Optimizing GPU Kernels for Irregular Batch Workloads: A Case Study for Cholesky Factorization

Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra
Innovative Computing Laboratory
University of Tennessee
Knoxville, USA
{ahmad,haidar,tomov,dongarra}@icl.utk.edu

***Abstract*—This paper introduces several frameworks for the design and implementation of high performance GPU kernels that target batch workloads with irregular sizes. Such workloads are ubiquitous in many scientific applications, including sparse direct solvers, astrophysics, and quantum chemistry. The paper addresses two main categories of frameworks, taking the Cholesky factorization as a case study. The first uses host-side kernel launches, and the second uses device-side launches. Within each category, different design options are introduced, with an emphasis on the advantages and the disadvantages of each approach. Our best performing design outperforms the state-of-the-art CPU implementation, scoring up to $4.7\times$ speedup in double precision on a Pascal P100 GPU.**

***Index Terms*—Matrix Factorization, Batch Linear Algebra, GPU Computing**

I. Introduction

A batch routine applies the same operation on many independent problems, potentially in a parallel fashion. In the context of dense linear algebra, a batch routine applies a basic linear algebra subprogram (BLAS) or Linear Algebra PACKage (LAPACK) operation to an ideally large number of relatively small independent matrices. The batch can have problems of the same size (fixed size) or different sizes (variable size).

Many higher-level solvers and scientific applications have a dependency on high-performance batch dense linear algebra software. In fact, the absence of a mature software for such workloads has sparked some in-house developments of batch routines for specific purposes. For example, batch LU factorization has been used in subsurface transport simulation [1], [2]. Batch Cholesky factorization and triangular solve have been also used to accelerate an *Alternating Least Square* (ALS) solver [3], [4]. Batch matrix-matrix multiplication (`GEMM`) is at the core of many tensor contraction problems [5], [6]. Sparse linear algebra software, such as textSuiteSparse,[1] has huge dependencies on many batch BLAS and LAPACK routines [7], [8], including matrix multiplication, one-sided factorization (LU, QR, and Cholesky), and matrix inversion [9]. The workload pattern of small independent problems is also very important to computations on Hierarchical matrices (H-matrices) [10].

Most numerical linear algebra libraries are specifically designed and tuned to perform well on large problem sizes. For example, the Matrix Algebra on GPU and Multicore Architectures (MAGMA) library[2] uses a hybrid CPU-GPU design for its implementation of most LAPACK algorithms. This is a nearly perfect strategy for large problems. MAGMA is built on the assumption that trailing matrix updates on the GPU can hide both the CPU activity and the CPU-GPU communication [11]. Such an assumption is invalid for small problems: the computation becomes more memory bound, and the communication turns out to be the bottleneck. Some recent efforts, however, have investigated a GPU-only design for large problems [12]. Such work can benefit from this paper by using the proposed optimized GPU kernels for memory-bound operations.

This paper introduces several design options to address variable-size batch computation on GPUs. The paper discusses two different design approaches. The first uses host-launch based solutions, with an emphasis on different scheduling policies for thread block execution. The second approach is based on the recent *dynamic parallelism* technology, where a GPU kernel can be launched from within another GPU kernel. Further, we address different design techniques for efficiently utilizing dynamic parallelism. Due to the lack of competitive vendor implementations in the CUDA Toolkit, the final performance results are compared against state-of-the-art CPU solutions.

II. Related Work

The growing demand for high-performance dense linear algebra on large batches of small matrices has led to early developments for batch matrix multiplication [13], [14], which is probably the most important operation in dense linear algebra. For higher-level algorithms that consist of calls to many BLAS kernels, researchers have followed two different approaches. The first uses LAPACK-style blocking, so that high performance is extracted from an optimized batch GEMM routine [15]–[18]. For example, Haidar et al. [19] adopt a blocking technique that is similar to LAPACK, but uses a much smaller range of blocking sizes.

The second type of designs uses a *one-kernel approach*, where all the computations are fused into a single GPU kernel. Such a technique works well for very small sizes but fails to scale on bigger sizes due to its heavy use of resources, which limits the execution throughput. For example, Wang et al. [20] introduced a field-programmable gate array (FPGA)-based parallel LU factorization of large sparse matrices, where the algorithm is reduced to factorizing many small matrices concurrently (up to size 118). Villa et al. [1] developed a GPU-based batched LU factorization (up to size 128) for subsurface transport simulation. Masliah et al. developed batched GEMM for very small sizes for both CPUs and GPUs [21]. Batch QR factorization and singular value decompositions (SVDs) [15], [22], [23] have been introduced to accelerate hierarchical compression of very large dense matrices. Kim et al. also introduced batched matrix multiplication (GEMM), triangular solve (TRSM), and LU factorization (no pivoting) for CPUs and Intel's Xeon Phi architectures, based on a compact interleaved data layout [24].

