OpenMP target task: tasking and target offloading on heterogeneous systems *

Pedro Valero-Lara[1][0000-0002-1479-4310], Jungwon Kim[1][0000-0001-6594-6225], Oscar Hernandez[1], and Jeffrey Vetter[1][0000-0002-2449-6720]

Oak Ridge National Laboratory, Oak Ridge TN 37830, USA
{valerolarap, kimj, oscar, vetter}@ornl.gov
https://www.ornl.gov/

Abstract. This work evaluated the use of OpenMP tasking with target GPU offloading as a potential solution for programming productivity and performance on heterogeneous systems. Also, it is proposed a new OpenMP specification to make the implementation of heterogeneous codes simpler by using OpenMP target task, which integrates both OpenMP tasking and target GPU offloading in a single OpenMP pragma. As a test case, the authors used one of the most popular and widely used Basic Linear Algebra Subprogram Level-3 routines: triangular solver (TRSM). To benefit from the heterogeneity of the current high-performance computing systems, the authors propose a different parallelization of the algorithm by using a nonuniform decomposition of the problem. This work used target GPU offloading inside OpenMP tasks to address the heterogeneity found in the hardware. This new approach can outperform the state-of-the-art algorithms, which use a uniform decomposition of the data, on both the CPU-only and hybrid CPU-GPU systems, reaching speedups of up to one order of magnitude. The performance that this approach achieves is faster than the IBM ESSL math library on CPU and competitive relative to a highly optimized heterogeneous CUDA version. One node of Oak Ridge National Laboratory's supercomputer, Summit, was used for performance analysis.

Keywords: Tasking · Heterogeneity · OpenMP · CUDA · Linear Algebra · TRSM · BLAS

1 Introduction

The motivation of this work was to analyze the OpenMP 4.5 specification for programming productivity and performance on heterogeneous systems via the

* Notice: This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

integration of tasking and target GPU offloading. Triangular solve (TRSM) was used as a motivating example because it is one of the most popular Basic Linear Algebra Subprograms (BLAS) routines. The authors also included a highly optimized Compute Unified Device Architecture (CUDA) code in their analysis to compare not only the performance but also the programming productivity.

Many other studies efficiently used task-based programming models for linear algebra computations. Examples include CPU-only math libraries, such as Parallel Linear Algebra Software for Multicore Architectures (PLASMA) [7], which is based on OpenMP and Quark [10]; Chamaleon, which is based on StarPU [2]; and Linear Algebra routines on OmpSs (LASs) [18], which is based on OmpSs [8].

Tasking [19,20] is an efficient tool for addressing irregular problems, such as sparse [4,3] and dense [17,18] linear algebra kernels. Unlike conventional directive-based clauses used for a uniform work sharing (i.e., decomposition) of loops, tasking provides the flexibility, transparency, and programming productivity necessary for handling irregular and nonbalanced applications in which each task has a different computational cost. Tasking is also well positioned to address the heterogeneity of the current and upcoming high-performance computing systems [16,2]. Since OpenMP 4.0, it is possible to use target GPU offloading in OpenMP codes. Using GPU offloading in OpenMP tasking could be a transparent and simple way to implement heterogeneous codes without compromising performance. Multiple implementations of the OpenMP 4.5 specification are found in multiple vendor compilers, such as Intel [12], NVIDIA [15], AMD [1], Cray [5], and IBM [11], as well as in open-source compilers [9,13], which would make OpenMP a portable specification among many different heterogeneous CPU+GPU systems. However, this study should be extended by using other architectures, such as AMD and Intel GPUs, to verify the portability of OpenMP 4.5.

The parallel algorithm of the TRSM routine comprises two main components: TRSM and general matrix-matrix multiplication (GEMM). Although GEMM can reach a very high—nearly peak—performance on GPU, TRSM is a more complex routine, reaching a lower performance on GPUs. In terms of the percentage of the peak performance reached, TRSM works better on CPUs than on GPUs, reaching about 75–80% of the CPU peak performance but only about 50% of the GPU peak performance. To achieve high performance on GPU and CPU, the authors propose to use CPU to compute the TRSM blocks while using GPU for the GEMM blocks. The use of heterogeneous (i.e., CPU+GPU) systems for linear algebra solvers is not new. For example, the MAGMA library [14] has different linear algebra factorizations (i.e., LAPACK routines), such as LU factorization/solve and Cholesky factorization/solve. These routines are distributed and computed on CPU and GPU by using vendor programming models, such as CUDA or HIP on NVIDIA and AMD GPUs and math libraries, such as cuBLAS/hipBLAS on NVIDIA/AMD GPUs or MKL on Intel CPUs.

