Tiling Framework for Heterogeneous Computing of Matrix-Based Tiled Algorithms

Narasinga Rao Miniskar, Mohammad Alaul Haque Monil,
Pedro Valero-Lara, Frank Liu, Jeffrey S. Vetter
Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN, USA
(miniskarnr,monilm,valerolarap,liufy,vetter)@ornl.gov

ABSTRACT

Tiling matrix operations can improve the load balancing and performance of applications on heterogeneous computing resources. Writing a tile-based algorithm for each operation with a traditional, hand-tuned tiling approach that uses *for* loops in C/C++ is cumbersome and error prone. Moreover, it must enable and support the heterogeneous memory management of data objects and also explore architecture-supported, native, tiled–data transfer APIs instead of copying the tiled data to continuous memory before the data transfer. The tiling framework provides a tiled data structure for heterogeneous memory mapping and parameterization to a heterogeneous task specification API. We have integrated our tiled framework into MatRIS (Math kernels library using IRIS). IRIS is a heterogeneous run-time framework with a heterogeneous programming model, memory model, and task execution model. Experiments reveal that the tiled framework for BLAS operations has improved the programmability of tiled BLAS and improved performance by ∼20% when compared against the traditional method that copies the data to continuous memory locations for heterogeneous computing.

KEYWORDS

Heterogeneous computing, Tiling, IRIS, Heterogeneous memories, Task programming

ACM Reference Format:
Narasinga Rao Miniskar, Mohammad Alaul Haque Monil,, Pedro Valero-Lara, Frank Liu, Jeffrey S. Vetter. 2023. Tiling Framework for Heterogeneous Computing of Matrix-Based Tiled Algorithms. In *Proceedings of The 2nd International Workshop on Extreme Heterogeneity Solutions (ExHET Workshop'23).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Math libraries used in high-performance computing (HPC) are facing important challenges [2] due to ever-growing heterogeneity in current and future computer systems. Static decision-based mapping, which maps the math kernel to specific compute unit, is an outdated approach. Attempting to manually coordinate and schedule data placement/computation, which types of processors to select for certain calculations, and when computations will occur is becoming an intractable problem due to the growing complexity, diversity, and scale of HPC systems.

Researchers have developed tiled algorithms for math kernels to distribute their loads on multiple compute units and for scalability purposes. However, these algorithms are best suited for load distribution of homogeneous compute units but not for heterogeneous compute units. Compared with tiled algorithms for homogeneous compute units, the heterogeneous tiled algorithms require tiling specifications and must also manage the heterogeneous device memories and heterogeneous task creation. There is no tiling framework approach in the state of the art for heterogeneous tiling that handles both tiling specification and heterogeneous memories.

This paper proposes a unique tiling framework that binds a run-time specific heterogeneous device memory object handler to each tile. It also proposes the necessary sequence iterators and index-based tile access APIs for writing the tiled algorithms in a more readable and performance efficient way. Our approach is performance efficient because it seamlessly enables native architecture–specific 2D and 3D data transfer APIs for the tiled data movements. The proposed tiled framework handles the run-time system internally, and it is completely abstracted from the programmer, who can focus on writing the tiled program with operations between tiles. The proposed approach uses the IRIS [7] run-time framework for heterogeneous task creation and to handle heterogeneous memories.

We evaluated our tiled framework with two tiled algorithms: tiled matrix multiplication [8] and tiled LU factorization [9]. The proposed tiling framework for heterogeneous systems has achieved a ∼20% performance uplift when compared against a traditional, hand-tuned tiling approach that does not explore the native 2D/3D data transfer APIs.

2 BACKGROUND

2.1 IRIS

IRIS [7] is a programming system for extremely heterogeneous architectures that enables application developers to write portable applications across diverse heterogeneous programming platforms, including CUDA, HIP, Level Zero, OpenCL, and OpenMP. IRIS orchestrates multiple programming platforms in a system into a single execution/programming environment by providing portable

ExHET Workshop'23, February 25, 2023, Montreal, CA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

