CoCoPeLia: Communication-Computation Overlap Prediction for Efficient Linear Algebra on GPUs

Petros Anastasiadis[†], Nikela Papadopoulou[†], Georgios Goumas[†] and Nectarios Koziris[†]
[†]Computing Systems Laboratory, National Technical University of Athens, Greece
Email: {panastas,nikela,goumas,nkoziris}@cslab.ece.ntua.gr

Abstract—**Graphics Processing Units (GPUs) are well established in HPC systems and frequently used to accelerate linear algebra routines. Since data transfers pose a severe bottleneck for GPU offloading, modern GPUs provide the ability to overlap communication with computation by splitting the problem to fine-grained sub-kernels that are executed in a pipelined manner. This optimization is currently underutilized by GPU BLAS libraries, since it requires an approach to select an efficient tiling size, which in turn leads to a challenging problem that needs to consider routine, system, data, and problem-specific characteristics. In this work, we introduce an elaborate 3-way concurrency model for GPU BLAS offload time that considers previously neglected features regarding data access and machine behavior. We then incorporate our model in an automated, end-to-end framework (called CoCoPeLia) that supports overlap prediction, tile selection and effective tile scheduling. We validate our model's efficacy for dgemm, sgemm, and daxpy on two testbeds, with our experimental results showing that it achieves significantly lower prediction error than previous models and provides near-optimal tiling sizes for all problems. We also demonstrate that CoCoPeLia leads to considerable performance improvements compared to the state of the art BLAS routine implementations for GPUs.**

I. Introduction

Dense linear algebra operations appear very frequently in high-performance computing (HPC) applications rendering their performance highly critical for their scalability. The standardization of the Basic Linear Algebra suprograms (BLAS) [1] in the early days of HPC has eased the development of scientific code since domain experts have been relying on standardized and performance-optimized libraries to build more complex simulations at scale. The regular parallelism of BLAS routines makes them a good fit for GPUs, hence the existence of many GPU-BLAS libraries, the most common being *cuBLAS*, a CUDA-like BLAS library for NVIDIA GPUs [2]. *cuBLAS* offers highly optimized primitive BLAS operations, however requires input data to reside on the GPU memory, a task that is left to the programmer.

To ease the programming effort of moving data between the GPU and host memory, NVIDIA introduced unified memory in CUDA 6.0, enabling the GPU to directly access host memory. Despite the improvement in programmability, critical performance issues exist: the limited bandwidth between the host and the GPU imposes a transfer overhead, while the unified memory abstraction can lead to additional performance loss [3]–[5]. A common practice to reduce these overheads is to overlap host-to-device (h2d) and device-to-host (d2h) transfers with computation, known as *3-way-concurrency*. This practice relies on splitting the initial problem data into smaller chunks and offloading it to the GPU in a pipelined manner, allowing the computation on a chunk to be performed while output data from the previous chunk and input data for the next chunk are being transferred. This optimization is usually left to the programmer for level-1 and level-2 BLAS but is integrated into the state-of-art multi-GPU level-3 BLAS libraries like *cuBLASXt* [6], the multi-GPU cuBLAS extension for level-3 BLAS, *NVBLAS* [7], a LAPACK-compatible wrapper for *cuBLASXt*, as well as *BLASx* [8] and *XKBlas* [9], two approaches focusing on reducing avoidable transfers and load-balancing overheads to improve performance. All these libraries are CUDA-based and their input data may reside on host memory, GPU memory or a combination of both.

3-way-concurrency for BLAS on GPUs requires an approach to select the size for the computational chunks, (henceforth *tiling size* T) which is a complex problem [8], since tiling size affects the granularity of computation and communication, together with other parameters like the nature of the routines itself, the data layout, the initial data locations, the problem size, and the underlying architecture. Existing libraries trade performance for programmability or conversely, to simplify the problem of tiling size selection. *cuBLASXt* extends BLAS parameters with the tiling size and assigns its selection to the programmer. This approach cannot guarantee optimal performance unless the programmer tests the library and tunes the tiling size for each specific problem size. *BLASX*, and *NVBLAS*, on the other hand, set the tiling size internally to a static value, presumed to provide an average performance gain across a range of commonly used system characteristics, routine parameters, and problem sizes. While this approach is more user-friendly, it still sacrifices performance for generality. *XKBlas* provides both options through wrapping.

In order to get the best out of both worlds (performance and programmability), we need to incorporate within an optimized GPU BLAS library an adequately accurate performance prediction model for 3-way concurrency that will support effective tile selection. Prior work copes with the problem of 3-way-concurrency modeling [10]–[12], but because of the problem's complexity these models need to make restricting assumptions regarding data location, shape, and kernel execution characteristics deeming them inaccurate or completely inapplicable to BLAS - especially level-3 BLAS. Werkhoven et al. [11] propose the most relevant generic 3-way-concurrency model for the total offload time. Their model takes as input the kernel execution time on the GPU and estimates transfer times with semi-empirical sampling, but does not take into account data reuse and non-linearities in execution time. These restrictions make the model more applicable to lower-level BLAS routines but inaccurate for many scenarios of level-3 BLAS GPU offloading.

In this paper we provide a solution for the 3-way-concurrency optimization problem on GPUs that supports near-optimal offload BLAS performance. To achieve this, we focus on 1) *how to accurately predict the performance of GPU BLAS kernels including 3-way-concurrency* and 2) *how to utilize this prediction to reach near-optimal automated overlap*. Overall, this paper makes the following contributions:

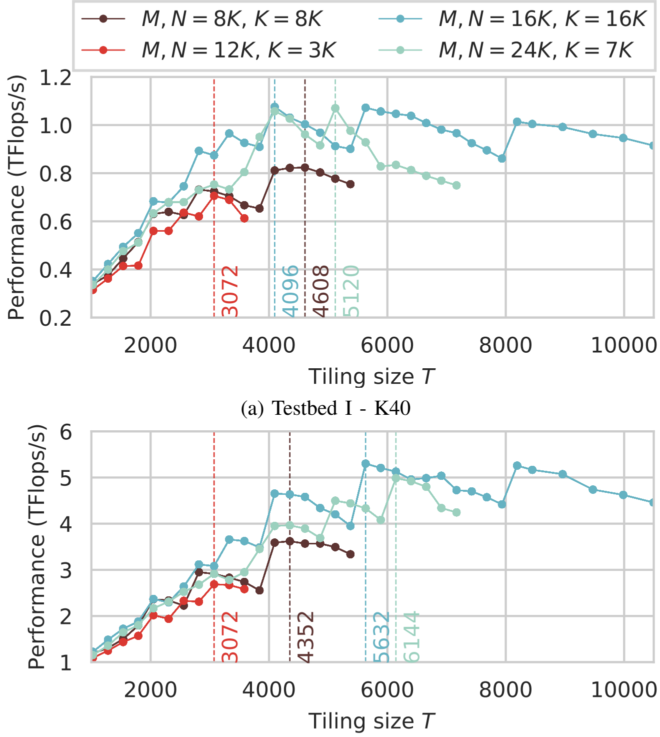
1) It introduces two 3-way-concurrency analytical models for BLAS GPU offload time, for cases with and without data reuse (Section III).
2) It develops an automated empirical methodology to instantiate these models on a system and offers three example models for `daxpy`, `dgemm`, and `sgemm` (Section IV).
3) It combines the above with a runtime tile scheduler into *CoCoPeLia*, an end-to-end GPU BLAS framework utilizing automatic tiling size selection (Section IV), which demonstrates considerable performance improvement over similar state-of-the-art libraries (Section V).

II. Motivation-Background

A. Motivation

To achieve performance gain with 3-way-concurrency, GPU BLAS libraries need to internally split the initial problem size into tiles (more details are provided in Sections III and IV). Most optimized overlap libraries use square tiling, i.e. they split matrices to equal squares $T \times T$, where T is the *tiling size*. Irrespective of the tile distribution mechanism of each library, selecting the appropriate tiling size significantly affects the resulting performance [8], [9], [11], [12]. Figure 1 illustrates the effect of tiling sizes T on performance for `cuBLASXtDgemm` on two testbeds. As the tiling size decreases, the performance increases due to better overlap, but after reaching one or two maxima, it rapidly degrades. These maxima "break-points" vary greatly across the two testbeds and problem sizes.

