The final publication is available at ACM via http://dx.doi.org/10.1145/3524059.3532385

Seamless Optimization of the GEMM Kernel for Task-based Programming Models

Arthur F. Lorenzon
aflorenzon@unipampa.edu.br
Federal University of Pampa
Alegrete, RS, Brazil

Sandro M. V. N. Marques
sandromarques.aluno@unipampa.edu.br
Federal University of Pampa
Alegrete, RS, Brazil

Antoni Navarro
antoni.navarro@bsc.es
Barcelona Supercomputing Center (BSC)
Barcelona, Spain

Vicenç Beltran
vbeltran@bsc.es
Barcelona Supercomputing Center (BSC)
Barcelona, Spain

ABSTRACT

The general matrix-matrix multiplication (GEMM) kernel is a fundamental building block of many scientific applications. Many libraries such as Intel MKL and BLIS provide highly optimized sequential and parallel versions of this kernel. The parallel implementations of the GEMM kernel rely on the well-known fork-join execution model to exploit multi-core systems efficiently. However, these implementations are not well suited for task-based applications as they break the data-flow execution model. In this paper, we present a task-based implementation of the GEMM kernel that can be seamlessly leveraged by task-based applications while providing better performance than the fork-join version. Our implementation leverages several advanced features of the OmpSs-2 programming model and a new heuristic to select the best parallelization strategy and blocking parameters based on the matrix and hardware characteristics. When evaluating the performance and energy consumption on two modern multi-core systems, we show that our implementations provide significant performance improvements over an optimized OpenMP fork-join implementation, and can beat vendor implementations of the GEMM (e.g., Intel MKL and AMD AOCL). We also demonstrate that a real application can leverage our optimized task-based implementation to enhance performance.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**;
• **Theory of computation** → **Parallel computing models**.

KEYWORDS

GEMM, Malleability, Parallel Computing, Energy-efficiency

1 INTRODUCTION

Many applications from traditional and emerging domains rely on the Basic Linear Algebra Subprograms (BLAS [9]) library to exploit multi-core systems. Among the routines implemented by BLAS, the general matrix-matrix multiplication (GEMM) is widely employed by machine learning algorithms, climate models, and applications used to rank the systems on the Top500 list of the fastest supercomputers in the world (e.g., High-Performance Linpack Benchmark – HPL). As a consequence, different GEMM optimized implementations for a given underlying hardware have been proposed, e.g., Intel MKL [1], AMD Core Math Library [2], OpenBLAS [34], ATLAS [9], and BLIS [33].

These high-performance libraries are usually parallelized following the fork-join paradigm (e.g., OpenMP `parallel for`), providing significant performance improvements compared to the sequential GEMM-BLAS implementation on multi-core systems. However, their rigid execution model and lack of malleability may not deal with some hardware and software aspects (e.g., data synchronization and cache contention) when there is variability in the application behavior or execution environment, preventing linear improvements. When scenarios like these arise, the rigid fork-join-based GEMM implementations can increase the energy consumption and execution time. Task-based programming models can overcome some of the fork-join limitations providing more malleability and better load-balancing on multi-core systems.

However, there is no optimized implementation of the GEMM routine with task-based models available to the best of our knowledge. Hence, to develop a parallel application that requires a high-performance GEMM implementation (e.g., Cholesky decomposition), software developers have the following options: (*i*) The entire application is forced to use the fork-join model that is optimized for performance but does not present malleability. (*ii*) The application is parallelized with tasks, but the GEMM kernel is limited to sequential versions, discarding an optimized parallel version's potential benefits (e.g., panel reuse between cores). Finally (*iii*), combining a task-based application with a GEMM kernel that is parallelized using the fork-join model. However, this last option usually results in the worst performance due to the oversubscription effects between the tasking and fork-join runtimes. This highlights the need for an optimized task-based implementation, benefiting high-level frameworks with a more performant and dynamic implementation.

With that in mind, we propose two highly optimized task-based implementations that leverage advanced features from the OmpSs-2 programming model (discussed in Section 2). The former implementation uses the `taskloop`[21] construct with support for data dependencies, which eliminates the need for coarse-grained synchronization. The second implementation extends the previous one with the `task for` [20] clause, which provides the best of both tasks and fork-join execution models, work-sharing tasks. Both implementations have been integrated into BLIS [33], a software framework that allows programmers and vendors to instantiate high-performance libraries with BLAS functionalities. With that,

end-users can take advantage of the malleability leveraged by our implementations and transparently employ it in a wide variety of linear algebra methods.

Simultaneously, as shown in our work, finding adequate task granularities becomes critical to yield competitive performance: an excessive number of fine-grained tasks will increase task overheads, but too few coarse-grained tasks will hinder the available parallelism. Therefore, to not burden the software developer with dealing with this challenging task and offer more malleability to the applications, we propose a heuristic to automatically and transparently select the adequate task granularity that works at runtime and without the need for a training phase. In a nutshell, this work presents the following contributions:

- A generic task-based implementation of the GEMM routine that can be implemented with any task-based parallel programming interface;
- Two task-based implementations that leverage advanced features offered by the OmpSs-2 programming model; and
- A heuristic to automatically define the parallelization scheme and parameters (e.g., block size, task granularity, and workload distribution) at runtime, namely *ASOC* (automatically selection of optimal configurations).

We validate the contributions of our work through an extensive set of experiments on two modern high-performance computing systems (Intel and AMD). When running the GEMM routine over nineteen different workloads, we show that: (*i*) By finding an optimized configuration (parallelization scheme and parameters), a task-based implementation can deliver better performance than BLIS's optimized OpenMP fork-join implementation. (*ii*) Leveraging the malleability of the optimized task-based implementations causes the energy consumption to be reduced compared to the OpenMP fork-join implementation on the Intel Xeon processor. (*iii*) Our final implementation shows that combining tasking and work-sharing leads to extremely balanced workloads and minimal runtime overhead. (*iv*) Our task-based version can reach better performance and energy results than the optimized vendor's implementation of GEMM (e.g., Intel MKL and AMD Optimizing CPU Libraries – AOCL) for most workloads. On top of that, we have used the Cholesky decomposition algorithm to illustrate that our optimized task-based version can be seamlessly integrated with task-based applications. Our evaluation shows that our optimized task-based GEMM can significantly improve the performance of tasks-based algorithms that currently rely on sequential GEMM kernels.

The remainder of this paper is structured as follows. In Section 2, we describe the background and motivation of this study. In Section 3, we give a thorough description of the implemented strategies and the heuristic. Then, in Section 4, we describe the methodology and discuss the results of our contributions. In Section 5, we discuss the related work and highlight our contributions. Finally, we draw the conclusions and future work in Section 6.

2 BACKGROUND AND MOTIVATION

BLIS is a software framework that allows programmers and vendors to instantiate high-performance libraries with BLAS functionality [33]. Because of its simplicity and performance, BLIS has been employed in many linear algebra methods, and optimization tools (e.g., Cholesky factorization, LU Decomposition, and AMD AOCL) [3, 13]. Furthermore, BLIS provides a highly optimized micro-kernel to implement the GEMM kernel for each specific micro-architecture. Hence, we have implemented our task-based versions on top of the BLIS infrastructure. Next, we describe the GEMM sequential and parallel fork-join implementations provided by BLIS.