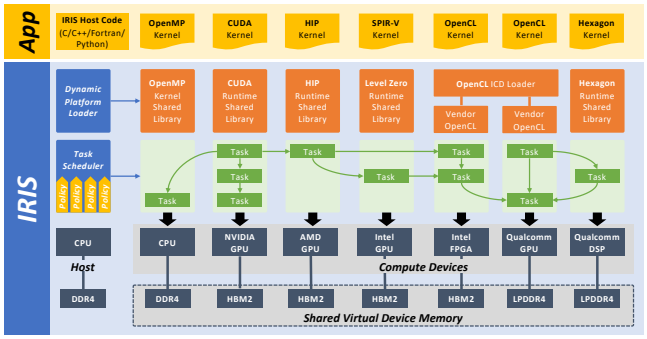


Figure 1: The IRIS architecture.

tasks and shared virtual device memory. Figure 1 illustrates the IRIS architecture.

IRIS provides a task-based programming model in which a task is a scheduling unit. A task runs on a single device, and it is portable across any compute device in the system, including accelerators (e.g., AMD and NVIDIA GPUs, FPGAs). A task contains zero or more commands. There are four types of commands: host-to-device memory copy command, device-to-host memory copy command, kernel launch command, and host command. A task can have a dependency on other tasks. When a task depends on other tasks, it cannot start until the prerequisite tasks complete. Therefore, writing an IRIS application means building directed acyclic graphs of tasks. Each task has a target device selection policy when it is submitted. The policy is specified by programmers, and it can be a device number, type (e.g., CPU, GPU, FPGA, DSP), or built-in policies (e.g., random, locality-aware, profile) provided by IRIS.

To achieve application portability and flexible task scheduling with effective data orchestration, IRIS provides shared virtual device memory across multiple, disjoint physical device memories. IRIS automatically transfers data across multiple devices to keep memory consistency across tasks. Therefore, all compute devices can share memory objects in the shared virtual device memory, and they can see the same content in the memory objects.

2.2 IRIS-BLAS

Still under development, IRIS-BLAS [8] is a novel, performance-portable BLAS library intended to address the portability challenges of BLAS for different heterogeneous architectures. IRIS-BLAS is built on top of the IRIS run-time and supports multiple vendor and open-source BLAS libraries, including OpenBLAS [13], Intel MKL [6], NVIDIA cuBLAS [10], and AMD hipBLAS [1]. In a heterogeneous system, IRIS-BLAS offloads the appropriate BLAS library kernel based on the task mapping at run-time. Thus, IRIS-BLAS is portable across a broad spectrum of architectures and BLAS libraries, thereby alleviating the worry of modifying the application source code. The effectiveness of IRIS-BLAS has been demonstrated on different CPUs (e.g., Intel Xeon Skylake, AMD 249 EPYC 7763, Qualcomm Snapdragon ARM cores) and GPUs (e.g,. NVIDIA A100, AMD MI100, Qualcomm Snapdragon Adreno). Although its objective is portability, IRIS-BLAS also provides competitive or even better performance compared with other state-of-the-art reference libraries [3]. By providing a vendor library kernel at run-time, IRIS-BLAS provides an important building block for implementing complex linear algebra algorithms.

2.3 LaRIS

LaRIS [9] (LaPACK library on the IRIS run-time) is a performance-portable LaPACK library that exploits the computational capabilities of a heterogeneous system to the fullest extent by using parameterized and auto-tuned tiled algorithms. LaRIS facilitates the inclusion of algorithm-specific performance models to guide scheduling in heterogeneous systems. Figure 2 shows the LaRIS software stack. Tiling is required to utilize all processors in a heterogeneous system, thereby enabling processing for larger matrix sizes. LaRIS provides automatic tiling and reconstructing. The tiling occurs before the creation of tasks and dependencies. While creating the graph, LaRIS associates memory chunks for different tiles to different tasks.

