A Fast Batched Cholesky Factorization on a GPU

Tingxing Dong*, Azzam Haidar*, Stanimire Tomov*, and Jack Dongarra*†‡
* Innovative Computing Laboratory
University of Tennessee, Knoxville
Knoxville, TN 37916
† Oak Ridge National Laboratory, USA
‡ University of Manchester, UK
{tdong, haidar, tomov, dongarra}@utk.edu

***Abstract*—Currently, state of the art libraries, like MAGMA, focus on very large linear algebra problems, while solving many small independent problems, which is usually referred to as batched problems, is not given adequate attention. In this paper, we proposed a batched Cholesky factorization on a GPU. Three algorithms – non-blocked, blocked, and recursive blocked – were examined. The left-looking version of the Cholesky factorization is used to factorize the panel, and the right-looking Cholesky version is used to update the trailing matrix in the recursive blocked algorithm. Our batched Cholesky achieves up to $1.8\times$ speedup compared to the optimized parallel implementation in the MKL library on two sockets of Intel Sandy Bridge CPUs. Further, we use the new routines to develop a single Cholesky factorization solver which targets large matrix sizes. Our approach differs from MAGMA by having an entirely GPU implementation where both the panel factorization and the trailing matrix updates are on the GPU. Such an implementation does not depend on the speed of the CPU. Compared to the MAGMA library, our full GPU solution achieves 85% of the hybrid MAGMA performance which uses 16 Sandy Bridge cores, in addition to a K40 Nvidia GPU. Moreover, we achieve 80% of the practical dgemm peak of the machine, while MAGMA achieves only 75%, and finally, in terms of energy consumption, we outperform MAGMA by 1.5× in performance-per-watt for large matrices.**

I. Introduction

Solving many small linear algebra problems is called batched problem. A batched problem consists of a large number of matrices (e.g., from thousands to millions matrices) to be factorized, where the size of each matrix is considered to be small (e.g., typically the size is around hundreds of rows or columns). For example, batched Cholesky factorization is widely used in computer vision, and anomaly detection of images [1], [2]. In Magnetic resonance imaging (MRI), billions small 8x8 and 32x32 eigenvalue problems need to be solved. Also, a batched 200x200 QR decomposition is required to be computed in radar signal processing [3]. Hydrodynamic simulations need to compute thousands of matrix-matrix (dgemm) or matrix-vector(dgemv) products of matrices of well over 100x100 [6]. NVIDIA also includes batched dgemm, LU and dtrsm in their recent CUBLAS release. Motivated by these applications, we proposed a batched Cholesky factorization that targets many small matrices.

The one sided factorizations such as the Cholesky, Gauss, and Householder factorizations are based on block outer-product updates of the trailing matrix. Algorithmically, this corresponds to a sequence of two distinct phases: panel factorization and trailing matrix update. Implementation of these two phases leads to a straightforward iterative scheme shown in Algorithm 1. Table I shows BLAS and LAPACK routines that should be substituted for the generic routines named in the algorithm.

Algorithm 1 Two-phase implementation of a one-sided factorization.

```
for $P_i \in \{P_1, P_2, \ldots, P_n\}$ do
    PanelFactorize($P_i$)
    TrailingMatrixUpdate($C^{(i)}$)
end for
```

	Cholesky	Householder	Gauss
PanelFactorize	xPOTF2 xTRSM	xGEQF2	xGETF2
TrailingMatrixUpdate	xSYRK xGEMM	xLARFB	xLASWP xTRSM xGEMM

TABLE I. Routines for panel factorization and the trailing matrix update.

MAGMA currently focuses on the performance of very large matrices using a hybrid (CPU-GPU) solution [4]. Since the panel consists of Level 2 BLAS operations, and hence is memory bound, MAGMA use the CPUs to performs these operations, "the panel factorization" and the GPU to update the trailing matrix. Note that in order to perform the update of the trailing matrix on the GPU, a memory transfer of the factorized panel from the CPU to the GPU is required in each step. By using an efficient scheduling technique, this memory transfer can be overlapped with GPU computation. Although this hybrid model makes use of both of the computing resources, sometimes it might be a bottleneck. In particular, when the GPU is working, the CPU might be needed to perform other work, such as I/O, and thus cannot be interleaved and synchronized with the GPU in every step. Moreover, many clusters have weak CPUs and slow PCI-E connections, so the panel factorization phase and the memory transfer becomes very slow, thereby affecting the overall performance of the hybrid algorithm. In these cases, a full-GPU implementation might be of great interest for many applications and users. In order to make our implementation cover the classical case, we propose a full-GPU implementation of the classical single Cholesky factorization dpotrf targeting toward large matrix.

II. Related work

Hatem et al. presented left looking a Cholesky factorization for multicore with GPU accelerators [8]. Volkov et al. implemented LU, Cholesky, and QR with right-looking on 8 GPUs [9]. Yang et al. factorized Cholesky on both FPGAs and GPUs with right-looking [10]. Molero et al. developed a batched Cholesky solver for the matrix in the hyperspectral image processing[1]. Their matrix size is around hundreds.

Our paper is organized as follows. First, we describe the Cholesky algorithms in III. Then, we detail our batched implementations and demonstrate their performance in IV. The performance and power of the CPU, the GPU and the hybrid Cholesky implementations are compared in V. Finally, we conclude in VI.

III. Algorithms

The Cholesky factorization (or Cholesky decomposition) is mainly used as a first step for the numerical solution of linear equations $Ax = b$, where A is symmetric and positive definite. Such systems often arise in physics applications, where A is positive definite due to the nature of the modeled physical phenomenon.