Algorithm 1 Sequential implementation of GEMM by BLIS

```
1:  for j_c = 0, ..., n - 1 in steps of n_c do
2:      B_jc = B + j_c * cs_b
3:      C_jc = C + j_c * cs_c
4:      for p_c = 0, ..., k - 1 in steps of k_c do
5:          A_pc = A + p_c * cs_a
6:          B_pc = B_jc + p_c * rs_b
7:          B̃ = pack(B_pc)
8:          for i_c = 0, ..., m - 1 in steps of m_c do
9:              A_ic = A_pc + i_c * rs_a
10:             C_ic = C_jc + i_c * rs_c
11:             Ã = pack(A_ic)
12:             for j_r = 0, ..., n_c - 1 in steps of n_r do
13:                 B_jr = B̃ + j_r * δ_B
14:                 C_jr = C_ic + j_r * cs_c * NR
15:                 for i_r = 0, ..., m_c - 1 in steps of m_r do
16:                     A_ir = Ã + i_r * δ_A
17:                     C_ir = C_jr + i_r * rs_c * MR
18:                     C(i_r : i_r + m_r − 1, j_r : j_r + n_r − 1)+ = ...
19:                 end for
20:             end for
21:         end for
22:     end for
23: end for
24: return C_mn
```

2.1 Sequential Implementation of the GEMM

GEMM can be formalized as $C = A \times B + C$, where A, B, and C are matrices of size $m \times k$, $k \times n$, and $m \times n$, respectively. It is worth mentioning that a BLIS implementation considers all matrices to be stored in row-major order to improve data locality. Hence, Algorithm 1 depicts the sequential version of the GEMM operation while Figure 1 illustrates each step of the algorithm[1]. In the j_c loop, matrices C and B are divided into column panels of size n_c. Next, in the p_c loop, A and the current column panel B_{jc} are partitioned into column and row panels of size k_c, respectively. Moreover, BLIS packs the current row panel of $B_{kc,jc}$ into $\tilde{B}$, a contiguous buffer in memory. Then in the i_c loop, C_{jc} and A_{pc} are partitioned into smaller blocks of m_c lines, and $A_{mc,kc}$ is packed into a contiguous buffer $\tilde{A}$. At this point, to better accommodate data into cache, the j_r loop partitions both C_{ic} and $\tilde{B}$ into column slivers of width n_r, while the inner-most i_r loop divides the current block of $\tilde{A}$ into row slivers of height m_r. Finally, the micro-kernel multiplies the slivers of $\tilde{A}$ by the slivers of $\tilde{B}$, and updates the $m_r \times n_r$ position of C_{ir}.

2.2 Identifying Parallelization Opportunities

When the i_r loop is parallelized, each created thread computes multiple instances of the micro-kernel operation. However, as it has a limited amount of parallelism (i.e., the number of iterations is defined by $\frac{m_c}{m_r}$), this loop is only parallelized when the ratio of m_c and m_r is large enough to overcome the overhead of managing

[1] cs_a, cs_b, and cs_c denote the distance to the next column. rs_a, rs_b, and rs_c denote the distance to the next row. δ_A and δ_B denote for values computed by the algorithm.

the parallelism. By parallelizing the j_r loop, each created thread multiplies the $\tilde{A}$ block with the sliver of $\tilde{B}$. It also has a limited amount of iterations, defined by the ratio of n_c and n_r. Hence, this loop is only worthy of parallelization when n_c is large enough. For the parallelization of the i_c loop, each created thread will get a distinct block of $\tilde{A}$ and share the same $\tilde{B}$ among all threads. It is a potential loop to be parallelized as the number of iterations only depends on the size of dimension m. If the parallelism is exploited over the p_c loop, each thread will be assigned a column panel B and a row panel A. Although the loop presents possibilities of parallelism exploitation, it needs a more advanced dependency system or a barrier, which would likely penalize the entire application's performance. Finally, when the outermost loop (indexed by j_c) is parallelized, each thread will be assigned a different row panel of B and C, and all threads will share matrix A. This loop presents the coarsest granularity that each thread will receive as it depends on the ratio of n and n_c.

2.3 Parallel Fork-join Implementation

BLIS provides two fork-join parallel implementations of the GEMM routine, one using OpenMP and the other using POSIX threads. On both versions the four loops are parallelized (j_c, i_c, j_r, and i_r). OpenMP [25] has a highly optimized fork-join execution model especially well-suited to exploit structured parallelism. The well-known `omp for` construct is commonly used to parallelize embarrassingly parallel loops. BLIS' strategy performs a weighted partitioning based on the relative length of the m and n dimensions. Then, threads are split between the loops and assigned greedily to a predefined maximum for each dimension. The parallelization is hierarchical, where the total number of running threads is the product of the amount of parallelism exploited on each loop. Although the end-user can tune the degree of parallelism for each loop, the common practice is defining the total number of threads so that BLIS can decide the degree of parallelism. It is worth mentioning that this implementation requires a barrier during the packing panels of matrices A and B into contiguous buffers, and after the execution of the i_c loop to ensure that different threads do not update the same position of C at the same time (between lines 21 and 22 of Algorithm 1). This parallel and high-performance implementation based on OpenMP is well-suited for accelerated applications that follow a fork-join model. However, it is sub-optimal for task-based applications following a data-flow execution model. Task-based applications require a global barrier before and after calling a kernel parallelized using the fork-join model, breaking the data-flow execution model. Thus, it is essential to develop task-based GEMM kernels seamlessly integrated with task-based applications.

2.4 Task-based Models & OmpSs-2 Advanced Features

The flexibility of the data-flow execution model relies on the dynamic management of data dependencies among tasks. However, the management of dependencies comes at a cost, and if done incorrectly, it may introduce a non-negligible overhead depending on the number of tasks [7]. Moreover, exploiting dynamic, irregular, and nested parallelism can become a challenge depending on the granularity of tasks. To mitigate this problem and widen the parallelization opportunities in task-based applications, the OmpSs-2 programming model offers advanced features to exploit multi-core systems. Moreover, it also provides advanced constructs to enable a more straightforward parallelization of loops, which transparently widens the available workload. These features are discussed next.

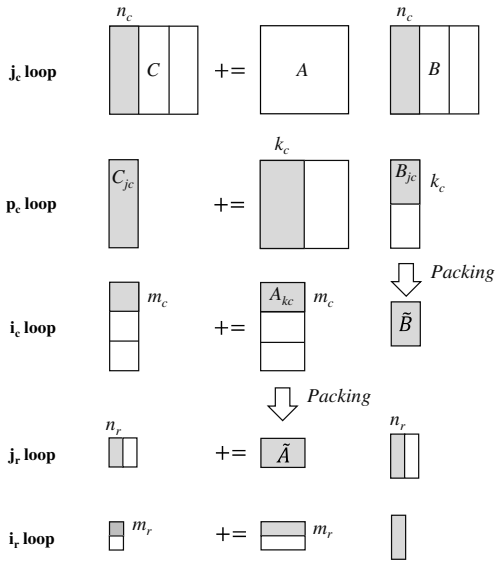


Figure 1: Illustration of Algorithm 1.

`Weak dependencies`[27]: Nesting in tasks requires the specification of data dependencies at every nesting level to link the domains of dependencies and ensure a correct execution order. Nonetheless, deferring the execution of an outer-level task due to its dependencies may not always be needed. If only inner sub-tasks access the managed data, the parent task must also specify those data dependencies, but it is unnecessary to defer its execution. For this purpose, OmpSs-2 offers the **weak** counterpart to all the dependency clauses. These variants indicate that a task does not require enforcing synchronization, but a future nested task will. Due to this, the execution of the parent task is not deferred and thus can start beforehand. Transparently, this causes applications to benefit from having multiple task-creators, which widens the workload faster and improves application performance.

`Taskloop Dependencies`[21]: The OpenMP *taskloop* construct does not support data dependencies. This leads to the need for a coarse-grained synchronization mechanism, such as `taskwaits`, and these usually translate to adverse effects in performance. To tackle this issue, OmpSs-2 offers data-dependency clauses within the *taskloop* construct. Through the induction variable of loops, programmers can specify dependencies. More specifically, each partition of the loop will have its dependency, resulting in a data-flow execution model that avoids the synchronization points imposed by `taskwaits`.