While most of the research effort focuses on problems of fixed sizes, there have been few contributions to address variable-size problems on the GPU. For example, the MAGMA library provides a set of Batch BLAS routines that support variable-size batch workloads [25]. Some of these routines were used in accelerating a biconjugate gradient stabilized (BiCGStab) solver on hierarchical matrices [26]. In general, these MAGMA kernels require finding the maximum size(s) across all dimensions before the kernel launch. The kernel grid is configured to accommodate the largest matrix in the batch. The irregularity in the batch workload is handled through the *Adaptive SubGrid Truncation* (ASGT) technique [25] which trims subgrids assigned to smaller problems in the batch.

This paper discusses, in depth, two different design approaches for variable-size batch computation on GPUs. The first follows in the footsteps of the ASGT technique [25], but further enriches it with scheduling policies that control the execution of thread blocks inside the GPU kernel. The second approach uses dynamic parallelism, which is a GPU technology that enables GPU kernels to be launched from within another GPU kernel. The latter enables easy development, and avoids the overhead of finding the maximum sizes.

III. Background

A. Cholesky Factorization

Symmetric positive definite (SPD) matrices are factorized using the Cholesky factorization. The algorithm factorizes an SPD matrix $A = LL^T$, where L is a lower triangular matrix. Two routines in LAPACK implement the algorithm. The first is called (POTF2). It uses an unblocked code, which factorizes one column at a time, and carries out the transformations to the trailing matrix using Level 1 and Level 2 BLAS routines. The second routine (POTRF) uses blocked code. It calls (POTF2) to factorize a panel of certain width ($nb > 1$), which leads to a compute-intensive update that utilizes Level 3 BLAS routines. According to LAPACK working notes [27], the Cholesky factorization algorithm performs $(\frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6})$ floating-point operations (FLOPs).

Algorithm 1 shows a pseudocode for the unblocked Cholesky factorization algorithm on an N×N matrix. At the i^{th} iteration, the algorithm inspects the diagonal element A[i, i] to ensure positive definiteness. If passed, the algorithm computes the square root of the diagonal element, scales the current column below the diagonal, and then performs a symmetric rank-1 update on the trailing submatrix to the right. On the other hand, Algorithm 2 shows the blocked Cholesky factorization. We adopt the left-looking variant of the algorithm, since it is mostly dominated by the GEMM operation, unlike the right-looking variant, which is dominated by the often-less-optimized SYRK routine. At each iteration, the current panel is updated first (SYRK + GEMM), then the factorization takes place on the updated panel (POTF2 + TRSM).

Algorithm 1: Unblocked Cholesky factorization.

```
for i=1 to N do
    if A[i, i]≤0 then
        // report error: non-SPD matrix
    end
    A[i, i] = sqrt(A[i, i])
    A[i+1:N,i] *= (1 / A[i,i]) //(DSCAL)
    // symmetric rank-1 update (DSYR)
    A[i+1:N,i+1:N] -= A[i+1:N,i] × A[i+1:N, i]^T
end
```

Algorithm 2: Blocked Cholesky factorization.

```
for i = 1 to N Step ib do
    if (i > 0) then
        // Update current panel (SYRK + GEMM)
        A[i:i+ib,i:i+ib] -= A[i:i+ib,1:i] × A[i:i+ib,1:i]^T;
        A[i+ib:N,i:i+ib] -= A[i+ib:N,1:i] × A[i:i+ib,1:i]^T;
    end
    // Panel factorization
    POTF2( A[i:i+ib, i:i+ib] );
    A[i+ib:N,i:i+ib] *= A[i:i+ib,i:i+ib]^-1 // TRSM;
end
```

B. Dynamic Parallelism in CUDA

Dynamic parallelism is a technology that enables launching GPU kernels inside other GPU kernels. Since the introduction of the Kepler architecture, the GPU has become able to generate work for itself without going back to the CPU. However, there is always a *parent kernel* that is launched from the host side. Only *child kernels* can be launched from the GPU side. In general, dynamic parallelism facilitates the programming of applications where the amount of work is determined only during runtime. One of the most notable applications to dynamic parallelism is adaptive mesh refinement (AMR).[3]

For variable-size batch computation, a parent kernel consists of independent threads, such that each thread is assigned to a single problem. Each parent thread reads the necessary information about the assigned problem, and launches the

corresponding child kernel. Figure 1 shows the difference between a host-launch–based technique, and a device-launch–based one. Typically, a host-launch–based technique uses a uniform subgrid size, so that each subgrid is guaranteed to fit the size of the assigned problem. The figure shows an example for a 4×4 subgrid configuration (left). The ASGT layer works on the subgrid level before any computation takes place. It detects and terminates unwanted thread blocks for a smaller problem size (colored in grey). The cost of the ASGT layer is the overhead of detection and termination, which depends on the size variation in the batch. In addition, the uniform subgrid configuration requires a search for the maximum dimensions across the batch, which is another source of overhead.