GPU BLAS libraries follow different strategies for tiling size selection, trading between programming ease and performance. We argue that none of these approaches are generic enough or performance-optimal. As an example, we annotate `dgemm` performance using the static tiling size of $T = 4096$, which offers the *best* average performance for *cuBLASXt* (details in Section V), for the examples in Figure 1; it results in up to 9.4% slowdown on testbed I and up to 14.7% slowdown for testbed II. Furthermore, static tiling sizes offer no performance guarantee for future machines with different transfer bandwidth/computation ratios and can result in increased slowdowns in such cases. These observations make a compelling case for dynamic tiling size selection, driven by accurate performance models.



(a) Testbed I - K40

(b) Testbed II - V100

Fig. 1. `cuBLASXtDgemm` performance on two different testbeds, relative to the tiling size T used for internal 3-way overlap with $T \times T$ tiles, for four different problem sizes (no transpose). The vertical lines outline the tiling size that achieves the best performance for each problem.

B. Related work

1) Models for GPU offloading: A signigicant amount of prior work focuses on modeling the computation time or performance of GPU kernels [13]–[18]. However, these models neglect offload time when data transfers are required before, after or during kernel execution. Gregg et al. [19] first highlighted this problem, arguing against the trend of scientific reporting of the performance of GPU applications, transfers overhead excluded, and propose a taxonomy for data transfers and their impact on offload performance. Numerous later works model CPU-GPU transfers, using variances of the linear latency-bandwidth model for PCIe transfers [20]–[23].

When modeling the serial offload scenario, including transfers improves prediction accuracy. However, if there is communication/computation overlap, simplistic models fail to predict the actual performance, therefore modeling overlap areas is a crucial step in performance modeling [24]. Towards this direction, Gómez-Luna et al. [10] explore the use of CUDA streams for 3-way concurrency, but they consider the stream creation time as the only overlap overhead. Werkhoven et al. [11] enhance this work by offering multiple performance models for communication/computation overlap for various common offload scenarios (RMA, 2-way, 3-way), introduce stream transfer overlap latency, and provide methods to obtain the optimal number of CUDA streams for a given problem. Their models offer high accuracy, however, their modeling approach does not capture all problem characteristics present in BLAS (details in Section III). Liu et al. [12] offer a mathematical framework for software pipelining on GPUs, using non-equal tiles and focus on partitioning, scheduling and granularity. However, they target problems defined by linear functions and with equal input/output bytes, which does not apply to most BLAS routines.

2) GPU BLAS tiling libraries: When considering GPU BLAS offloading, there is limited interest in level-1 and level-2 routines, which are usually heavily transfer bound and offloading them individually is inefficient. On the contrary, multiple level-3 BLAS libraries internally use tiling for better cache and memory utilization and to enable task parallelism [25]–[29]. Among these, PaRSEC [29] is the most efficient for multi-GPU execution, however, it is not compatible with the legacy LAPACK layout and also suffers from high scheduling overheads in small problems. Some existing libraries are specifically designed for level-3 BLAS multi-GPU usage [6]–[9]. cuBLASXt, and its wrapper NVBLAS [6], [7], the state-of-practice libraries introduced by NVIDIA for automatic GPU offloading, do not account for data reuse and leave tiling size tuning to the user. *BLASX* [8] overcomes the first problem with a runtime tile management engine, significantly decreasing the transfer volume and thus increasing performance. The tiling size in this case is static and selected at compile time, to provide good average performance. XKBLAS [9] focuses on problems that perform multiple subsequent BLAS invocations, like iterative solvers, and provides a library which further limits transfers, by taking into account intermediate data locations, but offers no insights regarding tiling size selection. In this work, we propose a performance-aware static workload distribution based on problem-specific tiling size selection, achieved through a prediction model for 3-way overlap offload time prediction and incorporate our method in an end-to-end framework, which offers optimized BLAS offloading through runtime tiling size prediction.

III. Models for GPU BLAS overlap

This section covers the core of this work, the CoCoPeLia 3-way concurrency prediction models. We first discuss 3-way concurrency on BLAS. Then, we build upon a baseline model and propose adjustments to improve prediction accuracy.

Table I describes the notation used throughout this Section, split in two categories; routine-specific (e.g. for a single `gemm` problem) and data specific (e.g. A,B,C - the matrices of the `gemm` routine) values. Certain parameters (e.g. opd, $dtype$) are inferred directly from the BLAS standard, others (e.g. $D1$, $D2$, $D3$) are problem-specific, while others are a combination of both (e.g. get_i, set_i, $S1_i$, $S2_i$). We provide details on obtaining or instantiating these parameters in Section IV. T is the optimization target parameter, namely the tiling size, and the formulas for k, k_{in} are defined later in this section.

A. BLAS overlap characteristics

In CUDA, concurrency is a term referring to software pipelining, in order to overlap CPU-GPU communication with GPU computation, for any problem that can be split in parts without complex dependencies. We are interested in 3-way concurrency for BLAS, where the problem is split into subkernels, each of which requires 1) host-to-device (h2d) transfers for its input, 2) computation on the GPU and 3) device-to-host (d2h) transfers of its output. The three steps must be executed serially for each subkernel, but can be overlapped with the steps of previous and following subkernels. 3-way concurrency execution time prediction relies on modeling computation time, transfer time and computation/communication overlap, with the latter being a highly challenging problem. In order to simplify this problem, overlap models often make a number of assumptions, outlined below.

TABLE I
BLAS modeling notation

$D1[, D2[, D3]$	routine problem size dimensions
$dtype$	routine datatype
k	the number of subproblems after tiling
k_{in}	the number of subproblems with input after tiling
T	tiling size
opd	number of input/output data structures
Per data structure:	$i : 0 \rightarrow opd$
get_i	flag to denote if data requires transferring from the host to the GPU
set_i	flag to denote if data requires transferring from the GPU to the host
$S1_i$, $S2_i$	initial dimensions, extracted from the routine problem size dimensions

1) Non-linear kernel execution times: Previous approaches make the assumption that if a problem is split in k subproblems and these are executed sequentially on the GPU, the total execution time will not change considerably, which does not hold for most BLAS routines. First, BLAS operations have internal dimensions and dependencies, which, in the case of tiling, may require additional reduction operations, while not necessarily producing output that requires transfers. Second, the performance of level-2/3 BLAS kernels does not depend linearly on their working set [26], [30], [31], since the problem shape (e.g. square vs fat-by-thin matrix multiplications) influences performance. Third, if a subproblem becomes too small, the GPU is underutilized and performance drops.

2) Data location awareness: Previous work [10]–[12] assumes that all data is initially resident on the memory of the host CPU, and only targets the full offload scenario. However, a common scenario for BLAS execution is for a kernel to be executed iteratively. In this case, some of the data may still be resident on the GPU [9].

3) Bidirectional overlap: 3-way concurrency goes beyond simple communication/computation overlap, considering also bidirectional host-device overlap (h2d with d2h transfers). While modern GPUs have virtually separate copy engines for h2d and d2h, both engines utilize the same communication medium and therefore simultaneous usage imposes a slowdown [11], [12]. This slowdown is asymmetric; usually the d2h transfers are more heavily affected, but the extent of this effect depends on the underlying interconnect [5].

4) Data reuse: Data reuse refers to the case when in a tiled problem, part of the data required for a subkernel's execution is also needed by subsequent subkernels. In level-3 BLAS,

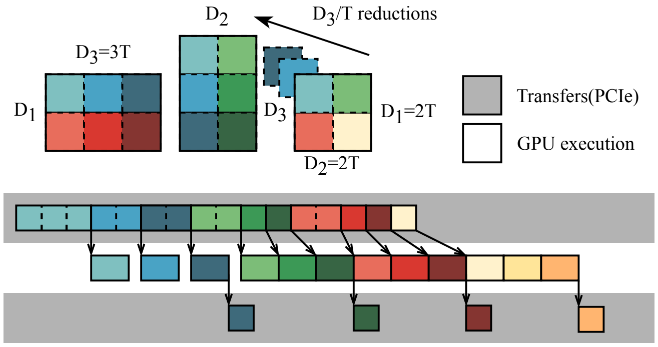


Fig. 2. An example of the 3-way concurrency pipeline for a `gemm` implementation which iterates through the tiles of the M, N, K dimensions, for a problem with $t^T_{h2d} < t^T_{GPU} < 2 \cdot t^T_{h2d}$.

it can be present in all three dimensions of the problem. Level-3 BLAS GPU libraries take advantage of data reuse, to better utilize caches and limit transfers [8], [9]. However, the reuse pattern and overlap percentage depend on the GPU architecture and the BLAS operation, making it difficult to accurately model data reuse.