`Task For`[20]: As aforementioned, the two most common parallelization strategies are fork-join and task-based. The former exploits structured parallelism while the latter exploits dynamic, irregular, and nested parallelism. Applications that show both types

Algorithm 2 Generic task-based implementation of GEMM

```
1:  get(n_cpus)
2:  panel_allocation(Ã[n_cpus])
3:  for j_c = 0, ..., n - 1 in steps of n_c do
4:      B_jc = B + j_c * cs_b
5:      C_jc = C + j_c * cs_c
6:      for p_c = 0, ..., k - 1 in steps of k_c do
7:          A_pc = A + p_c * cs_a
8:          B_pc = B_jc + p_c * rs_b
9:          B̃ = pack(B_pc)
10:         for i_c = 0, ..., m - 1 in steps of m_c do
11:             #pragma oss task inout(Ã[t_id%n_cpus], C_jc[i_c * rs_c])
12:             A_ic = A_pc + i_c * rs_a
13:             C_ic = C_jc + i_c * rs_c
14:             Ã[t_id%n_cpus] = pack(A_ic)
15:             for j_r = 0, ..., n_c - 1 in steps of n_r do
16:                 B_jr = B̃ + j_r * δ_B
17:                 C_jr = C_ic + j_r * cs_c * NR
18:                 for i_r = 0, ..., m_c - 1 in steps of m_r do
19:                     A_ir = Ã + i_r * δ_A
20:                     C_ir = C_jr + i_r * rs_c * MR
21:                     C(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)+ = ...
22:                 end for
23:             end for
24:         end for
25:     end for
26: end for
27: #pragma oss taskwait
28: return C_mn
```

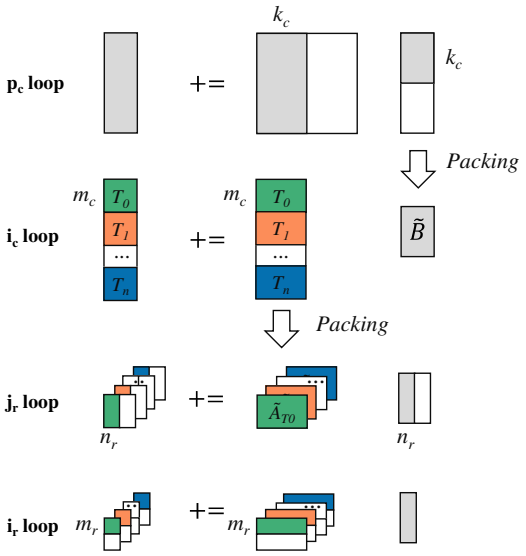


Figure 2: Illustration of Algorithm 2

of parallelism could benefit from both strategies, but it is not trivial to combine both while leaving performance unhindered. For this purpose, OmpSs-2 offers the `for` clause, which represents a task that internally leverages work-sharing techniques to exploit fine-grained loop-based parallelism. These tasks are helpful in scenarios where not enough parallelism per core is exposed. They can adapt to system demands due to their ability to run in several threads concurrently. This removes the need to choose between a large number of fine-grained tasks – which may lead to runtime overhead –, and a small number of coarse-grained tasks that are not enough to exploit all available cores.

3 OPTIMIZED TASK-BASED GEMM IMPLEMENTATION

In this section, we describe the proposed parallel task-based implementations of GEMM. As already mentioned, we have implemented them into the BLIS Sandbox infrastructure, which allows the implementation of different parallel schemes while benefiting from its existing build system. To ensure that the parallelization strategy is the only difference among the fork-join and task-based versions, our BLIS sandbox implementation considers a few rules. First, all matrices are allocated in row-major order to better accommodate data into the cache memory. Second, it applies the algorithms implemented by BLIS to pack the panels of matrices A and B into contiguous buffers. Finally, it uses the same optimized micro-kernel as the fork-join implementation.

We start by describing an implementation of the GEMM routine with the task-based model that uses common directives offered by different programming interfaces. Next, we present the two implementations that use advanced features from OmpSs-2. Finally, we describe the heuristic to select the best parallelization strategy and optimal parameters.

3.1 Generic Task-based Implementation

This section describes a generic task-based implementation to parallelize an optimized sequential GEMM that can be implemented with standard OpenMP task constructs. Given the discussion in Section 2.2 – and as OpenMP does not provide the weak dependencies feature – between the five loops around the micro-kernel, we choose to parallelize the i_c loop due to its available parallelism. Even though we use OmpSs-2 directives to describe our implementations, porting them to OpenMP is a straightforward task.

Algorithm 2 depicts the proposed implementation while Figure 2 illustrates the partitioning of the matrices. First, it obtains the number of CPUs (n_{cpus}) used by the application – settable by the user, by default the number of cores available in the architecture. When the application gets to the `#pragma oss task` directive (*line 11*), a new explicit task is generated from the associated structured block. Each task will receive a different shared block of C_j, a panel of $\tilde{A}$ based on its task id ($t_{id}\%n_{cpus}$) and will share the same panel $\tilde{B}$. In total, for each p_c loop iteration, $\frac{m}{m_c}$ tasks will be created.

Two `inout` dependencies are defined on a per-task basis to ensure correctness in the algorithm. First, $C_{jc}[i_c * rs_c]$ is protected so that the positions of the C_{jc} shared matrix are not being updated by more than one task at a time. Second, we protect $\tilde{A}[t_{id}\%n_{cpus}]$ to ensure that distinct tasks that are created at each p_c iteration will run one after another, sharing the same panel $\tilde{A}$. It is worth mentioning that defining these `inout` dependencies avoids the need to use a barrier after i_c loop iteration. Given this, the panels of $\tilde{A}$ can be allocated only once at the beginning (line *2*), so that the allocations barely affect performance and avoid increasing the amount of necessary memory. Finally, each created task will compute over a different position of C_{ic} matrix in parallel, as illustrated in Figure 2. The `oss taskwait` waits until all previous tasks have completed before returning the C_{mn} matrix (line *27*).

Algorithm 3 OmpSs-2 Taskloop Variant

```
1:  get_parameters(n_cpus, oss_nc, oss_mc, oss_bsx, oss_bsy)
2:  panel_allocation(Ã[n_cpus])
3:  panel_allocation(B̃[ n / oss_nc ])
4:  #pragma oss taskloop grainsize(oss_bsx)
5:  for j_c = 0, ..., n - 1 in steps of oss_nc do
6:      B_jc = B + j_c * cs_b
7:      C_jc = C + j_c * cs_c
8:      for p_c = 0, ..., k - 1 in steps of k_c do
9:          A_pc = A + p_c * cs_a
10:         B_pc = B_jc + p_c * rs_b
11:         B̃[tid_jc] = pack(B_pc)
12:         #pragma oss taskloop grainsize(oss_bsy) inout(Ã[t_ic%n_cpus], C_jc[ic*rs_c])
13:         for i_c = 0, ..., m - 1 in steps of oss_mc do
14:             A_ic = A_pc + i_c * rs_a
15:             C_ic = C_jc + i_c * rs_c
16:             Ã[t_ic%n_cpus] = pack(A_ic)
17:             for j_r = 0, ..., n_c - 1 in steps of n_r do
18:                 B_jr = B̃ + j_r * δ_B
19:                 C_jr = C_ic + j_r * cs_c * NR
20:                 for i_r = 0, ..., m_c - 1 in steps of m_r do
21:                     A_ir = Ã + i_r * δ_A
22:                     C_ir = C_jr + i_r * rs_c * MR
23:                     C(i_r : i_r + m_r − 1, j_r : j_r + n_r − 1)+ = ...
24:                 end for
25:             end for
26:         end for
27:     end for
28: end for
29: #pragma oss taskwait
30: return C_mn
```

3.2 OmpSs-2 Taskloop Variant

This strategy uses OmpSs-2 *taskloop dependencies*, as described in Section 2.4, on the parallelization of the j_c and i_c loops, as those are the loops with the highest parallelism available. We describe the parallelization scheme in Algorithm 3 and illustrate the partitioning of the matrices in Figure 3. The strategy considers as input four parameters defined by an environment variable that is used to distribute the workload among the created tasks: oss_{nc} and oss_{bsx}, which respectively define the block size assigned to each task and the number of iterations (granularity) that each task will receive from the j_c loop; and oss_{mc} and oss_{bsy}, which define the same parameters for the i_c loop. If the user does not define these, default values from BLIS are used instead, and oss_{bsx} and oss_{bsy} are set to *1*. All these values are initialized in line 1 of the Algorithm 3.

When the `#pragma oss taskloop` directive around the j_c loop is reached (line *4*), $tasks_{jc}$ tasks are created, defined by $\frac{n}{oss_{bsx} \times oss_{nc}}$. Each will be assigned to a different column panel of matrices C and B, sharing matrix A. Then, in the p_c loop, each task will share the same block of matrix A of size k_c and split the column panel of B_{jc} in k_c lines. Moreover, each task packs its column panel of B_{pc} into the contiguous buffer $\tilde{B}[tid_{jc}]$, where tid_{jc} is the task id (from 0 to $tasks_{jc} - 1$). Similarly to the design choice discussed in Section 3.1, all buffers are allocated at the beginning of the execution (line *3*).