On the other hand, dynamic parallelism launches the exact required number of thread bocks for each problem. There is no need to search for the maximum dimensions, and the ASGT layer is not required. However, the overhead of dynamic parallelism comes in the two-stage launch process. The CPU launches a parent kernel, which does not perform any computation. Each parent thread then independently launches a kernel for the assigned problem.

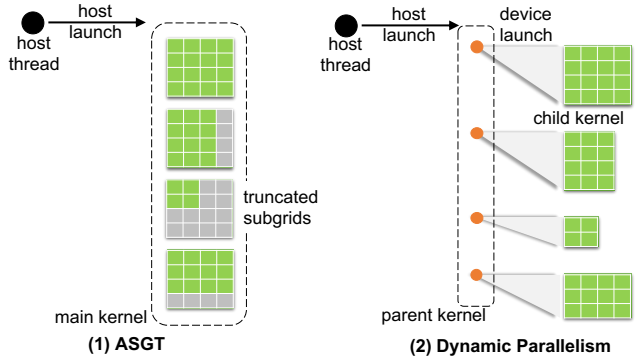


Fig. 1: The difference between a host-launch–based design (using ASGT), and a device-launch–based design (using dynamic parallelism).

IV. Experimental Setup

We illustrate our findings on a system with two 10-core Intel Haswell processors (E5-2650 v3, running at 2.3 GHz) and a Pascal GPU (Tesla P100). The GPU has 56 streaming multiprocessors, running at a 1.328 GHz clock rate. The GPU has a 16GB of CoWoS Stacked HBM2 memory, and is attached to the host CPU through a PCIe interconnect. All results are obtained using the CUDA 9.0RC toolkit.

Another experimental setup is the size distribution within the test batches. In this paper, we consider two sets of distributions. Given a maximum size N_{max}, the first distribution randomly samples the interval [1:N_{max}]. The distribution uses a random number generator that follows a uniform distribution. Figure 2 (left) shows an example for the first distribution. The second distribution is customized to impose more load imbalance for the designed kernels. This distribution generates 1% of the sizes that are equal to N_{max}; the other 99% are randomly sampled within the range [1:$\lfloor \frac{N}{10} \rfloor$]. Such a distribution is a challenging test case, as the majority of sizes are skewed within a very small size range, and there are still very few matrices that are at the end of the size spectrum, as shown in Figure 2 (right).

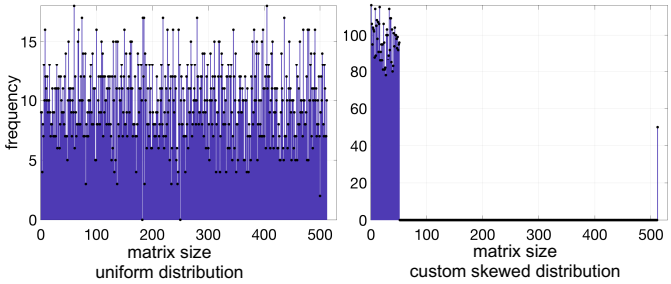


Fig. 2: Histogram of the sizes generated in a 5000 matrices batch using two different distributions with $N_{max} = 512$.

V. Design Details

This section discusses the different design candidates for a variable-size batch Cholesky factorization.

A. The Building Block Kernel

All the design candidates use the same factorization kernel as a building block. The kernel fuses the computational steps of Algorithm 2 into one entity. Because we are using a blocked code, the kernel performs the Cholesky factorization on a panel of size $(N - j) \times nb$, where N is the original size of the matrix, j is the size of the factorized submatrix, and nb is the blocking size. The kernel uses two main components that are written as inlined device functions. The first component is a *symmetric rank-k update* (SYRK) that is used in a left-looking manner to update a rectangular panel, as shown in Figure 3. The update function computes $C = C - A \times B^T$, where C is of size $(N - j) \times nb$. The function performs the update using an internal blocking size ib, so that $C = C - (a_0 b_0^T + a_1 b_1^T + \cdots)$. The function uses the register file to store a_i and b_i, and incorporates double buffers to prefetch a_{i+1} and b_{i+1} while the computation of the current product is taking place. The update function also takes advantage of the overlap between A and B to maximize data reuse. The panel C is read from global memory, and updated successively in shared memory. The second component of the building block kernel is the *panel factorization* function, which implements the unblocked Cholesky factorization according to Algorithm 1. The panel factorization function assumes that the panel is stored in shared memory, so that the updated C is reused.