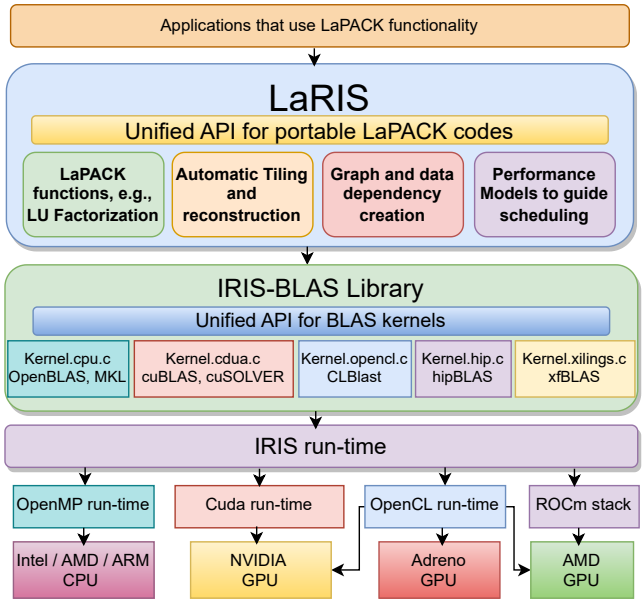


Figure 2: LaRIS and IRIS-BLAS software stack.

3 RELATED WORK

Although some tiling-based algorithms exist in the state of the art [5], they were proposed for homogeneous compute units or handled with rudimentary loop indexes with increments of indexes, and it is extremely difficult for the programmer to handle these tiles, and they are error prone. A small error in index usage in the tiled algorithm can lead to errors in run-time, and the issue then becomes more difficult to identify. Hence, a tiling framework is required to facilitate the programming of tiled algorithms. There are domain-specific compilers [4, 11, 12] defined for tiling. However, these approaches are defined for individual compute units (e.g., CPUs, GPUs, accelerators) but not for heterogeneous computing per se. The tiling for heterogeneous computing poses a new challenge of

how to manage the heterogeneous memories for tiles and how these tiled objects must be used to create tiled tasks for heterogeneous computing.

4 PROPOSED TILING FRAMEWORK

Tiling for heterogeneous computing involves dividing the matrices into tiles and managing the heterogeneous memories for the tiles. The proposed tiling framework addresses the configuration of tiling by binding the host tile memory object to IRIS memory for heterogeneous memory handling, provides iterators to access the tiles either in sequence or through indexing, and provides zip-on tiling iterators to ease the writing of tiled algorithms. The Tile2D data structure and its member variables and interfaces are shown in Figure 3.

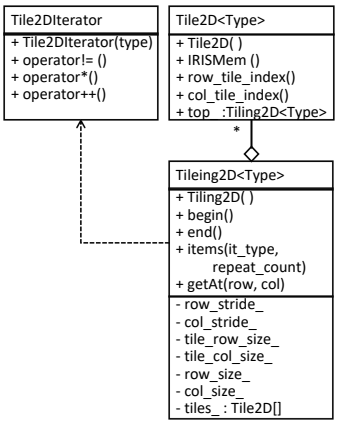


Figure 3: Tiling2D data structure with interfaces.

4.1 Configuration

The tiling object should be flexible enough to configure the tile specification and bind the tile to an IRIS memory object for the given input matrix.

4.1.1 Tile specification. The tile specification includes the size of the matrix in all dimensions and the size of the tile and stride between two adjacent tiles in all dimensions. The size of tile indicates the number of elements in the tile in each dimension. Stride indicates the offset between two adjacent tiles and is required for operations in which stride is a parameter for each dimension (e.g., convolution, image processing filters). As shown in Figure 3, the Tiling2D data structure has member variables for each 2D tile specification parameter and similar specification member variables for 3D tiling as well. The Tiling2D data structure maintains an array of Tile2D objects, in which each Tile2D object represents a row-tile index parameter and a column-tile index parameter. The Tile2D object also binds to an IRIS memory object.

4.1.2 Binding to IRIS memory. Binding a tile to an IRIS memory object is a unique and novel approach required for heterogeneous computing. Figure 4 illustrates the binding of a tile to an IRIS memory object. This binding enables the tiling data structure to map the tiles to heterogeneous memory locations through IRIS, and the programmer can focus on using these tiles to define the operations. Tiling objects will define the IRIS memory objects for tiles, manage the heterogeneous memories of kernels on these tiles, and manage the data transfers on these tiles via IRIS, thereby easing the process of writing tiled algorithms.

