Evaluating the NVIDIA Tegra Processor as a Low-Power Alternative for Sparse GPU Computations

José I. Aliaga[1], Ernesto Dufrechou[2(✉)], Pablo Ezzatti[2], and Enrique S. Quintana-Ortí[1]

[1] Dep. de Ingeniería y Ciencia de la Computación, Universidad Jaime I, 12701 Castellón, Spain
{aliaga,quintana}@icc.uji.es
[2] Instituto de Computación, Universidad de la República, 11300 Montevideo, Uruguay
{edufrechou,pezzatti}@fing.edu.uy

Abstract. In the last years, the presence of heterogeneous hardware platforms in the HPC field increased enormously. One of the major reason for this evolution is the necessity to contemplate energy consumption restrictions. As an alternative for reducing the power consumption of large clusters, new systems that include unconventional devices have been proposed. In particular, it is now common to encounter energy-efficient hardware such as GPUs and low-power ARM processors as part of hardware platforms intended for scientific computing.

A current line of our work aims to enhance the linear system solvers of ILUPACK by leveraging the combined computational power of GPUs and distributed memory platforms. One drawback of our solution is the limited level of parallelism offered by each sub-problem in the distributed version of ILUPACK, which is insufficient to exploit the conventional GPU architecture.

This work is a first step towards exploiting the use of energy efficient hardware to compute the ILUPACK solvers. Specifically, we developed a tuned implementation of the SPD linear system solver of ILUPACK for the NVIDIA Jetson TX1 platform, and evaluated its performance in problems that are unable to fully leverage the capabilities of high end GPUs. The positive results obtained motivate us to move our solution to a cluster composed by this kind of devices in the near future.

Keywords: ILUPACK · Jetson TX1 · Sparse linear systems
High performance

1 Introduction

In a large number of scientific applications one of the most important stages, from a computational point of view, is the solution of large sparse systems of

E. Mocskos and S. Nesmachnow (Eds.): CARLA 2017, CCIS 796, pp. 111–122, 2018.
https://doi.org/10.1007/978-3-319-73353-1_8

equations. Examples are in problems related with circuit and device simulation, quantum physics, large-scale eigenvalue computations, nonlinear sparse equations, and all sorts of applications that involve the discretization of partial differential equations (PDEs) [4].

ILUPACK[1] (incomplete LU decomposition PACKage) is a well known numerical toolbox that offers highly efficient sparse linear systems solvers, and can handle large-scale application problems. The solvers are iterative procedures based on Krylov subspace methods [13], preconditioned with an inverse-based multilevel incomplete LU (ILU) factorization, which keeps a unique control of the growth of the inverse triangular factors that determines its superior performance over other preconditioners in many cases [8,14,15].

Despite the remarkable mathematical properties of ILUPACK's preconditioner, it implies a larger number of computations when it is compared with other more simple ILU preconditioners, e.g. ILU with zero fill-in. In [1] we proposed the exploitation of the task-level parallelism in ILUPACK, for distributed memory platforms, focusing on symmetric positive definite (SPD) systems, by using the preconditioned Conjugate Gradient (PCG) method. Recently, we have aimed to enhance the task-parallel approach by leveraging the data-level parallelism present in some operations by the use of graphics accelerators.

One drawback of our solution is the limited level of parallelism offered by each sub-problem in the distributed version of ILUPACK, which is insufficient to exploit the conventional GPU architecture.

On the other hand, in the last years the presence of heterogeneous hardware platforms in the HPC field increased enormously. One of the major reasons for this evolution is the necessity to contemplate energy consumption restrictions. As an alternative for reducing the power consumption of large clusters, new systems that include unconventional devices have been proposed. In particular, it is now common to encounter energy efficient hardware such as GPUs and low-power ARM processors as part of hardware platforms intended for scientific computing.