B. BLAS 3-way concurrency models.

As previous models outline, to enable 3-way concurrency the initial problem is split to k equal parts. We split all problem dimensions, and use an equal tiling size T across them. This applies to level-1 BLAS, as well as level-2 and -3 BLAS square tiling, with the latter being the typical approach in BLAS GPU libraries [6], [8]. Thus, k is given by:

$$k = \frac{D1}{T}[\times \frac{D2}{T}][\times \frac{D3}{T}]$$

where $D2$ ($D2$ and $D3$) applies to level-2 (level-3) BLAS.

In our approach, modeling 3-way concurrency requires knowledge of the execution times for each of the overlapped parts. Since we do not assume a linear relation of the total time with the subkernel execution times, we consider that the total execution time (overlap time) is a function of k and the individual times for h2d transfer t^T_{h2d}, kernel execution t^T_{GPU}, and d2h transfer t^T_{d2h}, for each subproblem with tiling size T that is overlapped. Thus, we consider the following contributions to the total execution time:

$$t^T_{h2d} = f_1(system, dtype, T[, T])$$
$$t^T_{d2h} = f_2(system, dtype, T[, T])$$
$$t^T_{GPU} = f_3(routine, dtype, T[, T[, T]])$$

where t^T_{h2d}, t^T_{d2h} are the system-wide transfer times for a single tile of size T (if it is a vector) or $T \times T$ (if it is a matrix), and t^T_{GPU} is the BLAS routine-specific execution time for a kernel where $D1[= D2[= D3]] = T$. These times are empirically collected in the CoCoPeLia framework (see Section IV).

We assume that in the 3-way-concurrency scenario, each subkernel execution on the GPU is overlapped with 1) the subsequent subkernel input and 2) the previous subkernel output, in a pipelined manner [10]–[12]. Under the assumption that all data initially reside on the CPU and are both input and output data, the 3-way-concurrency execution time for a BLAS routine is then:

$$\begin{aligned} t^{baseline}_{total} &= max(t^T_{GPU}, opd \cdot t^T_{h2d}, opd \cdot t^T_{d2h}) \times (k-1) \\ &+ opd \cdot t^T_{h2d} + t^T_{GPU} + opd \cdot t^T_{d2h} \end{aligned} \qquad (1)$$

1) Data Location Modeling: In practice, Eq. 1 overestimates transfers to and from the GPU; by including the *opd* multiplier, we assume that all data is both input and output data and therefore must be transferred. To avoid this, we use the get_i, set_i flags, which determine which of the *opd* tiles require to be fetched to the GPU or returned to the host. We define t^T_{in} and t^T_{out} as the time required to transfer all tiles for which $get_i = 1$ and $set_i = 1$, respectively, as follows:

$$t^T_{in} = \sum_{i=0}^{opd} get_i \cdot t^T_{h2d} \quad \text{and} \quad t^T_{out} = \sum_{i=0}^{opd} set_i \cdot t^T_{d2h}$$

Following the notion of Eq. 1, the location-aware execution time of a BLAS problem using 3-way concurrency is:

$$\begin{aligned} t^{loc}_{total} &= max(t^T_{GPU}, t^T_{in}, t^T_{out}) \times (k-1) \\ &+ t^T_{in} + t^T_{GPU} + t^T_{out} \end{aligned} \qquad (2)$$

2) Bidirectional Slowdown Modeling: As simultaneous h2d and d2h transfers impose a slowdown on both sides, we define the slowdown factors $sl_{h2d\_bid}, sl_{d2h\_bid}$ for each direction as scaling factors applied to transfer time, in the case the opposite direction is also in use for the whole duration of the transfer. We estimate the slowdown factors empirically in the CoCoPeLia framework (Section IV). The bidirectional transfer time $t^T_{\{in,out\}\_bid}$ of a h2d or d2h transfer, when the opposite link is also in use, then becomes:

$$t^T_{\{in,out\}\_bid} = sl_{\{h2d,d2h\}\_bid} \cdot t^T_{\{in,out\}}$$

However, full bidirectional overlap only applies in practice if $t^T_{in} = t^T_{out}$. In a common case, two simultaneous opposite transfers have different duration, and the total overlap time t^T_{over} is split in two parts; 1) the part during which actual overlap occurs and 2) the single-way transfer of the remaining partially-complete transfer:

$$t^T_{over} = \begin{cases} t^T_{out\_bid} + \frac{t^T_{in\_bid} - t^T_{out\_bid}}{sl_{h2d\_bid}}, & \text{if } t^T_{in\_bid} \geq t^T_{out\_bid} \\ \\ t^T_{in\_bid} + \frac{t^T_{out\_bid} - t^T_{in\_bid}}{sl_{d2h\_bid}}, & \text{otherwise} \end{cases} \qquad (3)$$

The fraction in the equation corresponds to the time required to transfer the remaining part of the longer transfer. Eq. 2 therefore evolves to account for bidirectional overlap as follows:

$$t^{bi}_{total} = max(t^T_{GPU}, t^T_{over}) \times (k-1) + t^T_{in} + t^T_{GPU} + t^T_{out} \qquad (4)$$

3) Data Reuse Modeling: All previous models are not accurate for optimized level-3 BLAS problems, as they do not account for data reuse. Reuse exists in both level-2 BLAS (vector reuse) and level-3 BLAS (matrix reuse), but is mostly relevant to level-3 BLAS performance. Figure 2 shows an example of a level-3 BLAS routine with data reuse. Initially, the problem is transfer-bound. Then, h2d transfers decrease due to data reuse, and the problem becomes execution-bound. The example refers to a specific t^T_{h2d}, t^T_{GPU} ratio, and the amount of reuse and this ratio can significantly alter performance. We construct a generic model, for the ideal reuse case, namely full reuse, where all available tile reuse potential is utilized.

Given the tiling size T, we can compute how many tiles an initial matrix i of dimensions $S1_i$, $S2_i$ is split into, as follows:

$$tiles_i = \frac{S1_i}{T} \cdot \frac{S2_i}{T}, i : 0 \rightarrow opd$$

In level-3 BLAS, we opt to account for the transfer of tiles only once, assuming that they then become available for all subsequent subkernels that use them. Depending on its dependencies and what has already been transferred to the GPU, a subkernel may require the transfer of zero up to three tiles (i.e., the case of the first subkernel).

We compute the number of subkernels among the k total subkernels that require one or two tile transfers, as follows:

$$k_{in} = \sum_{i=0}^{opd} (get_i \cdot tiles_i - 1)$$

For a more optimized implementation, the larger percentage of k_{in} collapses to single tile transfers. We follow this assumption for our final 3-way-concurrency offload time model with reuse, which is given by the following model:

$$t^{re}_{total} = max(t^T_{h2d}, t^T_{GPU}) \cdot k_{in} + t^T_{GPU} \cdot (k - k_{in}) + t^T_{in} + t^T_{out} \tag{5}$$

C. Model overview per level

Level-1 BLAS routines perform vector-vector operations and their working set is $D1 = N$. These transfer bound routines have no working set overlaps and are therefore modeled effectively by Eq. 4.

Level-2 BLAS routines perform matrix-vector operations and have two problem dimensions $D1, D2$ where $D1$ is the output vector length and $D2$ is the remaining dimension of the multiplied matrix. There exists a minor working set overlap among sub-kernels for the vector, but it is relatively small ($D1$) compared to the matrix dimensions ($D1 \times D2$), and therefore Eq. 4 is still sufficient for modeling them.