The Cholesky factorization of an $n \times n$ real symmetric positive definite matrix A has the form $A = LL^T$, where L is an $n \times n$ real lower triangular matrix with positive diagonal elements. In LAPACK, the double precision algorithm is implemented by the dpotrf routine. A single step of the algorithm is implemented by a sequence of calls to the LAPACK and BLAS routines: dsyrk (symmetric rank-k update), dpotf2 (unblocked Cholesky factorization), dgemm (general matrix-matrix multiplication), dtrsm (triangular solver). Throughout the paper, we take the double precision as an example to describe how we implemented, though other precisions, including single, single complex and double complex are also implemented.

A. Non-blocked Cholesky factorization

The following notations will be used throughout the rest of the paper: $a(i,j)$ is used to denote the (i,j) element of the matrix A. The submatrix consisting of i-th through j-th row and m-th through n-th column is denoted as $a(i:j, m:n)$.

A non-blocked Cholesky factorization (dpotf2) is outlined in Figure 1. Due to the symmetry, the matrix can be factorized either as an upper triangular matrix or as a lower triangular matrix (e.g., only the shaded data is accessed if the lower side is to be factorized). If A is $n \times n$, there are n steps. Steps go from the upper left corner to lower right corner along the diagonal. At a step j, the column vector $a(j:n,j)$ is to be computed. First, a dot product of the row vector $a(j, 0:j)$ is needed to update the element $a(j,j)$(in black). Then the column vector $a(j+1 : n-1, j)$ (in red) is updated by a dgemv $a(j+1 : n-1, 0 : j-1) \times a(j, 0 : j-1)$ followed by a scaling operation with the updated element $a(j,j)$. This non-blocked Cholesky factorization involves two Level 1 BLAS routines (dot and scal), as well as a Level 2 BLAS routine dgemv. Since there are n steps, these routines are called n times and thus one can expect that the performance of this variants will depend on the performances of Level 2 and Level 1 BLAS operations, hence it is a slow memory bound algorithm.

B. Blocked right-looking

The blocked right-looking algorithm is described in Algorithm 2 and depicted in Figure 2. The factorization of the $n \times n$ matrix A proceeds in n/nb steps of size nb. A single step is implemented by a sequence of calls to the BLAS and the LAPACK routines: dpotf2 (unblocked Cholesky factorization), dtrsm (triangular solve) and dsyrk (symmetric rank-k update) as described in Algorithm 2.

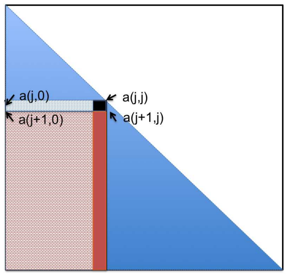

Fig. 1. Non-blocked Cholesky factorization

Once a panel A_iB_i at a step i is computed it will never be accessed. The trailing matrix C_i is now considered as a new matrix and the loop is repeated. This algorithm keeps updating the right hand side trailing matrix C_i, so it is called right-looking. Note that the dtrsm and the dsyrk routines are Level-3 BLAS [11], thus they perform efficiently on both CPUs and GPUs architectures, for this reason, the blocked implementation performs very well and reaches high flops per seconds.

Algorithm 2 The blocked right looking Cholesky factorization.

```
for $i \in \{1, 2, 3, \dots, n/nb\}$ do
    Panel Factorize $L_i$ :=Cholesky($A_i$) (dpotf2)
    Compute $B_i = B_i(L_i^T)^{-1}$ (dtrsm)
    Trailing Matrix Update $C_i = C_i - B_iB_i^T$ (dsyrk) where $C_i = a(i \times nb : n, i \times nb : n)$
end for
```

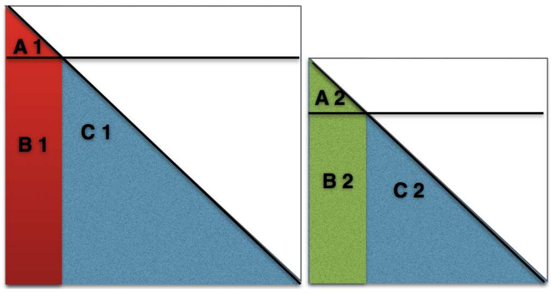

Fig. 2. Blocked right-looking

C. Blocked left-looking

The difference between the left-looking and the right-looking variants is in the update of the trailing matrix. The right-looking variant operates in a panel and applies its corresponding updates to the right (see Figure 2). The left-looking applies all updates coming from the left up, to the current panel, then factorize it, (as described in Algorithm 3), and therefore delays subsequent updates of the remaining right side columns of the matrix. For example, in the second step of Figure 3, the panel A_2B_2 is first updated by the resulting L_i of step 1 then factorized and so on for next steps. Yet, in the update of panel A_3B_3, the data in B_1 and B_2 will be read again. Because this algorithm needs to access all the previous panel matrices B_i in the left side, it is called left-looking.

Algorithm 3 The blocked left looking Cholesky factorization.

```
for $i \in \{1, 2, 3, \dots, n/nb\}$ do
    if (i $>$ 1) then
        Update Current panel $A_iB_i = A_iB_i - (T)'_{i-1}(T)'^{T}_{i-1}$ (dgemm), where $(T)'_{i-1} = a((i-1) \times nb : n, 0 : (i-1) \times nb)$
    end if
    Panel Factorize $L_i$ :=Cholesky($A_i$) (dpotf2)
    Compute $B_i = B_i(L_i^T)^{-1}$ (dtrsm)
end for
```

Both the right-looking and the left-looking variants have the same costs, $n^3/3$ operations. Previous study showed that there is little performance difference in the serial code. However, one can be favored than the other in a parallel design. The right-looking variant generates more parallelism, but also has more writes since the output matrix is large compared to a small input, while the left-looking variant emphasizes the data locality but have more reads. This difference is important to our CUDA code implementations, and we found that a merged implementation of both into a recursive algorithm can provide us the best performance.