This work is a first step towards exploiting the use of energy efficient hardware to compute the ILUPACK solvers. Specifically, we developed a tuned implementation of the SPD linear system solver of ILUPACK for the NVIDIA Jetson TX1 platform, based on an NVIDIA Tegra X1 processor, and evaluated its performance in problems that are unable to fully leverage the capabilities of high end GPUs. The obtained results show that the use of this kind of lightweight devices achieves interesting runtimes, specially if it is considered that we are addressing a memory-bound problem and the gap between the memory bandwidth of a general GPU and the Jetson GPU is in the order of 8–10×.

The rest of the paper is structured as follows. In Sect. 2 we revisit the SPD solver integrated in ILUPACK and we offer a brief study about the application of the parallel preconditioner. This is followed by a description of our implementations of ILUPACK to run over the Jetson TX1 in Sect. 3, and the experimental evaluation in Sect. 4. Finally, Sect. 5 summarizes the work and makes some concluding remarks, stating the most important lines of future work derived from this effort.

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

2 Accelerated Solution of Sparse Linear Systems with ILUPACK

ILUPACK is a software package that deals with solving the linear system $Ax = b$, where the $n \times n$ coefficient matrix A is large and sparse, and both the right-hand side vector b and the sought-after solution x contain n elements. It does so by applying Krylov subspace-based iterative solvers preconditioned with an inverse-based multilevel ILU preconditioner M, of dimension $n \times n$. The package includes numerical methods for different matrix types, precisions, and arithmetic, covering Hermitian positive definite/indefinite and general real and complex matrices. Although the preconditioner has proven to effectively boost the convergence rate in many problems, its application is usually the most computationally demanding task of each iteration of the solver.

Following, we briefly describe the main aspects related to the construction of the preconditioner in order to ease the understanding of its application in the context of iterative solvers, which is the focus of our work.

2.1 Computation of the Preconditioner

For simplicity, we base the following description on the SPD real case, where $A, M \in \mathbb{R}^{n \times n}$ are SPD, and $x, b \in \mathbb{R}^n$. The computation of ILUPACK's preconditioner is organized as follows:

1. First, a preprocessing stage scales A and reorders the result in order to reduce the fill-in in the factorization.
2. Later, $\hat{A} \approx LDL^T$ is computed by an incomplete Cholesky factorization process, where $L \in \mathbb{R}^{n \times n}$ is unit lower triangular and $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix. The partial ILU factorization obtained in this stage is of the form:

$$\hat{A} \equiv \begin{pmatrix} B & G^T \\ G & C \end{pmatrix} = LDL^T + E = \begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & S_c \end{pmatrix} \begin{pmatrix} L_B^T & L_G^T \\ 0 & I \end{pmatrix} + E. \quad (1)$$

   where E contains the elements "dropped" during the ILU factorization. The factorization procedure postpones the processing of a row/column of A whenever it estimates that it would produce $\|L^{-1}\| \gtrapprox \kappa$, being κ a user-defined threshold. The postponed rows and columns are moved to the bottom right corner of the matrix and processed in a subsequent stage, yielding a multilevel structure. In the previous equation S_C represents the approximate Schur complement assembled from the "rejected" rows and columns.
3. The process is then restarted with $A = S_c$, (until S_c is void or "dense enough" to be handled by a dense solver,) yielding a multilevel approach.

At level l, the multilevel preconditioner can be expressed as

$$M_l \approx \begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} L_B^T & L_G^T \\ 0 & I \end{pmatrix} \quad (2)$$

where L_B and D_B are blocks of the factors of the multilevel LDL^T preconditioner (with L_B unit lower triangular and D_B diagonal); and M_{l+1} stands for the preconditioner computed at level $l+1$.

A detailed explanation of each stage of the process can be found in [6].

2.2 Application of the Preconditioner During the Iterative Solve

The application of the preconditioner at a given level l requires solving a system of linear equation involving the preconditioner M_l and the permuted and scaled residual $\hat{r}$:

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} L_B^T & L_G^T \\ 0 & I \end{pmatrix} w = \hat{r}. \tag{3}$$

This is then solved for w in three steps,

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} y = \hat{r}, \quad \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} x = y, \quad \begin{pmatrix} L_B^T & L_G^T \\ 0 & I \end{pmatrix} w = x, \tag{4}$$

where the recursion is defined in the second one.