Unlike MAGMA, which uses CUDA or HIP, the authors propose to use OpenMP tasking with target GPU offloading as the orchestrator of the blocks and tasks. Unfortunately, there is not a heterogeneous TRSM implementation in

MAGMA that can be compared with this. Thus, the authors included a highly optimized asynchronous and heterogeneous CUDA implementation of TRSM in their analysis, following the same programming approach used by MAGMA. Like MAGMA and many other math linear algebra libraries, the authors used vendor libraries—IBM ESSL on CPU and NVIDIA cuBLAS on GPU—to compute the different components or blocks of the algorithms. To the best of the authors' knowledge, this is the first time that target GPU offloading within OpenMP tasking has been used for heterogeneous linear algebra operations on heterogeneous systems.

The remainder of the paper is structured as follows. Section 2 describes the main characteristics of TRSM. Section 3 explains the main details of the OpenMP and CUDA implementations. Section 4 evaluates these implementations and Section5 presents the proposal for a novel OpenMP construct which integrates OpenMP tasking and target offloading in a single OpenMP pragma. Finally, Section 6 summarizes the conclusions and future directions.

2 Motivating Example: TRSM BLAS Level-3 Routine

The TRSM BLAS Level-3 routine is one of the most popular and widely used BLAS routines. It is used in multiple applications and in some of the most important LAPACK operations, such as LU and Cholesky solve. It solves a triangular system, which can be defined as:

$$op(A) \cdot X = ALPHA \cdot B, or X \cdot op(A) = ALPHA \cdot B. \quad (1)$$

TRSM has a triangular matrix as input and a regular dense matrix as output. Matrix A is a triangular matrix, which can be "lower" or "upper," depending on the locations of the nonzero elements within it. The result of this operation is stored in matrix X. This matrix can be positioned on the left or right of matrix A, which affects the order of the operations to be computed. Matrix B is a dense matrix, and $ALPHA$ is a scalar matrix. *Op(A)* can be a transposed (i.e., A^T) or nontransposed matrix. For clarity and simplicity, the remainder of this document will focus only on one of the possible cases of this operation, which consists of computing a triangular system in which matrix A is a lower triangular matrix, is not transposed, and is positioned to the left of matrix X. More information about this operation and other BLAS Level-3 operations can be found in Dongarra et al. [6] or on the BLAS website.[1]

3 Task-Based Implementation of TRSM

One of the most common ways to parallelize this type of operation is to decompose the matrix into tiles of the same size, defining the dependencies between the tiles and operations to be computed on each tile. This can be efficiently implemented via tasking [18,17,7].

[1] `http://www.netlib.org/blas/`

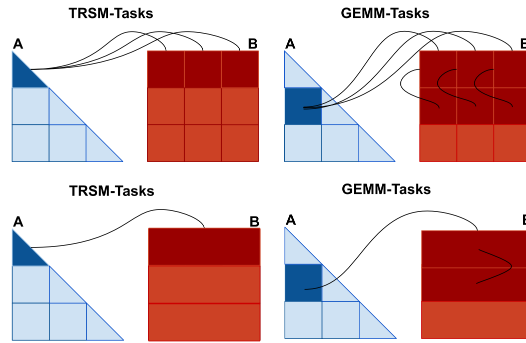

Fig. 1. Uniform (top) and nonuniform (bottom) tiled TRSM decomposition.

The top image in Figure 1 illustrates the dependencies among the tiles and operations to be computed on the tiles. The algorithm computes TRSM on the diagonal tiles of the input and triangular tiled matrix A and the tiles of the first row of the output and regular dense tiled matrix B. Once complete, a set of GEMM operations must be run by using the output (i.e., tiles) of the previous TRSM operations and the tiles of the column below the diagonal tile as input. The output corresponds to the tiles located in the second row through the last row of the output tiled matrix. This process is repeated until the last diagonal tile of the input matrix is computed.

