Received –; | Revised –; | Accepted –

DOI: xxx/xxxx

ARTICLE TYPE

Frequency Scaling and Energy Efficiency regarding the Gauss-Jordan Elimination Scheme with Application to the Matrix-Sign-Function on OpenPOWER 8

Martin Köhler* | **Jens Saak**

[1]Computational Methods in Systems and Control Theory, Max Planck Institute for Dynamics of Complex Technical Systems, D-39104 Magdeburg, Germany

Correspondence
*Email: koehlerm@mpi-magdeburg.mpg.de

Summary

The Gauss-Jordan Elimination scheme is an alternative to the LU decomposition for solving linear systems or computing the inverse of a matrix. We develop a multi-GPU aware implementation of this algorithm on an OpenPOWER 8 system with application to the Matrix Sign Function. Thereby, we analyze the influence of the CPU clock frequency scaling on the overall energy consumption. The results show possible energy saving of 14.2% without a noteworthy increase of the runtime.

KEYWORDS:
Gauss-Jordan-Elimination, Matrix Sign Function, GPU Acceleration, Energy Efficiency

1 | INTRODUCTION

The solution of the linear system $Ax = b$, $A \in \mathbf{R}^{m\times m}$, is one of the most fundamental tasks in numerical linear algebra. It is usually performed using the LU decomposition avoiding the explicit computation of the inverse A^{-1}. However, some tasks, like Newton's Method for computing the Matrix Sign Function (1, 2, 3, 4) or the Polar Decomposition (5), require the explicit inverse, where our main focus lies on. In those cases, one can either use the three step scheme implemented in LAPACK (6) on top of the LU decomposition, or the Gauss-Jordan Elimination (7) to obtain A^{-1}.

The LAPACK approach works in three steps:

1. compute the pivoted LU decomposition of A using `GETRF`,
2. invert the upper triangular matrix U using `TRTRI`,
3. and solve the linear system $LA^{-1} = U^{-1}$ for A^{-1}.

This procedure requires $2m^3$ flops and at least three sweeps over the memory location of A, if the algorithms work in-place as in LAPACK. Furthermore, in this approach it is necessary to find optimal implementations for all three routines. Moreover, the two steps working on triangular matrices are complicated to parallelize by their nature. Even inside the LU decomposition triangular operations are involved, but only on small substructures of a matrix, such that they have no critical impact on distributed or multi-GPU algorithms. On the other hand, we have the Gauss-Jordan Elimination computing the inverse A^{-1} by rearranging the operations in the three step LAPACK scheme into one single algorithm (7) requiring the same number of flops. Additionally, in (7) it was shown that this rearrangement can also be applied to positive definite matrices and the Cholesky decomposition, where in the end most operations are performed with the triangular matrices, as well.

Comparing the performance of a triangular solve (TRSM) operation with a general matrix-matrix multiply (GEMM) of the same size on one GPU in our system, described later in this section, we see that the general matrix reaches a 6% higher floprate than the triangular solves as show in Figure 1 for the maximum size fitting into the memory of one GPU. If we restrict to rank-k updates, we see (even for moderate size k) that the rank-k updates perform better than the triangular solve. For smaller problem sizes we see that the GEMM operations reaches their maximum performance handling much smaller problems. Using the Gauss-Jordan Elimination, which is free of any operations dealing with triangular matrices and mainly consists of general matrix-matrix products, we use this advantage to perform the $2m^3$ flops necessary to invert the matrix A. The straight forward

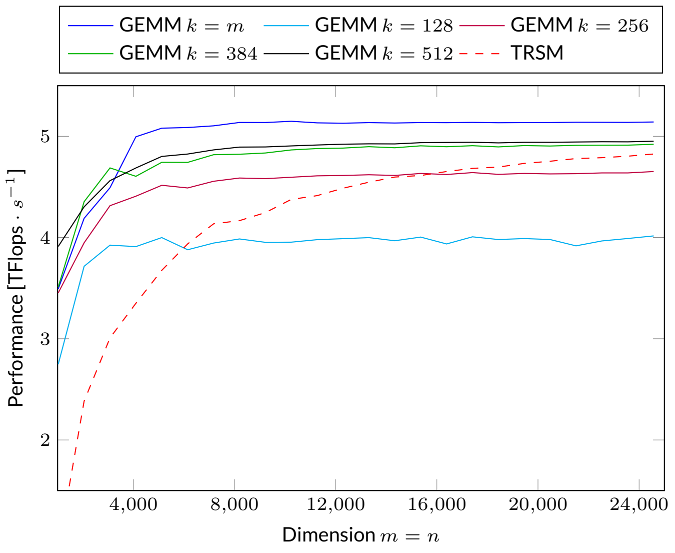


FIGURE 1 Performance of the GEMM, the TRSM and the rank k update on one P100 accelerator.

parallelization and data distribution of the general matrix-matrix product across many computational devices makes the algorithm preferable on (massively) parallel architectures, like multi-core or multi-accelerator based systems. Furthermore, one can show that the Gauss-Jordan Elimination reduces the number of memory accesses (8) and by using general matrix-matrix multiplies the data locality for the single operations of the algorithm is improved. Moreover, when an architecture appears on the market, the matrix-matrix is usually the first well optimized routine available.

In our contribution, we focus on the efficient implementation of the Gauss-Jordan matrix inversion and the Matrix Sign Function on the OpenPOWER 8 platform. The system is equipped with two 10-core IBM POWER 8 CPUs and two nVidia Tesla P100 accelerators using NVLink interconnect and 256 GB DDR4 memory. The CPUs' memory can be accessed with 230 GB/s. The CPU-GPU interconnect allows to transfer 40 GB/s between CPUs and each GPU. The memory bandwidth on the accelerator boards is 750 GB/s in theory and 500 GB/s in practical measurements.

The most important differences to previous, mostly x86-64 based, GPU accelerated systems and the named POWER 8 system are:

- The usage of NVLink as interconnect between CPU and GPU. This increases the transfer rate between their memories by a factor of 2 to 3 in comparison to the latest PCI-Express bus. In this way, data transfers between the CPU and the GPU are cheaper (with respect to runtime) than on older systems.
- The ratio of the peak performances between both CPUs and GPUs is a factor of 20. This constitutes an increase by a factor of 5 if we compare it to an older system, like the 16 core Intel Xeon Haswell with two nVidia K20 accelerators, which we used in our related work (8).
- While keeping the energy consumption for the GPUs in the same order of magnitude as for the old K20 GPUs the energy consumption of the POWER 8 CPUs is 480 Watts in idle state and 920 Watts under full CPU load. In contrast a dual socket Intel Hasweel Xeon system with similar CPU performance takes 145 Watts in idle state and only 350 Watts having load on all CPU cores. This is a factor of 3.3 in idle state and a factor of 2.6 under full load.

The last point makes the difference from the energy point of view. For this reason we want to focus on reducing the power consumption of the CPUs by changing their clock frequency and/or changing the CPU frequency governors that control the automatic adjustment of the CPU clock frequency.

The contribution is organized in the following way. First, we recall the Gauss-Jordan Elimination scheme based on Gauss-Transforms. On top of this idea, we derive its level-3 formulation, which only requires a distributed multi-GPU implementation of the rank-k update. Furthermore, we show how to avoid a direct level-2 fallback for the occurring smaller subproblems in the level-3 algorithm. The last part of Section 2 presents efficiency improving details about a practical implementation of the Gauss-Jordan Elimination scheme. The third section presents our application for the computation of the inverse of a matrix. We use a Newton approach to compute the sign function of a matrix, again using GPU acceleration. The overall procedure extends the (repeated) inversion by a number of communication intensive operations. Beside the these mathematical aspects, we will show the energy consumption related aspects and frequency scaling issues before we validate our approach with a set of numerical experiments. The ecological and economical rating of the results uses the Energy-Delay-Product (EDP) (9, 10).