Considering a partition of y and $\hat{r}$ conformable with the factors, the expressions in (4) can also be solved by

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} y_B \\ y_C \end{pmatrix} = \begin{pmatrix} \hat{r}_B \\ \hat{r}_C \end{pmatrix} \;\Rightarrow\; L_B y_B = \hat{r}_B,\; y_C := \hat{r}_C - L_G y_B. \tag{5}$$

Splitting the vectors in a similar way, the expression in the middle of (4) involves a diagonal-matrix multiplication and the effective recursion:

$$\begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} x_B \\ x_C \end{pmatrix} = \begin{pmatrix} y_B \\ y_C \end{pmatrix} \;\Rightarrow\; x_B := D_B^{-1} y_B,\; x_C := M_{l+1}^{-1} y_C. \tag{6}$$

In the base step of the recursion, M_{l+1} is void and only x_B has to be computed. Finally, the expression on the right of (4) can be reformulated as

$$\begin{pmatrix} L_B^T & L_G^T \\ 0 & I \end{pmatrix} \begin{pmatrix} w_B \\ w_C \end{pmatrix} = \begin{pmatrix} x_B \\ x_C \end{pmatrix} \;\Rightarrow\; w_C := x_C,\; L_B^T w_B = x_B - L_G^T w_C, \tag{7}$$

where z is simply obtained from $z := \tilde{D}(\tilde{P}(\hat{P}w))$.

To save memory, ILUPACK discards the off-diagonal blocks L_G once it is done calculating the level of the preconditioner, keeping only the much sparser rectangular matrix G. Thus, (5) is changed into:

$$L_G = G^T L_B^{-T} D_B^{-1} \quad \Rightarrow \quad y_C := \hat{r}_C - G^T L_B^{-T} D_B^{-1} y_B = \hat{r}_C - G^T L^{-T} D_B^{-1} L_B^{-1} \hat{r}_B, \tag{8}$$

while the expressions related to (7) are modified to

$$L_G = G L_B^{-T} D_B^{-1} \quad \Rightarrow \quad L_B^T w_B = D_B^{-1} y_B - D_B^{-1} L_B^{-1} G^T w_C. \tag{9}$$

Operating with care, the final expressions are thus obtained,

$$\begin{array}{l} L_B D_B L_B^T s_B = \hat{r}_B \\ L_B D_B L_B^T \hat{s}_B = G^T w_C \end{array} \Rightarrow \begin{array}{l} y_C := \hat{r}_C - G s_B \\ w_B := s_B - \hat{s}_B \end{array} \tag{10}$$

In summary, the application of the preconditioner requires, at each level of the factorization, two SpMV, solving two linear systems with coefficient matrix of the form LDU, and a few vector kernels.

3 Proposal

In this section we describe the design and implementation details of the solution that will be analyzed numerically in the following sections. First, we provide a general description of our data-parallel variant of ILUPACK and then, we specify the particular strategies that were adopted for the Jetson TX1.

3.1 Exploiting the Data Parallelism in ILUPACK

Although ILUPACK is a sophisticated preconditioner that manages to significantly improve the convergence of Krylov subspace methods in many cases [5,6,16], its application is computationally expensive. We then aim to reduce the cost of the iteration by exploiting the data-level parallelism present in the operations that compose the application of the preconditioner using GPUs.

In most cases, the portion of the runtime related with the application of the preconditioner is concentrated by two main types of operation, i.e. sparse triangular system solves (SpTrSV) and the SpMV that appears in Eq. (10). NVIDIA's CUSPARSE library provides efficient implementations for these operations, so we rely on the library to offload this kernels to the GPU. The vector scalings and reorderings that are performed in the GPU via *ad-hoc* CUDA kernels, and their execution time is completely negligible when compared to that of the triangular systems or the SpMV.