The complete factorization of an $N \times N$ matrix is achieved by calling the building block kernel $\lceil \frac{N}{nb} \rceil$ times. At each call, the kernel updates a new panel in shared memory, performs an unblocked factorization in shared memory, and writes the factorized panel in global memory. For an $(N-j) \times nb$ panel, the kernel requires one thread block of $(N - j)$ threads. We now describe how to use this building block kernel to handle a batch of matrices of different sizes.

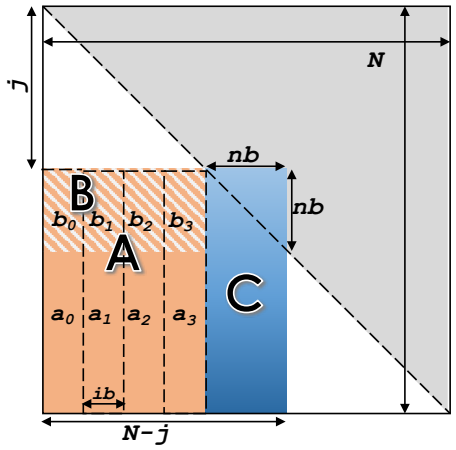


Fig. 3: The design of the Cholesky factorization kernel.

B. Batching Using Host-Launch

As mentioned before, a host-launch kernel is organized as an array of subgrids, where each subgrid handles a single problem. Since the CUDA programming model enforces homogeneous subgrid configurations, the host must use a configuration that fits the largest matrix size. A preparatory step involves launching a GPU kernel that searches the size array for the maximum matrix size N_{max}, which is then passed to the CPU to configure the kernel.

The ASGT layer is implemented on top of the building block kernel of Section V-A. For each matrix having a size $N < N_{max}$, the ASGT layer terminates all the extra threads. Recalling that the kernel is called multiple times to perform the factorization, smaller matrices in the batch require fewer calls to the kernel than bigger ones. Because the kernel does not keep track of matrices that have been fully factorized in previous calls, some thread blocks can be entirely terminated upon launch if the assigned matrix is fully factorized.

Another design point is the scheduling of smaller matrices in the batch. In general, we need $\lceil \frac{N_{max}}{nb} \rceil$ iterations to factorize the entire batch. Factorization on smaller matrices can start at iteration 0, or at a later iteration provided that the total number of iterations remains $\lceil \frac{N_{max}}{nb} \rceil$. In this regard, we discuss two *scheduling policies*. The first is called *Same Start Different Ends* (SSDE), and the second one is called *Different Starts Same End* (DSSE). An SSDE scheduling policy starts on all matrices at the first iteration. As the computation carries on, smaller matrices are entirely factorized and require no further processing. Table I shows an example for a batch of 4 matrices that requires a total of 7 iterations, where the blocking size nb is set to 8. The SSDE scheduling policy is easy to implement, but it suffers from low occupancy near the end of computation. It also produces significant variations in the sizes at each iteration.

The DSSE policy considers fewer matrices at the beginning, which correspond to matrices with the biggest set of sizes in the batch. Factorizations of different matrices begin at different

Org. size	Panel size per iteration						
	iter 0	iter 1	iter 2	iter 3	iter 4	iter 5	iter 6
50×50	50×8	42×8	34×8	26×8	18×8	10×8	2×2
32×32	32×8	24×8	16×8	8×8	—	—	—
17×17	17×8	9×8	1×1	—	—	—	—
5×5	5×5	—	—	—	—	—	—

TABLE I: The SSDE scheduling policy. A blank cell means that the corresponding matrix has been fully factorized.

iterations, but they all finish at the last iteration. As shown in Table II, the small number of matrices encountered at the beginning is mitigated by the fact that the sizes are relatively large and create enough work for the GPU. The DSSE policy also results in fewer size variations per iteration: at each iteration, it considers a maximum size range equal to nb. Table II shows the scheduling of factorization on the same four matrices mentioned above.

Org. size	Panel size per iteration						
	iter 0	iter 1	iter 2	iter 3	iter 4	iter 5	iter 6
50×50	50×8	42×8	34×8	26×8	18×8	10×8	2×2
32×32	—	—	—	32×8	24×8	16×8	8×8
17×17	—	—	—	—	17×8	9×8	1×1
5×5	—	—	—	—	—	—	5×5

TABLE II: The DSSE scheduling policy. A blank cell means that the corresponding matrix has been delayed to start at a later iteration.