2 | GAUSS-JORDAN ELIMINATION

The Gauss-Jordan Elimination scheme can be interpreted as a combination of the LU decomposition and the computation of the inverse with reordered operations (7). For a given matrix $A = (a_{ij})_{i,j=1}^{m} \in \mathbf{R}$ we consider the Gauss-Transform $G_i \in \mathbf{R}^{m \times m}$:

$$G_i = \begin{bmatrix} 1 & & & -\frac{a_{1i}}{a_{ii}} & & & \\ & \ddots & & \vdots & & & \\ & & 1 & -\frac{a_{(i-1)i}}{a_{ii}} & & & \\ & & & \frac{1}{a_{ii}} & & & \\ & & & -\frac{a_{(i+1)i}}{a_{ii}} & 1 & & \\ & & & \vdots & & \ddots & \\ & & & -\frac{a_{mi}}{a_{ii}} & & & 1 \end{bmatrix}, \tag{1}$$

which annihilates all off diagonal elements in the i-th column of the matrix G_iA and sets the diagonal entry in this column to one. In order to preserve numerical stability we have to introduce a pivoting strategy as known from the LU decomposition (11). Therefore, we apply the Gauss-Transform G_i to the permuted matrix P_iA, where P_i exchanges row i with row $k := \operatorname{argmax}_{k \geq i} |a_{ki}|$. Together with the pivoting matrix we denote $\widetilde{G_i} := G_iP_i$ as pivoted Gauss-Transform. Setting $A^{(0)} = A$ and $A^{(k)} = \widetilde{G_k}A^{(k-1)}$ yields

$$A^{(m)} = I$$

and

$$\widetilde{G_m}\widetilde{G_{m-1}} \cdots \widetilde{G_2}\widetilde{G_1} = A^{-1}. \tag{2}$$

Without loss of generality, we neglect the pivoting matrices P_i for deriving the level-3 formulation of the algorithm. The integration of the pivoting is straight forward as described in (7, 8). Note that the application of a Gauss-Transform G_i from the left to a matrix A can be reformulated into a rank-1 update

$$A := A - \underbrace{\frac{1}{a_{ii}}\left(a_{1i}, \cdots, a_{(i-1)i}, 0, a_{(i+1)i}, \ldots, a_{mi}\right)^T}_{h^T} A_{i,\cdot} \tag{3a}$$

with the subsequent row scaling

$$A_{i,\cdot} := \frac{1}{a_{ii}} A_{i,\cdot}. \tag{3b}$$

The partitioning of the matrix A into

$$A := \left[\begin{array}{c|c|c} A_{11} & A_{12} & A_{13} \\ \hline A_{21} & A_{22} & A_{23} \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right], \tag{4}$$

where A_{22} is of dimension $N_B \times N_B$ turns the rank-1 update (3) into a rank-N_B update. This leads us to the level-3 formulation:

$$A \leftarrow \left[\begin{array}{c|c|c} A_{11} & 0 & A_{13} \\ \hline 0 & 0 & 0 \\ \hline A_{31} & 0 & A_{33} \end{array}\right] + \underbrace{\left[\begin{array}{c} -A_{12}A_{22}^{-1} \\ \hline A_{22}^{-1} \\ \hline -A_{32}A_{22}^{-1} \end{array}\right]}_{H} \begin{bmatrix} A_{21} & I_{N_B} & A_{23} \end{bmatrix}. \tag{5}$$

Thereby, the matrix $H \in \mathbf{R}^{m \times N_B}$ is the matrix valued counterpart of the vector h in Equation (3). Depending on the architectures and their properties, like cache hierarchies, parallelization features or the optimization of the BLAS/LAPACK library, different approaches to obtain H exist. The first one computes H via using the LU decomposition of $\begin{bmatrix} A_{22}^T & A_{32}^T \end{bmatrix}$, which leads to

$$P \begin{bmatrix} A_{22} \\ A_{32} \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} U_1,$$

with $L_1 \in \mathbf{R}^{N_B \times N_B}$ and a pivoting matrix P. Then H can be computed using several triangular solves (i.e., `TRSM` operations):

$$H = \begin{bmatrix} -A_{12}U_1^{-1}L_1^{-1} \\ U_1^{-1}L_1^{-1} \\ -L_2L_1^{-1} \end{bmatrix} = \begin{bmatrix} H^{(1)} \\ H^{(2)} \\ H^{(3)} \end{bmatrix}.$$

Algorithm 1 Matrix Inversion using Gauss-Jordan-Elimination

```
1: procedure GAUSS-JORDAN-ELIMINATION($A, N_B$)    ▷ $A \in \mathbf{R}^{m\times m}$, $N_B$ width of the block columns in $A$
2:     for $J := 1, 1+N_B, 1+2N_B, \dots, m$ do
3:         $J_B := \min(N_B, m-J+1)$
4:         Partition $A := \left[\begin{array}{c|c|c} A_{11} & A_{12} & A_{13} \\ \hline A_{21} & A_{22} & A_{23} \\ \hline A_{31} & A_{32} & A_{33} \end{array}\right]$, where $A_{11} \in \mathbf{R}^{J-1\times J-1}$ and $A_{22} \in \mathbf{R}^{J_B\times J_B}$.
5:         Compute $H$ from $\begin{bmatrix} A_{12}^T & A_{22}^T & A_{32}^T \end{bmatrix}$ using $LU$ decomposition or recursive Gauss-Jordan Elimination.
6:         Update $A$ using $A := \left[\begin{array}{c|c|c} A_{11} & 0 & A_{13} \\ \hline 0 & 0 & 0 \\ \hline A_{31} & 0 & A_{33} \end{array}\right] + H \begin{bmatrix} A_{21} & I_{N_B} & A_{23} \end{bmatrix}$.
7:     end for
8: end procedure
```

Alternatively one can compute H from $\begin{bmatrix} A_{12}^T & A_{22}^T & A_{32}^T \end{bmatrix}$ by applying Gauss-Transforms (1) again. This either can be done using rank-1 updates as in Equation (3) or in a more advanced scheme, which does not directly fall back into the level-2 BLAS case immediately. Therefore, we introduce a recursive strategy to increase the amount of level-3 BLAS operations, as well as the data locality. This idea introduced by Toledo (12) for the LU decomposition is already used in LAPACK since version 3.6.0 and in a similar way also available for the QR decomposition (13). By partitioning $H \in \mathbf{R}^{m\times n}$ into two block columns

$$H = \begin{bmatrix} H_1 & H_2 \end{bmatrix},$$

where $H_1 \in \mathbf{R}^{m\times\lceil\frac{n}{2}\rceil}$ and $H_2 \in \mathbf{R}^{m\times n-\lceil\frac{n}{2}\rceil}$ the problem of computing H rearranges to:

1. the computation of the Gauss-Jordan Elimination of H_1,
2. the update of the second block column H_2:

$$H_2 = \begin{bmatrix} H_2^{(1)} \\ H_2^{(2)} \\ H_2^{(3)} \end{bmatrix} \leftarrow \begin{bmatrix} H_2^{(1)} \\ 0 \\ H_2^{(3)} \end{bmatrix} + H_1 H_2^{(2)}, \tag{6}$$

3. the computation of the Gauss-Jordan Elimination of H_2,
4. and the final update of the block column H_1:

$$H_1 = \begin{bmatrix} H_1^{(1)} \\ H_1^{(2)} \\ H_1^{(3)} \end{bmatrix} \leftarrow \begin{bmatrix} H_1^{(1)} \\ 0 \\ H_1^{(3)} \end{bmatrix} + H_2 H_1^{(2)}. \tag{7}$$

Applying this strategy in steps 1 and 3 again, we obtain a recursive scheme to compute H. The original work of Toledo (12) and the current implementation of the LU decomposition in LAPACK stop this recursion if $\frac{n}{2} = 1$. In our case that means that H_1 or H_2 consists of a single column and the computation of the Gauss-Jordan Elimination reduces to scaling a vector. Unfortunately, this also reduces the rank-$\frac{n}{2}$, i.e. rank-1, update in (6) or (7) to level-2 BLAS operations. The CPUs of modern computer architectures are mostly equipped with vector registers dealing with 2, 4, or 8 double precision values at once. In order to get the Updates (6) and (7) performed by a `GEMM` operation efficiently, $\frac{n}{2}$ should not get smaller than the length of the vector registers. Consequently, we stop the recursion if $\frac{n}{2}$ gets smaller or equal to the length of the vector registers. The remaining (small) block column H_1 or H_2 is computed using the level-2 BLAS formulation of the Gauss-Transforms (3).

Algorithm 1 shows a sketch of the overall procedure to compute the matrix inverse based on level-3 BLAS operations. Thereby, the update in Step 6 can be either implemented using six `GEMM` operations:

$$A \leftarrow \left[\begin{array}{c|c|c} A_{11} + H^{(1)}A_{21} & H^{(1)} & A_{13} + H^{(1)}A_{23} \\ \hline H^{(2)}A_{21}A_{21} & H^{(2)} & H^{(2)}A_{23} \\ \hline A_{31} + H^{(3)}A_{21} & H^{(3)} & A_{33} + H^{(3)}A_{23} \end{array}\right] \tag{8}$$

or as two large `GEMM` operations:

$$\begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix} \leftarrow \begin{bmatrix} A_{11} \\ 0 \\ A_{31} \end{bmatrix} + HA_{21} \quad \text{and} \quad \begin{bmatrix} A_{13} \\ A_{23} \\ A_{33} \end{bmatrix} \leftarrow \begin{bmatrix} A_{13} \\ 0 \\ A_{33} \end{bmatrix} + HA_{23}, \tag{9}$$

where the A_{21} and the A_{23} part of the matrix needs to be copied to a temporary memory location before the update in order to introduce the necessary zeros. Although this approach needs more memory and some additional operations, we prefer this variant because the increased size of the `GEMM` operations results in a higher performance due to an improved parallel execution inside the BLAS calls.

If pivoting is integrated into the algorithm, following (7), we invert $\tilde{A} = PA$ instead of A, where P is the product of all row permutations performed during the inversion. Hence, we have to apply the inverse permutation P^T to the columns of $\tilde{A}^{-1}$ to obtain $A^{-1} := \tilde{A}^{-1}P^T$.