3.1 OpenMP

Figure 2 shows a pseudocode for the tiled TRSM decomposition, which is illustrated in the top image in Figure 1, on a CPU-GPU heterogeneous system. The algorithm used in this implementation is identical to the one used by the PLASMA, Chameleon, and LASs math libraries. As mentioned previously, the goal is to compute TRSM blocks on CPU and GEMM blocks on GPU. The data and task dependencies are defined by using the `#pragma omp task depend` clause. A few more lines of code must be provided to compute GEMM on GPU and encapsulate target GPU offloading into OpenMP tasking. These new lines consist of (1) describing the data moving from CPU and GPU via `#pragma omp target enter data map`, (2) specifying that the pointers used in the GEMM call are GPU pointers via `#pragma omp target data use_device_ptr`, and (3) identifying which data must be copied back to CPU via `#pragma omp target exit data map`.

Unlike the previous OpenMP code (Figure 2) in which the matrices are decomposed into square tiles of the same size, the authors propose a different and irregular decomposition and/or parallelization, as shown in the bottom image of Figure 1, in which matrix A is decomposed into square tiles, but matrix B is decomposed into rectangular matrices. This different decomposition, which is illustrated in Figure 1 and implemented in the code shown in Figure 2, allows

```
1  aSIZE = TILE_SIZE*TILE_SIZE;
2  for(d = 0; d < dt; d++) {
3    for(c = 0; c < ct; c++) {
4      #pragma omp task depend(in:TILE_A[d][d]) \
5                       depend(inout:TILE_B[d][c])
6      CPU-TRSM(L, L, N, N,
7               TILE_SIZE, TILE_SIZE,
8               ALPHA, TILE_A[d][d], TILE_SIZE,
9                      TILE_B[d][c], TILE_SIZE);
10   }//End for c
11   for(r = d+1; r < rt; r++) {
12     for(c = 0; c < ct; c++) {
13       #pragma omp task depend(in:TILE_A[r][d]) \
14                        depend(in:TILE_B[d][c]) \
15                        depend(inout:TILE_B[r][c]) {
16         TILE_A=TILE_A[r][d];TILE_B=TILE_B[d][c];TILE_C=TILE_B[r][c];
17         #pragma omp target enter data map
           ↪   (to:TILE_A[0:aSIZE],TILE_B[0:aSIZE],TILE_C[0:aSIZE])
18         #pragma omp target data use_device_ptr(TILE_A,TILE_B,TILE_C) {
19           GPU-GEMM(N, N,
20                    TILE_SIZE, TILE_SIZE, TILE_SIZE,
21                    -1.0, TILE_A, TILE_SIZE,
22                          TILE_B, TILE_SIZE,
23                   ALPHA, TILE_C, TILE_SIZE);
24         }//End pragma target
25         #pragma omp target exit data map(from:TILE_C[aSIZE])
26       }//End pragma task
27     }//End for c
28   }//End for r
29 }//End for d
```

Fig. 2. CPU-GPU OpenMP code of the tiled TRSM decomposition.

```
1  aSIZE = TILE_SIZE*TILE_SIZE;
2  bSIZE = TILE_SIZE*MATRIX_SIZE;
3  for(d = 0; d < dt; d++) {
4    #pragma omp task depend(in:TILE_A[d][d]) \
5                     depend(inout:TILE_B[d])
6    CPU-TRSM(L, L, N, N,
7             TILE_SIZE, MATRIX_SIZE,
8             ALPHA, TILE_A[d][d], TILE_SIZE,
9                    TILE_B[d], TILE_SIZE);
10   for(r = d+1; r < rt; r++) {
11     #pragma omp task depend(in:TILE_A[d][r]) \
12                      depend(in:TILE_B[d]) \
13                      depend(inout:TILE_B[r]) {
14       TILE_A=TILE_A[r][d];TILE_B=TILE_B[d];TILE_C=TILE_B[r];
15       #pragma omp target enter data map
         ↪   (to:TILE_A[0:aSIZE],TILE_B[0:bSIZE],TILE_C[0:bSIZE])
16       #pragma omp target data use_device_ptr(TILE_A,TILE_B,TILE_C) {
17         GPU-GEMM(N, N,
18                  TILE_SIZE, MATRIX_SIZE, TILE_SIZE,
19                  -1.0, TILE_A, TILE_SIZE,
20                        TILE_B, TILE_SIZE,
21                 ALPHA, TILE_C, TILE_SIZE);
22       }//End pragma target
23       #pragma omp target exit data map(from:TILE_C[bSIZE])
24     }//End pragma task
25   }//End for r
26 }//End for d
```

Fig. 3. CPU-GPU OpenMP code of the optimized tiled TRSM decomposition.

the occupancy of the CPU and GPU to be maximized, as well as helps overlap more of the CPU-GPU communication and computation. Also, a lower number of tasks is necessary, which minimizes the scheduler overhead. These modifications in the code (Figure 3) consists of (1) removing those for loops related to the columns of matrix B, (2) using a unidimensional array for the tiles of matrix B, and (3) changing the input of the BLAS calls in which a whole block of rows of matrix B is computed instead of a square tile. These modifications also reduce the number of lines of code relative to the previous approach.