Another advantage of binding tiles to an IRIS memory object is that it maintains the tile host memory for the IRIS memory object with offsets from the base/start address of the matrix. Tile binding enables the IRIS memory to explore the architecture-supported, native, tiled-memory copy (data transfer) operations on the host memory's tile instead of first copying the tiled host memory to sequential memory. This copy would otherwise be an additional overhead in the traditional approach without a native tiled-memory copy operation (e.g., *cudaMemcpy2D* for CUDA GPUs, *hipMemcpy2D* for AMD GPUs). One can always write these APIs, but our tiling framework simplifies and abstracts the issue from programmers and enables them to focus on their actual work. Furthermore, this approach may enhance the performance of tiled algorithm execution when compared against traditional tiling because it avoids the memory copy of the host tile to a sequential memory address space in another host memory location.

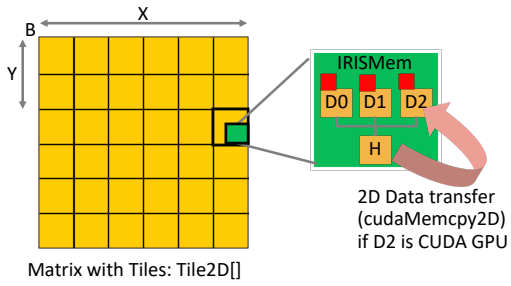


Figure 4: Binding an IRISMem object with a tile; IRISMem host memory (H) points to the matrix's base (B) start address with a 2D offset (X, Y).

4.2 Iterators

A tiling object should provide the necessary iterators, and it should be extendable. Operations on tiles need two types of iterators (1) sequence iterators for regularized controlled access and (2) indexed-based iterators for irregular or complex tile access.

4.2.1 Sequence iterators. The matrix addition with a tiled algorithm requires the iterators of both input matrices, *A* and *B*, in the same direction, such as iterating the tiles in column-wise fashion for each row (row major) or iterating the tiles in row-wise fashion for each column (column major). We have provided three types of sequence iterators for heterogeneous tiling operations: (1) a row major iterator, (2) a column major iterator, and (3) a right-down tree-wise iterator for LU factorization.

4.2.2 Index-based iterators. A tiled matrix multiplication operation can be defined in multiple ways, but the input matrices must be iterated in different directions (e.g., *A* matrix in column major order and *B* matrix in row major order). Moreover, the *A* and *B* matrix tiles must be accessed multiple times. This cannot be directly represented with a zip iterator with multiple sequence iterators of *A* and *B*. More complex examples include a tiled convolution operation. In these

scenarios, it is best to access the tiles by using indexing. An example of a tiled matrix multiplication is shown in Figure 6.

4.3 Zip-on Tiling Iterators

A *zip* iterator enables parallel iteration over several controlled sequence iterators (i.e., *A*, *B*, and *C* iterators) simultaneously, as shown in Figure 5. This approach would be useful for all regular sequence-based tiling iterators for point-wise operators (e.g., matrix addition, matrix subtraction, dot product, logical operators). The point-wise matrix operator would access the input tile only once in a predefined sequence iterator, either in row major or column major.

```
int tiled_matris_addition( int target, double *A, double *B, double *C, int
↪  SIZE, int tile_size) {
  Tiling2D<double> A_tiling(A, SIZE, SIZE, tile_size, tile_size);
  Tiling2D<double> B_tiling(B, SIZE, SIZE, tile_size, tile_size);
  Tiling2D<double> C_tiling(C, SIZE, SIZE, tile_size, tile_size);
  iris_graph graph; iris_graph_create(&graph);
  vector<iris_task> tasks;
  for(auto && it : zip(A_tiling.items(), B_tiling.items(), C_tiling.items()) {
    iris_task task;  iris_task_create(&task);
    Tile2D<double> & A_tile = std::get<0>(it);
    Tile2D<double> & B_tile = std::get<1>(it);
    Tile2D<double> & C_tile = std::get<2>(it);
    matris_task_add_matrix(task, A_tile.IRISMem(),
        B_tile.IRISMem(), C_tile.IRISMem());
    iris_graph_task(graph, task, target);
  }
  iris_graph_submit(graph);
}
```

Figure 5: Zip iterator used in a tiled matrix addition.

4.4 Tiled Algorithm Examples

4.4.1 Tiled matrix addition. Figure 5 shows an example of a tiled matrix addition algorithm. This algorithm requires regular sequencing of tiles in row-major order for two input matrices, *A* and *B*, and for the output matrix, *C*. The `matris_task_matrix_addition` is a heterogeneous BLAS API call for the addition of two matrices, and it takes IRIS memory objects as inputs and outputs, which are bonded to a tile in the host matrix's address space. The IRIS memory object creations, the tiling index management, and the data transfers on the tiles are completely abstracted from the programmer. A zip iterator is used to simultaneously iterate both input and output tiles.