Operation Count

Before focusing on a GPU accelerated implementation, we briefly discuss the operation count inside Algorithm 1. Especially, we want to show that the number of flops for each iteration J in Algorithm 1 is independent of J and hence each update of A in Step 6 needs the same amount of time. Without loss of generality we assume the two-`GEMM` variant described above and $m = k \cdot N_B$, where $k \in \mathbf{N}$. The derivation for the six `GEMM` variant (8) is straight forward. A general observation is that each update in (9) is of rank N_B. Regarding the left block column in the matrix A the update modifies $J-1$ columns, which costs $2mN_B(J-1)$ flops. The update of the right block column of A modifies $m-J-N_B+1$ columns and costs $2m(m-J-N_B+1)N_B$ flops. Summing up gives us an overall cost of $2mN_B(m-N_B) = 2m^2N_B - 2mN_B^2$ flops for each iteration independent of J. Having $k = \frac{m}{N_B}$ iterations we obtain

$$k\left(2m^2N_B - 2mN_B^2\right) = 2m^3 - 2m^2N_B \tag{10}$$

flops for all updates in Step 6 of Algorithm 1. The computation of H using the rank-1 updates from Equation (3) costs $mN_B + 2mN_B^2$ flops for each iteration J. Performing this $k = \frac{m}{N_B}$ times yields $N + 2m^2N_B$ flops. Together with Equation (10) this gives $2m^3 + N$ flops, which is asymptotically equivalent to the number of flops required by the LU decomposition approach stated in the Introduction.

2.1 | GPU Implementation

The previous section showed how to obtain a level-3 BLAS enabled version of the Gauss-Jordan Elimination scheme from the Gauss-Transform based idea. We have seen that the algorithm decomposes into two parts. The first one is the computation of H which is done inside a block column of the matrix. The second one is the update of the matrix A, which is not affected by the computation of H. Using a classic hybrid CPU-GPU approach we compute the matrix H on the CPU and perform the update on the remaining parts of the matrix A on the GPUs. Due to the fact that we only have a small number of GPUs inside one system, we chose the Column Block Cyclic (CBC) distribution of A across the available GPUs (8, 14). The structure of the rank-N_B updates (9) then allows an easy parallel execution by only distributing the matrix H to all GPUs in each iteration. Putting these ideas together one obtains an easy CPU-GPU variant of Algorithm 1. The CPU computes the matrix H and copies it into the memory of all participating GPUs. Now, the GPUs compute the update (9) or (8). This naive approach is improved using a tailored data layout together with a lookahead strategy and asynchronous operations, as described in the following paragraphs.

Look-Ahead and Asynchronous Operation

The basic CPU-GPU scheme has the main disadvantage that only the CPU or the GPU will work at the same time. This causes a huge performance loss due to the fact the fast GPUs are idle while the CPU is computing and vice versa. The ability of the GPUs to transfer data between their memory and the CPU's memory while they computing can be used to hide the communication between host and accelerator behind the computations. By implementing a look-ahead strategy we can get the CPUs and GPUs working in parallel. To this end, we partition the rightmost block column of A from (4) into

$$\begin{bmatrix} A_{13} \\ A_{23} \\ A_{33} \end{bmatrix} := \begin{bmatrix} \hat{A}_{13} & \bar{A}_{13} \\ \hat{A}_{23} & \bar{A}_{23} \\ \hat{A}_{33} & \bar{A}_{33} \end{bmatrix}, \tag{11}$$

where $\hat{A}_{13}$, $\hat{A}_{23}$, and $\hat{A}_{33}$ have N_B columns. The GPUs now compute

$$\begin{bmatrix} \hat{A}_{13} \\ \hat{A}_{13} \\ \hat{A}_{23} \end{bmatrix} \leftarrow \begin{bmatrix} \hat{A}_{13} \\ 0 \\ \hat{A}_{23} \end{bmatrix} + H\hat{A}_{23} \tag{12}$$

first before they perform the remaining parts of the update from (9). After finishing the computation of (12) the block column $\left[\hat{A}_{13}^T\ \hat{A}_{13}^T\ \hat{A}_{23}^T\right]^T$ is moved back to the host and the CPU computes H for the next iterate $J + N_B$. Due to the fact that all function calls to the GPUs can be done asynchronous, i.e. they do not interrupt the CPU program flow, the CPUs can work in parallel here. On the other hand, this behavior of the GPUs function requires synchronization to ensure that the operations on the GPUs are finished before the data is overwritten by the data requires for the next step. Here, this synchronization takes part after the next panel is copied to the GPUs again before the next look-ahead (12) starts on the GPUs. The dynamic block size adjustment as described by Catalán et. al. in (15) for the LU decomposition is not applicable here, since the workload (in terms of necessary flops) is independent of the iteration index J in Algorithm 1. This allows an a-priori tuning to select the a single optimal block size N_B for the whole algorithm. The optimal block size, on the one hand, minimizes the waiting time for the CPU at the synchronization point in the algorithm and, on the other hand, increases the performance of the rank-N_B update on the GPUs. Algorithm 2 shows how this technique is integrated into a hybrid CPU-GPU algorithm. The different parallel execution paths on the GPU are referred as "stream 1" and "stream 2" following the NVidia CUDA terminology. Although current accelerators allow at least three streams running in parallel (one for computational tasks, one copying data to the device and one copying data back to the host) we can restrict ourselves to two of them in our implementation. All work done inside one stream is executed in order but independent from computations or data transfers performed in other streams or on the CPU. If a stream only contains data transfers most GPUs can handle them parallel to computational work. This allows us to perform computations on the GPUs while copying data back and forth to the host memory. This property is used in the formulation of Algorithm 2 to hide the data transfers between GPUs and host behind the computations on the accelerators.

Data Layout Observations

Beside the CBC data distribution to obtain a multi-GPU aware algorithm, we have to fit the data layout on each GPU to the operations we have to perform. The GPU versions of BLAS, namely nVidia cuBLAS and clBLAS, are designed to use the same column-major (i.e., Fortran-like) matrix storage scheme as in BLAS and LAPACK on the host. As long as no pivoting is integrated in the algorithm we can use this storage scheme on the GPUs as well. When integrating the pivoting, i.e. adding the necessity to swap rows on the GPUs, this becomes a crucial point in the algorithm. Previous work on a Gauss-Jordan Elimination based linear solver on the nVidia Kepler architecture (see (8)) showed that between 20% and 37.5% of the total runtime are spent in exchanging the rows on an nVidia K20 accelerator. Because the basic design properties between the Kepler and the Pascal based GPUs did not change, we expect the same problem here.

The design of the GPUs and their memory access only allows to fetch and store data in chunks. This so called cache-line is typically 128 or 256 bytes long and is the smallest amount of data which can be read/written from/to the memory. That means, even if we only want to access a single double precision value, which takes 8 bytes in the memory, we have to fetch a whole cache-line. If we now assume a cache-line length of 128 bytes, we have 16 double precision value in there. Then, the column-major matrix storage scheme the row swap operation will results in one of the following two cases:

1. Both row entries in one column are inside the same cache-line. In this case two out of sixteen double precision entries of the cache-line are used in the swap operation. $\rightarrow$ 87.5% of the fetched and stored data are not used.

2. The row entries are in different cache-lines. That means for exchanging these two elements two cache-lines need to be fetched and stored. $\rightarrow$ 93.5% of the transferred data is not used.

This overhead in the data transfer during the pivoting wastes time and energy and ultimately slows down the algorithm. Changing the matrix storage scheme to row-major, we can use all 16 double precision entries of all cache-lines during the row exchange. In this way swapping two arbitrary rows of length 16 requires only the transfer of 256 bytes instead of 4kB in the worst case. Thus, we have to change the matrix storage scheme on the GPUs to row-major in contrast to the column-major scheme used by LAPACK on the host. The previous studies on the Kepler architecture (8) already show that the overhead of combining the CPU to GPU transfers (as well as the GPU to CPU transfers) with a transpose operation can be neglected.

By changing the storage scheme on the GPUs, we have to adjust the `GEMM` operation. Typically the call to `GEMM(`α`,A,B,`β`,C)`, as implemented in cuBLAS or clBLAS, computes: $C := \alpha AB + \beta C$, where $A \in \mathbf{R}^{m\times k}$, $B \in \mathbf{R}^{k\times n}$, and $C \in \mathbf{R}^{m\times n}$ are stored in column-major storage. Observing that A^T in column-major storage has the same memory layout as A in row-major storage, we see that calling `GEMM(`α`,B,A,`β`,C)` computes $C^T := \alpha B^T A^T + \beta C^T$, if A, B, and C are stored in row-major storage, although the operation works on column-major storage. That means, exchanging the the roles of A and B (and the corresponding dimension arguments) in the call to the `GEMM` routine allows us to use the optimized implementation, also for the row-major storage, without reimplementing this routine.