The use of CUSPARSE makes necessary to make an adaptation of the structures employed by ILUPACK to store the submatrices of the multilevel preconditioner to the format accepted by CUSPARSE routines. Specifically, the modified CSR format [4] in which ILUPACK stores the L_B needs to be rearranged into plain CSR. This transformation was done only once, during the construction of each level of the preconditioner, and occurred entirely in the CPU. In devices equipped with physical Unified Memory[2], like the Jetson TX1, no transference is needed once this translation has been done, and the triangular systems involved in the preconditioner application can be solved via two consecutive calls to `cusparseDcsrsv_solve`. It is also necessary to perform the analysis phase of the CUSPARSE solver, in order to gather information about the data dependencies, generating a level structure in which variables of the same level

[2] See: JETSON TX1 DATASHEET DS-07224-010_v1.1.

can be eliminated in parallel. This is executed only once for each level of the preconditioner, and it runs asynchronously with respect to the host CPU.

Regarding the computation of the SpMV in the GPU, it is important to remember from Eq. (10) that each level of the preconditioner involves a matrix-vector multiplication with F and F^T. Although CUSPARSE provides a modifier of its SpMV routine that allows to work with the transposed matrix without storing it explicitly, this makes the routine dramatically slow. We then store both F and F^T in memory, accepting some storage overhead in order to avoid using the transposed routine.

3.2 Threshold Based Version

Our data parallel variant of ILUPACK is capable of offloading the entire application of the preconditioner to the accelerator. This strategy has the primary goal of accelerating the computations involved while minimizing the communications between the CPU and the GPU memory.

However, the multilevel structure of the preconditioner usually produces some levels of small dimension, which undermines the performance of some CUSPARSE library kernels. Specifically, the amount of data-parallelism available in the sparse triangular linear systems is severely reduced, leading to a poor performance of the whole solver.

To address this situation, in recent work [2] we have proposed the inclusion of a threshold that controls whether there is sufficient parallelism to take profit of computing a given level of the preconditioner in the GPU or if it is better to move the computations to the CPU. This has proven to boost the performance in some applications although it implies an additional CPU-GPU communication cost.

We further enhanced the procedure by moving only the triangular solves corresponding to low levels, given that it is this operation the one which dramatically degrades the performance, while in most cases we are able to take some advantage of the data parallelism present in the sparse matrix-vector products and the vector operations that remain. For the rest of the work we will consider three different implementations:

- *GPU 1 level*: computes the triangular systems of all levels but the first in the CPU while using the GPU for the rest of the operations.
- *GPU all levels*: computes the entire preconditioner application (all levels) on the GPU.
- *ARM-based*: makes all the computations in the ARM processor. This variant does not leverage any data parallelism.

We believe that the *GPU 1 level* strategy can be specially beneficial for devices like the Jetson, where each device can perform the operations for which it is better suited without adding the communication overhead necessary in other platforms.

4 Experimental Evaluation

In this section we present the results obtained in the experimental evaluation of our proposal. First we describe the hardware platform and the test cases employed in this stage, and later, we analyze the numerical and runtime results. Specifically, our primary goal is to evaluate if this kind of devices are able to solve sparse linear systems of moderate dimensions efficiently using ILUPACK. In order to do so, we start evaluating our developed variants to identify the best one. All experiments reported were obtained using IEEE single-precision arithmetic.

4.1 Experimental Setup

The evaluation was carried out in an NVIDIA Jetson TX1 that includes a 256-core Maxwell GPU and a 64-bit quad-core ARM A57 processor configured in maximum performance. The platform is also equipped with 4 GB of LPDDR4 RAM that has a theoretical bandwidth of 25.6 GB/s (see [12]).

The code was cross-compiled using the compiler `gcc 4.8.5` for `aarch64` with the `-O3` flag enabled, and the corresponding variant of CUDA Toolkit 8.0 for the Jetson, employing the appropriate libraries.