Level-3 BLAS routines perform matrix-matrix operations and have 3 problem dimensions $D1, D2, D3$ where $D1, D2$ are the output matrix dimensions and $D3$ is the internal matrix multiplication dimension. Although Eq. 4 can provide an estimate for the offload time of many applications, optimized implementations that employ tiling and data reuse lead to higher performance. Since our focus is tiling size selection for state of the art performance, we devise Eq. 5 to account for data reuse in level-3 BLAS.

IV. Putting it all together: The CoCoPeLia *framework*

Having presented elaborate 3-way concurrency prediction models for BLAS routines on GPUs, we now turn our attention on how to utilize them in practice. We present the **CoCoPeLia** framework that handles all the necessary steps including the automatic instantiation of the model for a specific machine and the development of a proof-of-concept library that utilizes the model itself at runtime. Figure 3 shows the complete CoCoPeLia framework. At the heart of the framework lies the Tile selection runtime which employs the prediction models described in detail in Section III. The deployment module feeds the runtime with the proper transfer and execution sub-models (predictors for t^T_{h2d}, t^T_{d2h} and t^T_{GPU}, sl_{h2d_bid}, $sl_{d2h_{b}id}$ – details in Section IV-A) while the library implements an optimized subset of the BLAS prototype on top of basic GPU BLAS kernels (details in Section IV-C). During application execution, when a BLAS routine with a specific set of parameters is invoked for the first time, the CoCoPeLia model is consulted in order to pick the best tiling size.

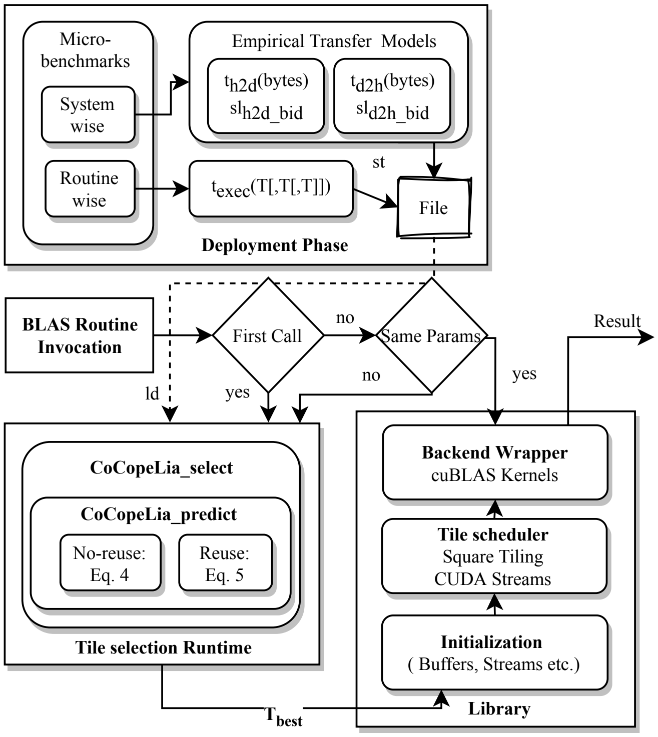


Fig. 3. The CoCoPeLia framework pipeline. During the offline deployment phase the framework performs micro-benchmarks. Then, when a BLAS routine is invoked with some problem parameters for the first time, the tile selection runtime uses them in conjugation with the values obtained during deployment to predict the best tiling size T for this problem. Finally the library is invoked to perform the operation for the given T_{best} and produce the routine result. In case the routine has been called with the same problem parameters before, all unnecessary steps are skipped and the previous tiling scheme and T_{best} is reused.

A. Deployment

To instantiate the models of Section III, we first model the *transfer time* of the target system, with a semi-empirical approach. We perform a set of micro-benchmarks offline and use them to fit the coefficients of basic linear models for transfer time. We use the well-accepted latency/bandwidth model [10]–[12], [21], [30], which estimates transfer time as a function of *bytes*. In our case the latency/bandwidth model for host to device transfers takes the form:

TABLE II
TRANSFER SUB-MODELS FOR THE TWO TESTBEDS.

System						
	t_i	$1/t_b$	RSE	$1/t_b$ bid.	RSE bid.	sl
Testbed I (Nvidia Tesla K40)						
$h2d$	$2.4e^{-6}$	$3.15e^9$	$1.1e^{-6}$	$2.94e^9$	$2.7e^{-6}$	1.07
$d2h$	$2.2e^{-6}$	$3.29e^9$	$2.1e^{-6}$	$2.84e^9$	$3.4e^{-6}$	1.16
Testbed II (Nvidia Tesla V100)						
$h2d$	$2.5e^{-6}$	$12.18e^9$	$1.7e^{-6}$	$9.59e^9$	$3.4e^{-6}$	1.27
$d2h$	$2.5e^{-6}$	$12.98e^9$	$2.8e^{-6}$	$9.21e^9$	$4.2e^{-6}$	1.41

$$t^T_{h2d} = t_i + t_b \cdot (\overbrace{T[\cdot T] \cdot sizeof(dtype)}^{\textbf{bytes}})$$

As discussed, we assume that bidirectional overlap imposes a constant slowdown (sl) to transfer, and therefore the bidirectional transfer time can be estimated by:

$$t_{h2d\_bid} = sl_{h2d} \times t_{h2d}$$

Similarly, t^T_{d2h}, t_{d2h_bid} are modeled with the same equations. Therefore, the system-wise transfer parameters required for prediction are t_i, t_b and sl for h2d and d2h (six in total). To fit these coefficients, we conduct a set of micro-benchmarks, subset of those proposed by Pearson [5]. For all transfer experiments, we use the `cublas{Set/Get}MatrixAsync` routines for h2d, d2h transfers respectively, with pinned host memory, as required by these asynchronous calls. We obtain t_i empirically as the average latency of multiple single-byte transfers. For t_b, we run benchmarks for square transfers with $dtype = double$, for $D1 = D2 = 256 \xrightarrow{\text{step=256}} max\_device\_memory/2$, and follow the same approach for sl, but couple the entire transfer with a concurrent transfer towards the opposite direction. We use least square regressions on the 64 samples to compute t_b, both in the case of unidirectional and in the case of bi-directional transfers, excluding t_i from the transfer time during the regression (assuming zero intercept), in the manner of [32], and then estimate sl. We ensure the statistical robustness of the empirical values by collecting repetitive measurements, until the 95% confidence interval of the mean falls within 5% of the reported mean value, for all micro-benchmarks. The micro-benchmarks for transfer times are lightweight (requiring less than 10 minutes and less than 3 minutes on Testbed I and II, respectively), and only need to be run once on every new system CoCoPeLia is deployed.

Table II contains the obtained values for the two testbeds used in this work, described later in Section V-A. The displayed $1/t_b$ is equal to each system's PCIe bandwidth for each direction. Testbed II has almost 3× higher bandwidth than testbed I, but also has much larger bidirectional slowdowns sl for both directions, indicating that overlapping h2d and d2h transfers is not going to be as effective. The least square regression coefficient p-values for t_b and sl_{bid} are $< 2.2e-16$, and the RSEs are between 1 and $4.2e^{-6}$, which is comparable to t_i, however this only affects the prediction of small transfers.

Second, we estimate routine *GPU execution time*. We are only interested in the time of fine-grained chunks of specific small tiling sizes, therefore we measure the execution time for a set of tiling sizes T for each routine, store them and perform value lookups at runtime, for usage in our models. The usage of empirical estimates is favored by the tiled execution and the non-linear execution time assumption in CoCoPeLia, since micro-benchmarks for these chunks are much more lightweight than an approach that would require the full problem's execution time, as in [11]. For example, empirically estimating the execution time for a `gemm` problem of size $M = N = K = 32K$ using $T = 2048$ would require $t_{exex\_routine}(M, N, K)$ for [11], while for our case $t_{exex\_routine}(T, T, T)$, which requires 4096 times less computations, would be sufficient. To measure the GPU BLAS execution time, we use cuBLAS, but given that kernel execution is wrapped and all libraries follow the BLAS standard, this benchmarking method is applicable to any BLAS GPU library with minimal adjustments.

