MAGMA Embedded: Towards a Dense Linear Algebra Library for Energy Efficient Extreme Computing

Azzam Haidar, Stanimire Tomov, Piotr Luszczek
University of Tennessee
Knoxville, TN 37916
{haidar,tomov,luszczek,dongarra}@eecs.utk.edu

Jack Dongarra
University of Tennessee, Knoxville
Oak Ridge National Laboratory, USA
University of Manchester, UK
dongarra@eecs.utk.edu

***Abstract*—Embedded computing, not only in large systems like drones and hybrid vehicles, but also in small portable devices like smart phones and watches, gets more extreme to meet ever increasing demands for extended and improved functionalities. This, combined with the typical constrains for low power consumption and small sizes, makes the design of numerical libraries for embedded systems challenging. In this paper, we present the design and implementation of embedded system aware algorithms, that target these challenges in the area of dense linear algebra. We consider the fundamental problems of solving linear systems of equations and least squares problems, using the LU, QR, and Cholesky factorizations, and illustrate our results, both in terms of performance and energy efficiency, on the Jetson TK1 development kit. We developed performance optimizations for both small and large problems. In contrast to the corresponding LAPACK algorithms, the new designs target the use of many-cores, readily available now even in mobile devices like the Jetson TK1, e.g., featuring 192 CUDA cores. The implementations presented will form the core of a MAGMA Embedded library, to be released as part of the MAGMA libraries.**

I. Introduction

While working on high performance linear algebra libraries such as PLASMA [28], [4] and MAGMA [1] or their runtime systems [29] we are often reminded of the shift that happened a few years back: the shift to the "mobile first" market strategy [8]. It has happened for at least the two most recent generations of processors from AMD, Intel, and NVIDIA. As we show below, responding to that trend with the adequate software is a challenge due to a variety of hardware design choices that are made differently on the mobile platforms. For example, the balance between the processor and the accelerator is skewed, the fractions of various performance points is drastically different, and the entire software tool chain might be missing essential pieces. Any of these obstacles would constitute a serious impediment but taken together, they create an environment that requires drastic shift in numerical software design and implementation.

In this paper, we describe a methodology we used to port an important numerical algorithm from a server-class accelerated system to a mobile form factor. Namely, we are interested in the fundamental problems of directly solving a linear system of equations, or least squares problem. The main component of these solvers is a direct factorization, be it sparse or dense, illustrated in Figure 1.

To provide parallelism in these solvers, the computation can be expressed as a Directed Acyclic Graph (DAG) of tasks with labeled edges designating data dependencies, which naturally leads to the need to handle many linear algebra

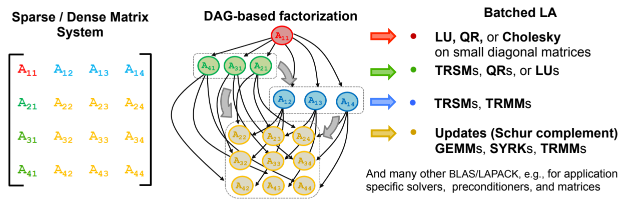

Sparse / Dense Matrix System
A_{11} A_{12} A_{13} A_{14}
A_{21} A_{22} A_{23} A_{24}
A_{31} A_{32} A_{33} A_{34}
A_{41} A_{42} A_{43} A_{44}
DAG-based factorization
Batched LA
LU, QR, or Cholesky on small diagonal matrices
TRSMs, QRs, or LUs
TRSMs, TRMMs
Updates (Schur complement) GEMMs, SYRKs, TRMMs
And many other BLAS/LAPACK, e.g., for application specific solvers, preconditioners, and matrices

Fig. 1. Direct sparse or dense factorizations—a DAG approach that needs efficient computation of many linear algebra tasks, varying in size along the factorization process. Thin DAG edges represent data dependencies among individual tasks. Small data-parallel tasks can also be grouped together, in *batches*, and the thick edges represent dependencies among the resulting batched tasks.

problems of varying sizes in parallel. To provide this in a numerical library, we must design dense linear algebra factorizations like LU, QR, and Cholesky, and tune them for variable size problems. Our work with vendors (through vendor recognition centers), collaborators (from the HPC community), and application developers has resulted in the accumulation of expertise, technologies, and numerical software [4], [18], [28], [20], [1], [28], [26], [27], [21], [3], [19], [9], [5], [11] that we leverage in this paper to develop state-of-the-art dense solvers for embedded systems. We illustrate the approach using the QR factorization.