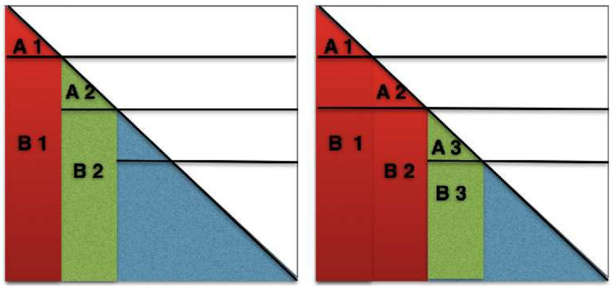

Fig. 3. Blocked left-looking

D. Recursive blocked Cholesky

In this section we propose a recursive mixed implementation of both left and right looking variants. Our main algorithm proceeds as a right looking variant by steps of size nb. The difference comes from improving the panel factorization, which consists of the Level 2 BLAS operations dpotf2 and dtrsm. In order to achieve higher performance, especially on GPU architecture, we should make efforts to increase the use of blocked techniques. The panel matrix $A_i = L_i L_i^T$ (dpotf2) and the triangular solve $B_i = B_i (L_i^T)^{-1}$ (dtrsm) of Algorithm 2 can also be factorized using the blocked algorithm instead of dpotf2 [12]. In theory, the matrix can be blocked recursively until the blocking size can equal to a single element. For that, we create a second level of blocking by developing a new blocked CUDA implementation of the panel factorization (dpotf2+dtrsm) routines. However, achieving high performance is not straightforward. We proposed a mixed left-right looking recursive technique to factorize each panel and replace the (dpotf2+dtrsm) routines. The panel factorization of $A_i = A(i : n, nb)$ follows a recursive pattern as described below.

Algorithm 4 The blocked right looking Cholesky factorization.

```
define $ib = nb$
for $i \in \{1, 2, 3, \ldots, n/nb\}$ do
    Panel Factorize of $A(i \times nb : n, nb)$
    a- define $ib = ib/2$
    for $k \in \{1, 2\}$ do
        if $ib < minblock$ then
            unblocked panel factorization dpotf2
        else
            go to (a)
        end if
        b-update next panel using dgemm and dtrsm
    end for
    Trailing Matrix Update $C_i = C_i - B_i B_i^T$ (dsyrk) where $C_i = a(i \times nb : n, i \times nb : n)$
end for
```

IV. Batched Implementation and Performance on a GPU

A. Hardware Description and Setup

We conducted our experiments on a Intel multicore system with dual-socket, 8-core Intel Xeon E5-2670 (Sandy Bridge) processors, each running at 2.6 GHz. Each socket had 20 MB of shared L3 cache, and each core had a private 256 KB L2 and 64 KB L1 cache. The system is equipped with 52 GB of memory and the theoretical peak in double precision is 20.8 Gflop/s per core. The TDP (Thermal Design Power) of each socket is 115Watts. It is also equipped with a NVIDIA K40c cards with 11.6 GB memory per card running at 825 MHz, connected to the host via two PCIe I/O hubs at 6 GB/s bandwidth. The TDP of K40c is 235Watts. A number of software packages were used for the experiments. On the CPU side, we used the MKL (Math Kernel Library) [5] and on the GPU accelerator we used CUDA version 5.5.

B. Batched CUDA routines

In a batched problem, there are many small dense matrices that must be factorized simultaneously. Each matrix consists of an independent Cholesky problem, where the factorization itself is a sequence of BLAS calls. A natural way to implement this batched model in CUDA is to organize it as a sequence of batched BLAS routines. This means that all the matrices will be processed simultaneously by the same kernel. Yet, each matrix problem is still solved independently, identified by a unique batch ID. We follow this model in our batched implementations and developed a set of new batched CUDA kernels. For the remainder of the paper, the routine name refers to a batched version of the respective routine without explicitly noted.

The batched CUDA routines that we implemented and study in this paper are:

dsyrk,

dot,

scal,

dgemv, and

dgemm.

C. Batched non-blocked Cholesky

The performance of the non-blocked Cholesky is shown in Figure 7. A K40 GPU is used for the tests throughout the paper. A breakdown of the execution time is shown in Figure 4. It shows that when the matrix size increase, the dgemv dominates the overall execution time. Since dgemv is a bandwidth bound routine, the algorithm's performance will be limited by the GPU's bandwidth. The performance achieved by the standard dgemv routine is shown in Figure 5.