When each parallel task reaches the `#pragma oss taskloop` directive around the i_c loop (line 12), it creates as many tasks as represented by $tasks_{ic} = \frac{m}{oss_{bsy} \times oss_{mc}}$. There will be a total of $tasks_{jc} \times tasks_{ic}$ parallel tasks for each iteration of the p_c loop. Hence, to ensure the correct result of the GEMM routine, the algorithm applies the same dependencies discussed in Section 3.1 w.r.t the panel $\tilde{A}$ and the positions of matrix C_{jc}. Moreover, each

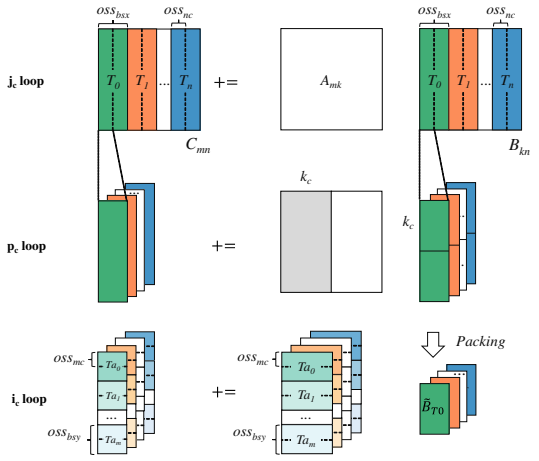


Figure 3: Illustration of Algorithm 3

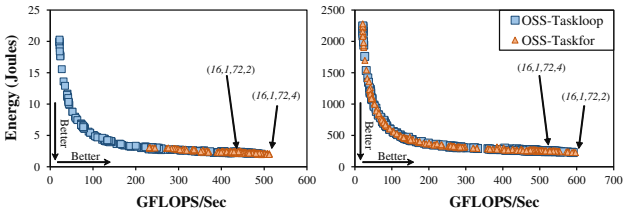
task will compute over a block C_{ic} of size $oss_{bsy} \times oss_{nc}$, A_{ic} of size $oss_{bsy} \times k_c$, and share $\tilde{B}$ of size $k_c \times oss_{nc}$ among all tasks created by its parent task. All tasks compute the loops indexed by j_r and i_r and the micro-kernel in parallel from this moment on. Finally, the `oss taskwait` directive ensures that all tasks finish the computation before returning matrix C_{mn} (line *29*).

3.3 OmpSs-2 Task For Variant

The second optimized strategy implemented with OmpSs-2 applies the `task for` feature (discussed in Section 2.4). As this strategy shares the same structure as the previous one, based on Algorithm 3, the only difference is the addition of the following directive in line 12: `task for chunksize(`oss_{bsy}`)`. Taskfors are work-sharing tasks that behave similarly to regular tasks. A single thread executes a regular task; nonetheless, a `taskfor` can be executed concurrently by several threads. Its synchronization is handled through data dependencies, without the need for explicit barriers. Its iteration space is partitioned into chunks, the size of which is defined by users via directive `chunksize(size)`. However, the difference is that creating these chunks does not imply an overhead such as the one associated with tasks, as it does not need an extra allocation or specific management. Threads obtain the boundaries and data environment to work with it. In summary, taskfors can be run concurrently by multiple threads without paying an extra overhead for their management.

3.4 ASOC: Automatic Selection of Optimal Configurations

Finding an optimized configuration (implementation and parameters) that delivers the best outcome in performance would require a huge design space exploration. In this section – and with the previously used terminology for parameters and variables in Algorithm 3 – we give an in-depth description of how ASOC obtains these ideal configurations. To highlight the importance of the chosen configuration, we have evaluated the performance and energy consumption obtained when using different configurations for each implementation (`oss taskloop` and `oss task for`): oss_{bsx}, oss_{bsy},



a) *4000 x 4000 x 240* b) *3000 x 3000 x 1024*

Figure 4: Performance and Energy results when different parameters and parallelization schemes are considered for two different input sets ($m \times n \times k$) on the Intel 44-core system

oss_{nc} – where n is divided by 1, 2, 4, 8, 16, 32, and 64 – and oss_{mc} – from 18 to 72 in steps of 6, as it must be a number divisible by 6 due to the micro-kernel organization. In total, 3,430 configurations have been evaluated for each input set.

In Figure 4 we showcase the results of two different input sets on the Intel 44-core system (described in Section 4). Blue squares represent executions with different parameters of the taskloop variation, while the orange triangles represent the taskfor version. As observed, the outcomes in performance (*x-axis*) and energy consumption (*y-axis*) are strongly affected by the chosen configuration. On top of that, there is not a unique configuration that delivers the best result for different inputs. For instance, the configuration which yields the lowest energy consumption and the highest performance for an input such as $4000 \times 4000 \times 240$ is 12% worse in performance for an input such as $3000 \times 3000 \times 1024$.

As observed, finding adequate task granularities becomes a critical objective to yield competitive performance. To automatically and transparently select the ideal implementation (`oss taskloop` or `oss task for`) and their respective parameters (oss_{mc}, oss_{nc}, oss_{bsx}, and oss_{bsy}), we integrated a heuristic within our BLIS sandbox. At runtime, and without the need for a previous training phase, ASOC finds an optimized set of parameters for an application by considering the dimensions of input matrices (m, n, and k) and features of the target architecture, such as the number of cores and memory hierarchy.

To define the parameters through the experiments performed in the design space exploration, we take the following design choices into account: (*i*) To overcome the overhead of managing and distributing the workload among the tasks, the minimum number of iterations assigned to each created task in the j_c loop is 64. If n is smaller than this number, the fifth loop is not parallelized; (*ii*) Creating more than $\frac{totalCores}{4}$ tasks in the j_c loop will increase the total amount of tasks created in the i_c loop, diminishing the workload size assigned to each task and increasing the overhead to manage tasks; (*iii*) Setting the granularity of loop j_c to 1 ($oss_{bsx} = 1$) can deliver better results, as it is associated to the number of tasks that will be created in loop i_c; (*iv*) When defining the oss_{mc} parameter for the i_c loop, the m dimension plays an important role. Thus, when m is smaller than *512*, it is better to set $oss_{mc} = \frac{mc}{2}$. Otherwise, it is better to keep the value defined by BLIS, as it is based on the structure of the optimized micro-kernel.

Algorithm 4 Automatic Selection of Optimal Configurations

```
1:  oss_nc ← n_c                          ▷ block size assigned to each task in the j_c loop
2:  oss_mc ← m_c                          ▷ block size assigned to each task in the i_c loop
3:  oss_bsx ← 1                           ▷ granularity of each task in the j_c loop
4:  oss_bsy ← 1                           ▷ granularity of each task in the i_c loop
5:
6:  m ≥ 512 ? oss_mc = m_c : oss_mc = m_c/2          ▷ defining oss_mc
7:
8:  for i = TotalCores/4, ...1 in steps of −1 do     ▷ defining oss_nc
9:      temp = n/i
10:     τ = size(C_oss_mc,temp + A_oss_mc,kc + B_kc,temp)
11:     if τ fits in L3_cache then
12:         oss_nc = temp
13:         tasks_jc = n/oss_nc
14:     end if
15: end for
16:
17: β = m/(oss_mc × oss_bsy)              ▷ calculating total number of tasks
18: if mod(m, β) > 0 then β = β + 1
19: end if
20: oss_tasks = β * tasks_jc              ▷ total number of tasks
21: oss_tasks' = mod(oss_tasks, oss_nc)
22:
23: for Ω = 1, ..., m/(oss_mc × Ω) in steps of 1 do   ▷ refining oss_bsy
24:     t = m/(oss_mc × Ω)
25:     mod(m, t) == 0 ? β = t : β = t + 1
26:     φ = mod(β × tasks_jc, oss_tasks)
27:     if φ > oss_tasks/2 then
28:         φ = oss_tasks − φ
29:     end if
30:     if φ ≤ oss_tasks' then
31:         oss_tasks = β × tasks_jc          ▷ total number of tasks
32:         oss_tasks' = φ
33:         oss_bsy = Ω                       ▷ new value for oss_bsy
34:     end if
35: end for
```

The procedure is described in Algorithm 4. It starts assigning default values to parameters based on the BLIS implementation (lines *1-4*). Afterwards, the first parameter to be defined is oss_{mc} following the design choice (*iv*) discussed above. It is the first to be defined because it is used to get the size of panels for matrices A and C when calculating the oss_{nc} value. Secondly, oss_{nc} is defined (lines *8-15*), playing an essential role in the distribution of elements from C and B to the tasks. The higher this value, the fewer tasks are created and the greater the workload assigned to each task becomes. Hence, it aims to create tasks so that each task's memory (τ) for the elements of matrices used within the p_c loop fits in the L3 cache. We consider this loop to ensure that tasks created in the i_c loop compute over data already loaded into shared cache levels. In the end, the number of tasks that will be created ($tasks_{jc}$) is defined by dividing the n dimension by the value of oss_{nc}.