II. Methodology and Algorithmic Design

The state-of-the-art methodology for server-class accelerated systems is based on hybrid algorithms that use the DAG approach and properly schedule tasks for execution over the available CPU and GPU hardware components [26], [2], [6], [7], [12]. Benchmark software also uses hybridized methods [10]. Typically, small or memory bound tasks on the critical path of the algorithm are scheduled on the CPUs, and large data-parallel tasks on the GPUs. This is what we denote as the hybrid approach. While this methodology works very well, it could have significant drawbacks when the balance between the processor and the accelerator is skewed. A slow CPU for example, even after tuning, can make a fast GPU idle. For that we proposed another schema that use only the GPU to perform either the memory bound or the compute intensive tasks, in other term, that use only the GPU to perform the whole computation. We have shown that both hybrid and GPU-only high-performance linear algebra algorithms can be designed so that the computation is performed by calls to BLAS kernels, to the extent possible by the current BLAS API. This is important since the use of BLAS has been crucial for the high-performance sustainability of major numerical libraries for decades, and therefore we can also leverage the lessons learned from that success. However, to enable the effective use of a BLAS based approach, there is a need to develop highly efficient and optimized BLAS routines.

Our methodology to embedded systems, based on this approach, but designed and tuned for a single task, is described as follows.

A. Algorithmic Baseline

The QR factorization of an m-by-n matrix A (with $m_t = m/n_b$ and $n_t = n/n_b$) is of the form $A = QR$, where Q is an m-by-m orthonormal matrix, and R is an m-by-n upper-triangular matrix. The LAPACK routine **GEQRF** implements a right-looking QR factorization algorithm, whose first step consists of the following two phases:

1) *Panel factorization*: The first panel $A_{:,1}$ is transformed into an upper-triangular matrix.
   a) **GEQR2** computes an m-by-m Householder matrix H_1 such that $H_1^T A_{:,1} = \begin{pmatrix} R_{1,1} \\ 0 \end{pmatrix}$, and $R_{1,1}$ is an n_b-by-n_b upper-triangular matrix.
   b) **LARFT** computes a block representation of the transformation H_1, i.e., $H_1 = I - V_1 T_1 V_1^H$, where V_1 is an m-by-n_b matrix and T_1 is an n_b-by-n_b upper-triangular matrix.
2) *Trailing submatrix update*: **LARFB** applies the transformation computed by **GEQR2** and **LARFT** to the submatrix $A_{:,2:n_t}$:

$$\begin{pmatrix} R_{1,2:n_t} \\ \widehat{A} \end{pmatrix} := (I - V_1 T_1 V_1^H) \begin{pmatrix} A_{1,2:n_t} \\ A_{2:m_t,2:n_t} \end{pmatrix}.$$

Then, the QR factorization of A is computed by repeating the same transformation to the submatrix $\widehat{A}$ and so on. The transformations V_j are stored in the lower-triangular part of A, while R is stored in the upper-triangular part. Additional n_b-by-n_b storage is required to store T_j [25].

B. Optimized and parametrized BLAS kernels

We developed the most needed and performance-critical Level 3 and Level 2 BLAS routines, tuned for small sizes. Namely, we developed the gemm (general matrix-matrix multiplication), trsm (triangular matrix solver), and gemv (general matrix-vector product) routines, as well as a number of Level 1 BLAS such as the dot product, the norm functionality, and the scal scaling routine. There are a number of feasible design choices for BLAS on small matrices, each best suited for a particular case. Therefore, to capture as many of them as possible, we designed a space for BLAS on small matrices that includes parametrized algorithms enabling an ease of tuning for modern and future hardware and taking into account the matrix size. Thus, a parametrized-tuned approach can find the optimal implementation within the confines of the said design space.

We developed our kernel in a parametrized fashion, that uses multiple levels of blocking, including shared memory and register blocking, as well as double buffering techniques to hide the data communication with the computation. This technique allowed us to optimize and tune the our BLAS routine for differen matrix sizes — originally for Fermi GPUs [21], and later for the Kepler GPUs. Recently, we extended it to a small gemm and a batch of small gemms [14], [13], which is now available through MAGMA 1.6.1 [15]. The extension was done by autotuning the basic kernel, and, in the case of batched execution, by adding one more thread dimension to account for the batch count. Our goal is to develop optimized components that can be used easily as a plug-in device routine to provide many of the Level 3 and Level 2 BLAS routines.