The benchmark utilized for the test is a SPD case of scalable size derived from a finite difference discretization of the 3D Laplace problem. We generated 3 instances of different dimension; see Table 1. In the linear systems, the right-hand side vector b was initialized to $A(1, 1, \ldots, 1)^T$, and the preconditioned CG iteration was started with the initial guess $x_0 \equiv 0$. For the tests, the parameter that controls the convergence of the iterative process in ILUPACK, *restol*, was set to 10^8. The drop tolerance, and the bound to the condition number of the inverse factors, that influence ILUPACK's multilevel incomplete factorization process, where set to 0.1 and 5 respectively.

Table 1. Matrices employed in the experimental evaluation.

Matrix	Dimension n	nnz	nnz/n
A126	2,000,376	7,953,876	3.98
A159	4,019,679	16,002,873	3.98
A171	5,000,211	19,913,121	3.98

It should be noted that these test cases present dimensions that are often too small to take profit of regular GPUs. These dimensions are comparable with those of each sub-problem derived from the application of the distributed ILUPACK variant on large matrices.

4.2 Results

The first experiment evaluates the performance of our 3 variants of ILUPACK developed for the Jetson TX1 platform to solve the sparse linear systems from the Laplace problem. In this line, Table 2 summarizes the runtimes implied by *ARM-based*, *GPU 1 level* and *GPU all levels* versions to solve the test cases described in Table 1. Specifically, we include the number of iterations taken by each variant to converge (*iters*), the runtimes for the application of the preconditioner (*Prec. time*), the total runtime (*Total time*), the numerical precision (i.e. the numerical error computed as $\mathcal{R}(x^*) := ||b - Ax^*||_2/||x^*||_2$) and finally, the speedup associated with both the accelerated stage with the GPU (*Prec. speedup*) and the whole method (*Total speedup*).

Table 2. Runtime (in seconds) of the three data-parallel variants of ILUPACK in Jetson TX1. *Prec. time* corresponds to the time spent applying the preconditioner during the entire solver.

Variant	Case	*Iters*	*Prec. time*	*Total time*	*Error*	*Prec. speedup*	*Total speedup*
ARM-based	A126	156	60.33	84.57	2.31E−07	-	-
GPU 1 level		156	59.00	84.85	2.37E−07	1.02	1.00
GPU all levels		156	44.36	70.30	2.45E−07	1.36	1.20
ARM-based	A159	206	161.90	228.26	3.07E−07	-	-
GPU 1 level		206	177.33	243.09	3.15E−07	0.91	0.94
GPU all levels		206	123.93	187.93	3.15E−07	1.31	1.21
ARM-based	A171	222	218.53	306.78	3.02E−07	-	
GPU 1 level		222	170.76	253.84	3.03E−07	1.28	1.21
GPU all levels		222	146.09	229.45	3.10E−07	1.50	1.34

First of all we focus on the numerical aspects of our variants. In this sense, it can be noted that all variants needed the same number of iterations to reach the convergence criteria for each of the test cases addressed. In the other hand, the residual errors attained are not exactly the same. However, the differences are not at all significant and can be explained by the use of single precision floating point arithmetic in conjunction with the parallel execution.

Considering the performance results, it should be highlighted that the *GPU all levels* version outperforms the *GPU 1 level* counterpart for all test cases. This result is not aligned with our previous experiences (see [2]) and can be explained by the elimination of the overhead caused by transferring data between both processors (ARM and GPU), allowed by the Unified Memory capabilities of the Jetson platform.

In the same line, the *GPU all levels* version implies lower runtimes than the *ARM-based* variant, but these improvements are decreasing with the dimension

of the addressed problem. This result is consistent with other experiments, and relates to the fact that GPUs requires large volumes of data to really exploit their computational power.