We choose three representative routines; `axpy` for 1D splitting and `gemm` for single and double precision for square tiling. For `daxpy`, we run 256 benchmarks for $D1 = N = 2^{18} \xrightarrow{\text{step}=2^{18}} 2^{26}$. For `dgemm` and `sgemm`, we use 64 benchmarks with square dimensions $D1 = D2 = D3 = 256 \xrightarrow{\text{step=256}} 16384$ for value lookup of tiled sub-kernels. Therefore, the sub-model t^T_{GPU} can only predict time for these 380 and 64 tile sizes (via direct value lookup) for `daxpy` and `gemm` respectively. We repeat the micro-benchmarks, until the 95% confidence interval of the mean falls within 5% of the reported mean value. The required time for the benchmark execution is less than 6 minutes for each routine on Testbed I, and less than 2 minutes on Testbed II.

CoCoPeLia automates the micro-benchmark execution on any new system without modifications. Upper limits for the required benchmarks are extracted based on the available GPU memory. Additionally, CoCoPeLia can be easily extended for any BLAS routine by modifying the existing micro-benchmark template scripts with the new routine and its parameters.

B. Tile selection

The CoCoPeLia runtime includes two functions for tile selection. The function `CoCoPeLia_predict` combines the empirical values obtained at deployment with the problem-specific parameters used in the BLAS routine invocation listed in Table I, to provide the execution time of a BLAS routine, as a function of the tiling size T, using Eq. 4 and 5 for cases without and with data reuse. The problem dimensions $D1[, D2[, D3]]$ are inferred from the BLAS dimensions M, N, K, and $S1_i, S2_i$ are then calculated based on the above. Finally, get_i, set_i are obtained by querying the pointers of the respective $i{-}th$ data structure (e.g. matrix, vector) using `cudaPointerGetAttributes`. The function `CoCoPeLia_select` is used to provide the best tiling size T for a specific problem, using the `CoCoPeLia_predict` routine to find T_{best}, which minimizes the total offload time, by iterating through all sizes T obtained at deployment for the target routine. The function can be extended to include different optimization criteria (e.g. GPU utilization, memory etc.). We have measured model initialization to take 2-3 ms and prediction time to be negligible (less than 100 μs).

The extension of the CoCoPeLia Tile selection runtime with additional BLAS routines, besides the micro-benchmarks explained in IV-A, requires the following modifications: i) the extension of a skeleton for a `CoCoPeLia_{routine}_init` function, that matches the routine's parameters to the struct with the model parameters of table I, and ii) the selection of a `CoCoPeLia_predict_{ModelName}` function for Tile prediction of this routine. The extension of CoCoPeLia with new prediction models is possible by defining a new `CoCoPeLia_predict_{ModelName}`. However, if any additional parameters are required, the struct of table I must be also modified accordingly.

C. Library-Tile scheduler

While selecting an appropriate tiling size T should suffice for a 3-way concurrency optimized library to achieve near-optimal performance, existing libraries do not optimize level-1 and -2 BLAS routines, while *cuBLASXt* and *BLASX* often result in less performance than what Eq. 5 hints.

To validate the accuracy of the proposed data-reuse model and fill this performance gap, as a part of CoCoPeLia, we implement an optimized end-to-end library for a subset of the BLAS prototype on top of state-of-the art primitive libraries. We use cuBLAS as the GPU execution and data transfer backend, utilizing `cublas{Dtype}{Routine}` and `cublas{Set/Get}MatrixAsync` routines respectively. For 3-way concurrency, we use CUDA streams, utilizing one stream per operation (h2d transfer, d2h transfer, kernel execution). The tile splitting, address matching and distribution (tile scheduler) are implemented based on the square tiling approach (as implied in Eq 5), also used by [6], [8]. After calculating these, the tile scheduler hands over all underlying transfers and execution to the aforementioned cuBLAS calls. Additionally, we enable GPU buffer and CUDA stream reuse after the first routine call, to avoid allocation/de-allocation overheads, as proposed by *BLASX* [8] to emulate an iterative use-case scenario. Finally, CoCoPeLia routines support either passing the tiling size T as an extra BLAS parameter, similar to *cuBLASXt*, for validation reasons, or using `CoCoPeLia_select` internally to predict T_{best} during invocation. CoCoPeLia routines also take advantage of model reuse in the second case; they initialize the corresponding model only the first time a user makes a call to CoCoPeLia with a set of parameters (routine, problem size, flags, etc) and use the preobtained T_{best} in subsequent calls. The tile scheduler is generalized per BLAS-level. To add a new BLAS routine that utilizes the tile scheduler requires the creation of a routine wrapper, since the specifics of each BLAS operation differ, but transfer/ execution overlap is then achieved by the tile scheduler without modifications. The underlying backend functions are wrapped and can also be modified, as long as an overlap mechanism similar to CUDA streams is available.

TABLE III
TESTBED CHARACTERISTICS

	Testbed I	**Testbed II**
CPU	Intel Core i7-4820K 3.7GHz	Intel Xeon Silver 4114 2.2GHz
GPU	NVIDIA Tesla K40 FP Peak 4.3 TFlop/s DP Peak 1.4 TFlop/s	NVIDIA Tesla V100 FP Peak 14 TFlop/s DP Peak 7 TFlop/s
PCIe	Gen2 x8	Gen3 x16
Compiler (host)	g++ 4.7.2	g++ 7.5.0
Compiler flags	-O3, -lm, -std=gnu99	-O3, -lm, -std=gnu99
CUDA	9.2	9.2
Compiler flags (GPU)	-O3, -arch=sm_35	-O3, -arch=sm_75

V. EVALUATION

In this section, we evaluate the CoCoPeLia framework for three example kernels; `daxpy`, `sgemm`, and `dgemm`. First, we present our experimental setup, and describe the micro-benchmark and validation sets we used. Then, we validate the proposed 3-way concurrency time prediction model error and the ability of CoCoPeLia to select near-optimal tile sizes. Finally, we present the end-to-end performance achieved with CoCoPeLia using our own 3-way concurrency implementation coupled with automatic tile selection, and compare it with the state of the art.

A. Experimental setup

We perform experiments for the validation of our models and the evaluation of CoCoPeLia on two different testbeds, the details of which are presented in Table III, along with the information on code compilation. For time measurements we use `clock_gettime`, with device synchronization (`cudaDeviceSynchronize()`) also included; both timer and synchronization overhead were less than 1% of the benchmarked times. We perform 100 executions for each benchmark, after a warmup run, not accounted for, and we report the average time for all models, unless otherwise noted. The allocation time needed for CPU/GPU buffers is not modeled or included in the total time, and all matrices/vectors are initialized with random values before execution. We use pinned host memory to enable Async CUDA calls and the cashes/buffers are not flushed between runs.

B. Validation sets

To validate different initial memory locations and problem sizes, we select four large problem sizes ($N = \{8, 64, 128, 256\} \cdot 2^{20}$) for `daxpy`, for all $2^2 - 1 = 3$ location combinations (15 problems). Similarly, for `sgemm` and `dgemm` we want to validate different locations, problem sizes and shapes. We use four square problem sizes $M = N = K = \{4, 8, 12, 16\} \cdot 2^{10}$, for all $2^3 - 1 = 7$ location combinations (28 problems) to validate the location-problem size, and 3 problem sizes with $M \cdot N \cdot K = \{4, 8, 12, 16\} \cdot 2^{10 \cdot 3}$ for 3 fat-by-thin ratios $M = N = K \cdot \frac{r^3}{8}, r \in [3, 4, 5]$ and 3 thin-by-fat ratios $M = N = K \cdot \frac{8}{r^3}, r \in [3, 4, 5]$ for the scenario of all data initially residing on the CPU (24 problems). We exclude the scenario where all data is located on the GPU, since there is no overlap. All selected problem sizes can fit in