We conducted two performance tests to compare the SSDE and DSSE scheduling policies. The first test uses a uniform distribution similar to Figure 2 (left), while the second uses the custom distribution of Figure 2 (right). Recall that both policies share the same computational kernel, such that differences in performance are due to the scheduling of thread block execution. Figure 4 shows a performance comparison of both distributions. Both scheduling policies have similar performances for the uniform distribution. A typical uniform distribution, especially using large batches, introduces small variations in the matrix sizes. This is why such a distribution cannot distinguish between the two scheduling policies, except for a slight advantage on the part of the DSSE policy that is up to 7.8%. Such an advantage decreases consistently as the size become larger.

Figure 4 also shows the performance against the customized distribution. This is a test case where we see a consistent advantage for the DSSE policy. Unlike the uniform distribution, which has a nicely balanced workload, the custom distribution of Figure 2 (right) imposes a more challenging range of sizes. In fact, the DSSE policy is always faster than the SSDE policy, scoring speedups that range between 5% and 20%. The main reason for such speedups is the shared memory requirements—the DSSE policy reduces the amount of unused shared memory by reducing the size variations per kernel launch, which has a positive effect on the kernel's overall occupancy. As mentioned before, the DSSE policy balances workloads at each iteration, and also increases the occupancy near the end of computation to compensate for the decreasing amount of work per matrix.

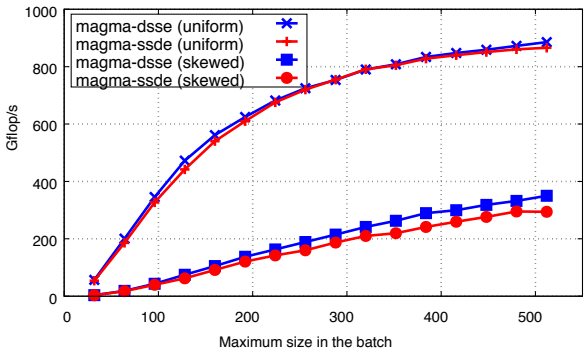


Fig. 4: Performance comparison between the SSDE/DSSE scheduling policies against two different size distributions for 5000 matrices using double precision on a P100 GPU.

Therefore, the DSSE policy is a more robust scheduling policy for different size distributions.

C. Batching Using Device-Launch

The device-launch–based design uses the CUDA dynamic parallelism. It invokes the same computational kernel described in Section V-A. We begin with a simple design where the parent kernel is configured with as many threads as the number of matrices in the batch (`batchCount`). We call this design the *brute-force* design. Each parent thread reads the necessary information about the assigned problem, and configures and launches the child kernel accordingly.

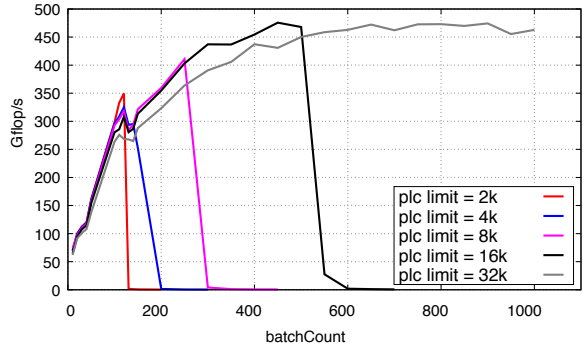


Fig. 5: The impact of the pending launch count on performance of dynamic parallelism kenels. Results are shown for $N_{max} = 512$ using double precision on a P100 GPU.

The initial design exposes an important limitation of dynamic parallelism, which is called the *Pending Launch Count Limit* (PLC limit). The CUDA runtime system keeps a buffer in global memory for setting up the arguments and the launching of child kernels. Such a buffer has a default limit of $2k$ pending launches. If the buffer is full, and more pending launches are generated, the runtime resizes the buffer while the kernel is running, resulting in a severe performance penalty that looks like a cliff in the performance graph. The CUDA runtime system allows the buffer to be resized at an initialization step. Figure 5 shows the effect of the PLC limit on performance. The experiment uses $N_{max} = 512$, and shows the observed performance while increasing the number of matrices (`batchCount`). It is clear that enlarging the buffer beforehand helps delaying the cliff point. However, it does not completely prevent it, as the maximum PLC limit is $64k$. Note that enlarging the buffer results in less available memory for the user's program. We also point out that the cliff points encountered in single precision (not shown) were exact to those encountered in double precision. This is due to the fact that size of the kernel arguments (mostly integers and pointers) do not differ from one precision to another, thus leading to the same behavior regardless of the precision.