3.2 CUDA

The CUDA code uses the same matrix decomposition used in the optimized OpenMP code shown in Figure 3. To overlap communication with computation—as well as CPU computation with GPU computation, when possible—the asynchronous application programming interface of the cuBLAS library and CUDA streams must be used. Also, CUDA events must be used to guarantee the data dependencies among those blocks of the algorithm computed on the CPU and on GPU. The use of async memory transfers between CPU and GPU requires the use of pinned memory. This is done by using `cudaHostAlloc` to allocate host CPU memory. Finally, a stream must be associated with the CUDA handle before running GEMM via `cublasSetStream`. To achieve fully overlapping computation and communication, the authors used a different stream in each consecutive GEMM block.

Figure 4 shows a pseudocode corresponding to the first iteration of the CUDA CPU-GPU asynchronous TRSM code. The authors implemented this code to minimize the overhead of CPU-GPU communication as much as possible, as well as to maximize CPU and GPU use. During the first iteration of the code, all rectangular tiles of matrix B must be transferred from CPU to GPU. Once these tiles are in GPU memory, only these must be copied back to the CPU after the computation of the first GEMM block of each iteration because TRSM must compute them on the CPU at the beginning of the following iteration (Figure 4).

4 Evaluation

The authors conducted the performance evaluation by using one node of Oak Ridge National Laboratory's heterogeneous supercomputer, Summit, which is currently listed on the TOP500 list as the second fastest supercomputer in the world. Summit features 2× IBM Power9 8335-GTH at 2.4 GHz, 32 GB RAM memory, and 6× NVIDIA V100 (Volta) GPU with 16 GB HBM2 and NVLink2 for high-bandwidth communication between CPU and GPU. In this study, the authors used one IBM Power9 CPU (21 cores) and one NVIDIA GPU (V100); all computations were done in double precision. The math libraries IBM ESSL (6.1.0-2) and NVIDIA cuBLAS (CUDA version 10.1.243) were used to compute the different components of the algorithm (i.e., CPU-TRSM and GPU-GEMM

```
cuStream_t stream0, stream1;
cuEvent_t event;
cuHandle_t handle;
//First iteration
cblas_dtrsm(L, L, N, N,
         TILE_SIZE, MATRIX_SIZE,
         ALPHA, TILE_A[0][0], TILE_SIZE,
                     TILE_B0, TILE_SIZE);
cudaEventRecord(event);
cudaEventSynchronize(event);
cublasSetMatrixAsync(TILE_SIZE,MATRIX_SIZE,sizeof(precision),
                      TILE_B[0], TILE_SIZE,
                     TILE_BGPU0, TILE_SIZE, streams[0]);
//First GEMM block
cublasSetMatrixAsync(TILE_SIZE, TILE_SIZE, sizeof(precision),
                  TILE_A[1][0], TILE_SIZE,
                    TILE_AGPU1, TILE_SIZE, streams[0]);
cublasSetMatrixAsync(TILE_SIZE,MATRIX_SIZE,sizeof(precision),
                     TILE_B[1], TILE_SIZE,
                     TILE_GPUB1, TILE_SIZE, streams[0]);
cublasSetStream(handle, streams[0]);
GPU-GEMM(N, N,
         TILE_SIZE, MATRIX_SIZE, TILE_SIZE,
              -1.0, TILE_GPUA1, TILE_SIZE,
                    TILE_GPUB0, TILE_SIZE,
             ALPHA, TILE_GPUB1, TILE_SIZE);
cublasGetMatrixAsync(TILE_SIZE,MATRIX_SIZE,sizeof(precision),
                    TILE_GPUB1, TILE_SIZE,
                     TILE_B[1], TILE_SIZE, streams[0]);
//Second GEMM block
cublasSetMatrixAsync(TILE_SIZE, TILE_SIZE, sizeof(precision),
                  TILE_A[2][0], TILE_SIZE,
                    TILE_AGPU2, TILE_SIZE, streams[1]);
cublasSetMatrixAsync(TILE_SIZE,MATRIX_SIZE,sizeof(precision),
                     TILE_B[2], TILE_SIZE,
                     TILE_GPUB2, TILE_SIZE, streams[1]);
cublasSetStream(handle, streams[1]);
cublasDgemm(N, N,
         TILE_SIZE,MATRIX_SIZE,TILE_SIZE,
              -1.0, TILE_GPUA2, TILE_SIZE,
                    TILE_GPUB0, TILE_SIZE,
             ALPHA, TILE_GPUB2, TILE_SIZE);
...
```