Next, we compute the number of tasks created in the i_c loop, represented by β. It is worth mentioning that oss_{tasks} means the total number of tasks that will be created. β is initially defined w.r.t the standard value of oss_{bsy} (lines 17-21). Then, the operations performed between lines 23 and 35 refine oss_{bsy}, using Ω as its temporary value. Our heuristic aims at finding a value for oss_{bsy} in which the number of created tasks is a multiple of the available number of cores, or at least a close number to that, to ensure optimal workload balancing. Finally, according to the number of times the loop i_c will be executed, we define if the strategy to be used is either `task loop` or `task for`. If data is to be re-used, `task for` is selected due to its data re-using features (as discussed in Section 2.4). Otherwise, `task loop` is selected.

Table 1: Configurations of Input Matrices

	Input Size	
	m and n	***k***
Case I	2000, 3000, ..., 14400	240
Case II	14400	200, 300, ..., 1000
Case III	2000, 3000, ..., 10000	2048
Case IV	10240	1000, 1200, ..., 4000

Table 2: Main characteristics of each multicore architecture

	Intel Xeon E5-2699 v4	**AMD EPYC 7742**
Microarchitecture	Broadwell	Zen2
#Physical Cores	44(22+22)	64
Base Oper. Freq.	2.2GHz	2.25GHz
L1 Data Cache	32KB	32KB
L2 Cache	256KB	512KB
L3 Cache (total)	110MB	256MB
Main memory	256GB	1024GB
Node name	Intel-Xeon	AMD-EPYC

Table 3: Evaluated Configurations

Config.	**Description**
BLIS omp-fj	GEMM with the highly optimized OpenMP fork-join model implemented by BLIS, where the number of threads matches the number of physical cores, thread affinity set to close, and thread placement to cores
BLIS pt-fj	Optimized BLIS implementation of GEMM with PThreads
omp-tasks	Generic task-based implementation of GEMM with OpenMP (as described in Section 3.1)
oss	Generic task-based implementation of GEMM with OmpSs-2 (as described in Section 3.1)
oss-loop	OmpSs-2 Task Loop implementation with parameters found by the design space exploration
oss-for	OmpSs-2 Task For implementation, with the best parameters found by the design space exploration
oss-asoc	Results achieved when the automatic selection of optimal configurations proposed in Section 3.4 is applied
AMD Aocl	Vendor AMD Optimizing CPU Library
Intel MKL	Vendor Intel Math Kernel Library

4 EVALUATION

4.1 Methodology

Input Matrices. Following the methodology used to validate the standard OpenMP fork-join parallel implementation of BLIS [29], we consider different test cases, as depicted in Table 1. Furthermore, to keep the compatibility of data among executions, the matrices are generated by BLIS routines.

Execution environment. We evaluate the configurations described in Table 3. We performed the experiments on two modern HPC systems, as depicted in Table 2. We used Ubuntu (Kernel v. 5.11) on all the machines. The following methodology was employed to reduce noise in executions: (*i*) CPU frequencies were set to their base operating value, (*ii*) the executions consider a warm-up step (a single execution of the GEMM routine) to avoid any cache and memory influence on the measured performance, and (*iii*) we consider the average of 50 executions for each configuration (input set, implementation, and target machine) with a standard deviation lower than 0.5%. The applications were compiled with the LLVM compiler infrastructure (v. 12.0, using the *-O3* flag). To measure the performance of only the GEMM kernel, without the initialization and termination time of the matrices and warm-up, we used the *omp_get_wtime()* function. As for the energy consumption, we used the Running Average Power Limit library, which obtains the energy of entire CPU packages and DRAM modules [12].

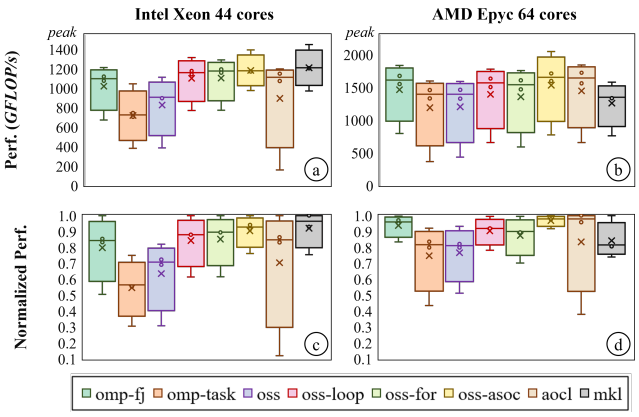


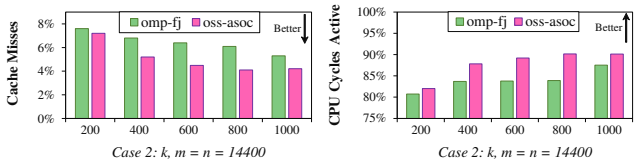
Figure 5: Performance results for all configurations and input cases: (a) and (b) show the GFLOP/s raw numbers while (c) and (d) present the performance normalized to the best result for each test case (↑ values = ↑ performance).

4.2 Performance Evaluation

In this section, we compare the performance results of each parallelization scheme running with the input cases on the target processors, as described in Section 4.1. For that, they are organized as follows: Figures 5 (a) and (b) depict the distribution of raw performance numbers (*GFLOPS/Sec*) for all input cases for Intel and AMD systems respectively. These figures highlight each version's average, maximum, minimum, and median values (represented by each box), so the higher the values, the better. Both plots also show the theoretical peak performance for each architecture. Similarly, Figures 5 (c) and (d) show the distribution of the results normalized to the best performance achieved on each test case – the closer the value to 1.0, the better the performance[2].

We start by highlighting that generic task-based implementations (**omp-tasks** and **oss**) cannot deliver competitive performance when compared to **omp-fj**. The main reason for this behavior is the lack of parallelism exploitation, as only one loop is parallelized due to the model's limitation that does not offer advanced dependency managing features (as described in Section 3.1). On the other hand, the task-based implementations that apply advanced dependency features (**oss-loop** and **oss-for**) can outperform the optimized OpenMP fork-join implementation. On top of that, it is worth noting that for some test cases, **oss-loop** achieves better results than **oss-for** (e.g., on AMD Epyc), while for other cases, **oss-for** is better (e.g., on the Intel Xeon). This scenario highlights the need for an

[2]It is worth mentioning that for the fork-join versions, we only show the results of **omp-fj**, as it delivered better performance than **pt-fj** since BLIS does not implement any thread affinity for the **pt-fj** version. In this scenario, the number of context switches and data movements – as threads are not pinned to a given core during the entire execution – played an essential role in the performance of **pt-fj**.



(a) Cache Misses **(b) CPU Cycles Active**

Figure 6: CPU/Memory usage of omp-fj and oss-asoc

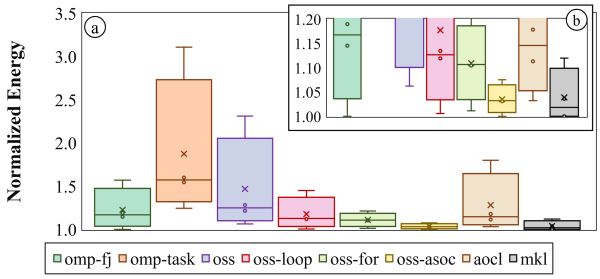


Figure 7: (a) Distribution of energy normalized to the best result of each input case for all test cases on the Intel Xeon processor (↓ values = ↑ energy efficiency). (b) Zoomed-in view of Fig. 7a.

automatic selection of optimized configurations. Hence, as **oss-asoc** automatically defines an optimized parallelization scheme and parameters to run the GEMM routine, in general, it can deliver better performance than the **omp-fj** and all the other task-based versions. When considering the average of all experiments, the **oss-asoc** configuration achieved 11% and 4% better performance than **omp-fj** on Intel Xeon and AMD Epyc, respectively.