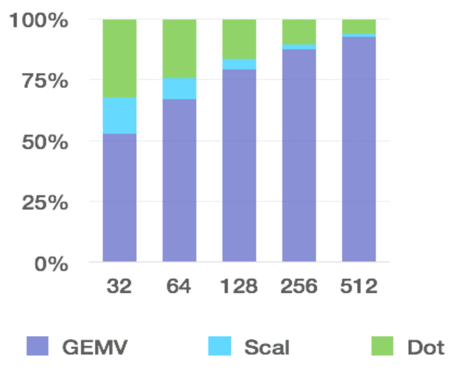

Fig. 4. Execution time breakdown of the non-blocked Cholesky factorization for matrices of different sizes on a K40 GPU

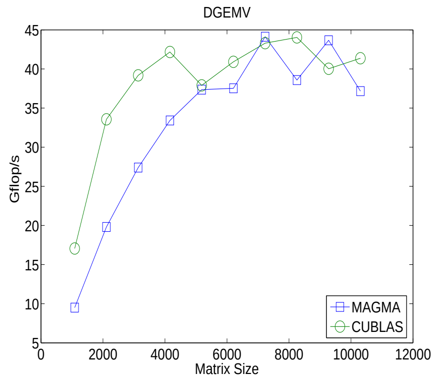

Fig. 5. Performance in Gflop/s achieved by the dgemv on a K40 GPU

D. Batched blocked right-looking Cholesky

This implementation follows the steps described in Algorithm 2, but it differ by merging the factorization of the panel submatrix A_i and B_i into one kernel to minimize the overhead of calling CUDA kernel for small tasks. The trailing matrix C_i in Algorithm 2 is updated using the Level 3 BLAS routine dsyrk, as shown in Figure 2. Compared to the non-blocked algorithm, a large number of Level 2 BLAS dgemv operations are replaced by Level 3 BLAS dsyrk operations in the blocked right-looking.

The performances using various batch sizes is shown in Figure 6. For matrices of size larger than 500, the performance impact of increasing the batch count is insignificant, because the streaming multi-processors of the GPU are slowly getting saturated. The blocking size of our algorithm is tunable, and experimentally we determined that the optimal size for a K40 GPU is four. The performance of the blocked algorithm slightly exceeds the non-blocked one, as shown in Figure 7. Although this algorithm outperforms the bandwidth-limited dgemv, it is still not very satisfactory (see next).

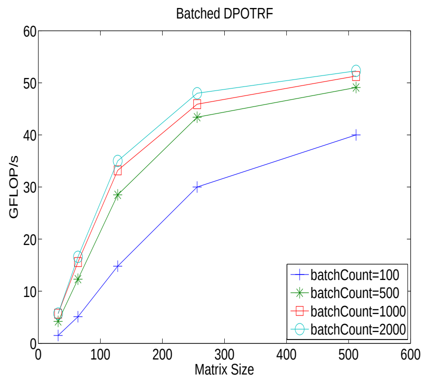

Fig. 6. Performance of the batched blocked right-looking Cholesky for various matrix and batch sizes

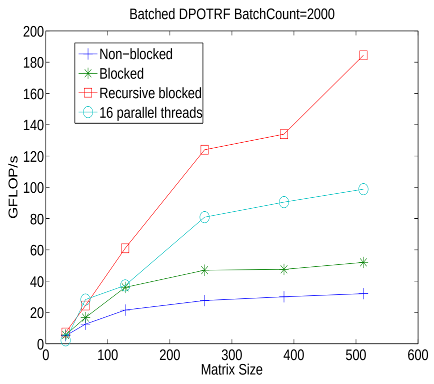

Fig. 7. Performance of the three algorithms

E. Batched recursive blocked Cholesky

The technique of the batched recursive Cholesky is similar to the one described in Algorithm 4. The reason for this choice is that on the first hand, the right looking variant used for updating the trailing matrix provides a high level of parallelism – the update is for the entire trailing matrix – and thus can be performed efficiently on a GPU. On the other hand, the panels are factorized using the recursive left schema. This provides better results as it minimizes the costly writes back to the main memory by keeping the data in cache for its reuse, and thus writes back the final result only once and also it recursively increases the inner blocking of the local panel operations which will gives a Level 2.5 BLAS. Note that caching is possible because of the small panel sizes.

The performance of the three algorithms described above, along with the comparison to an optimized parallel batched Cholesky on CPUs using the Intel MKL library, is shown in Figure 7. The matrix sizes range from 32 to 512. To be fair, we tuned the CPU implementation as optimal as we can. A simple way of using the CPU batched Cholesky is to call the multi-thread version of the MKL Library and to factorize the small matrices one after one. Such implementation provided a bad performance around 30 Gflop/s. Since the matrix size is in an order of hundreds, thus more than 8 small matrices can fit into the L3 cache level of the CPU and so an optimized CPU implementation might be achieved by threading "*pthread*" independent sequential factorization on each thread using the sequential MKL BLAS. Our experiments shows that this technique is the best among many other CPU implementations. Therefore, for the batched problem the threading should be on the batched level rather than inside the processing of each matrix. In our test, 2,000 matrices were distributed onto 16 OpenMP threads running sequential MKL BLAS. The results shows that except for matrices of size 64, where the matrix fit the private L2 cache level of each thread which makes the CPU variant faster, our proposed recursive blocked algorithm on the GPU always gives the best performance. Compared to MKL, the recursive algorithm achieves up to 1.8 $\times$ speedup.

The recursive blocked algorithm has the least number of BLAS-2 operations and achieves a performance that is characteristic for Level 3 BLAS. However, all the algorithms have the same amount of BLAS-1 scal and dot routines. A breakdown of the time is shown in Figure 8. A specific CUDA dgemm kernel that we developed is used in the left looking panel factorization. dsyrk is used in the update of the right looking trailing matrix. The optimal blocking size in the recursive algorithm, which is the size of the panel matrix A, is 32. The optimal blocking size of A is 8. Since 32 is the panel size, there is no dsyrk at size 32. For small matrix sizes, the performance of the dot product is critical for the overall performance. The dot product (reduction) is performed along the rows, resulting in consecutive threads reading elements at a step of lda. Since the data is stored in column major, the memory accesses are non-coalesced, which has negative effect on its performance. With the matrix size increasing, this reduction is not significant compared to the more flops intensive dsyrk routine.