The brute-force design can be modified to delay the encounter of the cliff point, so that we can use the same PLC buffer size for a greater number of problems while maintaining the same performance. The modified design adopts a *round robin* design by launching a finite number of parent threads regardless of the value of `batchCount`. Parent threads loop over the problem in a round-robin style, and launch child kernels accordingly. This modification helps delay, but not prevent, the cliff effect. As an example, when the PLC limit is set to $16k$, the brute-force kernels encounter the cliff point at batches of size 550. Using the round-robin approach, the kernel can now process batches of a thousand matrices using the same PLC limit.

The round-robin kernel uses a fixed number of threads, which can be a tuning parameter for the kernel. We conducted a tuning experiment for the number of threads on 5000 matrices with a uniform size distribution. We tried kernel configurations with 32, 64, 96, and 128 threads. We also considered 56 and 112 thread configurations, since the GPU has 56 multiprocessors. The experiment shows that as long as the number of parent threads is ≤ 112, the performance gets better with more parent threads. Beyond this threshold we observe a performance decay, until a dramatic drop in performance is encountered for more than 128 parent threads. The bottom line with device-based launches is to closely monitor the number of pending launch counts that can be generated. The round-robin design is a partial solution. The number of parent threads must be tuned according to the GPU architecture, and to the launch pattern of child kernels.

VI. Final Performance Results

The vendor library (cuBLAS [28]) does not provide variable-size batch routines. This is why the final performance results are compared against a parallel CPU implementation that calls the Intel MKL library (version 2017.0.3) within a dynamically scheduled `parallel for` loop. All results are reported using double-precision arithmetic.

The first experiment assesses the overheads of using host-launches and device-launches. Figure 6 shows the performance on batches of fixed-size matrices. The performance of both approaches is compared against the MAGMA fixed-size batch routine, which can be considered as an upper bound

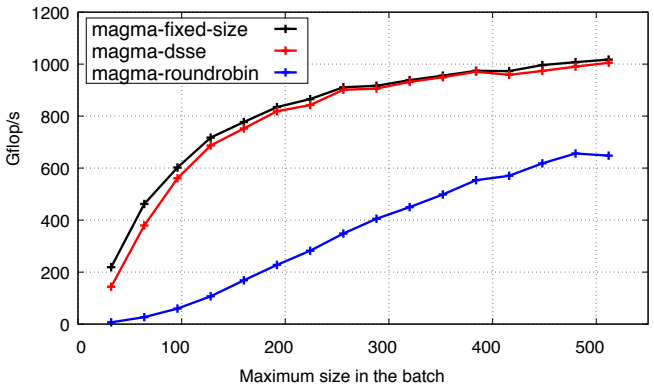
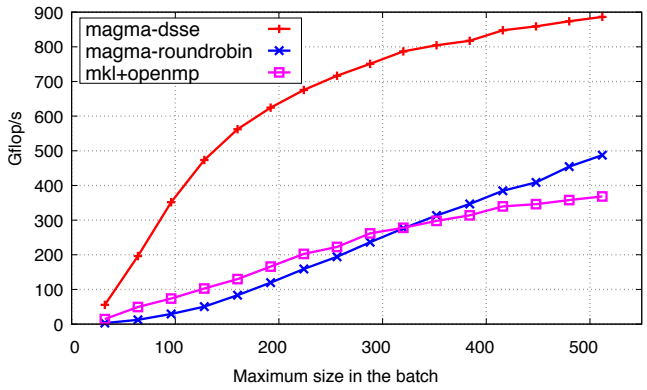


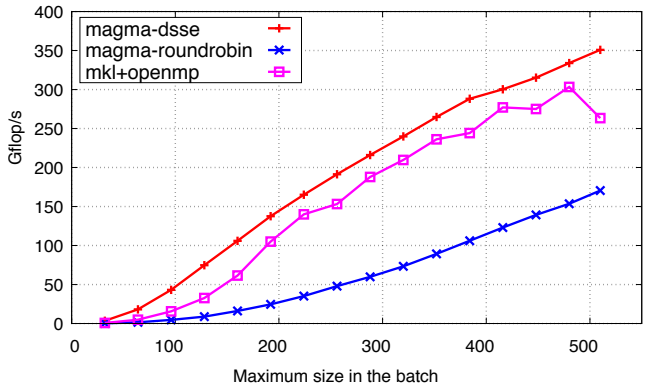
Fig. 6: Performance on a batch of fixed-size matrices. Results are for 5000 matrices using double precision on a P100 GPU.

for performance. The performance of the host-launch–based implementation using the DSSE policy is very close to the performance of the fixed-size batch routine. This implies that the overheads associated with the ASGT layer are almost negligible, especially when the sizes grow. On the other hand, the dynamic parallelism approach using the round-robin design suffers from a huge overhead, although it uses the same computational kernel. In fact, Figure 6 shows an overhead that is up to 90%. The overheads consistently decrease as the matrix sizes increase, which means that the overheads due to device-launches are independent from the matrix sizes. As the size grows bigger, more time is spent on the child kernels than on launching them.