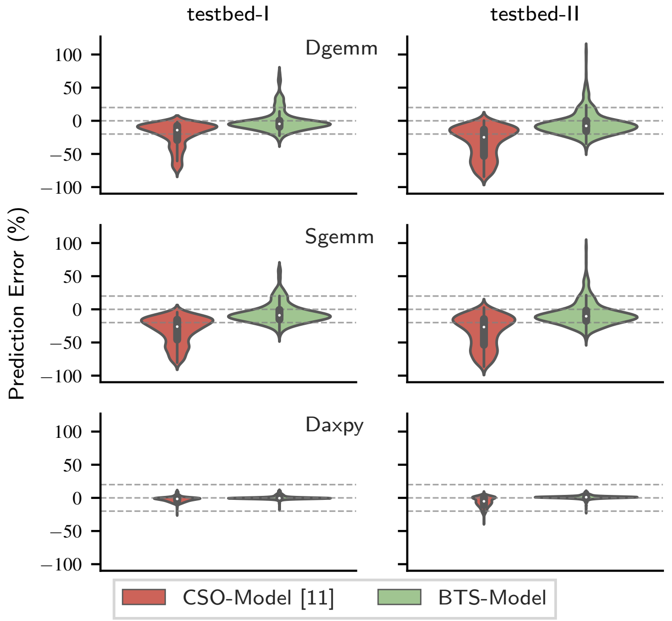


Fig. 4. Error distribution of the *CSO-Model* and the *BTS-Model* for `daxpy` and `cublasXt_{D,S}gemm` without data reuse on testbeds I and II.

the device memory; we do not consider larger problem sizes since that would require a considerably more sophisticated implementation of overlap with memory constraints, which is outside the scope of this work. For each problem size, we measure the execution time t of 1) the CoCoPeLia wrapper and 2) *cuBLASXt*, for all tile sizes $T = 1024 \xrightarrow{\text{step=256}} 16384$ for which $T \leq \frac{min(D1[,D2,D3])}{1.5}$.

C. Time prediction validation

We first focus on validating the prediction ability of our bidirectional transfer overlap-aware model of Eq. 4, hereafter referred to as *BTS-Model*, and our data reuse-aware model of Eq. 5, hereafter referred to as *DR-Model*, used in t_{over} = `CoCoPeLia_predict` of Figure 3. We examine their error over measured execution time and compare their predictive power against the analytical CUDA stream overlap model with two copy engines, proposed in [11], hereafter referred to as *CSO-Model*. We evaluate the percentage (relative) error $e_{\%} = 100 \cdot (t_{predicted} - t_{measured})/t_{measured}$. We highlight that both models include empirical parts, which impose second order errors. Nonetheless, the comparison between different models is fair, as we rely on the same micro-benchmarks to collect the empirical values.

We first validate the prediction accuracy of the *BTS-Model*, which is suitable for problems without data reuse between subkernels, using the level-1 BLAS `daxpy`, which does not reuse data, and *cuBLASXt* `sgemm` and `dgemm`, which do not sufficiently utilize data reuse to minimize transfers [8], [9]. Figure 4 shows the relative error distribution for `daxpy`, `sgemm` and `dgemm`, for both testbeds, in the form of violinplots. First, we note that, on both testbeds, the *BTS-Model* achieves very high prediction accuracy for `daxpy` with median errors between 1 to 2%, while the *CSO-Model* underpredicts execution time with median errors between -3% to -7%. This is attributed to the *CSO-Model* not accurately modeling the actual bidirectional overlap, and is more evident on Testbed-II, where the slowdown is larger on both directions. Second, for both `sgemm` and `dgemm`, the prediction error is higher. The *CSO-Model* again significantly underestimates the execution time in almost all cases, with median errors between -20% to -34%. On the other hand, the *BTS-Model* demonstrates smaller median errors between -10% to -15% and a better error distribution with no bias towards underprediction.

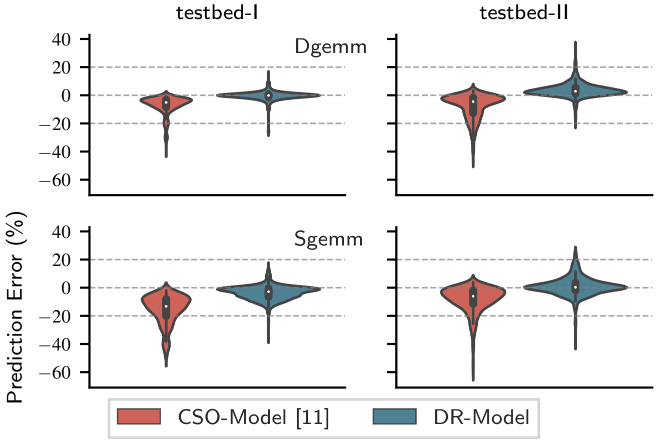
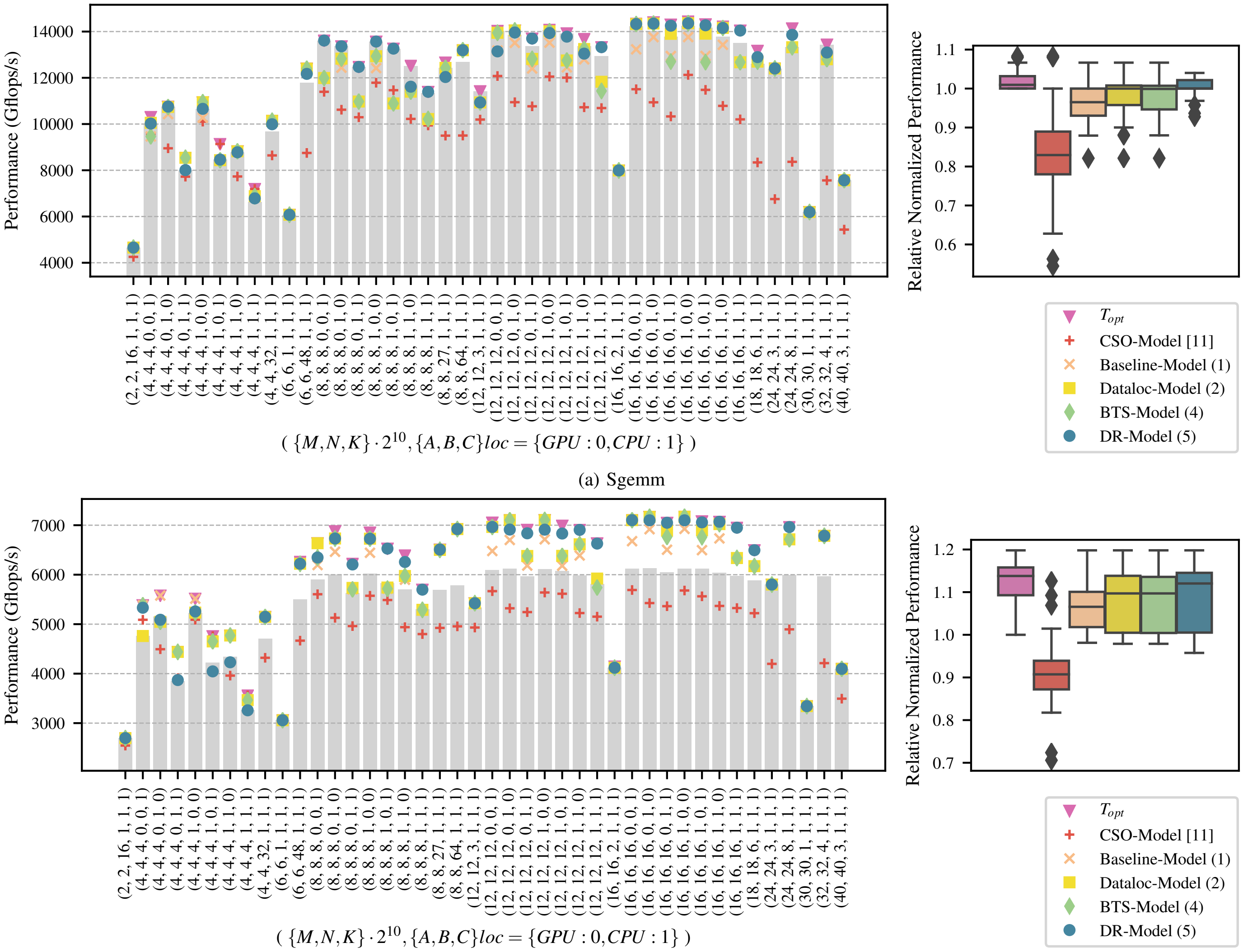


Fig. 5. Error distribution of the *CSO-Model* and the *DR-Model* for our CoCoPeLia wrapper BLAS implementation of `sgemm` and `dgemm` on testbeds I and II.