Fig. 4. CUDA code of the optimized tiled TRSM decomposition.

in the codes illustrated in Figures 2 and 3). The IBM compiler xl 16.1.1-5 was also used.

The performance analysis corresponds to a set of runs by using different square matrix sizes ranging from 512 to 16,384. In every run, the authors used a tile size that was 1/8 the size of the matrix. For example, for a matrix size of 512^2, a tile size of 64^2 was used for matrices A and B in the OpenMP version (Figures 2 and 5) in which the matrices are uniformly decomposed (top image in Figure 1). Additionally, a tile size of 64×512 was used for matrix B in the other OpenMP implementation (Figures 3 and 5) and CUDA code (Figure 4). Thus, depending on the matrix decomposition used, the authors used the same number of tasks and blocks. For comparison and completeness, two different CUDA implementations were included: one synchronous and one asynchronous.

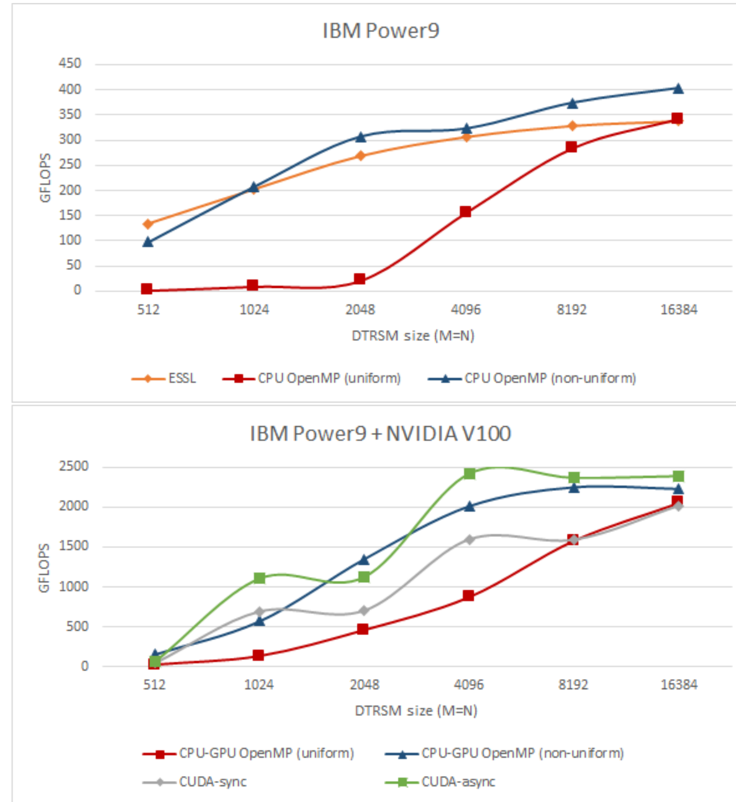

Fig. 5. Performance in terms of giga-floating point operations per second of the different implementations on CPU (top) and CPU+GPU (bottom).

These are shown as CUDA-sync and CUDA-async in Figure 5. This helps show the impact of using asynchronous CPU-GPU communication and computation.

First, the results were evaluated on CPU only, as shown in the top image of Figure 5. The OpenMP uniform code is based on the implementations of the CPU-only math libraries, including PLASMA [7] and LASs [18] libraries. As shown in Figure 4, the use of a nonuniform decomposition can outperform the OpenMP uniform code, achieving a speedup of up to one order of magnitude in some cases (23× for a matrix size of 1,024 and 15× for a matrix size of 2,048). An irregular distribution of the workload can achieve a high performance, even on relatively small matrix sizes. Finally, as expected, both variants achieved the best performance on the biggest matrix tested, being the OpenMP nonuniform code about 1.2× faster. Also, this code is faster than the multithreading IBM ESSL library in most cases, performing up to a 1.2× speedup.