The second experiment compares the final performance on a uniform distribution identical to Figure 2. Similarly, Figure 7a shows that the MAGMA-DSSE kernel is a clear winner, with speedups between 2.4×-4.7× against the CPU implementation. The MAGMA-DSSE kernel is also much faster than the dynamic parallelism approach. It scores speedups in the range of 1.8×-15×. Finally, Figure 7b shows a performance test which uses the customized size distribution. This is a scenario where we observe a much smaller asymptotic gap between the MAGMA-DSSE routine and the CPU implementation. The MAGMA-DSSE performance has an advantage in the range of 1.08×-4.4× against MKL running over OpenMP. There are two reasons for this behavior: firstly, the distribution of Figure 2 (right) causes the MAGMA-DSSE kernel to launch many extra thread blocks that are terminated through the ASGT layer, which is a considerable overhead on a majority of small size; secondly, most of the matrices are small enough to fit the CPU cache, making CPU performance very competitive with GPU performance. The MAGMA–round-robin kernel is at least 2× slower than the MAGMA-DSSE kernel. This is an expected behavior, since the overhead of a device-launch is huge for small matrices.



(a) Uniform distribution



(b) Custom distribution

Fig. 7: Final performance against two different distributions. Results are shown using double precision on a P100 GPU.

VII. Conclusion and Future Work

We introduced several design techniques to optimize GPU kernels for variable-size batch workloads. The paper introduced a single computational kernel that can be launched in two different ways (host side vs. device side). Both approaches are thoroughly discussed and evaluated. While the device-launch–based approach did not prove competitive, it is a promising approach if the technology behind dynamic parallelism sufficiently improves in the future. The host-launch design is the winning approach to date, and thanks to an optimized scheduling of thread blocks, it is able to outperform high-performance CPU based solutions, even on unbalanced workloads.

Acknowledgments

This work is partially supported by NSF grant No. OAC-1740250 and CSR 1514286, NVIDIA, and by the Exascale Computing Project (17-SC-20-SC).

References

[1] O. Villa, M. Fatica, N. Gawande, and A. Tumeo, *Power/Performance Trade-Offs of Small Batched LU Based Solvers on GPUs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 813–825.

[2] O. Villa, N. Gawande, and A. Tumeo, "Accelerating subsurface transport simulation on heterogeneous clusters," in *2013 IEEE International Conference on Cluster Computing, CLUSTER 2013, Indianapolis, IN, USA, September 23-27, 2013*, 2013, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CLUSTER.2013.6702656

[3] M. Gates, H. Anzt, J. Kurzak, and J. J. Dongarra, "Accelerating collaborative filtering using concepts from high performance computing," in *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, 2015, pp. 667–676. [Online]. Available: https://doi.org/10.1109/BigData.2015.7363811

[4] J. Kurzak, H. Anzt, M. Gates, and J. J. Dongarra, "Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 7, pp. 2036–2048, 2016. [Online]. Available: https://doi.org/10.1109/TPDS.2015.2481890

[5] A. Abdelfattah, M. Baboulin, V. Dobrev, J. J. Dongarra, C. W. Earl, J. Falcou, A. Haidar, I. Karlin, T. V. Kolev, I. Masliah, and S. Tomov, "High-Performance Tensor Contractions for GPUs," in *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, 2016, pp. 108–118. [Online]. Available: https://doi.org/10.1016/j.procs.2016.05.302

[6] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka, "Tensor Contractions with Extended BLAS Kernels on CPU and GPU," in *23rd IEEE International Conference on High Performance Computing, HiPC 2016, Hyderabad, India, December 19-22, 2016*, 2016, pp. 193–202. [Online]. Available: https://doi.org/10.1109/HiPC.2016.031

[7] S. N. Yeralan, T. A. Davis, W. M. Sid-Lakhdar, and S. Ranka, "Algorithm 980: Sparse QR Factorization on the GPU," *ACM Trans. Math. Softw.*, vol. 44, no. 2, pp. 17:1–17:29, Aug. 2017. [Online]. Available: http://doi.acm.org/10.1145/3065870