F. Comparison with CUBLAS batched routines

CUBLAS does not have a batched dsyrk routine, but has a batched dgemm – cublasDgemmBatched. To be a fair, we compared it with our batched dgemm, as shown in Figure 9. Our batched dgemm is 2 $\times$ faster than the CUBLAS routine. In our code, we use our batched dsyrk version which is 4 $\times$ faster (than a dsyrk based on the batched dgemm from CUBLAS). The only difference is our dsyrk writes in the lower triangular part of the output matrix. The performance of dsyrk is important to the overall performance, since it takes a big part of the running time. Our test showed that if we took the cublasDgemmbatched, the overall performance will be down to 100 Gflop/s at size 512.

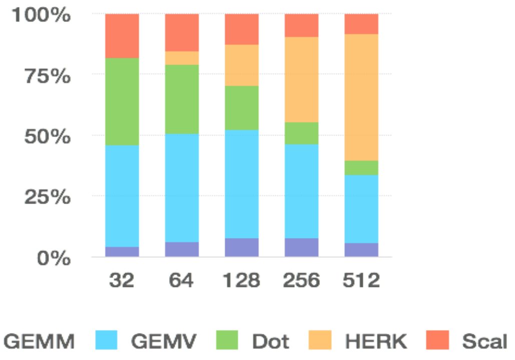

Fig. 8. A breakdown of the recursive blocked Cholesky

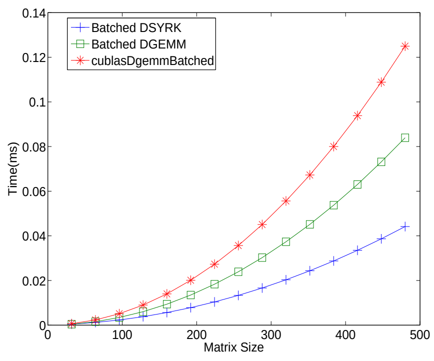

Fig. 9. Performance of our batched DSYRK, batched DGEMM and cublasDgemmbatched

CUBLAS has included a batched LU (getrfBatched) since version 5.5, but does not have a batched Cholesky. Cholesky can be treated as a special version of LU decomposition tailored to symmetric and positive definite matrices. For the same data input, LU is two times slower than Cholesky, because LU accesses the whole matrix instead of a lower or upper side. We compared our implementation with cublasDgetrfBatched and our Cholesky is up to 9$\times$ faster than the CUBLAS routine as shown in Figure 10.

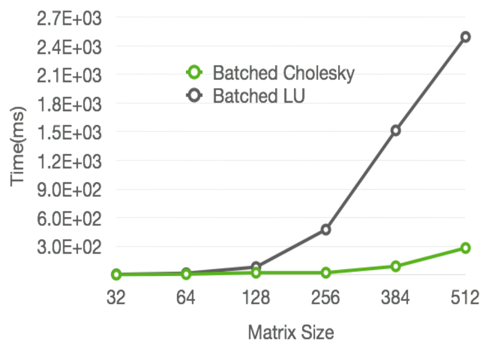

Fig. 10. Performance of our batched Cholesky vs cublasDgetrf-Batched

V. CHOLESKY FOR LARGE MATRICES AND ITS ENERGY CONSUMPTION

In contrast to the batched problems on small matrices from the previous sections, this section focuses on a classical Cholesky dpotrf for large matrices. The entire matrix size is at least a few thousands, while its panel size is in the order of hundreds. The difference between MAGMA and our implementation is that the factorization of the panel is performed on the GPU, while in MAGMA is on the CPU. The differences are shown in Table II. In this section, we call our implementation full-GPU-dpotrf to make the name more self-explanatory.

MAGMA has adopted the left-looking Cholesky algorithm. The panel factorization on the CPU is overlapped and hidden by dgemm computations on the GPU [7].

The panel factorization in our algorithm is based on our batched recursive blocked algorithm.Since the panel size is small and therefore its computation will not saturate all computational resources of the GPU, one can try to overlap the panel computation with the trailing matrix update. One way of doing it, is to put the two computations through two different CUDA streams. However, experiments showed that, it might not be fully overlapped with the dgemm computations. The first explanation of this behavior is that the CUDA kernels are non-preemptive [17]. Once a kernel is issued and starts running on the GPU, it will try to occupy all the computational resources it needs. If the kernel uses all the computing units of the GPU, then no other kernel can be started. Thus the CUDA scheduler might not be able to initiate and overlap all the small BLAS in the recursive blocked panel with the large dgemm computation. Therefore, the panel factorization will prevent the full-GPU-dpotrf algorithm to match the MAGMA performance. In our case, since the dgemm computation for the trailing matrix update is large, it takes up all the resources after it is issued. Only close to the end of its computation, a few of the panel factorization kernels can be launched, as shown in Figure 12. Therefore, the panel factorization can not be completely overlapped. Despite this, our profiler show that the GPU is fully busy doing either the panel factorization or the trailing matrix update.