Except for the smallest matrix size computed, all heterogeneous versions are faster than using CPU only. As in the CPU case, using OpenMP tasking with OpenMP target via a nonuniform decomposition of the matrices, as shown in the

bottom image of Figure 5, achieves a better result than using a uniform matrix decomposition. As in the CPU case, the OpenMP nonuniform code can achieve better results, even on small and medium matrices. The OpenMP nonuniform code was 4.3× faster on a matrix size of 1,240, 2.3× faster on a matrix size of 4,096, and 1.1× faster on a matrix size of 16,384. Also, the OpenMP nonuniform code surpasses the CUDA synchronous implementation in most experiments because it is about 2× faster. Finally, although the asynchronous version of the CUDA implementation, as shown in the bottom image of Figure 5, is faster than the heterogeneous OpenMP nonuniform code, the performance of the OpenMP code is competitive with respect to the performance reached by the CUDA code because OpenMP tasking with the target offloading code achieved about 85–95% of the asynchronous CUDA code performance.

5 A proposal for OpenMP target tasking

Given the good results shown in the previous section by using target offloading in OpenMP tasks, we propose the integration of both OpenMP tasking and target offloading by using a new OpenMP construct: OpenMP target task. An example of this new construct with respect to the current OpenMP specification can be seen in Figure 6. As shown, the use of OpenMP target tasking would simplify the implementation of heterogeneous codes considerably. Additionally, a better CPU-GPU communication could be performed by keeping the data in GPU memory when other GPU tasks need to access to the same data.

```
1  //Current specification
2  #pragma omp task depend(in:TILE_A[d][r]) \
3                   depend(in:TILE_B[d]) \
4                   depend(inout:TILE_B[r]) {
5    TILE_A=TILE_A[r][d];TILE_B=TILE_B[d];TILE_C=TILE_B[r];
6    #pragma omp target enter data map
      ↪   (to:TILE_A[0:aSIZE],TILE_B[0:bSIZE],TILE_C[0:bSIZE])
7    #pragma omp target data use_device_ptr(TILE_A,TILE_B,TILE_C) {
8      GPU-GEMM(N, N,
9               TILE_SIZE, MATRIX_SIZE, TILE_SIZE,
10              -1.0, TILE_A, TILE_SIZE,
11                    TILE_B, TILE_SIZE,
12             ALPHA, TILE_C, TILE_SIZE);
13   }//End pragma target
14   #pragma omp target exit data map(from:TILE_C[bSIZE])
15 }//End pragma task
16 //Proposed specification
17 #pragma omp target task depend(in:TILE_A[d][r]) \
18                         depend(in:TILE_B[d]) \
19                         depend(inout:TILE_B[r]) {
20   GPU-GEMM(N, N,
21            TILE_SIZE, MATRIX_SIZE, TILE_SIZE,
22            -1.0, TILE_A[d][r], TILE_SIZE,
23                  TILE_B[d], TILE_SIZE,
24           ALPHA,    TILE_B[r], TILE_SIZE);
25 }//End pragma target task
```

Fig. 6. Proposal for OpenMP target task.

6 Conclusions and Future Work

This paper proposes a new parallel approach for the BLAS Level-3 routine TRSM by using a nonuniform data decomposition that better matches the characteristics of heterogeneous systems. This new approach can achieve an important acceleration compared with the reference implementations by using a nonuniform data decomposition on both CPU and the heterogeneous CPU+GPU implementations. The authors implemented two different codes using a nonuniform data decomposition: one based on OpenMP tasking and target GPU offloading and one based on CUDA. Although slower than the asynchronous CUDA code, the OpenMP code can achieve about 85–95% of the performance achieved by the CUDA code with a much lower programming effort, reaching a high programming productivity without compromising much performance. To simplify the implementation of heterogeneous codes, we propose OpenMP target tasking, a new OpenMP construct which combines OpenMP tasking and target offloading in a single OpenMP pragma.

In future work, the authors plan to (1) extend this work to other dense and sparse linear algebra kernels and other heterogeneous systems and (2) achieve better problem tuning by using and including algorithm and hardware factors that are susceptible to tuning in the code and OpenMP specification.