In particular, let us consider the QR decomposition. We developed the equivalent of LAPACK's geqr2 routine to perform the Householder panel factorizations. For a panel of n_b columns, it consists of n_b steps where each step calls a sequence of the larfg and the larf routines. At every step (to compute one column), the larfg involves a norm computation followed by a scal that uses the results of the norm computation in addition to some underflow/overflow checking. These Level 1 BLAS kernels have been developed as device component routines to all for easy plug-in when needed. The norm computation is a sum reduce and thus a synchronization step. To accelerate it, we implemented a two-layer tree reduction where, for sizes larger than 32, all 32 threads of a warp progress to do a tree reduction similar to the MPI_REDUCE operation, and the last 32 elements are reduced by a single thread. Our parametrized technique lets us run our autotuner and tune these kernels. As a result, custom implementations of both larfg and the larf have been developed. When the panel size is small enough, we use the shared memory to load the whole panel and to perform its computation in fast memory. For larger panel sizes, we load only the vector that is annihilated at each step, meaning that the norm, scal, and thus the larfg computation operate on data in shared memory; the larf reads data from shared memory, but writes data in main memory since it cannot fit into the shared memory. When the panel is large, the BLAS kernel operates using many thread-blocks and an atomic synchronization.

C. Development of GPU-only LAPACK algorithms

The development of GPU-only LAPACK algorithms and implementations is our main example of how to use the small BLAS for higher-level algorithms. We show an approach based on small BLAS and architecture-specific algorithmic improvements that overcomes the challanges for solving small-size problems on embedded systems. Similarly to the small BLAS, we build a design space for GPU-only LAPACK that includes parametrized algorithms that are architecture and matrix size aware. An autotuning approach is used to find the best implementation within the provisioned design space.

We developed technologies for deriving high-performance from GPU-only implementations to solve single small problems, sets of small linear algebra problems (as in LAPACK) in parallel, as well as single large problems. Note that GPU-only implementations have been avoided up until recently in numerical libraries, especially for small and difficult to parallelize tasks like the ones targeted by here. Indeed, hybridization approaches were at the forefront of developing large scale solvers as they were successfully resolving the problem by using CPUs for the memory bound tasks [26], [2], [6], [7], [12]. For large problems, the panel factorizations (the source of memory bound, not easy to parallelize tasks) are always performed on the CPU. For small problems, as we already mentioned, this is not possible, and our experience has shown that hybrid algorithms would not be as efficient as they are for large problems.

1) Recursive Multilevel Nested Blocking.: The panel factorizations (geqr2) described above factorize the n_b columns one after another, similarly to the LAPACK algorithm. At each of the n_b steps, a rank-1 update is required to update the vectors to the right of the factorized column i. This operation is done by the larf kernel. Since we cannot load the entire panel into the shared memory of the GPU, the columns to the right are loaded back and forth from the main memory at every step except for the very small size cases (e.g., size less than 32×8). Thus, one can expect that this is the most time consuming part of the panel factorization.

Our analysis using the NVIDIA Visual Profiler [22] shows that a large fraction of even a highly optimized small-size factorization is spent in the panels, e.g., 40% of the time for the QR decomposition. The profiler reveals that the larf kernel requires more than 75% of the panel time by itself. The inefficient behavior of these routines is also due to the memory access. To resolve this challenge, we propose to improve the efficiency of the panel and to reduce the memory access by using a two-level nested blocking technique as depicted in Figure 2. First, we recursively split the panel to an acceptable block size n_b as described in Figure 2a. In principle, the panel can be blocked recursively until a single element remains. Yet, in practice, 2-3 blocked levels (an $n_b = 32$ for double precision was the best) are sufficient to achieve high performance. Then, the routine that performs the panel factorization (geqr2) must be optimized, which complicates the implementation. This optimization can bring between 30% to 40% improvement depending on the panel and the matrix size. In order to reach our optimization goal, we also blocked the panel routine using the classical blocking fashion to small blocks of size ib ($ib = 8$ was the optimized choice for double precision) as described in Figure 2b. More than a 25% boost in performance is obtained with this optimization.

