions and applications. J.V. acknowledges financial support by the European Union FP7 in the form of an ERC Starting Grant under contract no. 277910. Calculations were enabled by a grant of the Swiss National Su-

percomputer Centre (CSCS) under project IDs s238, s441, and h05. In preparation of this research, resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725, have been used. The research leading to these results has received funding from the Swiss University Conference through the High Performance and High Productivity Computing (HP2C) Programme.

References

(1) Goedecker, S. *Rev. Mod. Phys.* **1999**, *71*, 1085–1123.

(2) Bowler, D. R.; Miyazaki, T. *Rep. Prog. Phys.* **2012**, *75*, 036503.

(3) The CP2K developers group, CP2K is freely available from: http://www.cp2k.org/, 2013.

(4) VandeVondele, J.; Borstnik, U.; Hutter, J. *J. Chem. Theory Comput.* **2012**, *8*, 3565–3573.

(5) Higham, N. J. *Functions of Matrices: Theory and Computation*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2008; pp xx+425.

(6) Beylkin, G.; Coult, N.; Mohlenkamp, M. *J. Comput. Phys.* **1999**, *152*, 32–54.

(7) Nemeth, K.; Scuseria, G. *J. Chem. Phys.* **2000**, *113*, 6035–6041.

(8) Palser, A.; Manolopoulos, D. *Phys. Rev. B* **1998**, *58*, 12704–12711.

(9) Niklasson, A. M. N.; Tymczak, C. J.; Challacombe, M. *J. Chem. Phys.* **2003**, *118*, 8611–8620.

(10) Li, X.; Nunes, R.; Vanderbilt, D. *Phys. Rev. B* **1993**, *47*, 10891–10894.

(11) Helgaker, T.; Larsen, H.; Olsen, J.; Jorgensen, P. *Chem. Phys. Lett.* **2000**, *327*, 397–403.

(12) Shao, Y.; Saravanan, C.; Head-Gordon, M.; White, C. *J. Chem. Phys.* **2003**, *118*, 6144–6151.

(13) Salek, P.; Host, S.; Thogersen, L.; Jorgensen, P.; Manninen, P.; Olsen, J.; Jansik, B.; Reine, S.; Pawlowski, F.; Tellgren, E.; Helgaker, T.; Coriani, S. *J. Chem. Phys.* **2007**, *126*, 114110.

(14) Borstnik, U.; VandeVondele, J.; Weber, V.; Hutter, J. *Submitted* **2013**, *0*, 0.

(15) Cannon, L. *Montana State Univ., Bozeman, MT, USA* **1969**,

(16) Chatterjee, S.; Lebeck, A.; Patnala, P.; Thottethodi, M. *Parallel and Distributed Systems, IEEE Transactions on* **2002**, *13*, 1105–1123.

(17) VandeVondele, J.; Hutter, J. *J. Chem. Phys.* **2007**, *127*, 114105.