References

1. AMD: Aomp (Jun 2021), `https://rocmdocs.amd.com/en/latest/Programming_Guides/aomp.html`
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Exp. **23**(2), 187–198 (2011). https://doi.org/10.1002/cpe.1631
3. Catalán, S., Martorell, X., Labarta, J., Usui, T., Díaz, L.A.T., Valero-Lara, P.: Accelerating conjugate gradient using ompss. In: 20th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2019, Gold Coast, Australia, December 5-7, 2019. pp. 121–126. IEEE (2019). https://doi.org/10.1109/PDCAT46702.2019.00033
4. Catalán, S., Usui, T., Toledo, L., Martorell, X., Labarta, J., Valero-Lara, P.: Towards an auto-tuned and task-based spmv (lass library). In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) OpenMP: Portable Multi-Level Parallelism on Modern Systems - 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22-24, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12295, pp. 115–129. Springer (2020). https://doi.org/10.1007/978-3-030-58144-2_8
5. Cray: Cce openmp (Jun 2021), `https://pubs.cray.com/bundle/Cray_Fortran_Reference_Manual_S-3901_11-0/page/OpenMP_Overview.html`
6. Dongarra, J.J., Croz, J.D., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. **16**(1), 1–17 (1990). https://doi.org/10.1145/77626.79170
7. Dongarra, J.J., Gates, M., Haidar, A., Kurzak, J., Luszczek, P., Wu, P., Yamazaki, I., YarKhan, A., Abalenkovs, M., Bagherpour, N., Hammarling, S., Šístek, J.,

Stevens, D., Zounon, M., Relton, S.D.: PLASMA: parallel linear algebra software for multicore using openmp. ACM Trans. Math. Softw. **45**(2), 16:1–16:35 (2019). https://doi.org/10.1145/3264491
8. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(2), 173–193 (2011). https://doi.org/10.1142/S0129626411000151
9. GNU: Gcc openmp (Jun 2021), https://gcc.gnu.org/wiki/Offloading
10. Haidar, A., Ltaief, H., Dongarra, J.J.: Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In: Lathrop, S.A., Costa, J., Kramer, W. (eds.) Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011. pp. 8:1–8:11. ACM (2011). https://doi.org/10.1145/2063384.2063394
11. IBM: Xlc openmp (Jun 2021), https://www.ibm.com/docs/en/xl-c-and-cpp-linux/13.1.6?topic=gpus-programming-openmp-device-constructs
12. Intel: Oneapi (Jun 2021), https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-cpp-fortran-compiler-openmp/top.html
13. LLVM: Openmp (Jun 2021), https://llvm.org/docs/AMDGPUUsage.html#target-triples
14. Nath, R., Tomov, S., Dongarra, J.J.: An improved magma gemm for fermi graphics processing units. Int. J. High Perform. Comput. Appl. **24**(4), 511–515 (2010). https://doi.org/10.1177/1094342010385729
15. NVIDIA: Nvcc openmp (Jun 2021), https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html#openmp-use
16. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Self-adaptive ompss tasks in heterogeneous environments. In: 27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013. pp. 138–149. IEEE Computer Society (2013). https://doi.org/10.1109/IPDPS.2013.53
17. Valero-Lara, P., Catalán, S., Martorell, X., Labarta, J.: BLAS-3 optimized by ompss regions (lass library). In: 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 13-15, 2019. pp. 25–32. IEEE (2019). https://doi.org/10.1109/EMPDP.2019.8671545
18. Valero-Lara, P., Catalán, S., Martorell, X., Usui, T., Labarta, J.: slass: A fully automatic auto-tuned linear algebra library based on openmp extensions implemented in ompss (lass library). J. Parallel Distributed Comput. **138**, 153–171 (2020). https://doi.org/10.1016/j.jpdc.2019.12.002
19. Valero-Lara, P., Sirvent, R., Peña, A.J., Labarta, J.: Mpi+openmp tasking scalability for multi-morphology simulations of the human brain. Parallel Comput. **84**, 50–61 (2019). https://doi.org/10.1016/j.parco.2019.03.006
20. Valero-Lara, P., Sirvent, R., Peña, A.J., Martorell, X., Labarta, J.: Mpi+openmp tasking scalability for the simulation of the human brain: Human brain project. In: Proceedings of the 25th European MPI Users' Group Meeting, Barcelona, Spain, September 23-26, 2018. pp. 5:1–5:8. ACM (2018). https://doi.org/10.1145/3236367.3236373