4.4.2 Tiled matrix multiplication (DGEMM). It is not possible to use zip for matrix multiplication, which requires fine-grain control of tile access inputs and outputs using their tile indexes. Hence, we have used index-based tiling access for tiled matrix multiplication algorithm, as shown in Figure 6.

4.4.3 LU factorization. The tiled matrix LU factorization algorithm is shown in Figure 7. This is a non-recursive approach. In the top-level iterator, the tiles are accessed by using the right-down tree-wise iterator. This iterator first accesses the first row of tiles by starting from the *(0, 0)* index and then accesses the first column of tiles by starting from *(0, 0)*. After iterating through the first row and the first column, the iterator proceeds with the second row and the second column by starting from index *(1, 1)* and continues until

```
int tiled_matris_multiplication( int target, double *A, double *B, double *C,
↪  int SIZE, int tile_size) {
    Tiling2D<DTYPE> A_tiling(A, SIZE, SIZE, tile_size, tile_size);
    Tiling2D<DTYPE> B_tiling(B, SIZE, SIZE, tile_size, tile_size);
    Tiling2D<DTYPE> C_tiling(C, SIZE, SIZE, tile_size, tile_size);
    iris_graph graph; iris_graph_create(&graph);
    for(size_t i=0; i<A_tiling.row_tiles_count(); i++) {
        for(size_t j=0; j<B_tiling.col_tiles_count(); j++) {
            Tile2D<DTYPE> & c_tile = C_tiling.getAt(i, j);
            iris_task prev_task = NULL;
            for(size_t k=0; k<B_tiling.row_tiles_count(); k++) {
                Tile2D<DTYPE> & a_tile = A_tiling.getAt(i, k);
                Tile2D<DTYPE> & b_tile = B_tiling.getAt(k, j);
                iris_task task;  iris_task_create(&task);
                matris_task_dgemm
                    (task, MATRIS_ROW_MAJOR, MATRIS_NO_TRANS, MATRIS_NO_TRANS,
                     a_tile.row_tile_size(), b_tile.row_tile_size(),
                     b_tile.col_tile_size(),
                     1.0f, a_tile.IRISMem(), tile_size,
                     b_tile.IRISMem(), tile_size,
                     1.0f, c_tile.IRISMem(), tile_size);
                if (prev_task != NULL) {
                    iris_task gemm_depend_tasks[] =  { prev_task };
                    iris_task_depend(task, 1, gemm_depend_tasks);
                }
                iris_graph_task(graph, task, target_dev);
                prev_task = task;
            }
            iris_task_dmem_flush_out(prev_task, c_tile.IRISMem());
        }
    }
    iris_graph_submit(graph);
}
```

Figure 6: Tiled matrix multiplication with index-based tile accesses.

it reaches the last row and the last column of the tile. For each tile access in the top iterator, it will apply the following algorithm:

- If the row tile index and column tile index are the same for the tile, then it is considered an entry into a new step of the LU factorization. If it is step 0 (first step), then the GETRF operation is applied to this tile. Otherwise, the DGEMM operation is applied on all tiles in the step followed by the GETRF on the new step's entry tile. The second-level DGEMM iterator uses a simple row major iterator but starts with *(step, step)* indexing. However, no DGEMM operation is applied for the first step (i.e., *[0, 0]*). This algorithm creates the necessary DGEMM and GETRF tasks, passes the appropriate input and output IRISMem memory objects, and creates the appropriate task dependencies.
- If the row tile index is same as the current step, then apply the left TRSM operation with an assumption that the input is already in column major/transposed.
- If the column tile index is the same as the current step, then apply the top TRSM operation.