If we take the performance of *GPU all levels* in other kind of hardware platform into consideration, it is easy to see the benefits offered by the Jetson device. To illustrate this aspect we compared our previous results for the GPU-based ILUPACK from [3], run on an NVIDIA K20 GPU, to compare the runtimes. Table 3 summarizes these results, focusing only in the preconditioner application runtime, and contrasting it with the one obtained in the Jetson TX1 platform.

It should be recalled that ILUPACK, as a typical iterative linear system solver, is a memory-bounded algorithm. Hence, when comparing the performance allowed by the Jetson with other GPU-based general hardware platforms, it is necessary to analyze the differences between their memory bandwidth. As an example, the NVIDIA K20 GPU offers a peak memory bandwidth of 208 GB/s [11], while the NVIDIA Jetson only allows to reach 25.6 GB/s, i.e. a difference above 8×.

Table 3. Runtime (in seconds) of GPU-based ILUPACK in a K20 (from [3], using double-precision arithmetic) and the GPU-all variant of ILUPACK in Jetson TX1 (in single-precision).

Case	K20			Jetson		
	Iters	*Prec. time*	*Time by iter*	*Iters*	*Prec. time*	*Time by iter*
A126	44	11.38	.26	156	44.36	.28
A159	52	19.75	.38	206	123.93	.60
A171	-	-	-	222	146.09	.66
A200	76	28.28	.37	-	-	-

Before analyzing the results, it is important to remark that the computations in both works are not exactly the same. Note that preconditioners generated by executing ILUPACK have a different drop tolerance input parameter, and hence are distinct since this parameters affects the amount of fill-in allowed in the triangular factors. However, the runtime by iteration is an acceptable estimator for the performance of each version.

The results summarized in Table 3 show that the time per iteration for the smallest case is similar in the two platforms. Considering that the experiments in the K20 GPU were performed using double precision, it is reasonable to expect this runtimes to be reduced in half[3] if single precision is used. This means that the difference in performance is of about 2× in favour of the K20. Nevertheless, this gap is considerably smaller than the difference in the bandwidth of both devices, which is of approximately 8×. However, it can also be observed that the benefits offered by the Jetson hardware start to diminish when the dimension of

[3] Assuming a memory-bound procedure.

the test cases grow (note that in the case A159 is near to 3× if we estimate the single precision performance of the K20 as before).

This result shows that when the dimension of the addressed test case is enough to leverage the computational power of high end GPUs this kind of lightweight devices are not competitive. On the other hand, in contexts where the problem characteristics do not allow exploiting commodity GPUs efficiently, this kind of devices (e.g. the Jetson TX1) are a really good option. Additionally, the important difference in power consumption between the two devices (the K20 has a peak power consumption of 225W[4], while the Jetson only 15W[5]) should also be taken into account.

With the obtained results, our next step is to develop a distributed variant of ILUPACK specially design to run over a cluster of low power devices, as a NVIDIA Jetson TX1, and evaluate the energy consumption aspects. It should be noted that this kind of clusters are not yet widespread, but some examples are the one built in the context of the Mont-Blanc project, leaded by the Barcelona Super Computing (BSC) Spain [7], and the one constructed by the ICARUS project of the Institute for Applied Mathematics, TU Dortmund, Germany [9,10].

5 Final Remarks and Future Work

In this work we have extended the data-parallel version of ILUPACK with the aim to contemplate low power processors, as the NVIDIA Jetson TX1 hardware platform. In particular, we implemented three different versions of the solution that take into account the particular characteristics of this kind of devices, two GPU-based variants (*GPU 1 level* and *GPU all levels*) and the other centered on the use of the ARM processor (*ARM-based*).

The numerical evaluation exhibits that the *GPU all levels* variant outperforms the other options for the test cases addressed. However, when the dimension of the problems decreases the *ARM-based* version starts to be more competitive. Additionally, as the dimension of the problem grows the benefits related to the use of restrictive platforms such as the Jetson start to disappear, and the utilization of conventional GPU platforms becomes more convenient.