2) Block Recursive dlarft Algorithm.: The larft is used to compute the upper triangular matrix T that is needed by the QR factorization in order to update either the trailing matrix or the right hand side of the recursive portion of the QR panel. The classical LAPACK computes T column by column in a loop over the n_b columns as described in Algorithm 1. Such an implementation takes up to 50% of the total QR factorization time. This is due to the fact that the kernels needed – gemv and trmv – require implementations where threads go through the matrix in different directions (horizontal vs. vertical, respectively). An analysis of the mathematical formula of computing T allowed us to redesign the algorithm to use Level 3 BLAS and to increase the data reuse by putting the column of T in shared memory. One can observe that the loop can be split into two loops – one for gemv and one for trmv. The gemv loop that computes each column of $\widehat{T}$ (see the notation in Algorithm 1) can be replaced by one gemm to compute all the columns of $\widehat{T}$ if the triangular upper portion of A is zero and the diagonal is made of ones. For our implementation, replacing a gemv loop with one gemm is already done for the trailing matrix update in the larfb routine, and thus can be exploited here as well. For the trmv phase, we load the T matrix into shared memory as this allows all threads to read/write from/into shared memory during the n_b steps of the loop. The redesign of this routine is depicted in Algorithm 2. Since we developed a recursive blocking algorithm, we must compute the T matrix for every level of the recursion. Nevertheless, the analysis of Algorithm 2 leads us to conclude that the portion of the T's computed in the lower recursion level are the same as the diagonal blocks of the T of the upper level (yellow diagonal blocks in Figure 3), and thus we can avoid their (re-)computation. For that we modified Algorithm 2 in order to compute either the whole T or the upper rectangular portion that is missed (red/yellow portions in Figure 3). Redesigning the algorithm to block the computation using Level 3 BLAS accelerated the overall algorithm on average by about $20 - 30\%$ (depending on various parameters).

```
for $j \in \{1, 2, \ldots, n_b\}$ do
    dgemv to compute $\widehat{T}_{1:j-1,j} = A^H_{j:m,1:j-1} \times A_{j:m,j}$
    dtrmv to compute $T_{1:j-1,j} = T_{1:j-1,1:j-1} \times \widehat{T}_{1:j-1,j}$
    $T_{j,j} = \tau_j$
```

Algorithm 1: Classical implementation of the dlarft routine.

```
dgemm to compute $\widehat{T}_{1:n_b,1:n_b} = A^H_{1:m,1:n_b} \times A_{1:m,1:n_b}$
load $\widehat{T}_{1:n_b,1:n_b}$ to the shared memory.
for $j \in \{1, 2, \ldots, n_b\}$ do
    dtrmv to compute $T_{1:j-1,j} = T_{1:j-1,1:j-1} \times \widehat{T}_{1:j-1,j}$
    $T_{j,j} = \tau_j$
write back $T$ to the main memory.
```

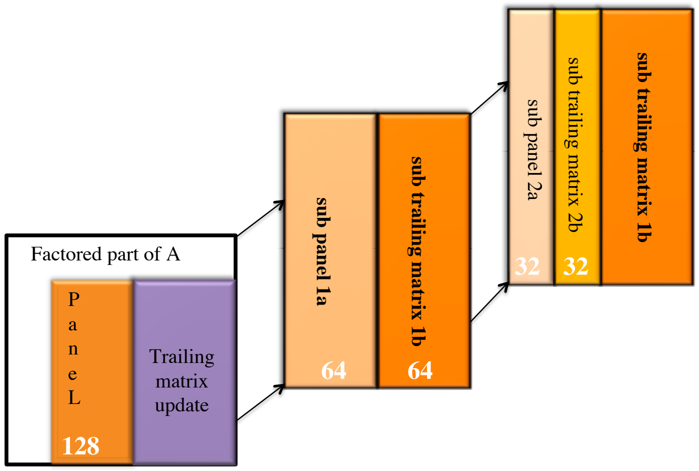
Algorithm 2: Block recursive dlarft routine.

3) Trading extra computation for higher performance.: The goal here is to replace the use of low performance kernels with higher performance ones—often for the cost of more flops, e.g., trmm used by the larfb can be replaced by gemm. The QR trailing matrix update uses the larfb routine to perform $A = (I - VT^HV^H)A$. The upper triangle of V is zero with ones on the diagonal, and also the matrix T is upper triangular. The classical larfb uses trmm to perform the multiplication with T and with the upper portion of V. If one can guarantee that the lower portion of T is filled with zeroes and the upper portion of V is filled with zeros and ones on the diagonal, then the trmm can be replaced by gemm. Thus we implemented a GPU-only larfb for small-size problems that uses three gemm kernels by initializing the lower portion of T with zeros, and filling up the upper portion of V with zeroes and ones on the diagonal. Note that this brings $3n_b^3$ extra operations. The benefits again depend on various parameters, but on current architectures we observe an average of 10% improvement, and see a trend where its effect on the acceleration grows from older to newer systems.