Before returning the inverted matrix, the pivoting needs extra attention also when recovering $A^{-1} = \tilde{A}^{-1}P^T$. On the one hand, we now have the same problem with the matrix storage scheme as in the application of the row permutation but in the opposite direction. The column permutation is only efficient on the GPUs if the matrix is stored in column-major storage. Otherwise the previously described cache-line problem appears. On the other hand, in case of a multi-GPU implementation, the matrix is distributed in a column block cyclic way. That means that swapping two columns might cause lots of communications between the GPUs. Although the OpenPOWER 8 system uses the NVLink interconnect between the GPUs we only have a limited bandwidth of 40 GB/s available for this operation. Furthermore, for each transfer of $\mathcal{O}(m)$ data the latency for initiating the transfer must be taken into account. This will slow down the matrix inversion in its final step. On the other hand the CPUs on the OpenPOWER 8

Algorithm 2 Matrix Inversion using Gauss-Jordan-Elimination on GPUs

```
1: procedure MULTI-GPU GAUSS-JORDAN-ELIMINATION($A$, $N_B$)    ▷ $A \in \mathbf{R}^{m\times m}$, $N_B$ width of the block columns in $A$
2:     Asynchronous distribution and row-major storage conversion of $A$ across all GPUs with panel width $N_B$.
3:     for $J := 1, 1+N_B, 1+2N_B, \dots, m$ do
4:         $J_B := \min(N_B, m-J+1)$
5:         if $J = 1$ then
6:             CPU: Compute $H$ and pivoting from $A(1:m, 1:J_B)$
7:         else
8:             CPU: Wait for the download of $\hat{A}_{13}$, $\hat{A}_{23}$, and $\hat{A}_{33}$ in stream 1 and compute $H$ and pivoting from them.
9:         end if
10:        CPU: Upload $H$ to all GPUs in stream 1.
11:        GPU: Convert $H$ to row-major storage in stream 1 and apply the pivoting to $A$ in stream 2.
12:        GPU: Synchronize stream 1 and stream 2.
13:        GPU: Partition $A := \left[\begin{array}{c|c|c|c} A_{11} & A_{12} & \hat{A}_{13} & \bar{A}_{13} \\ \hline A_{21} & A_{22} & \hat{A}_{23} & \bar{A}_{23} \\ \hline A_{31} & A_{32} & \hat{A}_{33} & \bar{A}_{33} \end{array}\right]$, where $A_{11} \in \mathbf{R}^{J-1\times J-1}$, $A_{22} \in \mathbf{R}^{J_B\times J_B}$, and $\hat{A}_{*3}$ has $m-(J+JB)+1$ columns.
14:        GPU: Perform Look-ahead update $\begin{bmatrix} \hat{A}_{13} \\ \hat{A}_{13} \\ \hat{A}_{23} \end{bmatrix} \leftarrow \begin{bmatrix} \hat{A}_{13} \\ 0 \\ \hat{A}_{23} \end{bmatrix} + H\hat{A}_{23}$ and convert the result to column-major storage in stream 2.
15:        CPU: Synchronize stream 2 and download $\hat{A}_{13}$, $\hat{A}_{23}$, and $\hat{A}_{33}$ in stream 1.
16:        GPU: Compute $\begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix} \leftarrow \begin{bmatrix} A_{11} \\ 0 \\ A_{31} \end{bmatrix} + HA_{21}$, $\begin{bmatrix} A_{12} \\ A_{22} \\ A_{32} \end{bmatrix} \leftarrow H$, and $\begin{bmatrix} \bar{A}_{13} \\ \bar{A}_{23} \\ \bar{A}_{33} \end{bmatrix} \leftarrow \begin{bmatrix} \bar{A}_{13} \\ 0 \\ \bar{A}_{33} \end{bmatrix} + H\bar{A}_{23}$ in stream 2.
17:    end for
18:    GPU: Synchronize all streams and convert $A$ to column-major storage.
19:    CPU: Download $A$ and revert the column pivoting.
20: end procedure
```

architecture reach a memory bandwidth of up to 230 GB/s. So our strategy is to move the Matrix from the GPU's memory to the CPU memory with 40 GB/s per GPU and apply the permutation on the host where we mostly need the result as well. In this way we utilize all GPU-CPU interconnects instead of using only one GPU-GPU interconnect.

3 | NEWTON'S METHOD FOR THE MATRIX SIGN FUNCTION

As already stated in the introduction, there are only few applications where the inverse of a matrix needs to be setup. One of these application is the computation of the Matrix Sign Function. The matrix sign function $\operatorname{sign}(A)$ is defined as follows. Let $A \in \mathbf{R}^{m\times m}$ be a matrix with no purely imaginary eigenvalues and $YJY^{-1} = A$ its Jordan canonical form. The Jordan blocks in $J = \begin{bmatrix} J_1 & \\ & J_2 \end{bmatrix}$ are ordered such that $J_1 \in \mathbf{C}^{p\times p}$ contains all Jordan blocks for eigenvalues in the left half-plane and $J_2 \in \mathbf{C}^{(m-p)\times(m-p)}$ contains all Jordan blocks for the eigenvalues in the right half-plane. Then, we define

$$\operatorname{sign}(A) := Y \begin{bmatrix} -I_p & 0 \\ 0 & I_{m-p} \end{bmatrix} Y^{-1} \tag{13}$$

to be the sign of the matrix A. This can be regarded as the generalization of the scalar sign function to the matrix valued case.

The property $(\operatorname{sign}(A))^2 = I$, see (1), can be used to compute the matrix sign function efficiently via a Newton-type method. Therefore, we apply Newton's method to the equation $X^2 = I$ with the initial value $X_0 = A$. The resulting scheme can be written as the following iteration:

$$X_{k+1} := \frac{1}{2}\left(c_k X_k + c_k^{-1} X_k^{-1}\right) \quad \text{and} \quad X_0 = A, \tag{14}$$

where c_k is a scaling factor. Typically, three scaling strategies are used (see, e.g. (1, 3)):

1. *determinantal scaling:* $c_k = |\det(X_K)|^{-\frac{1}{n}}$,
2. *spectral scaling:* $c_k = \sqrt{\frac{\rho(X_k^{-1})}{\rho(X_k)}}$, where $\rho(X)$ denotes the spectral radius of X,

Algorithm 3 GPU accelerated Newton Iteration for the Matrix Sign Function

```
 1: procedure NETWON'S METHOD($A, \tau, \texttt{maxit}$)    ▷ $A \in \mathbf{R}^{m\times m}$, $\tau$ stopping tolerance, maxit maximum iteration number.
 2:     GPU: Allocation $Y \in \mathbf{R}^{m\times m}$ and $W \in \mathbf{R}^{m\times m}$.
 3:     CPU: Upload of $A$ to $Y$ and $W$.
 4:     for $k = 1, \dots, \texttt{maxit}$ do
 5:         CPU/GPU: Invert $W$ using Algorithm 1.    ▷ $W^{-1}$ is assembled on the CPU.
 6:         CPU/GPU Determine the scaling factor $c_k$ or set $c_k = 1$.
 7:         CPU: Upload $W^{-1}$ into $W$ on the GPU.
 8:         GPU: Compute the next iterate in $W$ and determine $F_D$ and $F_Y$ using (16).
 9:         if $\sqrt{\frac{F_D}{F_y}} < \tau$ then    ▷ convergence check.
10:             break
11:         end if
12:         GPU: Copy $W$ to $Y$.
13:     end for
14:     CPU: Download $W$ containing $\operatorname{sign}(A)$ on convergence.
15: end procedure
```

3. and *norm scaling:* $c_k = \sqrt{\frac{||X_k^{-1}||}{||X_k||}}$, where $||\cdot||$ is an arbitrary matrix norm.

Upon convergence $\operatorname{sign}(A) = X_\infty$ holds. In practical implementation mostly the determinantal scaling or the norm scaling are used due to their cheap evaluation. The determinantal scaling is obtained from the Gauss-Jordan Elimination of the matrix A as follows:

$$|\det(A)|^{-\frac{1}{n}} = \prod_{i=1}^{m} \left|A_{ii}^{(i-1)}\right|^{-\frac{1}{n}},$$

where $A^{(i-1)}$ is the matrix in the i-th step of the Gauss-Jordan Elimination as in (2). It can be proven that the iteration (14) converges quadratically to $\operatorname{sign}(A)$ if A has no purely imaginary eigenvalues. Beside a maximum iteration number we use

$$\frac{||X_k - X_{k+1}||_F}{||X_k||_F} < \tau, \tag{15}$$

as convergence criterion, where $\tau > 0$ is a given tolerance.

The GPU implementation of this algorithm is nearly straight forward since we have Gauss-Jordan Elimination available on the GPUs. Beside that, we obtain the scaling factor for determinantal scaling directly from the Gauss-Jordan Elimination. Finally, the update of the iterate and the computation of the convergence criterion are bandwidth limited operations. They require at least $2m^2$ elements additional memory to backup the previous iterate. Choosing the identical data layout for both memory locations, the previous iterate and the inversion process, the update (14) and the convergence criterion (15) are performed on all GPUs in parallel.