5 EXPERIMENTAL RESULTS

Although the proposed tiling framework makes writing the tiled algorithms easier for heterogeneous computing, the effectiveness of the tiling framework itself can be demonstrated with two metrics: (1) the overhead of the tiling framework and (2) the performance enhancement of tiling with native tile data transfers. These two metrics were derived on a truly heterogeneous system with multi-core AMD EPYC CPUs (128 CPU cores), four NVIDIA A100 GPUs, and four AMD MI100 GPUs. We considered two benchmarks for

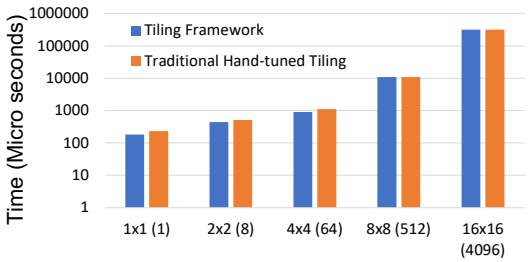
```
int tiled_matris_lufactorization( int target, double *A,
    int SIZE, int tile_size) {
    Tiling2D<DTYPE> A_tiling(A, SIZE, SIZE, tile_size, tile_size);
    size_t n_row_tiles = A_tiling.row_tiles_count(),
        n_col_tiles = A_tiling.col_tiles_count();
    iris_task getrf_tasks[n_row_tiles], gemm_tasks[n_row_tiles][n_col_tiles];
    iris_task top_trsm_tasks[n_col_tiles],  left_trsm_tasks[n_row_tiles];
    size_t step = 0;
    for(auto & a_tile : A_tiling.items(TILE2D_RIGHT_DOWN_TREE_WISE)) {
        if (a_tile.row_tile_index() == a_tile.col_tile_index()) {
           // First check whether GEMM has to be applied and proceed with GETRF
            step = a_tile.row_tile_index();
           if (step != 0) for(auto & gemm_tile : A_tiling.items(step, step)) {
              size_t tile_jj = gemm_tile.row_tile_index();
              size_t tile_ii = gemm_tile.col_tile_index();
             Tile2D<DTYPE> & left_trsm_tile = A_tiling.getAt(step-1, tile_ii);
              Tile2D<DTYPE> & top_trsm_tile = A_tiling.getAt(tile_jj, step-1);
              iris_task task; iris_task_create(&task);
              laris_task_dgemm(graph, task, target, left_trsm_tile.IRISMem(),
                    top_trsm_tile.IRISMem(), gemm_tile.IRISMem(), tile_size);
              if (step-1 == 0) {
                    iris_task gemm_depend_tasks[] =  {
                        left_trsm_tasks[tile_ii], top_trsm_tasks[tile_jj] };
                    iris_task_depend(task, 2, gemm_depend_tasks);
              }
              else {
                   iris_task gemm_depend_tasks[] =  { left_trsm_tasks[tile_ii],
                      top_trsm_tasks[tile_jj], gemm_tasks[tile_jj][tile_ii] };
                    iris_task_depend(task, 3, gemm_depend_tasks);
              }
              gemm_tasks[tile_jj][tile_ii] = task;
           }
           // Now Do GETRF: GETRF of new step
           iris_task_create( &getrf_tasks[step]);
           laris_graph_getrf(getrf_tasks[step], target,
                    step, tile_size, a_tile.IRISMem() );
           if (step != 0) {
               iris_task getrf_depend_tasks[] =  { gemm_tasks[step][step] };
               iris_task_depend(getrf_tasks[step], 1, getrf_depend_tasks);
           }
        }
        else if (a_tile.row_tile_index() == step) {
           // Do LEFT TRSM (Assuming it is transformed)
           size_t tile_ii = a_tile.col_tile_index();
           Tile2D<DTYPE> & getrf_tile = A_tiling.getAt(step, step);
           iris_task_create(&left_trsm_tasks[tile_ii]);
           laris_left_graph_trsm(graph, left_trsm_tasks[tile_ii], target,
                    step, tile_size, getrf_tile.IRISMem(), a_tile.IRISMem());
           iris_task parent_gemm_task=NULL;
           if (step != 0) parent_gemm_task = gemm_tasks[step][tile_ii];
           iris_task trsm_depend_tasks[] =  {
                    getrf_tasks[step], parent_gemm_task };
           iris_task_depend(left_trsm_tasks[tile_ii], 2, trsm_depend_tasks);
        }
        else if (a_tile.col_tile_index() == step) {
           // Do TOP TRSM (Assuming it is transformed)
           size_t tile_jj = a_tile.row_tile_index();
           Tile2D<DTYPE> & getrf_tile = A_tiling.getAt(step, step);
           iris_task_create(&top_trsm_tasks[tile_jj]);
           laris_top_graph_trsm(graph, top_trsm_tasks[tile_jj], target,
               tile_size, getrf_tile.IRISMem(), a_tile.IRISMem());
           iris_task parent_gemm_task = NULL;
           if (step != 0) parent_gemm_task = gemm_tasks[tile_jj][step];
           iris_task trsm_depend_tasks[] =  {
                    getrf_tasks[step], parent_gemm_Task };
           iris_task_depend(top_trsm_tasks[tile_jj], 2, trsm_depend_tasks);
        }
    }
    iris_graph_submit(graph);
}
```