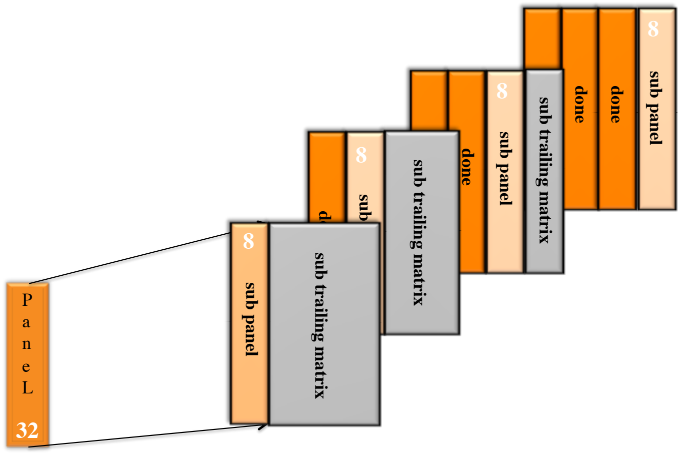
III. Experimental Results

A. Hardware platforms

In our experiments we use NVIDIA's Jetson K1 platform (for Tegra K1 embedded applications). It features a Kepler GPU with 192 CUDA cores running at 0.85 GHz and a 4-Plus-1 quad-core ARM Cortex A15 CPU at 1.5 GHz. The GPU contributes over 300 Gflop/s worth of performance and CPU only about 10 Gflop/s. With the factor of 30 more performance coming from the accelerator, the balance for this mobile platform is heavily skewed when compared with the server-based Kepler GPUs and their CPU counterparts such as Intel Haswell – the difference is at most 3-to-1 in favor of the accelerator. With such a GPU-CPU performance ratio, it is imperative to distribute the workload much differently on the mobile platform than on the server.



(a) Recursive nested blocking fashion.



(b) Classical blocking fashion.

Fig. 2. The recursive two-level nested blocking fashion is used in our implementation to achieve high-performance small-size kernels.

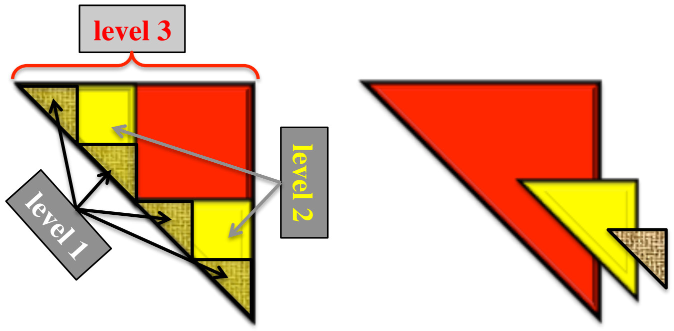


Fig. 3. The shape of the matrix T for different level of the recursion during the QR decomposition.

For completeness, it is worth explaining the word *core* that might easily become ambiguous if used interchangeably in different context. We primarily use the word in two primary ways: CPU core and CUDA core. The former represents an independent processing unit, a processor of yore, that features a complete set of circuit to issue, dispatch, execute, and retire instructions. From the numerical software perspective, a CPU core features 2 (Intel), 1 (ARM), half (AMD) floating-point unit (FPU), which may feature short vector extension ,for example, AltiVec, SSE or AVX. CUDA cores are specific to NVIDIA GPUs and may be considered an equivalent of single precision FPU. It is sometimes suggested that a more appropriate equivalent of a CPU core is a GPU SMX – a streaming multiprocessor that features many CUDA cores. In fact, in case of the Kepler GPUs, mobile or server, a single SMX features 192 CUDA cores.