Let us now compare the results achieved by the vendor optimized implementations of GEMM (**aocl** and **mkl**). For the AMD Epyc processor, **oss-asoc** can deliver better performance than the **aocl** version as **aocl** is based on the **omp-fj** implementation and therefore suffers from the very same drawbacks of fork-join models. Even though it implements dynamic concurrency throttling to optimize the number of running threads, the lack of malleability of the runtime system limits the performance improvements. Furthermore, the **mkl** version presented the worst results as it is not optimized for AMD processors. On average of all test cases, **oss-asoc** was 3% and 20% better than the **aocl** and **mkl** versions, respectively. As for the Intel Xeon processor, the **oss-asoc** implementation reached competitive levels of performance when compared to the highly optimized **mkl** version. On average of all test cases, **oss-asoc** achieved only 1.5% less performance than **mkl** and 10.5% more performance than the **aocl** version.

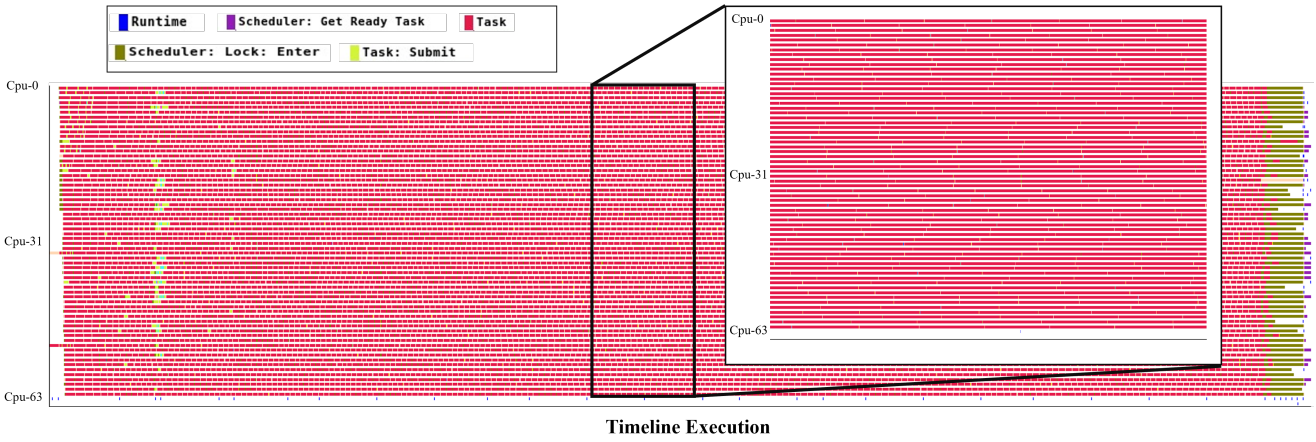
While **oss-asoc** achieves overall performance levels closer to the theoretical peak, it also presents the lowest variability on normalized performance, as observed in Figures 5 (c) and (d). In other words, the **oss-asoc** implementation can reach the best performance, or at least get close to it, regardless of the test case. This behavior highlights the ability of our optimized GEMM task-based version to adapt itself according to the input set and microarchitecture at hand. As a result of this malleability provided by running an optimized task-based configuration (*oss-asoc*), hardware resources such as cache memories and processing units are better utilized. Figure 6 illustrates this by considering the behavior of both **omp-fj** and **oss-asoc** when running with *Case 2* input set on the Intel Xeon processor. It shows the cache miss ratio and the percentage of CPU cycles active during execution (the number of cycles where the CPU is doing useful computation). As observed, **oss-asoc** significantly decreases the number of cache misses due to the immediate successor scheduling policy. As soon as a CPU finishes executing a task, if a successor task can be executed, it is done immediately to benefit from data re-use in the cache memories. Similarly, since the optimized task-based version does not use barriers, there is no implicit overhead due to synchronization. Thus, the outcome is a higher number of active CPU cycles.

Even though the **oss-asoc** execution can deliver better performance than the **omp-fj**, **mkl**, and **aocl** versions in most cases, there are specific scenarios where the optimized fork-join versions were better due to the overhead of the OmpSs-2 runtime system in managing the tasks and workload distribution. In such scenarios (e.g., *Case 1, m=n=2000* on Intel Xeon and *Case IV, k=3000* on AMD Epyc), the **oss-asoc** configuration was unable to deliver competitive performance results. To demonstrate this scenario, we illustrate in Figure 8 execution timeline traces for three different configurations: (*i*) the most common across all configurations, a balanced workload (top) with a zoomed-in view showcasing the lack of gaps – thus the high parallelism available –, (*ii*) the previously mentioned under-performing scenario (*Case IV, k=3000* on AMD Epyc), where there is a lack of parallelism (bottom left), and (*iii*) the execution of multiple kernels of the previous under-performing scenario at the same time. It is important to note that these three traces highlight different executions, as the lack of parallelism in (b) is shadowed by the execution of multiple chained GEMM kernels. Thus, the timelines of all these traces are completely different.

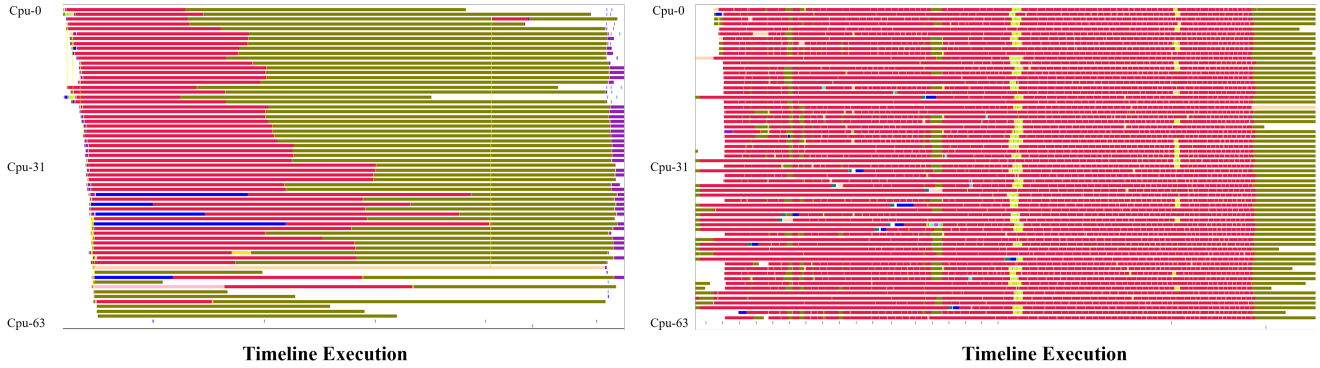
As shown, for most configurations, the trace presents no gaps, which coupled with the previous CPU utilization plots demonstrates the benefits from our implementations which combine both tasks and work-sharing. Nevertheless, there are scenarios (bottom left trace) in which a lack of parallelism causes the execution trace to be unbalanced. Furthermore, the evident lack of workload also causes the time to manage the tasks to be greater than their execution. Both these drawbacks lead to an undesired reduction in performance. However, this is an unlikely scenario in task-based applications, as other tasks could fill the gaps on the GEMM kernel. To illustrate this behavior, we simultaneously executed fifteen micro-kernels with the same configuration (bottom right) to exemplify a more realistic scenario. As shown at the beginning, the runtime exposes a high amount of parallelism, and the time to manage tasks becomes minimal in comparison. Due to this, the execution remains as balanced as our previous experiment (top).