Figure 7: Tiled LU factorization algorithm.

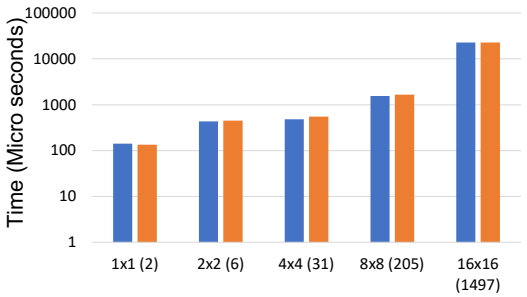
these experiments: tiled matrix multiplication (IRIS-BLAS) and LU factorization (LaRIS).

5.1 Overhead of Tiling Framework

We compared our tiling framework overhead with traditional, hand-written tiling with indexes, as shown in Figure 8. The overhead is the time taken to create heterogeneous tasks and a task graph with our tiling framework. We compared the times for our framework against the traditional, hand-tuned tiling approach. Our tiling framework's overhead is significantly lower (in the order of microseconds) when compared with the traditional method. In the traditional approach, the tiled algorithm loops handle tiling indexes but are error prone. Our tiling framework approach needs less code to write the tiled algorithms, and the overhead of iterators is comparable to the traditional approach. Our results are consistent for both the tiled DGEMM operation and the tiled LU factorization. The varying x-axis (tile count) leads to a varying number of tasks in the task graph. Our tiling framework overhead for an increasing number of tasks is still negligible when compared with the traditional, hand-tuned tiling approach.



(a) Varying X-axis Tile Count: M x M (# DGEMM Tasks)



(b) Varying X-axis Tile Count: M x M (# LU Factorization Tasks)

Figure 8: Overhead of tiling framework compared with traditional, hand-tuned tiling. Average overhead when compared with traditional approach is ~90 microseconds.

5.2 Performance Enhancement with Native Tile Data Transfers

Our tiling framework enables native 2D/3D tile data transfer APIs (i.e., for CPUs and GPUs) to transfer the data from a host tile to device memory through the IRIS run-time to provide performance enhancement of a tiled algorithm's execution on heterogeneous computing resources. The comparison of these two approaches for the tiled DGEMM and LU factorization algorithms is shown in Figure 9. We varied the matrix size, as shown on the x-axis, and measured the execution time of a tiled algorithm's task graph. The traditional, hand-tuned approach introduces a memory copy operation for flattening the tile to a continuous host memory location. On average, we observed a ~20% performance uplift compared with the traditional approach.

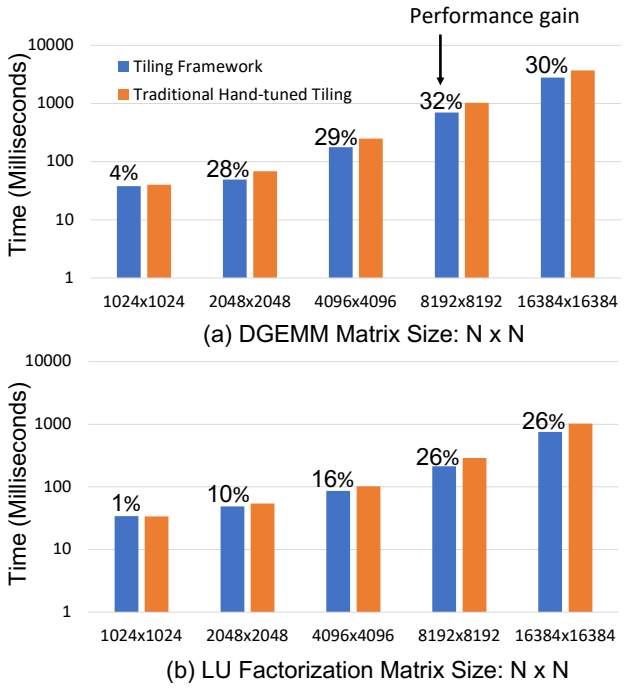


Figure 9: Performance enhancement of tiling framework with native tile data transfer APIs (Execution time in log scale)