Given the importance of the results obtained we plan to advance in several directions, which include:

- Assessing the use of other kinds of small devices, such as the recently released Jetson TX2 (with 8 GB of memory and 59.7 GB/s of memory bandwidth).
- Developing a distributed variant of ILUPACK specially designed to run over a cluster of lightweight devices, as a NVIDIA Jetson TX1.
- Evaluate the distributed variant in a large Jetson-based cluster, such as the ones developed in the context of Mont-Blanc or ICARUS projects.
- Studying the energy consumption aspects of this distributed version of ILUPACK for small devices.

[4] TESLA K20 GPU ACCELERATOR - Board Specifications - BD-06455-001_v05.
[5] JETSON TX1 DATASHEET DS-07224-010_v1.1.

Acknowledgments. The researchers from the *Universidad Jaime I* were supported by the CICYT project TIN2014-53495R of The researchers from *UdelaR* were supported by PEDECIBA and CAP-UdelaR Grant.

References

1. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors. In: Jónasson, K. (ed.) PARA 2010. LNCS, vol. 7133, pp. 162–172. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28151-8_16
2. Aliaga, J.I., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Design of a task-parallel version of ILUPACK for graphics processors. In: Barrios Hernández, C.J., Gitler, I., Klapp, J. (eds.) CARLA 2016. CCIS, vol. 697, pp. 91–103. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57972-6_7
3. Aliaga, J.I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Leveraging data-parallelism in ILUPACK using graphics processors. In: Muntean, T., Rolland, R., Mugwaneza, L. (eds.) IEEE 13th International Symposium on Parallel and Distributed Computing, ISPDC 2014, Marseille, France, 24–27 June 2014, pp. 119–126. IEEE (2014)
4. Barrett, R., Berry, M.W., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Van der Vorst, H.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, vol. 43. SIAM, Philadelphia (1994)
5. Bollhöfer, M., Grote, M.J., Schenk, O.: Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media. SIAM J. Sci. Comput. **31**(5), 3781–3805 (2009)
6. Bollhöfer, M., Saad, Y.: Multilevel preconditioners constructed from inverse-based ILUs. SIAM J. Sci. Comput. **27**(5), 1627–1650 (2006)
7. Anonymous Contributors: start—mont-blanc prototype (2016). Accessed 10 July 2017
8. George, T., Gupta, A., Sarin, V.: An empirical analysis of the performance of preconditioners for SPD systems. ACM Trans. Math. Softw. **38**(4), 24:1–24:30 (2012)
9. Geveler, M., Ribbrock, D., Donner, D., Ruelmann, H., Höppke, C., Schneider, D., Tomaschewski, D., Turek, S.: The ICARUS white paper: a scalable, energy-efficient, solar-powered HPC center based on low power GPUs. In: Desprez, F., et al. (eds.) Euro-Par 2016. LNCS, vol. 10104, pp. 737–749. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58943-5_59. ISBN 978-3-319-58943-5
10. Geveler, M., Turek, S.: How applied sciences can accelerate the energy revolution-a pleading for energy awareness in scientific computing. In: Newsletter of the European Community on Computational Methods in Applied Sciences, January 2017, accepted
11. NVIDIA: TESLA K20 GPU Accelerator (2013). https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf. Accessed 10 July 2017
12. NVIDIA: NVIDIA Tegra X1 NVIDIAs New Mobile Superchip (2015). http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf. Accessed 10 July 2017
13. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM Publications, Philadelphia (2003)

14. Schenk, O., Wächter, A., Weiser, M.: Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. SIAM J. Sci. Comput. **31**(2), 939–960 (2009)
15. Schenk, O., Bollhöfer, M., Römer, R.A.: On large scale diagonalization techniques for the Anderson model of localization. SIAM Rev. **50**, 91–112 (2008)
16. Schenk, O., Wächter, A., Weiser, M.: Inertia revealing preconditioning for large-scale nonconvex constrained optimization. SIAM J. Sci. Comput. **31**(2), 939–960 (2008)