TABLE II. THREE CHOLESKY IMPLEMENTATIONS

Name	Panel Matrix A	Panel Matrix B	Trailing Matrix C
MKL	CPU	CPU	CPU
MAGMA	CPU	GPU	GPU
full-GPU-dpotrf	GPU	GPU	GPU

The performance of the three implementations is shown in Figure 11. For small matrix sizes, less than 1,500, MKL is the fastest. This is expected as matrices of size up to 2,200 fit the L3 cache of the CPU. For matrices larger than 2,200, MKL stagnates at the same performance which is the peak it can reach. MAGMA's performance rises quickly before size of 10,000, and outperforms the full-GPU-dpotrf by 300 Gflop/s at 10,000. After that, MAGMA's performance slowly levels, while the full-GPU-dpotrf's performance continues to rise steadily. The difference is narrowed to less than 100 Gflop/s at around matrices of size 25,000. Since the practical peak performance of the CPUs is around 300 Gflop/s, we consider that compared to MAGMA which uses the CPUs, a difference below 200 Gflop/s to be acceptable, and for less than 100 Gflop/s to be very good. We compute the ratio that can be reached by either of the Cholesky algorithms in proportion to the resources used. Our implementation reaches around 80% of the available practical dgemm peak on the K40c GPU which is around 1200 Gflop/s, while MAGMA reaches around 75% of the practical peak of the resources it uses (1200+300 Gflop/s).

A. *Performance-per-watt*

Besides performance, energy consumption has become another major concern in HPC. An indicator for it is given by the performance-per-watt measure. As Eq. 3 demonstrates, it evaluates how many flops are performed for one joule of energy. The higher the number, the more efficient the computation is.

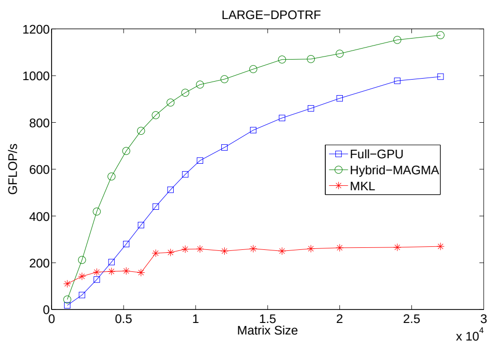

Fig. 11. Performance of the full-GPU, MAGMA Hybrid and MKL solution of Choleksy factorization

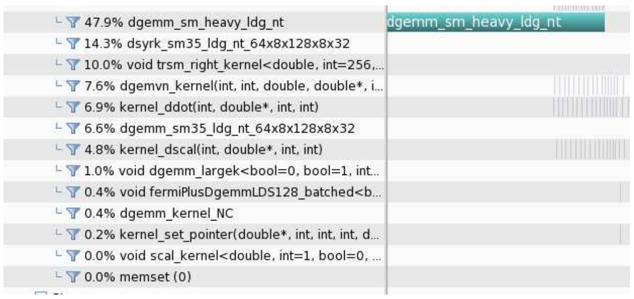
```
47.9% dgemm_sm_heavy_ldg_nt
14.3% dsyrk_sm35_ldg_nt_64x8x128x8x32
10.0% void trsm_right_kernel<double, int=256,...
7.6% dgemvn_kernel(int, int, double, double*, i...
6.9% kernel_ddot(int, double*, int, int)
6.6% dgemm_sm35_ldg_nt_64x8x128x8x32
4.8% kernel_dscal(int, double*, int, int)
1.0% void dgemm_largek<bool=0, bool=1, int...
0.4% void fermiPlusDgemmLDS128_batched<b...
0.4% dgemm_kernel_NC
0.2% kernel_set_pointer(double*, int, int, int, d...
0.0% void scal_kernel<double, int=1, bool=0, ...
0.0% memset (0)
```

Fig. 12. The CUDA profiler showed that only a few routines in panel factorization were able to be launched at the end stage of dgemm. The vertical lines are the footage of the routines. The width represents the running time.

$$Performance(Flop/s) = \frac{Flops}{Time(Sec)} \tag{1}$$

$$Power(Watt) = \frac{Energy(Joule)}{Time(Sec)} \tag{2}$$

Dividing equation 1 by 2 give:

$$Performance - per - watt = \frac{Flops}{Joule} \tag{3}$$

We use PAPI to measure the CPU power [14] [15] and NVML to measure the GPU power [16]. The measurement frequency provided by these tools is one millisecond. The K40 GPU power usage of the MAGMA hybrid and the full-GPU-dpotrf are shown in Figures 13 and 15, respectively. To collect more meaningful data, we made one hundred runs for each routine. This is needed because the kernels for small matrix sizes run at hundreds of microseconds, which is less than the resolution (millisecond) provided by PAPI and NVML.

While the hybrid MAGMA runs faster, its power is also higher than that of the full-GPU solution for the same matrix size, because the GPU during the MAGMA run only performs the flop intensive BLAS-3 routines.

The two socket CPU power usage of the three implementations is shown in Figures 14, 16 and 17. The red line is one socket and the blue line is the other socket. In these tests, we gradually increase the matrix size from 1,088 to 10,320. Each spike represents one run. The bigger the matrix, the longer the run time and the wider the bar. From the power usage, we can see that the hybrid and the full-GPU-dpotrf only use a small part of the CPU capability, because they only achieved 50-70 W, while MKL achieves 110 W for one socket. This indicates that a less powerful CPU is enough to achieve the same performance. For the full-GPU Cholesky the CPU is only used to drive the GPU code. CPU's narrow power bars in this case mean that the CPU is at the idle power most of the time.

The performance and GPU power of the full-GPU-dpotrf is shown in Figure 18. With the size increasing, the performance increases linearly, but the power rises slowly and levels off.