Because both additional steps are bandwidth limited, we investigate them in more detail. The update (14) fetches $4m^2$ elements from the memory and writes $2m^2$ elements back to the memory. If the convergence criterion is evaluated afterwards $4m^2$ elements are read again. This gives $8m^2$ elements read and $2m^2$ elements written. Even with a memory bandwidth of 732 GB/s on each P100 card this gets a communication intensive, and in this way time and energy consuming, process. A more efficient scheme works as follows: Let $Y = X_k$ be the memory location of the current iterate and $W = X_k^{-1}$ the location of its inverse, computed by the Gauss-Jordan Elimination. Furthermore, let F_Y be the squared Frobenius norm of X_k and F_D the squared Frobenius norm of the difference in (15). Then the update (14) and the convergence criterion (15) are evaluated in a single step using:

$$W_{ij} \leftarrow \frac{1}{2}\left(c_k Y_{ij} + \frac{1}{c_k} W_{ij}\right), \quad F_Y \leftarrow F_Y + Y_{ij}^2, \quad F_D \leftarrow F_D + (Y_{ij} - W_{ij})^2 \quad \forall i,j \in \{1,\dots,m\}, \tag{16}$$

with $F_D = F_Y = 0$ at the begin of the update. This saves $4m^2$ ($= 40\%$) data transfers in each step of the Newton iteration. Finally, the values contained in W need to be duplicated in Y to create the backup of the current iterate for the next step. Because all data necessary for the next iteration resides in the GPU memory the initial data movement to the GPU in Algorithm 2 is not required any longer. Together with Algorithm 2 we obtain Algorithm 3 as GPU accelerated procedure to compute the Matrix Sign Function.

TABLE 1 Maxium Performance Measures for the OpenPower 822LC System with two nVidia P100 accelerators.

	CPUs	GPUs
P_{elec}	540 W (TDP)	610 W (TDP)
P_{comp}	643 GFlops/s	10.6 TFlops/s

4 | ENERGY CONSUMPTION AND FREQUENCY SCALING

Regarding the hardware aspects of hybrid CPU-GPU accelerated systems we have to take several performance measures into accout. For both, CPUs and GPUs, we have the computational performance P_{comp} (in terms of double precision flops per second) and (electrical) power consumption P_{elec}. For our OpenPOWER 8 system described in the introduction we obtain the values shown in Table 1 .

A simplified connection between the performance values and the clock frequency of a CPU (or a GPU) is described by the following equations (see, e.g., (16)):

$$P_{comp} = w_{flop} \cdot f \tag{17}$$

and

$$P_{elec} = w_0 + w_1 \cdot f + w_2 \cdot f^2, \tag{18}$$

where f is the clock frequency of the computational unit and w_{flop}, w_0, w_1, and w_2 are weights to fit the CPUs' or the GPUs' parameters. Thereby, w_0 represents the clock frequency invariant power consumption, like peripheral components. The factors w_1 and w_2 represent the frequency dependent power consumption. Due to missing accurate measurements of the single components in the considered hardware setup, we cannot give concrete values for w_0, w_1, w_2 here. However, the model shows that the clock frequency has a quadratic influence on the power consumption.

Regarding the ratio between theoretical computational performance and the maximum electrical power we see that the CPUs only provide approximately 6% of the overall compute power, while consuming 47% of the total energy. This makes it preferable to reduce their energy consumption by modifying the CPU part of the algorithm. Our implementation of the Gauss-Jordan Elimination shown in Section 2 and the extension to the Newton scheme in Section 3 only needs the CPU to compute the next panel matrix H and to apply the column permutation after the Gauss-Jordan Elimination. All remaining operations are performed on the GPUs, which have a much better ratio between energy consumption and computational performance as already shown in Table 1 . Observing the quadratic influence of the clock frequency on the power consumption (18), on the one hand, but only a linear impact on the computational performance, on the other hand, we see that halving the clock frequency of the POWER8 CPU from 4GHz to 2GHz will only reduce the overall performance from 11.2 TFlops/s to 10.9 TFlops. However, the linearly influenced power consumption of the CPU is reduced by a factor of 2 and the quadratically influenced one is consequently reduced by a factor of 4. This suggests to slow down the CPU as much as possible to maximize the energy savings. Regarding the hybrid algorithm to compute A^{-1} the operations on the GPU depend on the results of the CPU. If we reduce clock frequency of the CPU too much the panel matrix H will not be ready before the GPU finishes the ongoing computations. That means that the GPU has to wait until the CPU finishes and during this time wastes energy being idle. On the one hand, the selection of an optimal block size N_B in Algorithm 2 is used to match a point where the GPU does not have to wait for the CPU. On the other hand, this block size does not necessarily have to be the one for achieving the best performance for the rank-k update on the GPU. Especially, for large problems the block size giving the optimal performance on the GPUs is optimal on the CPU. In many cases the CPU finishes the preparation of the next panel H earlier than the GPUs are ready for the next step. Here, we are able to slow down the CPU such that its computational workload takes as long as the GPUs are busy. We see in the experiments that this enables us to reduce the clock frequency of the CPU by up to 1 GHz without a noteworthy performance loss for large problems.

The IBM POWER 8 CPU allows a fine grained frequency selection using the Linux kernel frequency governors. At the moment Linux supports four automatic governors and one manual one, which are listed in Table 2 . Out of those, only the `userspace` governor allows to fix the frequency to a desired value. The settings done by the Linux kernel are directly transferred to the hardware for each individual physical CPU core but a separate hardware controller can supersede them. This is the case if, e.g. the temperature of the CPU package reaches its maximum safe-operating temperature, or the energy supply can no longer be guaranteed. Especially, this happens if the CPU works close to its maximum frequency (4.023 GHz) and cannot be superseded by the `userspace` governor. Typically, the frequency is scaled down to $\approx$ 3.9 GHz, for a short time, until the power capabilities and the temperature are in the allowed operation range again. Due to the fact that we use all available CPU cores for the BLAS and LAPACK operations, we set the frequency of all available CPU cores at once.

In contrast, the default behavior of Intel CPUs, since the Sandybridge generation, is to implement a race-to-idle strategies to adjust the clock frequency, where the user can only switch between the `powersave` or the `performance` governor. Both are implemented by the *intel_pstate* driver

TABLE 2 Available Frequency Governors in Linux 4.8 for the OpenPOWER 8 (S822LC) platform.

CPU Frequency Governor	Description
`performance`	The performance governor sets the clock frequency statically to the highest possible value. In our case 4.023 GHz.
`powersave`	The powersave governor sets the clock frequency statically to the lowest possible value. In our case 2.061 GHz.
`ondemand`	The ondemand governor sets the clock frequency depending on the system load obtained by the kernel's process scheduler. Thereby, it switches directly between lowest and highest possible frequencies. This governor can be regarded as software implementation of the race-to-idle strategy.
`conservative`	The conservative governors works similar to the ondemand governor setting the frequency with respect to the system load. However it does a graceful step-by-step frequency adjustment rather than jumping between the both extreme states as in the ondemand case.
`userspace`	The clock frequency is set to a user supplied value. In case of our system this allow frequencies from 2.061 GHz to 4.026 GHz in steps of 33.3 MHz.

inside the Linux Kernel. In performance mode this results in running the CPU cores as fast as possible at the TDP (thermal design power) limit, to get the work done as fast as possible. The driver can be disabled and replaced by an older driver[1] similar to the one used on the OpenPOWER platform but this leads to several problems. On the one hand, the *intel_pstate* driver is deeply integrated into the Linux kernel to deal with the different power states. On the other hand, the highest clock frequencies, also know as turbo-mode, are not available if the old driver is used. In the Linux Administrator Manual[2] it says that "the maximum supported frequency reported by acpi-cpufreq [the old driver] is higher by 1 MHz than the frequency of the highest supported non-turbo P-state". This makes turning off the new driver for frequency scaling experiments uninteresting because one can not check the full frequency range, especially the turbo mode is not accessible.

Selecting a lower clock frequency for the CPUs will also limit their maximum computational performance as a direct result from Formula (17). This performance loss may increase the runtime of the algorithm if large parts of the CPUs workload are in the critical path of the algorithm. This is the case in our application where the CPUs have to prepare the data for the updates performed on the GPUs one step in advance. Considering the case, where we selected a clock frequency which minimizes the energy consumption, on the one hand, but results in a too slow execution of the overall process, we have to introduce a combined measure which rates the ecological aspects as well as the economical ones. Therefore, the *Energy-Delay-Product (EDP)* (9, 10) defined by

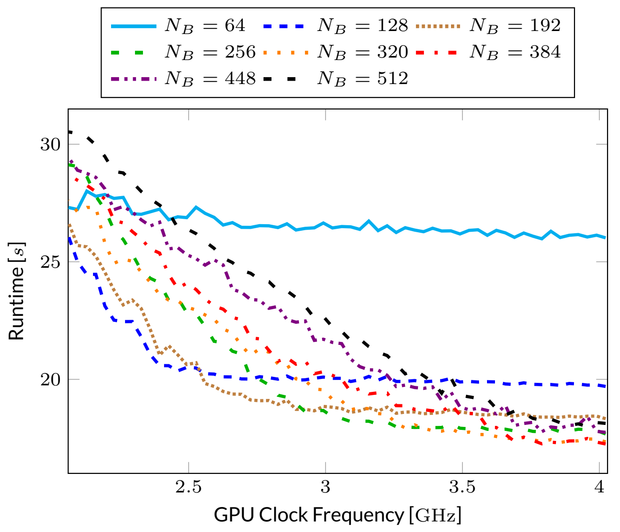
$$\mathrm{EDP}(p) = E \cdot T^p \tag{19}$$

is used. Thereby, E is the energy-to-solution, T is the time-to-solution and p a weight factor to penalize the runtime. This allows to compare several configurations of an algorithm and to select the optimal one with respect to the chosen importance of the runtime. The optimality used in this sense is that as well runtime and energy needs to minimized or if the energy consumption increases the runtime needs to be reduced in order to stay on the save optimality level. In order to avoid such cases, where the energy consumption is reduced by increasing the runtime to a not acceptable value the penalty value p is used to avoid this. A larger penalty value moves the focus more on the runtime, i.e. the economic measure "time" gets more important than the ecological measure "energy". Typical values of p are 1 to obtain an equal weighting between runtime and energy and $p = 2$ and $p = 3$ to focus more on the runtime. In general other value for p are possible but usually not used (9, 10).