4.3 Energy Consumption Evaluation

In this section, we compare the energy consumption of the implementations and scenarios discussed in Section 4.1 on the Intel



(a) Balanced execution of GEMM



(b) Lack of parallelism and unbalanced execution of GEMM (left) and Simultaneous execution of multiple GEMM (right)

Figure 8: Execution traces exemplifying a balanced and an unbalanced workload

Xeon processor[3]. We illustrate in Figure 7 the distribution of energy results for each configuration, normalized to the best result on each test case – the closer the values are to 1.0, the lower the energy consumption. The outcome of offering malleability to the GEMM routine so it can adapt to the execution environment at hand and make better use of the hardware resource (as discussed in Section 4.2) is a significant reduction in energy consumption. When considering the task-based versions with advanced features (**oss-loop** and **oss-for**), the energy reductions vary according to the input set due to the intrinsic characteristics of each feature (as discussed in Section 3). Therefore, there are cases where the **oss-loop** execution delivers better results (e.g., *Case II, k=200*) and others where the **oss-for** is better (e.g., *Case II, k=1000*), reinforcing the need of the **oss-asoc** configuration. Therefore, as the **oss-asoc** configuration can cover a wider range of possibilities (parameters and implementation) than the **oss-loop** and **oss-for** simultaneously, it can deliver better energy consumption.

[3]As for the AMD Epyc system, we were unable to obtain such readings, as user permissions in this machine disabled some hardware counter readings.

4.4 Cholesky Decomposition Use-case

In this Section, we use the Cholesky decomposition algorithm to illustrate the benefits of our optimized task-based GEMM kernel. To that end, we have implemented several versions of this algorithm using OpenMP (**omp**) and OmpSs-2 (**oss**) tasks. The first two versions (**omp+blis-seq** and **oss+blis-seq**) use the traditional task-based parallelization strategy [10] where each task executes a sequential BLAS kernel. A well-known problem of this approach is that few tasks are available at the end of the execution. Thus, not all available cores are exploited on large multi-core systems. We have implemented three additional versions that rely on the previous task-based parallelization strategy to address this issue, which use parallel BLAS kernels instead. On the **omp+blis-omp-fj** version the BLAS kernels are parallelized using the OpenMP fork-join model, while the (**omp+blis-omp-tasks**), **oss+blis-oss-asoc**, and **oss+blis-oss-dse** versions use OpenMP tasks and OmpSs-2 tasks with the ASOC heuristic, respectively. We have executed each version with seventeen distinct input sets ranging from $m = n = k = 2048$ to $m = n = k = 65536$ on the AMD Epyc multicore architecture. Moreover, for each input set, we evaluated different

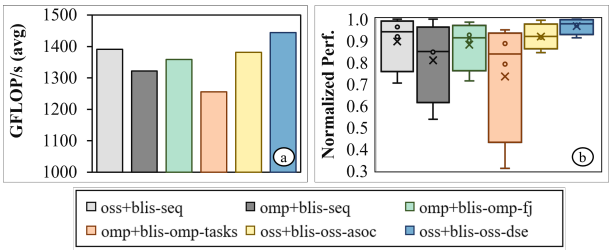


Figure 9: Results for the Cholesky evaluation: (a) average for all input sets; (b) distribution of the perf. normalized to the best result for all inputs; (↑ values = ↑ performance).

blocksizes from 64 to $\frac{n}{2}$ and discuss next the best performance found by each configuration on each input set.

Figure 9 depicts the (a) average performance of each version represented by the *Cholesky + BLIS GEMM implementation* w.r.t. all input sets and (b) the distribution of the results normalized to the best performance achieved on each input set. We also depicted in the same Figure the best results found by an exhaustive search that tries all possible combinations of *blocksize*, parameters to the GEMM routines, and the number of task granularity (**oss+blis-oss-dse**). As observed, when Cholesky uses the **oss-asoc** version, it can deliver better overall performance than the OpenMP fork-join and task-based versions. On top of that, it reaches performance levels close to the ones achieved by the **oss+blis-oss-dse** configuration. Although it shows that there is still room for enhancements on the proposed heuristic, it is essential to highlight that (**oss+blis-oss-asoc**) works at runtime and with no previous training phase nor information from the software developer.

In Figure 10 we show the execution trace of the **oss+blis-seq** (top) and the **oss+blis-oss-asoc** (bottom) versions with a $m = n = k = 8192$ problem size. The goal of this figure is to show when an application can take advantage of our GEMM task-based implementation. For that, we have chosen a configuration from the samples averaged in Figure 9 that does not exploit the GEMM kernel's task-level parallelism. These traces, which are on the same time scale, clearly illustrate the benefits of our optimized task-based GEMM kernel. The **oss-asoc** heuristic can find good configurations for the GEMM kernels and the runtime system can leverage the additional sub-tasks created by different BLIS kernels (e.g., syrk, trsm, and GEMM) to exploit all cores during the whole execution, which significantly reduces the execution time.

5 RELATED WORK

Many works have previously proposed optimizations of GEMM operations. For instance, Tan et al. [30] propose a performance model to optimize the performance of the GEMM operation in a Fermi GPU architecture. Similarly, Heinecke et al. [13] optimize GEMM routines for native and hybrid execution in the Intel Xeon Phi. Kurzak et al. [18] provide implementations of the QR, LU, and Cholesky algorithms but do not propose optimizations for the GEMM. The works of Agullo et al. [5, 6] focus on implementing the Cholesky algorithm on heterogeneous and distributed systems, which deviates from our approach. Benson and Ballard [8] propose

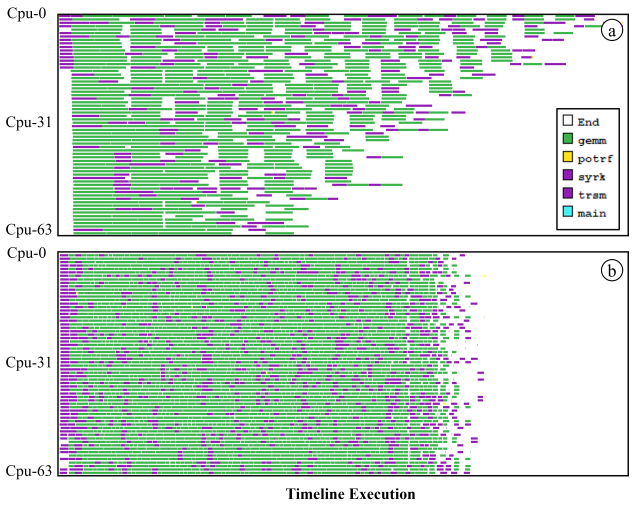


Figure 10: Execution traces demonstrating a better use of the HW resources by (b) *oss+blis-oss-asoc* compared to the (a) *oss+blis-seq*

a code generation tool to automatically implement versions of matrix multiplication algorithms with the OpenMP fork-join model. CompDGEMM [15] is an optimized OpenBLAS DGEMM routine for 64-bit ARMv8 architectures. Jiang et al. [16] propose a three-level blocking DGEMM algorithm to improve data-locality in the Sunway TaihuLight supercomputer. Lim et al. [19] optimize a DGEMM OpenMP fork-join version by choosing the proper block size and thread affinity to the Intel Xeon Phi. Abdelfattah et al. [4] propose HGEMM to improve the performance in GPU Tensor Cores. SLATE [11] is a PBLAS implementation to improve the performance in accelerators and distributed systems. BiQGEMM [14] is a GEMM routine for quantized neural networks that optimize the performance through removing computation redundancy. FLASH [23] is a framework that applies an analytical model to optimize the tile size and data movement of the GEMM kernel in accelerators. Park et al. [26] present a study of optimizations for the PDGEMM routine, by applying thread affinity/placement, parallelization scheme, and data blocking using AVX-512 instructions on Intel-based distributed systems.

Our Contributions. Compared to the works that propose different strategies for optimizing the GEMM operation [4, 8, 11, 13–17, 19, 23, 26, 30], our work is the first to present highly optimized task-based implementations of the GEMM routine. On top of that, although distinct works have evaluated the execution of task-based versions of the GEMM routine [22, 24, 28, 31, 32], none of them (*i*) implement a highly optimized task-based version; and (*ii*) propose a heuristic to select the best parallelization scheme and parameters; as we do in this work.