5.3 Source Code Reduction

We were able to significantly reduce the amount of source code required for these operations in our tiling framework: a 78% reduction for the tiled GEMM algorithm and a 45% reduction for the tiled LU factorization. The tiled GEMM application is more complex to write when using traditional tiling with indexes.

Table 1: Source code reduction (lines of code) with our tilling framework.

Algorithm	Traditional Tiling Lines	Proposed Tiling Lines	Line reduction
Tiled GEMM	230	50	78%
Tiled LU Factorization	125	68	45%

5.4 Future Work

The tiling framework described here addresses the challenges and burdens associated with writing tiled algorithms by handling the heterogeneous tile objects and enabling the creation of heterogeneous computing tasks with the tile objects. Adding a domain-specific compiler to our tiling framework could make it even more robust for writing the simplified tiled algorithms, and this is future work.

6 CONCLUSION

We proposed and described a novel framework for writing tiled algorithms for heterogeneous computing. This unique tiling framework creates the tiles, handles the heterogeneous memory objects, *and* enables them to be used to create heterogeneous tasks for heterogeneous computing. Our approach simplifies writing the tiled algorithms and is also performance efficient when compared against the traditional, handwritten tiling approaches, which do not exploit the architecture-native tile data transfer APIs. The gains are ~20% when compared to traditional approaches.

ACKNOWLEDGMENTS

Notice: This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for U.S. Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

REFERENCES

[1] AMD. 2022. hipBLAS, the Basic Linear Algebra Subroutine library. https://github.com/ROCmSoftwarePlatform/hipBLAS [Online; accessed 6-July-2022].
[2] James Ang, A. Andrew Chien, Simon David Hammond, Adolfy Hoisie, Ian Karlin, Scott Pakin, John Shalf, and Jeffrey Vetter. 2022. Reimagining Codesign for Advanced Scientific Computing: Report for the ASCR Workshop on Reimaging Codesign. https://www.osti.gov/biblio/1822199 [Online; accessed 6-July-2022].
[3] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.
[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2018. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. *CoRR* abs/1804.10694 (2018). arXiv:1804.10694 http://arxiv.org/abs/1804.10694
[5] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35, 1 (2009), 38–53.
[6] Intel. 2022. The Intel Math Kernel Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html?s=Newest [Online; accessed 6-July-2022].
[7] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S. Vetter. 2021. IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems. In *2021 IEEE High Performance Extreme Computing Conference, HPEC 2021, Waltham, MA, USA, September 20-24, 2021*. IEEE, 1–8. https://doi.org/10.1109/HPEC49654.2021.9622873
[8] Narasinga Rao Miniskar, Alaul Haque Monil Mohammad, Valero-Lara Pedro, Frank Liu, and Jeffrey S Vetter. 2022. IRIS-BLAS: Towards a Performance Portable and Heterogeneous BLAS Library. In *29th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2022, Bengaluru, India, December 18-21, 2022*. IEEE.
[9] Alaul Haque Monil Mohammad, Narasinga Rao Miniskar, Frank Liu, Jeffrey S Vetter, and Valero-Lara Pedro. 2022. Targeting Portability and Productivity for LAPACK Codes on Extreme Heterogeneous Systems by Using IRIS. In *SC 2022 Workshop. RSDHA: Redefining Scalability for Diversely Heterogeneous Architectures.*
[10] NVIDIA. 2022. cuBLAS, the CUDA Basic Linear Algebra Subroutine library. https://docs.nvidia.com/cuda/cublas/index.html [Online; accessed 6-July-2022].
[11] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Commun. ACM* 61, 1 (dec 2017), 106–115. https://doi.org/10.1145/3150211
[12] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973
[13] Martin Kroeker Zhang Xianyi. 2022. OpenBLAS. https://www.openblas.net/ [Online; accessed 6-July-2022].