5 | EXPERIMENTAL RESULTS

The hardware used for the experiments is already described in the introduction. Due to the novelty of the architecture we use a custom Linux Kernel version 4.8 with the frequency governors described in Section 4. The software ecosystem consists of CentOS 7 together with IBM XLC 13.1.5 as C compiler and IBM XLF 15.1.5 as Fortran Compiler. The IBM ESSL library 5.5 is used as BLAS and LAPACK library on the host. Due to the fact that the ESSL library does not provide a full featured LAPACK interface missing routines are included from the reference LAPACK in version 3.7.0. The GPU code uses nVidia CUDA 8.0 together with the cuBLAS library. The power measurements are taken using a ZES Zimmer LMG 450 at both power supplies of the system with a sampling rate of 20Hz.

[1] Actually, this driver was marked deprecated by the kernel developers and may thus dissapear any time now.
[2] https://www.kernel.org/doc/html/v4.13/admin-guide/pm/intel_pstate.html, accessed January 15th, 2018



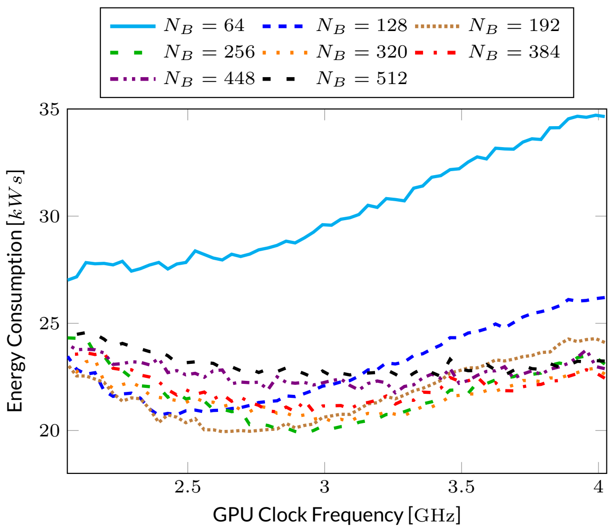


FIGURE 2 Matrix Inversion: Influence of the block-size N_B on runtime and energy consumption, $m = 40,960$, userspace governor

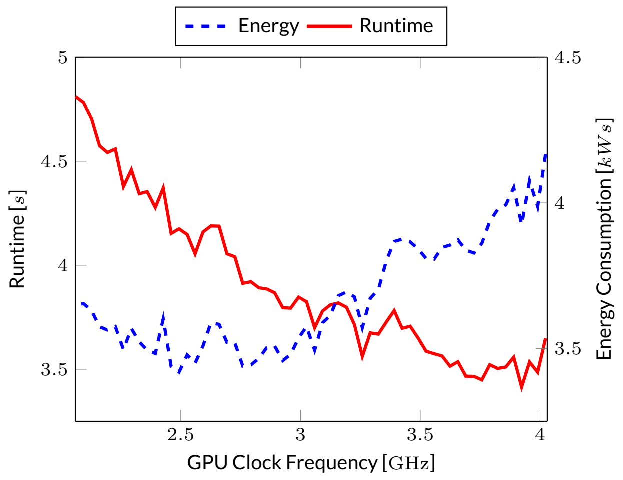


FIGURE 3 Matrix inversion: $m = 20,480$, userspace governor

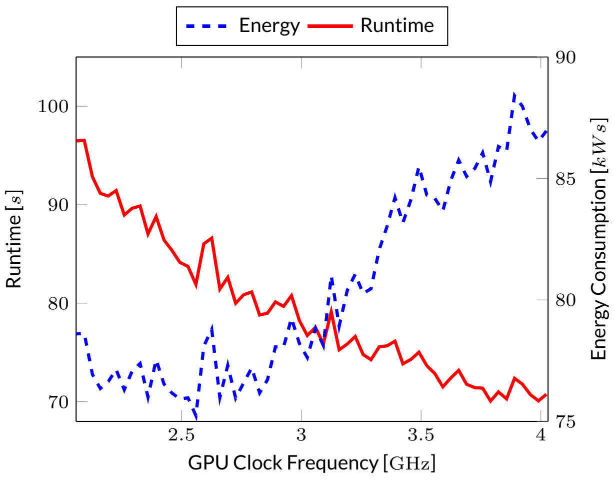


FIGURE 4 Matrix sign function: $m = 20,480$, userspace governor

We analyze both the matrix inversion and the computation of the matrix sign function in order to check whether the additional operations and the increased data transfers will influence the results. The input matrix $A \in \mathbf{R}^{m\times m}, m \in \{20,480, 40,960, 61,440\}$, is a uniformly-$(-1,1)$ distributed random matrix generated using `DLARNV` from LAPACK with an initial seed of $(1,1,1,1)$. The iteration to compute the matrix sign function is fixed to `maxit` $= 20$ without any further convergence criterion to guarantee the equivalent behavior in each test. The 20 iterations are chosen because most problems converge within this range (17).

The optimal block size N_B for Algorithm 2 was chosen such that it minimizes the runtime of the matrix inversion. For the medium sized problem Figure 2 shows the block size which minimizes the runtime for a desired CPU clock frequency. This block size coincides with the block size which minimizes the energy consumption. The figure also shows that with increasing clock frequency of the CPU the block size can be increased as well. Obviously, the CPU can then prepare larger panel matrices H while the GPU is performing its operations.

First, we regard the userspace frequency governor because this is the only one which allows us to analyze the connection between the clock frequency of the CPU and the runtime, as well as the energy consumption. In the Figures 3 and 4 we show both relations for computing the inverse and the matrix sign function for the smallest case $m = 20,480$. From both plots we see that operating with higher clock frequencies the runtime will decrease but the energy consumption will increase. Regarding the step from running at 3 GHz to 4 GHz for inverting the matrix we need 16.6% more energy by saving 4.5% in the runtime. The complete process of computing the matrix sign takes 7.6% less time by consuming 11.5% more energy in the same setup. This means that the runtime benefit does not increase as much as the energy consumption rises when we increase the

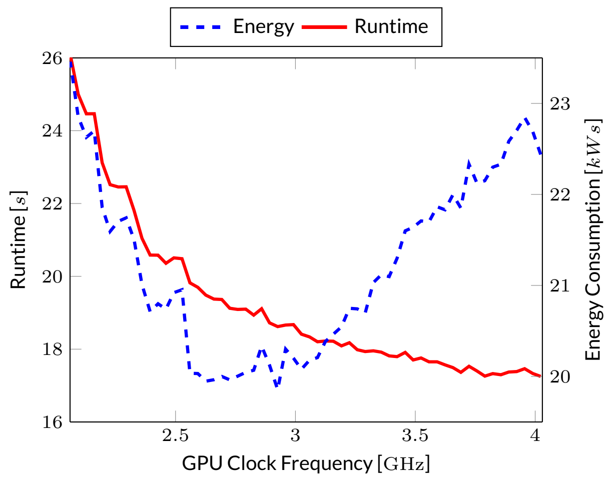


FIGURE 5 Matrix Inversion: $m = 40,960$, userspace governor

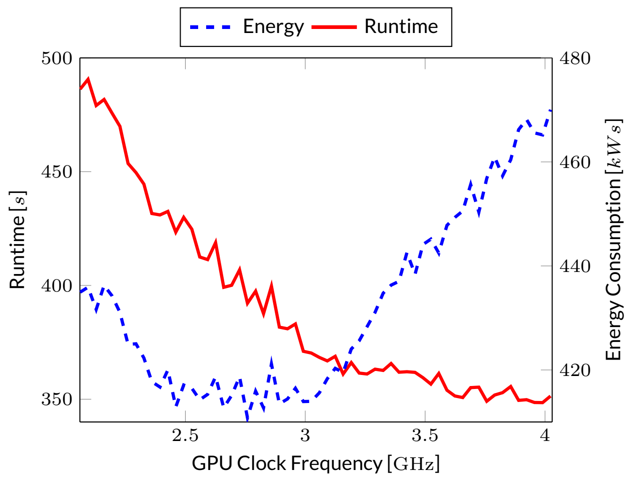


FIGURE 6 Matrix sign function: $m = 40,960$, userspace governor

TABLE 3 Matrix inversion: CPU Clock Frequency (in $[GHz]$) and Runtime (in $[s]$) minimizing the $\mathrm{EDP}(w)$.