[8] S. C. Rennich, D. Stosic, and T. A. Davis, "Accelerating sparse Cholesky factorization on GPUs," *Parallel Computing*, vol. 59, pp. 140–150, 2016. [Online]. Available: https://doi.org/10.1016/j.parco.2016.06.004

[9] H. Anzt, J. J. Dongarra, G. Flegar, and E. S. Quintana-Ortí, "Batched Gauss-Jordan Elimination for Block-Jacobi Preconditioner Generation on GPUs," in *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2017, Austin, TX, USA, February 5, 2017*, 2017, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/3026937.3026940

[10] W. Hackbusch, "A Sparse Matrix Arithmetic Based on H-matrices. Part I: Introduction to H-matrices," *Computing*, vol. 62, no. 2, pp. 89–108, May 1999. [Online]. Available: http://dx.doi.org/10.1007/s006070050015

[11] S. Tomov, J. J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010. [Online]. Available: https://doi.org/10.1016/j.parco.2009.12.005

[12] A. Haidar, A. Abdelfatah, S. Tomov, and J. Dongarra, "High-performance Cholesky Factorization for GPU-only Execution," in *Proceedings of the General Purpose GPUs*, ser. GPGPU-10. New York, NY, USA: ACM, 2017, pp. 42–52. [Online]. Available: http://doi.acm.org/10.1145/3038228.3038237

[13] L. Ng, K. Wong, A. Haidar, S. Tomov, and J. Dongarra, "Magmadnn high-performance data analytics for manycore gpus and cpus," December 2017, magma-DNN, 2017 Summer Research Experiences for Undergraduate (REU), Knoxville, TN. [Online]. Available: http://icl.cs.utk.edu/magma/software/

[14] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, Design, and Autotuning of Batched GEMM for GPUs," in *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, 2016, pp. 21–38.

[15] A. Haidar, T. Dong, S. Tomov, P. Luszczek, and J. Dongarra, "A framework for batched and GPU-resident factorization algorithms applied to block householder transformations," in *High Performance Computing*, ser. Lecture Notes in Computer Science, J. M. Kunkel and T. Ludwig, Eds. Springer International Publishing, 2015, vol. 9137, pp. 31–47.

[16] T. Dong, A. Haidar, S. Tomov, and J. Dongarra, "A Fast Batched Cholesky Factorization on a GPU," in *Proc. of 2014 International Conference on Parallel Processing (ICPP-2014)*, September 2014.

[17] T. Dong, A. Haidar, P. Luszczek, A. Harris, S. Tomov, and J. Dongarra, "LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU," in *Proceedings of 16th IEEE International Conference on High Performance and Communications (HPCC 2014)*, August 2014.

[18] A. Haidar, P. Luszczek, S. Tomov, and J. Dongarra, "Towards batched linear solvers on accelerated hardware platforms," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015, ACM. San Francisco, CA: ACM, 02/2015 2015.

[19] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra, "Batched Matrix Computations on Hardware Accelerators Based on GPUs," *IJHPCA*, vol. 29, no. 2, pp. 193–208, 2015.

[20] X. Wang and S. G. Ziavras, "Parallel LU Factorization of Sparse Matrices on FPGA-based Configurable Computing Engines," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 4, pp. 319–343, 2004. [Online]. Available: http://dx.doi.org/10.1002/cpe.748

[21] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. J. Dongarra, "High-Performance Matrix-Matrix Multiplications of Very Small Matrices," in *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, 2016, pp. 659–671.

[22] W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes, "Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression," *Parallel Computing*, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819117301461

[23] T. Dong, A. Haidar, S. Tomov, and J. Dongarra, "Accelerating the svd bi-diagonalization of a batch of small matrices using gpus," *Journal of Computational Science*, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187775031731150X

[24] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, "Designing Vector-friendly Compact BLAS and LAPACK Kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 55:1–55:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126941

[25] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Novel HPC Techniques to Batch Execution of Many Variable Size BLAS Computations on GPUs," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: ACM, 2017, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/3079079.3079103

[26] I. Yamazaki, A. Abdelfattah, A. Ida, S. Ohshima, S. Tomov, R. Yokotax, and J. Dongarra, "Performance of Hierarchical-matrix BiCGStab Solver on GPU Clusters," in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2018, (accepted).

[27] "LAPACK Working Note 41: Installation Guide for LAPACK," 1999, http://www.netlib.org/lapack/lawnspdf/lawn41.pdf.

[28] "NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS)," available at https://developer.nvidia.com/cublas.