The performance-per-watt is shown in Figure 19. The watts include both CPU and GPU power. For the full-GPU-dpotrf we consider two metrics:

- Full-GPU-1: where we consider the CPU power
- Full-GPU-2: where we do NOT consider the CPU power

We consider the Full-GPU-2, because the watt usage in the Full-GPU-1 are exaggerated since a less powerful CPU using less watts is able to achieve the same GPU performance. Second, we want to compare it with the peak performance-per-watt of the GPU. The peak performance-per-watt of a K40 in double precision is 6 Gflops/W [18]. It is interesting that the trend for the performance-per-watt in Figure 19 is almost the same as that in Figure 11. At the very beginning, MKL is the best, since the GPU wake-up power is a big contributor to other curves. For moderate matrix sizes in the range of 2,000 to 5,000, MAGMA performs the best due to its performance advantage in this range. Its performance-per-watt then levels off, as its performance also levels off. After that, the full-GPU takes the lead since its performance increases linearly before 10K, but the power levels off as in Figure 18. The performance-per-watt for our Cholesky factorization is 4.5, compared to 3 for MAGMA at matrix of size 10K. In all, our performance-per-watt is 75% of the theoretical performance-per-watt for the K40.

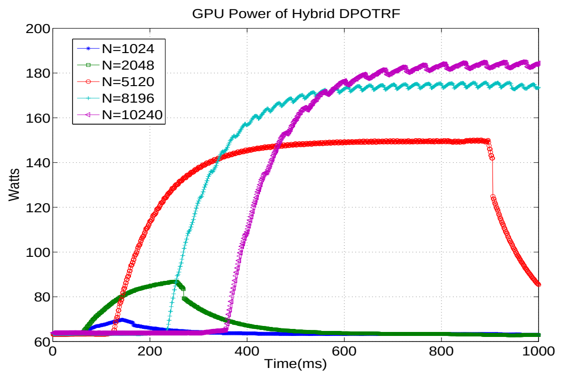

Fig. 13. GPU power of the MAGMA hybrid factorization.

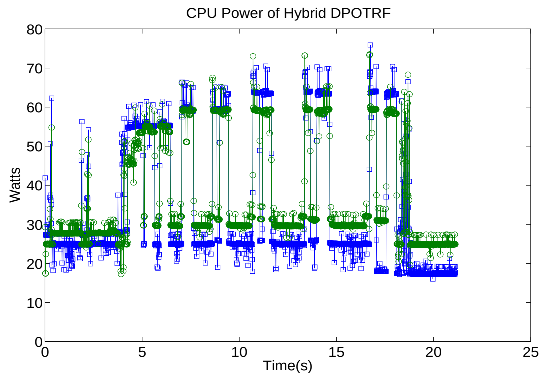

Fig. 14. CPU power of the MAGMA hybrid factorization.

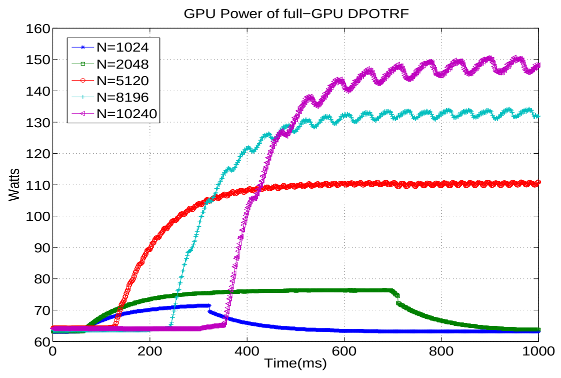

Fig. 15. GPU power of the full-GPU factorization.

VI. Conclusion

We designed different techniques for developing high-performance batched dense linear algebra kernels in a GPU accelerator environments. In particular, we have implemented a batched Cholesky factorization using GPUs

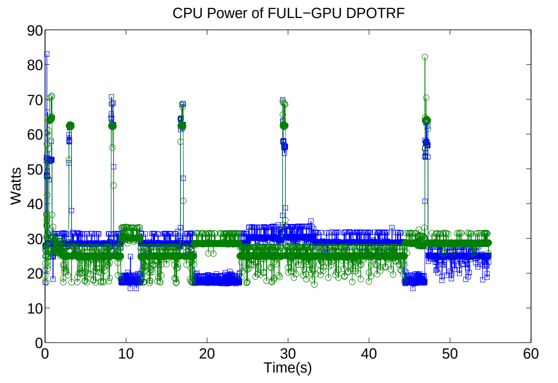

Fig. 16. CPU power of the full-GPU factorization.

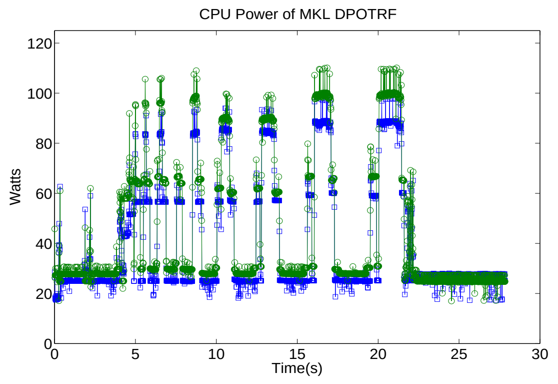

Fig. 17. CPU power of MKL with 32 threads.