6 CONCLUSIONS AND FUTURE WORK

We have presented a task-based implementation of the GEMM kernel that task-based applications can seamlessly leverage. We exploited several advanced OmpSs-2 features to minimize runtime

overhead and improve load-balancing and data locality. As the parameters and implementation that deliver the best performance change according to the input size of the matrices, we also proposed a heuristic to select the best parallelization scheme and parameters at runtime automatically. Through an extensive set of experiments, we have shown that our optimized task-based implementation delivers better performance and reduces energy consumption compared to the optimized BLIS OpenMP fork-join GEMM implementation. Furthermore, our implementation can deliver better performance than the optimized vendor implementations of GEMM (e.g., Intel MKL and AMD AOCL). Lastly, we have demonstrated that real applications can enhance their performance by leveraging our optimized task-based implementation. We intend to implement optimized task-based versions of other linear algebra routines as future work.

REFERENCES

[1] 2009. *Intel Math Kernel Library. Reference Manual.* Intel Corporation, Santa Clara, USA. ISBN 630813-054US.

[2] 2012. *AMD Core Math Library (ACML) User Guide.* Advanced Micro Systems (AMD), Santa Ana, USA. https://developer.amd.com/wordpress/media/2012/10/acml_userguide.pdf\.pdf

[3] 2021. *AMD Optimizing CPU Libraries User Guide.* Advanced Micro Systems (AMD), Santa Ana, USA. https://developer.amd.com/wp-content/resources/AOCL_User%20Guide_3.0.pdf/

[4] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. 2019. Fast batched matrix multiplication for small sizes using half-precision arithmetic on gpus. In *IEEE IPDPS*. IEEE, 111–122.

[5] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. 2010. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs.

[6] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. 2017. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems* (2017), 1–1. https://doi.org/10.1109/TPDS.2017.2766064

[7] David Álvarez, Kevin Sala, Marcos Maroñas, Aleix Roca, and Vincenç Beltran. 2021. Advanced Synchronization Techniques for Task-Based Runtime Systems. In *ACM SIGPLAN* (Virtual Event, Republic of Korea) *(PPoPP '21)*. ACM, New York, NY, USA, 334–347. https://doi.org/10.1145/3437801.3441601

[8] Austin R Benson and Grey Ballard. 2015. A framework for practical parallel fast matrix multiplication. *ACM SIGPLAN Notices* 50, 8 (2015), 42–53.

[9] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 16, 1 (March 1990), 1–17. https://doi.org/10.1145/77626.79170

[10] Joseph Dorris, Jakub Kurzak, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. 2016. Task-Based Cholesky Decomposition on Knights Corner Using OpenMP. In *High Performance Computing*, Michela Taufer, Bernd Mohr, and Julian M. Kunkel (Eds.). Springer International Publishing, Cham, 544–562.

[11] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. Slate: Design of a modern distributed and accelerated linear algebra library. In *Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. 1–18.

[12] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Performance Evaluation Rev.* 40, 3 (2012), 13–17.

[13] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G Shet, George Chrysos, and Pradeep Dubey. 2013. Design and implementation of the linpack benchmark for single and multi-node systems based on intel® xeon phi coprocessor. In *IEEE IPDPS*. IEEE, 126–137.

[14] Yongkweon Jeon, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Jeongin Yun, and Dongsoo Lee. 2020. BiQGEMM: matrix multiplication with lookup table for binary-coding-based quantized DNNs. In *SC20: Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[15] Hao Jiang, Feng Wang, Kuan Li, Canqun Yang, Kejia Zhao, and Chun Huang. 2015. Implementation of an accurate and efficient compensated dgemm for 64-bit armv8 multi-core processors. In *IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*. IEEE, 491–498.

[16] Lijuan Jiang, Chao Yang, Yulong Ao, Wanwang Yin, Wenjing Ma, Qiao Sun, Fangfang Liu, Rongfen Lin, and Peng Zhang. 2017. Towards highly efficient DGEMM on the emerging SW26010 many-core processor. In *Int. Conf. on Parallel Processing (ICPP)*. IEEE, 422–431.

[17] Raehyun Kim, Jaeyoung Choi, and Myungho Lee. 2019. Optimizing parallel GEMM routines using auto-tuning with Intel AVX-512. In *Int. Conf. on High Performance Computing in Asia-Pacific Region*. 101–110.

[18] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. 2010. Scheduling Dense Linear Algebra Operations on Multicore Processors. *Concurr. Comput.: Pract. Exper.* 22, 1 (jan 2010), 15–44.

[19] Roktaek Lim, Yeongha Lee, Raehyun Kim, and Jaeyoung Choi. 2018. OpenMP-based parallel implementation of matrix-matrix multiplication on the intel knights landing. In *Workshops of HPC Asia*. 63–66.

[20] Marcos Maronas, Kevin Sala, Sergi Mateo, Eduard Ayguadé, and Vicenç Beltran. 2019. Worksharing Tasks: An Efficient Way to Exploit Irregular and Fine-Grained Loop Parallelism. In *IEEE Int. Conf. on High Performance Computing, Data, and Analytics*. IEEE, 383–394. https://doi.org/10.1109/HiPC.2019.00053

[21] Marcos Maroñas, Xavier Teruel, and Vicenç Beltran. 2021. OpenMP Taskloop Dependences. In *OpenMP: Enabling Massive Node-Level Parallelism*, Simon McIntosh-Smith, Bronis R. de Supinski, and Jannis Klinkenberg (Eds.). Springer Int. Publishing, Cham, 50–64.

[22] Panagiotis D Michailidis and Konstantinos G Margaritis. 2012. Computational comparison of some multi-core programming tools for basic matrix computations. In *IEEE Int. Conf. on High Performance Computing and Communication & IEEe Int. Conf. on Embedded Software and Systems*. IEEE, 143–150.

[23] Gordon E Moon, Hyoukjun Kwon, Geonhwa Jeong, Prasanth Chatarasi, Sivasankaran Rajamanickam, and Tushar Krishna. 2021. Evaluating Spatial Accelerator Architectures with Tiled Matrix-Matrix Multiplication. *arXiv preprint arXiv:2106.10499* (2021).

[24] Gideon Nimako, Ekow J Otoo, and Daniel Ohene-Kwofie. 2012. Fast parallel algorithms for blocked dense matrix multiplication on shared memory architectures. In *Int. Conf. on Algorithms and Architectures for Parallel Processing*. Springer, 443–457.

[25] OpenMP Architecture Review Board. 2018. OpenMP Application Programming Interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf Accessed: 2019-03-24.

[26] Yoosang Park, Raehyun Kim, Thi My Tuyen Nguyen, and Jaeyoung Choi. 2021. Improving blocked matrix-matrix multiplication routine by utilizing AVX-512 instructions on intel knights landing and xeon scalable processors. *Cluster Computing* (2021), 1–11.

[27] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé. 2017. Improving the Integration of Task Nesting and Dependencies in OpenMP. In *IEEE IPDPS*. 809–818. https://doi.org/10.1109/IPDPS.2017.69

[28] Solmaz Salehian, Jiawen Liu, and Yonghong Yan. 2017. Comparison of threading programming models. In *IEEE IPDPSW*. IEEE, 766–774.

[29] Tyler M Smith, Robert Van De Geijn, Mikhail Smelyanskiy, Jeff R Hammond, and Field G Van Zee. 2014. Anatomy of high-performance many-threaded matrix multiplication. In *IEEE IPDPS*. IEEE, 1049–1059.

[30] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast implementation of DGEMM on Fermi GPU. In *Int. Conf. for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[31] Xavier Teruel, Michael Klemm, Kelvin Li, Xavier Martorell, Stephen L Olivier, and Christian Terboven. 2013. A proposal for task-generating loops in OpenMP. In *Int. Workshop on OpenMP*. Springer, 1–14.

[32] Pedro Valero-Lara, Ivan Martinez-Perez, Sergi Mateo, Raül Sirvent, Vicenç Beltran, Xavier Martorell, and Jesús Labarta. 2018. Variable batched DGEMM. In *Euromicro Int. Conf. on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 363–367.

[33] Field G Van Zee and Robert A Van De Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS)* 41, 3 (2015), 1–33.

[34] Zhang Xianyi, Wang Qian, and Werner Saar. 2021. OpenBLAS: An optimized BLAS library. *URL: http://xianyi. github. io/OpenBLAS* (2021).