For our energy efficiency measurements we used power and energy estimators built into the modern hardware platforms. In particular, on the tested CPU, an Intel Xeon E5-2690, we used RAPL (Runtime Average Power Limiting) hardware counters [17], [24]. Given the caveat that these counters are only an estimate, we can report that the idle power of the tested Sandy Bridge CPU, running at a fixed frequency of 2.6 GHz, is 20 W per socket. For the GPU measurements, we use NVIDIA's NVML (NVIDIA Management Library) library [23]. NVML provides a C/C++ API – a programmatic interface – for monitoring and managing various states within NVIDIA Tesla-grade GPUs. On the Fermi and Kepler GPUs,

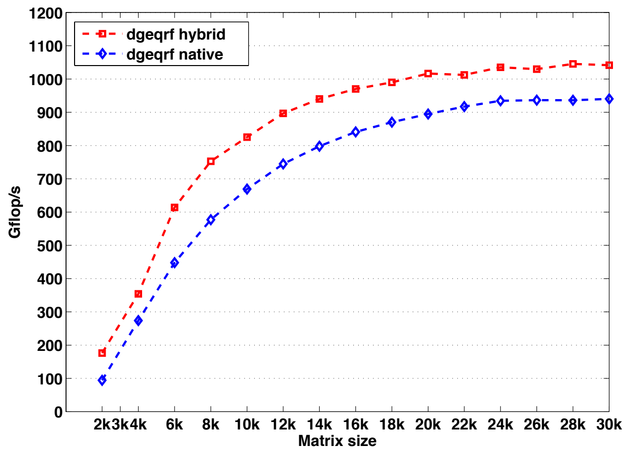


Fig. 4. Performance comparison of hybrid CPU+GPU *vs.* GPU-only approaches (QR factorization in double precision).

such as the K40c that we used in our tests, the readings are reported to be accurate to within ±5% of the current power draw. The idle state of the K40c GPU consumes about 20 W, but when the GPU is initialized the power goes up to 62 W.

B. *Hybrid* vs. *GPU-only approaches*

Figure 4 shows the performance of the QR factorization on server-class Kepler GPU paired with a server class Intel CPU with only the latency-bound workload executing on the CPU. For comparison, performance of double precision matrix-matrix multiply **DGEMM** is highly tuned on this platform and reaches over 1100 Gflop/s which is nearly 90% of the peak performance of the accelerator. The difference between the two approaches is considered to be small, it is around 100 Gflop/s. The GPU-only implementation is about 10% to 15% slower, but it does not use any CPU resources, while the hybrid algorithm use the CPU for the panel phase. The panel factorization consists of memory bound operations. For the hybrid routine, it is performed on the CPU in an overlapped fashion which mean the CPU factorize factorize the panel of step $k+1$ the GPU is doing the update of step k. The cost of the panel is hidden and we can see that the hybrid QR performance reach very close to the one obtained by the **DGEMM** routine.

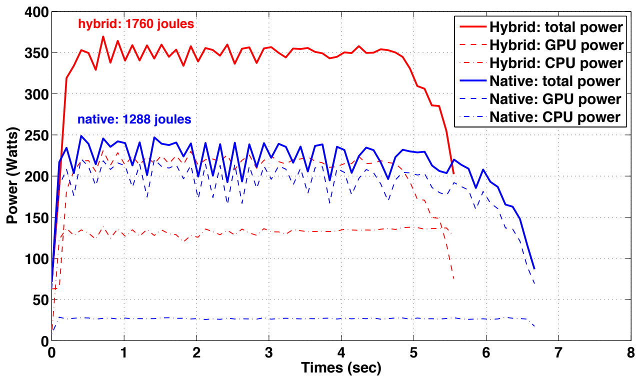


Fig. 5. Comparison of the power consumption for the QR decomposition of the GPU-only and the hybrid routine for a matrix of size 16000 in double precision).

Therefore, it is because of the panel computation that the GPU-only implementation cannot match the hybrid one. However, the results shown are considered attractive, the GPU profiler show that it is fully busy doing either the panel factorization or the trailing matrix update.

Figure 5 shows the comparison of the power consumption required by our two implementations of the QR decomposition. The problem solved here is of size 16,000. The red curve shows the power required by the hybrid CPU-GPU implementation. It includes the power of the GPU and the power consumption of the CPU as measured by the software through the hardware counters. Note, that the panel factorization phase is performed by the CPU using the multithreaded **DGEQRF** routine form the Intel MKL [16] library on the 16 Sandy Bridge cores while the compute intensive operations are handled by the GPU. Here, the panel operations raise the consumption of the CPU to about 105 W for the two sockets and about 30 W for the DRAM package making the total CPU power reaching a level of about 135 W. The GPU operation is mainly preoccupied by the **DGEMM** routine, that reaches the highest fraction of the peak performance, and it raises the power draw to about 225 W. The blue curve shows the power required by the GPU-only implementation. We also included the power of the CPUs (which are "idles") which is about 20 W in addition to the DRAM power which is about 8 W. Both the panel and the trailing matrix update phases are computed by the GPU and we can observe the saw-like behavior of the curve. When the GPU is working on the panel, the power goes to around 160 W and when it performs the **DGEMM**, the power goes up to about 215 W. We also illustrate the total energy required by each approach of the QR decomposition From Figures 4 and 5, we can compute the performance per Watt of both implementations. Our results showed that the hybrid implementation is able to reach about 3 Gflop/s/Watt while the GPU-only one can reach about 4 Gflop/s/Watt.