We then validate the prediction accuracy of the *DR-Model*, using our aforementioned implementations for `sgemm` and `dgemm`, in the CoCoPeLia library, which have near-optimal data reuse on single-GPU scenarios, when the problem fits the GPU memory. Figure 5 demonstrates the relative error distribution on both testbeds. The *CSO-Model* underestimates execution times, similarly to the *cuBLASXt* case in Figure 4, however with fewer underestimations with errors ranging from -20% to -60% and a lower median error of -7% to -15%. Again, our *DR-Model* is significantly more accurate, with median errors ranging from 2% to -5%, and a few high positive errors (overestimations). It is interesting to note that both models exhibit higher errors for `sgemm`, where the memory footprint is half than the equivalent of `dgemm`, and smaller problems are more prone to second order errors from the empirical value acquisition. Additionally, our *DR-Model* is more accurate for Testbed I, than Testbed II. This is due to spikes in performance on the NVIDIA Tesla V100 GPU of Testbed II for cublas_{D,S}gemm, which are not present in the NVIDIA Tesla K40 of Testbed I. We attribute these to the more complex GPU architecture of the former.

D. Validation of tiling size selection

Subsequently, we validate the CoCoPeLia tiling size selection ability when used in practice. The target of the CoCoPeLia framework is to predict the tiling size that leads to near-optimal performance. We hence consider the following scenario: for all validation cases in Section V-B, we explore the performance



(a) Sgemm

(b) Dgemm

Fig. 6. Evaluation of Tile selection ability for Sgemm (a) and Dgemm (b) on testbed II. The baseline performance (gray bars) is acquired using a static tiling size $T = 2048$, also used by BLASx. We compare this against the experimentally achieved performance using the optimal tiling size for each problem, $T = T_{opt}$, the performance achieved using the tiling size predicted with the `CSO-Model` [11] and c) the performance achieved using $T = T_{best}$ returned by $CoCoPeLia\_select$ using Equations 1, 2, 4, 5 respectively.

achieved by the prediction of each model, and how this compares with a good baseline tiling size and the maximum achievable performance, using the optimal tiling size T_{opt}.

Figure 6 shows the results of this comparison for `dgemm` and `sgemm` on Testbed II. The selected baseline tiling size is $T = 2048$. First, the CSO-Model mispredicts the optimal tiling sizes in both cases leading to performance degradation compared to the baseline. This happens mostly because the CSO-Model does not take into account the non-linearity of execution time, which results in favoring small tiles with limited performance. Additionally, it is evident that the baseline is enough to provide near-optimal performance for Figure 6a, where even T_{opt} provides a median performance improvement of 1%, and a maximum of 10%. The CoCoPeLia models provide performance close to the baseline, with the DR-Model surpassing its performance, but less than 1% (which is close to the T_{opt} median). On the other hand, in the case of Figure 6b, T_{opt} is able to provide improvements of a median of 13.5% and up to 20%. In this case, the incremental improvement of each CoCoPeLia model is more evident; the *Baseline-Model* (Equation 1) provides a median speedup of 7%, the *Dataloc-Model* (Equation 2) and *BTS-Model* (Equation 4) both provide median improvement of 10%, and the DR-Model (Equation 5) provides 12% improved performance, which is very close to the T_{opt} median. We note that bidirectional slowdown, considered by the *BTS-Model*, does not significantly affect the performance of `gemm`, which requires fewer d2h than h2d transfers. Its impact is more evident in level-1 BLAS functions with similar transfers to and from device memory.

E. Performance evaluation

To evaluate the end-to-end performance of the runtime scheme proposed in Figure 3, we extend the validation set of Section V-B. For `daxpy`, we select 11 large problem

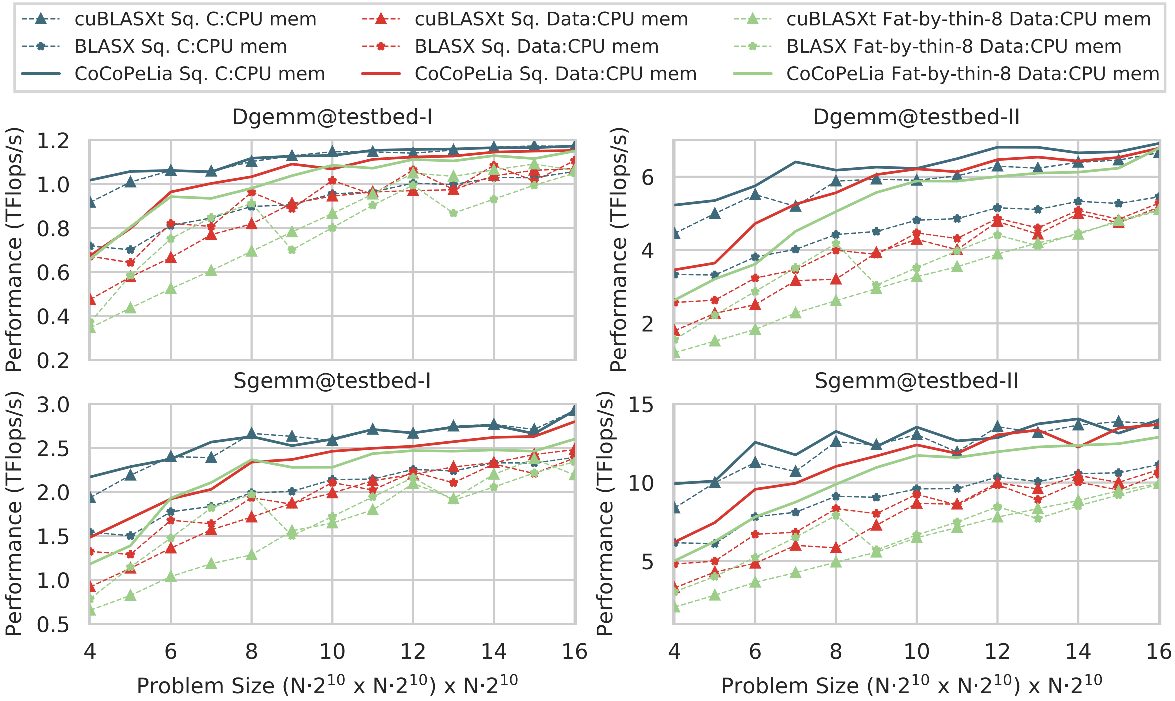


Fig. 7. `dgemm` and `sgemm` performance evaluation for various problem sizes on testbeds I,II. We use three scenarios with different transfer-to-computation ratios: 1) $M = N = K$ with A, B on the GPU and C requiring update from the CPU (blue), 2) $M = N = K$ with A, B, C on the CPU (red) and 3) $N = M = \frac{K}{8}$ with A, B, C on the CPU (green).

sizes $N = (1 \xrightarrow{\text{step}=N*2} 1024) \cdot 2^{20}$ for all three location combinations. For `sgemm` and `dgemm`, we select 25 square problem sizes $M = N = K = (4 \xrightarrow{\text{step}=0.5} 16) \cdot 2^{10}$ for all seven location combinations and for all thin/fat ratios of the aforementioned validation set. Overall, we evaluate 33 problems for `daxpy` and 325 for `dgemm` and `sgemm`.

We compare the performance of CoCoPeLia `sgemm` and `dgemm` against *cuBLASXt* and *BLASX*. These are both multi-GPU libraries, but *cuBLASXt* is the state of practice and *BLASX* offers the most performance for transfer-bound cases, deeming them the most relevant comparison targets for single GPU 3-way concurrency. We compare the performance of CoCoPeLia `daxpy` against a unified memory implementation with prefetching. Our *BLASX* results use the library default - static - tiling size $T = 2048$. For *cuBLASXt*, which accepts the tiling size as an input parameter, we test 10 different tiling sizes and choose the best for each problem. We note that this nearly-exhaustive tiling size selection gives a performance advantage to *cuBLASXt* over *BLASX*.