Dimension	$m = 20,480$		$m = 40,960$		$m = 61,440$	
	Freq.	Time	Freq.	Time	Freq.	Time
$\mathrm{EDP}(1)$	3.258	3.562	3.092	18.82	2.959	54.73
$\mathrm{EDP}(2)$	3.258	3.562	3.690	17.36	2.959	54.73
$\mathrm{EDP}(3)$	3.923	3.415	3.790	17.25	3.092	54.15

TABLE 4 Matrix Sign Function: CPU Clock Frequency (in $[GHz]$) and Runtime (in $[s]$) minimizing the $\mathrm{EDP}(w)$.

Dimension	$m = 20,480$		$m = 40,960$		$m = 61,440$	
	Freq.	Time	Freq.	Time	Freq.	Time
$\mathrm{EDP}(1)$	3.092	75.87	3.158	361.07	2.826	1161.98
$\mathrm{EDP}(2)$	3.790	70.07	3.158	361.07	2.826	1161.98
$\mathrm{EDP}(3)$	3.790	70.07	3.757	349.10	3.325	1123.71

CPU's clock frequency. On the other hand the power consumption nearly stagnates between 2 and 3 GHz but the algorithm work 20.5% faster in the case of the sign function. This difference is also covered by the EDP comparison in Tables 3 and 4 , where for the $m = 20,480$ case the EDP is minimized by choosing a clock frequency next to the middle of the allowed spectra for equal weighting of runtime and energy. Another effect is that for too small clock frequencies the energy consumption of the overall processes increases again. In the $m = 20\,480$ case this effect is only marginal but with increasing problem size this effect gets more and more visible as it can be seen in Figures 5 , 6 , 7 , and 8 . Thereby, the low clock frequencies yields a longer runtime such that the frequency independent part of the energy consumption increases its influence on the total power consumption. This coincides with the definition of the electrical energy in combination with the power model from Equation (18):

$$E_{elec} := \int_0^t P_{elec}\,\mathrm{d}t = \int_0^t w_0 + w_1 f + w_2 f^2\,\mathrm{d}t, \tag{20}$$

where the influence of the frequency independent part w_0 increases with the runtime t. Furthermore, the operations like the reordering of the columns which are influence by the speed of the CPU and its connection to the main memory are slowed down too much by the low clock frequencies. Computing the matrix sign function shows a similar behavior and will require a higher frequency to perform optimally in the sense of the EDP.

For the medium sized problem $m = 40,960$ we obtain a similar behaviour. We see in Figures 5 and 6 that in the interval between 2.5 GHz and 3 GHz the energy consumption is minimized. However, increasing the clock frequency of the CPU accelerates the algorithm. Beginning at 3.1 GHz we have a notable increase of the energy consumption but, in contrast, we only have small runtime savings. The jump from 3 GHz to 4 GHz costs 13.5% more energy but only saves 5% in terms of runtime. Here, the problem size reaches a critical point such that the GPU's workload is large enough to take longer than the CPU's preparation of the next panel matrix H. This means, we can slow down the CPUs clock frequency until CPU and GPU are ready, for the next step, at the same time. The evaluation of the EDP(1) in Tables 3 and 4 shows this behavior and the EDP(1) minimizes at a clock frequency a bit bigger than the middle of the operation range. Only if we penalize the runtime by a weight of 2 or 3 we should increase the clock frequency more to the fastest operation mode but the runtime savings stay marginal. By integrating the additional operations to compute the matrix sign function iteration behaves similar.

The largest test case, $m = 61,440$, fills up the memory of both GPUs completely during the inversion. Thereby, we see in Figures 7 and 8 that the point where the fast increase of the energy consumption begins, moves to a lower clock frequency again. The jump from 2.8 GHz to 4 GHz in

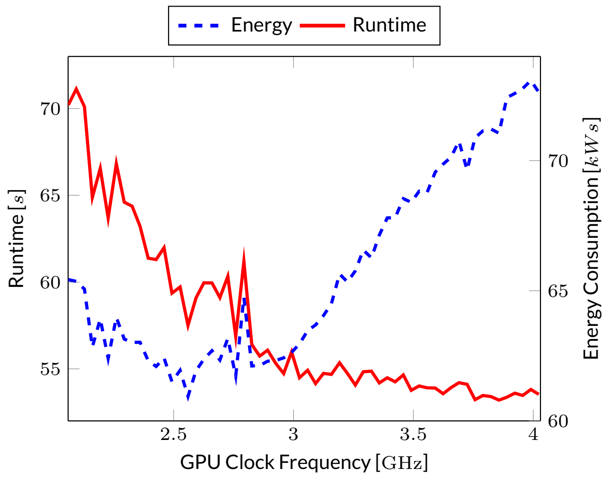


FIGURE 7 Matrix inversion: $m = 61,440$, userspace governor

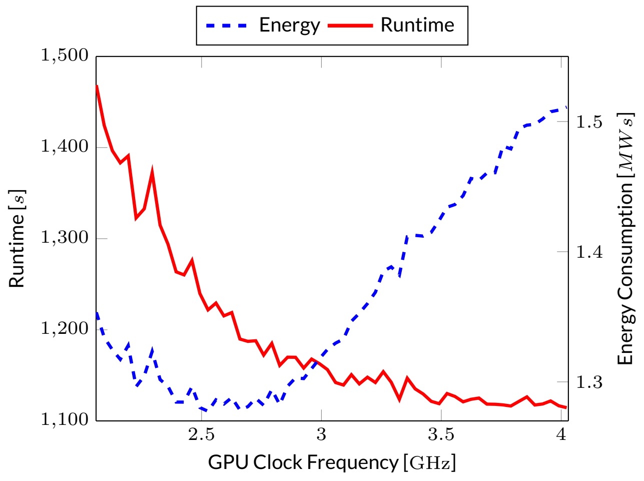


FIGURE 8 Matrix sign function: $m = 61,440$, userspace governor

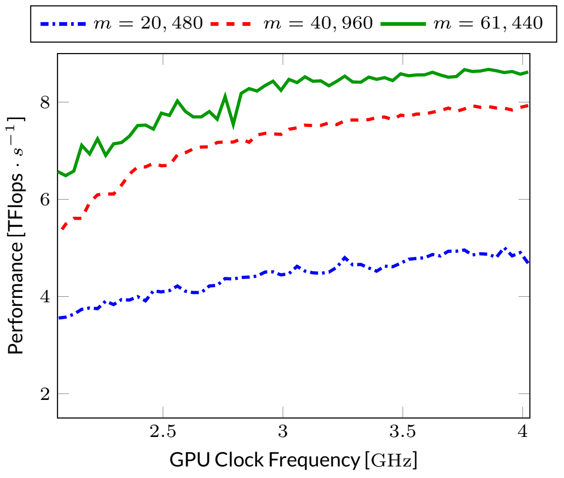


FIGURE 9 Performance comparison of the matrix inversion.

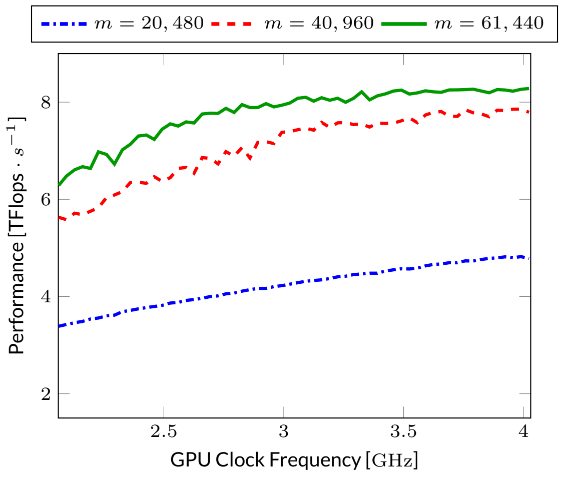


FIGURE 10 Newton's Method for the matrix sign function, $m = 40,960$

case of the matrix sign function will only give us a 4% faster result while using 17.8% more energy. Again, the EDP detects this situation as shown in Tables 3 and 4 . Even if we increase the influence of the runtime by choosing $w = 3$ the suggested clock frequency is still close to 3 GHz. Due to the fact that for $m = 61,440$ only one copy of the matrix could be held in the GPU's memory, the update of the Newton iteration (14) is completely moved to the host, here. The fact that this operation is bandwidth limited the CPU's clock frequency has only a small impact on this operation and the influence on the matrix inversion still dominates.

In general we see that the `userspace` frequency governor will give energy optimal results when operating with a clock frequency around 3 GHz. Even for the smallest problem, $m = 20,480$, the curve for the matrix sign function shows that beginning at 3 GHz the energy consumption increases. But there we still obtain a higher gain in the runtime. The Figures 9 and 10 show the achieved performance of the inversion and the matrix sign function computation. We see that for both large problems the floprate nearly stagnates once the EDP(1) optimal clock frequency is reached. Independent of the problem size we see that for low frequencies the the power consumption increases. The reason for this is that at this point the time spent to prepare the next panel on CPUs take longer than the GPUs are working. The design of the power supplies at the GPUs move the power consumption slowly to the idle state in order to react fast enough when the GPUs get in full operation mode again. This short period of time consuming energy without participating at the computation increases the overall energy consumption. Furthermore, as shown by the energy consumption model (18) and (20), the influence of the clock frequency independent part increases linearly with the runtime. So the a high base power consumption w_0 will influence long running configurations significantly more than the fast ones.