C. Performance and energy consumption on embedded systems

Figure 6 shows the performance of the QR factorization on the mobile development board – NVIDIA Jetson, which combines a mobile version of the Kepler GPU with a mobile ARM CPU. We show the performance of three different codes: the hybrid CPU-GPU one, GPU-only, and, for comparison, raw **SGEMM** performance. Of note is the performance of the hybrid code, that might be considered a direct port of the implementation that works extremely well on server-class hybrid systems.

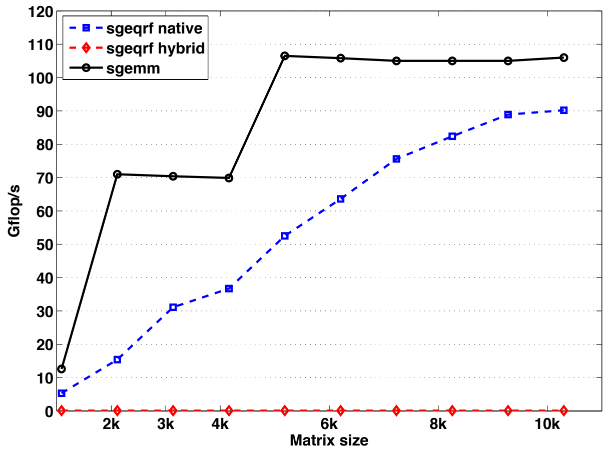


Fig. 6. Performance of QR factorization on a Jetson TK1 (single precision).

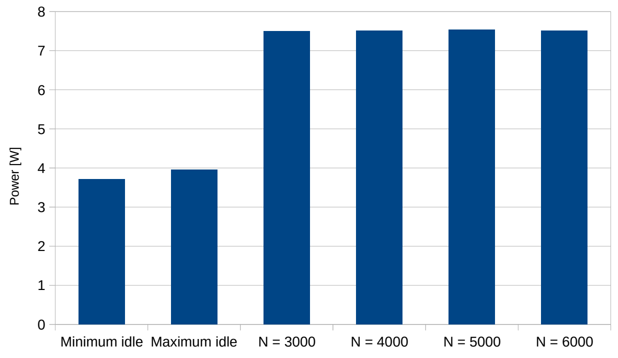


Fig. 7. Temporal (total) power measurement on the NVIDIA Jetson board for single precision matrix-matrix multiply (**SGEMM**).

This kind of port of code may be achieved with minimal effort and the reported performance is meant to show little value of sharing the code between the two hardware platforms. The next curve represents performance of the specialized GPU-only code that is meant to counteract the deficiencies of the mobile CPU and highlight the superior performance available from the GPU across a spectrum of workloads: latency-bound and bandwidth-sensitve panel factorization as well as compute-bound trailing matrix update that combines matrix-matrix product (**SGEMM**) and triangular matrix product (**STRSM**). The GPU-only implementation asymptotically reaches close to the **SGEMM** performance which highlights the efficiency of our kernels. The hybrid routine cannot achieve high performance, or any reasonable performance for that matter, and in the figure its graph is very close to the horizontal axis due to the fact that the panel computation on the ARM Cortex A15 is very slow and the total time becomes prohibitive and bound by the CPU computation, which makes the overall performance close to zero. We would like to note, that this situation could be somewhat mitigated by a careful tuning of some of the latency-sensitive and bandwidth-bound computational kernels for the ARM CPU. However, with 30-fold difference in performance between the mobile CPU and GPU, we see no prospects of significant improvements.

D. Temporal Power Measurement

Figure 7 shows temporal power draw for the Jetson mobile development board as measured at the DC power input. Thus the power measurement includes everything that consumes electricity on the board with only the power supply power dissipation excluded.

The performance of 120 Gflop/s of **SGEMM** for some matrix sizes translates to 16 Gflop/s per Watt and this counts all the board components combined – not just the GPU.