Figure 7 visualizes the performance of the three libraries for `dgemm` and `sgemm` on our two testbeds, for three scenarios of problem sizes and data locations. We first note that *BLASX* outperforms *cuBLASXt* in fat-by-thin matrices, while *cuBLASXt* shows better performance in the low transfer cases, where only the C matrix resides on the CPU. Second, CoCoPeLia outperforms both *BLASX* and *cuBLASXt* in all three scenarios. For the low-transfer scenario (blue), its performance is on par with *cuBLASXt*, but it considerably outperforms the other two libraries for the full offload scenario (red) and the transfer-heavy fat-by-thin matrix multiplication (green). Third, CoCoPeLia provides better relative performance on testbed II, which has a lower bandwidth/FLOP ratio and therefore transfers are a bigger bottleneck.

TABLE IV
(GEO)MEAN PERCENTILE COCOPELIA PERFORMANCE IMPROVEMENT OVER STATE OF THE ART GPU BLAS LIBRARIES.

System	Testbed I		Testbed II	
Offload Scenario	Full	Partial	Full	Partial
daxpy	21.5%	9.4%	19.9%	9.1%
dgemm	16.2%	5.8%	32.2%	15.6%
sgemm	20.6%	5.7%	33.3%	15.7%

In Table IV we summarize the mean percentile performance improvement of CoCoPeLia over the best among the two other libraries for each problem size, calculated using the geometric mean of the fraction of their times, respectively. We separate full and partial offload cases for reference with relevant literature, where full offload refers to all data residing on the CPU, and partial offload to some of the data residing on the GPU. The results are similar to the outlined cases in Figure 7; CoCoPeLia outperforms the other libraries by 16-33% in the full offload case and 5-15% in the partial offload case, indicating that it is able to improve cuBLAS performance without architecture-specific tuning or bias towards specific data shapes.

VI. Conclusion

In this work we outlined that 3-way concurrency is currently not well utilized in BLAS GPU offload libraries, since the efficient split tile size depends on routine, problem, and system-specific parameters. Since part of these only become available during runtime, we propose a) two models for the 3-way overlap offload time as a function of tile size, b) a micro-benchmark approach for initializing the empirical model parameters offline, and c) a runtime tile scheduler for efficient 3-way overlap and data reuse. We combine these into an end-to-end GPU BLAS framework, CoCoPeLia, and demonstrate its use for `dgemm`, `sgemm` and `daxpy`; our evaluation shows that it achieves lower errors than previous approaches and is usable in practice for efficient tile prediction. Furthermore, our BLAS wrapper with runtime tile prediction offers considerable performance improvement over previous offload approaches for all tested routines.

The proposed model offers many directions for future work, since all components in Figure 3 can be improved separately; the empirical modeling can be improved to limit second order errors; the models can be fine-tuned to specific implementations and/or other routines; the tile scheduling strategy can be further optimized. We plan to extend the model to more complex tiling schemes for level-3 BLAS, and include multi-GPU and host-assisted execution, with the vision of providing a portable auto-tuned heterogeneous BLAS library.

References

[1] J. Dongarra, "Basic linear algebra subprograms technical (blast) forum standard ii," IJHPCA, vol. 16, pp. 1–111, 05 2002.

[2] "developer.nvidia.com/cublas."

[3] W. Li, G. Jin, X. Cui, and S. See, "An evaluation of unified memory technology on nvidia gpus," in 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015, pp. 1092–1098.

[4] A. Mishra, L. Li, M. Kong, H. Finkel, and B. Chapman, "Benchmarking and evaluating unified memory for openmp gpu offloading," in Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, 2017, pp. 1–10.

[5] C. Pearson, A. Dakkak, S. Hashash, C. Li, I.-H. Chung, J. Xiong, and W.-M. Hwu, "Evaluating characteristics of cuda communication primitives on high-bandwidth interconnects," in Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, 2019, pp. 209–218.

[6] "developer.nvidia.com/cublasxt."

[7] "docs.nvidia.com/cuda/nvblas."

[8] L. Wang, W. Wu, J. Xiao, and Y. Yang, "BLASX: A high performance level-3 BLAS library for heterogeneous multi-gpu computing," CoRR, vol. abs/1510.05041, 2015. [Online]. Available: http://arxiv.org/abs/1510.05041

[9] T. Gautier and J. V. F. Lima, "Xkblas: a high performance implementation of blas-3 kernels on multi-gpu server," in 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2020, pp. 1–8.

[10] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil, "Performance models for asynchronous data transfers on consumer graphics processing units," Journal of Parallel and Distributed Computing, vol. 72, no. 9, pp. 1117 – 1126, 2012, accelerators for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731511001468

[11] B. v. Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal, "Performance models for cpu-gpu data transfers," in 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2014, pp. 11–20.

[12] B. Liu, W. Qiu, L. Jiang, and Z. Gong, "Software pipelining for graphic processing unit acceleration: Partition, scheduling and granularity," The International Journal of High Performance Computing Applications, vol. 30, no. 2, pp. 169–185, 2016. [Online]. Available: https://doi.org/10.1177/1094342015585845

[13] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," SIGARCH Comput. Archit. News, vol. 37, no. 3, p. 152–163, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1555815.1555775

[14] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," SIGPLAN Not., vol. 45, no. 5, p. 105–114, Jan. 2010. [Online]. Available: https://doi.org/10.1145/1837853.1693470

[15] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in 2011 IEEE 17th International Symposium on High Performance Computer Architecture, 2011, pp. 382–393.

[16] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, "Grophecy: Gpu performance projection from cpu code skeletons," in SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–11.

[17] M. Martell and H. Sato, "Linear performance-breakdown model: A framework for gpu kernel programs performance analysis," International Journal of Networking and Computing, vol. 5, pp. 86–104, 01 2015.

[18] E. Konstantinidis and Y. Cotronis, "A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling," Journal of Parallel and Distributed Computing, vol. 107, pp. 37 – 56, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731517301247

[19] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software, 2011, pp. 134–144.

[20] D. Schaa and D. Kaeli, "Exploring the multiple-gpu design space," in 2009 IEEE International Symposium on Parallel Distributed Processing, 2009, pp. 1–12.

[21] M. Boyer, J. Meng, and K. Kumaran, "Improving gpu performance prediction with data transfer modeling," in 2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, 2013, pp. 1097–1106.

[22] A. Riahi, A. Savadi, and M. Naghibzadeh, "Comparison of analytical and ml-based models for predicting cpu–gpu data transfer time," Computing, January 2020.

[23] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole, "Modeling and predicting performance of high performance computing applications on hardware accelerators," in 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum, 2012, pp. 1828–1837.

[24] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–12.

[25] E. Chan, F. Van Zee, P. Bientinesi, E. Quintana-Orti, G. Quintana-Orti, and R. Van de Geijn, SuperMatrix: a Multithreaded Runtime Scheduling System for Algorithms-by-blocks, ser. Technical report (University of Texas at Austin. Department of Computer Sciences). Computer Science Department, University of Texas at Austin, 2007. [Online]. Available: https://books.google.gr/books?id=ggn-jwEACAAJ

[26] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid gpu accelerated manycore systems," Parallel Computing, vol. 36, no. 5, pp. 232 – 240, 2010, parallel Matrix Algorithms and Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819109001276

[27] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs," in GPU Computing Gems, W. mei W. Hwu, Ed. Morgan Kaufmann, Sep. 2010, vol. 2. [Online]. Available: https://hal.inria.fr/inria-00547847

[28] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," ACM Transactions on Mathematical Software, vol. 41, no. 3, pp. 14:1–14:33, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2764454

[29] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical dag scheduling for hybrid distributed systems," in 2015 IEEE

International Parallel and Distributed Processing Symposium, 2015, pp. 156–165.

[30] G. Bernabé, J. Cuenca, L.-P. García, and D. Giménez, "Tuning basic linear algebra routines for hybrid cpu+gpu platforms," Procedia Computer Science, vol. 29, pp. 30 – 39, 2014, 2014 International Conference on Computational Science. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187705091400180X

[31] E. Peise and P. Bientinesi, "Performance modeling for dense linear algebra," CoRR, vol. abs/1209.2364, 2012. [Online]. Available: http://arxiv.org/abs/1209.2364

[32] T. Hoefler, T. Schneider, and A. Lumsdaine, "Loggp in theory and practice–an in-depth analysis of modern interconnection networks and benchmarking methods for collective operations," Simulation Modelling Practice and Theory, vol. 17, no. 9, pp. 1511–1521, 2009.