TABLE 5 Martix Sign Function: Runtime (in $[s]$) and Energy Consumption (in $[kWs]$) for automatic frequency governors.

	$m = 20,480$		$m = 40,960$		$m = 61,440$	
Governor	Time	Energy	Time	Energy	Time	Energy
Conservative	75.09	85.13	364.20	471.92	1112.53	1503.28
Ondemand	74.45	85.92	358.00	495.50	1130.85	1502.99
Performance	71.13	86.96	350.20	467.40	1122.10	1512.08
Powersave	101.20	97.08	495.60	443.10	1447.95	1342.56
Userspace – Runtime [3]	71.00	86.31	350.73	450.58	1116.34	1478.98
Frequency	3.823 GHz		3.657 GHz		3.790 GHz	
Userspace – Energy [4]	71.76	85.11	362.08	442.55	1150.71	1346.51
Frequency	3.690 GHz		3.424 GHz		3.125 GHz	

TABLE 6 Matrix Sign Function: Comparison of the Energy-Delay-Product for different Frequency-Governors

	$m = 20,480$			$m = 40,960$			$m = 61,440$		
Governor	$\mathrm{EDP}(1)$	$\mathrm{EDP}(2)$	$\mathrm{EDP}(3)$	$\mathrm{EDP}(1)$	$\mathrm{EDP}(2)$	$\mathrm{EDP}(3)$	$\mathrm{EDP}(1)$	$\mathrm{EDP}(2)$	$\mathrm{EDP}(3)$
Conservative	6.392e+06	4.800e+08	3.604e+10	1.719e+08	6.259e+10	2.279e+13	1.672e+09	1.861e+12	2.070e+15
Ondemand	6.397e+06	4.762e+08	3.546e+10	1.645e+08	5.889e+10	2.108e+13	1.700e+09	1.922e+12	2.174e+15
Performance	6.185e+06	4.399e+08	3.129e+10	1.637e+08	5.733e+10	2.008e+13	1.697e+09	1.904e+12	2.136e+15
Powersave	9.830e+06	9.952e+08	1.008e+11	2.196e+08	1.088e+11	5.393e+13	1.944e+09	2.815e+12	4.076e+15
Userspace (optimal)	5.928e+06	4.166e+08	2.919e+10	1.515e+08	5.469e+10	1.944e+13	1.489e+09	1.729e+12	1.961e+15
Frequency in $[GHz]$	3.092	3.790	3.790	3.158	3.158	3.757	2.826	2.826	3.325

After evaluating the user supplied frequency settings we benchmark the four automatic frequency governors implemented in the Linux kernel. Due to the fact that the results for the matrix inversion nearly coincides with the one from the matrix sign function, we only regard the later one here. Table 5 shows the measured runtime and energy.

Comparing the runtime shows, we get a similar behavior for the `conservative`, the `ondemand`, and the `performance` governors. Each of them results in a comparable runtime with a similar energy consumption. Only the `powersave` governor results in much longer runtimes, on the one hand, but also in considerable energy savings for the two large problems. On the one hand the influence of the CPU performance on the overall performance is still too high and on the other hand the long runtime causes a high influence of the frequency independent power consumption. Additionally, the behavior of the GPU's power supplies when switching to idle consumes energy without performing operations. All together this leads to the high power consumption for the `powersave` governor in the small example.

The selection of comparable results from the `userspace` governor shows that we can obtain the same nearly the same runtime as with one of the automatic governors with reduced the energy consumption. For the $m = 20,480$ case we save 0.8% energy, for the medium sized problem we save 3.7% energy and for the largest case need 1.6% less energy. The other way around, selecting the `userspace` governor results, with respect to approximately the same energy consumption, we get a 4.4% faster solution in the $m = 20,480$ case, a 37.3% faster solution in the $m = 40,960$ case and we need 25.9% less time in the largest problem setting. That means, by using the `userspace` governor we are able to achieve a higher numerical performance for the same amount of energy. Although the `powersave` governor tries to save energy by setting the clock speed as low as possible the Figures 4 , 6 , and 8 show that this is not the minimizer for the energy consumption and the systematic usage of the `userspace` governor allows larger energy savings.

Recapitulating all presented benchmarks, Table 6 compares the EDP values for all test cases computing the matrix sign function. Again, the `userspace` governor can achieved the best tradeoff between the ecological measurement in terms of energy and the economical rating which is reflected by the runtime. Even higher weights on the runtime will not alter this result.

[3]Userspace result selected to match the approximately match the minimal runtime of the other governors.
[4]Userspace result selected to match the approximately match the minimal energy consumption the other governors.

6 | CONCLUSIONS

In our contribution we showed that the frequency scaling of the CPU has a large impact on the runtime as well as on the energy consumption of computing the matrix sign function on the OpenPower 8 platform. We have seen that the automatic frequency scaling governors provided by the Linux kernel are not able to fulfill an ecological and an economical goal at the same time. The usage of the `userspace` governor, which allows a fine grained adjustment of the clock frequency, enables us to save up to 37% in the runtime by using the same amount of energy as the `powersave` governor and in general the best tradeoff between ecological and economical aspects. Beside the experimental evaluation we showed a hybrid implementation of the Gauss-Jordan Elimination and its extension to Newton's method for the matrix sign function.

Similar frequency scaling techniques also exist for the GPUs. Due to the large performance gap and the good floprate-power-ratio, shown in Section 4, we only focused on the CPU frequency scaling here. The combination of CPU and GPU frequency scaling will be part of future research.

References

[1] Kenney C., Laub A. J.. The Matrix Sign Function. *IEEE Trans. Autom. Control.* 1995;40(8):1330–1348.

[2] Denman E. D., Beavers A. N.. The Matrix Sign Function and Computations in Systems. *Appl. Math. Comput..* 1976;2:63–94.

[3] Byers R., He C., Mehrmann V.. The Matrix Sign Function Method and the Computation of Invariant Subspaces. *SIAM J. Matrix Anal. Appl..* 1997;18(3):615–632.

[4] Benner P., Ezzatti P., Quintana-Ortí E. S., Remón A.. Matrix Inversion on CPU-GPU Platforms with Applications in Control Theory. *Concurrency and Comput.: Pract. Exper..* 2013;25(8):1170–1182.

[5] Higham N. J.. Computing the Polar Decomposition—with Applications. *SIAM J. Sci. Statist. Comput..* 1986;7:1160–1174.

[6] Anderson E., Bai Z., Bischof C., et al. LAPACK Users' Guide. SIAMPhiladelphia, PAthird ed.1999.

[7] Quintana-Ortí E. S., Quintana-Ortí G., Sun X., Geijn R.. A Note On Parallel Matrix Inversion. *SIAM J. Sci. Comput..* 2001;22(5):1762–1771.

[8] Köhler M., Penke C., Saak J., Ezzatti P.. Energy-Aware Solution of Linear Systems with Many Right Hand Sides. *Comput. Sci. Res. Dev..* 2016;31(4):215-223.

[9] Freeh Vincent W., Lowenthal David K., Pan Feng, et al. Analyzing the Energy-Time Trade-Off in High-Performance Computing Applications. *IEEE Trans. Parallel Distrib. Syst..* 2007;18(6):835–848.

[10] Horowitz M., Indermaur T., Gonzalez R.. Low-power digital design. *Proceedings of 1994 IEEE Symposium on Low Power Electronics.* 1994;:8–11.

[11] Golub G. H., Van Loan C. F.. *Matrix Computations.* Johns Hopkins Studies in the Mathematical SciencesBaltimore: Johns Hopkins University Press; fourth ed.2013.

[12] Toledo S.. Locality of Reference in LU Decomposition with Partial Pivoting. *SIAM Journal on Matrix Analysis and Applications.* 1997;18(4):1065–1081.

[13] Elmroth E., Gustavson F. G.. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM Journal of Research and Development.* 2000;44(4):605–624.

[14] Yamazaki I., Tomov S., Dongarra J.. One-sided Dense Matrix Factorizations on a Multicore with Multiple GPU Accelerators. *Procedia Computer Science.* 2012;9:37–46.

[15] Catalán S., Herrero J. R., Quintana-Ortí E. S., Rodríguez-Sánchez R., Geijn R.. *A Case for Malleable Thread-Level Linear Algebra Libraries: The LU Factorization with Partial Pivoting.* e-print 1611.06365: arXiv; 2016. cs.DS.

[16] Hager G., Treibig J., Habich J., Wellein G.. Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and Comput.: Pract. Exper..* 2016;28(2):189–210.

[17] Kenney C., Laub A. J.. On Scaling Newton's Method for Polar Decomposition and the Matrix Sign Function. *SIAM J. Matrix Anal. Appl..* 1992;13:688–706.

How cite this article: Köhler M. and Saak, J. Frequency Scaling and Energy Efficiency regarding the Gauss-Jordan Elimination Scheme with Application to the Matrix-Sign-Function on OpenPOWER 8, , .