IV. Conclusions and Future Work

In this paper, we presented the methodology of implementing numerical linear algebra routines on the contemporary mobile hardware platforms that feature accelerators. We have provided sufficient evidence that the much different performance balance between the mobile CPU and mobile GPU gives rise to new techniques for writing and optimizing code on the mobile parts. We show our methodology successfully applied on the mobile development board.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant ACI-1339822, the Department of Energy, and NVIDIA. The results were obtained in part with the financial support of the Russian Scientific Fund, Agreement N14-11-00190.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

[2] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

[3] Emmanuel Agullo, Jack Dongarra, Rajib Nath, and Stanimire Tomov. Fully empirical autotuned qr factorization for multicore architectures. *CoRR*, abs/1102.5328, 2011.

[4] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov. The impact of multicore on math software. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing, 8th International Workshop, PARA*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2006.

[5] C. Cao, J. Dongarra, P. Du, M. Gates, P. Luszczek, and S. Tomov. clmagma: High performance dense linear algebra with opencl. In *The ACM International Conference Series*, Atlanta, GA, may 13-14 2013. (submitted).

[6] Tingxing Dong, Jack Dongarra, Thomas Schulthess, Raffaele Solca, Stanimire Tomov, and Ichitaro Yamazaki. Matrix-vector multiplication and tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. *Parallel Comput.*, July 2012. (submitted).

[7] J. Dongarra, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and A. YarKhan. Model-driven one-sided factorizations on multicore accelerated systems. *International Journal on Supercomputing Frontiers and Innovations*, 1(1), June 2014.

[8] Jack Dongarra and Piotr Luszczek. Anatomy of a globally recursive embedded linpack benchmark. In *HPEC*, pages 1–6. IEEE, 2012.

[9] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.*, 38(8):391–407, August 2012.

[10] Massimiliano Fatica. Accelerating LINPACK with CUDA on heterogenous clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*, pages 46–51. ACM, New York, NY, USA, 2009. DOI 10.1145/1513895.1513901.

[11] A. Haidar, S. Tomov, J. Dongarra, R. Solca, and T. Schulthess. A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks. *International Journal of High Performance Computing Applications*, September 2012.

[12] Azzam Haidar, Chongxiao Cao, Asim Yarkhan, Piotr Luszczek, Stanimire Tomov, Khairul Kabir, and Jack Dongarra. Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 491–500, Washington, DC, USA, 2014. IEEE Computer Society.

[13] Azzam Haidar, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Optimization for performance and energy for batched matrix computations on gpus. In *8th Workshop on General Purpose Processing Using GPUs (GPGPU 8) co-located with PPOPP 2015*, PPoPP 2015, San Francisco, CA, 02/2015 2015. ACM, ACM.

[14] Azzam Haidar, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Towards batched linear solvers on accelerated hardware platforms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, San Francisco, CA, 02/2015 2015. ACM, ACM.

[15] Matrix algebra on GPU and multicore architectures (MAGMA), MAGMA Release 1.6.1, 2015. Available at http://icl.cs.utk.edu/magma/.

[16] Intel Math Kernel Library, 2014. Available at http://software.intel.com/intel-mkl/.

[17] Intel® 64 and IA-32 architectures software developer's manual, July 20 2014. Available at http://download.intel.com/products/processor/manual/.

[18] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 2009 International Conference on Computational Science, ICCS'09*, Baton Roube, LA, May 25-27 2009. Springer.

[19] R. Nath, S. Tomov, T. Dong, and J. Dongarra. Optimizing symmetric dense matrix-vector multiplication on GPUs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, nov 2011.

[20] R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *Proceedings of the 2009 International Meeting on High Performance Computing for Computational Science, VECPAR'10*, Berkeley, CA, June 22-25 2010. Springer.

[21] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, November 2010.

[22] Nvidia visual profiler.

[23] NVIDIA management library, 2014. Available at https://developer.nvidia.com/nvidia-management-library-nvml.

[24] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eliezer Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, March/April 2012. ISSN: 0272-1732, 10.1109/MM.2012.12.

[25] R. Schreiber and C. Van Loan. A storage efficient WY representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57, 1989.

[26] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parellel Comput. Syst. Appl.*, 36(5-6):232–240, 2010.

[27] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[28] Stanimire Tomov and Jack Dongarra. *Scientific Computing with Multicore and Accelerators*, chapter Dense Linear Algebra for Hybrid GPU-based Systems. Chapman and Hall/CRC, 2010.

[29] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02*, 2011.