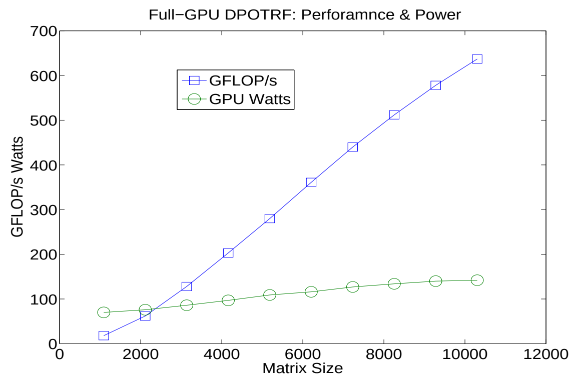

Fig. 18. Performance and power of the full-GPU Cholesky factorization.

hardware targeting thousands of small matrices of size hundreds by hundreds. We compared three variants of the algorithm, the non-blocked, the blocked right-looking and the recursive blocked left-right-looking implementation. The performance of the non-blocked version is bounded by the performance of the Level 2 BLAS routine dgemv. The blocked right-looking performs better than

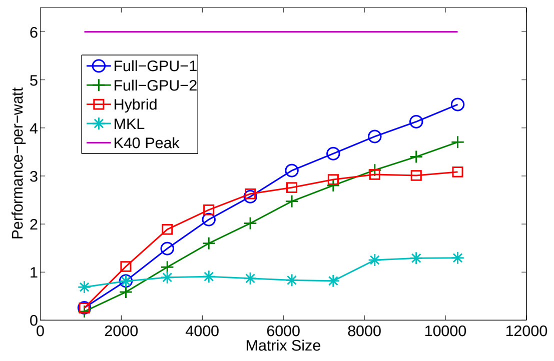

Fig. 19. Performance-per-watt.

the non-blocked one. We adopted a recursive hybrid algorithm, where a recursive left-looking technique is used in the panel factorization while a right-looking one is used in the update of the trailing matrix. To maximize the performance, we implemented and optimized a set of batched new CUDA kernels (routines). Their performance exceeded their counterparts in CUBLAS by 2 $\times$ and 9 $\times$. Our CUDA code achieved up to 1.8 $\times$ speedup than an optimized implementation that uses the Intel MKL library on two sockets of Intel Sandy Bridge CPUs.

We also implemented a full GPU implementation of the Cholesky factorization targeting to a larger matrix on the GPU. Compared to the MAGMA hybrid solution, our full-GPU solution achieved 85% of the practical peak performance with 1.5$\times$ performance-per-watt.

Furthermore, despite the complexity of the hardware, acceleration was achieved at a surprisingly low software development effort using a high-level methodology of developing hybrid techniques. In particular, we obtained high fraction of the practical peak performance of the GPU. The promise shown so far motivates and opens opportunities for future research and extensions, e.g., tackling more batched one-sided factorizations, like the Gaussian elimination "LU", and the QR decomposition and also batched eigenvalue solver.

Acknowledgment

The authors would like to thank the National Science Foundation, the Department of Energy, NVIDIA and MAGMA project support.

References

[1] J.M. Molero, E.M. Garzn, I. Garca,E.S. Quintana-Ort and A. Plaza. Poster: A Batched Cholesky Solver for Local RX Anomaly Detection on GPUs. PUMPS

[2] https://devtalk.nvidia.com/default/topic/527289/help-with-gpu-cholesky-factorization-/

[3] Anderson, M.J. Sheffield, D. Keutzer, K. A Predictive Model for Solving Small Linear Algebra Problems in GPU Registers, Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International

[4] http://icl.cs.utk.edu/magma/

[5] Intel Math Kernel Library. http://software.intel.com/intel-mkl/.

[6] Tingxing Dong, Veselin Dobrev, Tzanio Kolev, Robert Rieben, Stanimire Tomov, Jack Dongarra A Step towards Energy Efficient Computing: Redesigning A Hydrodynamic Application on CPU-GPU. Parallel Distributed Processing Symposium (IPDPS), 2014 IEEE 28th International

[7] Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra One-sided dense matrix factorizations on a multicore with multiple GPU accelerators in MAGMA. International Conference on Computational Science, ICCS 2012

[8] Hatem Ltaief , Stanimire Tomov , Rajib Nath , Jack Dongarra Hybrid Multicore Cholesky Factorization with Multiple GPU Accelerators. University of Tennessee Computer Science Technical Report, 2010.

[9] Vasily Volkov, James W. Demmel. LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs LAPACK Working Note 202

[10] Depeng Yang, Junqing Sun, JunKu Lee, Getao Liang, David D. Jenkins, Gregory D. Peterson, and Husheng Li. Performance Comparison of Cholesky Decomposition on GPUs and FPGAs, Symposium on Application Accelerators in High Performance Computing, 2010

[11] Gallivan, K., Jalby, W., and Meier, U. 1987. The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory. SIAM J. Sci. Stat. Comp. 8, 10791084.

[12] Gustavson, F. G. 1997. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. 346 IBM J. Res. Dev. 41, 6, 737755.

[13] CUDA Programming Guide v5.0: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[14] V.M. Weaver, M.Johnson, K.Kasichayanula, J.Ralph, P.Luszczek, D.Terpstra, S.Moore. *Measuring Energy and Power with PAPI*, Parallel Processing Workshops, 2012 41st International Conference on Sep, 2012

[15] Intel 64 and IA-32 Architectures Software Developer's. http://download.intel.com/products/processor/manual/

[16] https://developer.nvidia.com/nvidia-management-library-nvml

[17] Jon Calhoun, Hai Jiang, "Preemption of a CUDA Kernel Function," snpd, pp.247-252, 2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2012

[18] http://www.nvidia.com/object/tesla-servers.html