SC24: International Conference for High Performance Computing, Networking, Storage and Analysis | 979-8-3503-5291-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/SC41406.2024.00049

CUDASTF: Bridging the Gap Between CUDA and Task Parallelism

Cédric Augonnet
NVIDIA
caugonnet@nvidia.com

Andrei Alexandrescu
NVIDIA
andrei@nvidia.com

Albert Sidelnik
NVIDIA
asidelnik@nvidia.com

Michael Garland
NVIDIA
mgarland@nvidia.com

***Abstract*—Organizing computation as asynchronous tasks with data-driven dependencies is a simple and efficient model for single- and multi-GPU programs. Sequential Task Flow (STF) is such a model that derives task graphs from data dependencies.**

We propose CUDASTF, a C++ library that implements STF over CUDA APIs, fostering easy creation of scalable and composable algorithms. Users may easily elect to use CUDA Graphs instead of streams, which improves performance of small kernels. Structured kernels are automatically spread over multiple devices and can exercise fine-grained affinity control. Implementation-wise, CUDASTF makes a compelling argument for an event-based approach to asynchronous parallel libraries.

We obtain up to a 1.8x improvement over the cuSolverMg library on Cholesky decomposition. On a small weather simulation task we demonstrate near-optimal scalability of our multi-GPU kernels; also, on a single GPU, CUDA Graphs improve performance by up to 30%. Finally, we were able to author the first implementation of the CKKS fully homomorphic encryption scheme over multiple devices.

I. Introduction

Today's HPC systems developers need to grapple with complex algorithms, massive workloads, and diverse heterogeneous hardware environments. Asynchronous programming is necessary to fully exploit these increasingly complex machine organizations, but asynchrony is notoriously difficult to manage by hand. Low-level interfaces, such as CUDA, offer all the tools necessary to leverage complex hardware configurations, but leave it to the programmer to manage resources and to coordinate asynchronous operations correctly and efficiently. The majority of programmers would benefit from a higher-level programming framework that provides tools for efficient intra-process asynchrony/concurrency, integrates with existing compilers and libraries, works with inter-process communication layers, and does not force developers into a narrow model of computation.

To fill this gap, we propose CUDASTF, a modern C++ library that equips programmers to design composable asynchronous algorithms by describing the natural data dependencies of individual tasks, rather than manually prescribing the complex control flow dependencies implicit in their algorithm. Our approach is designed around the notions of *tasks*, which are units of computation with clearly defined inputs and outputs, and *logical data*, which represents data being produced and/or consumed by tasks. We provide a Sequential Task Flow (STF) programming model, which automatically transforms a sequence of tasks specified in program order into a directed acyclic graph (DAG) of tasks whose ordering constraints are inferred from named data dependencies. This model allows our runtime to efficiently parallelize and scale algorithms on a variety of hardware platforms without a need to modify any source code. Modern C++ constructs and a typed interface allow describing structured algorithms as well as structured compute kernels without the complexities of a domain-specific language.

We introduce new programming model abstractions for managing asynchrony and placement of work and data. These abstractions allow programmers to leverage advanced platform features, such as CUDA Graphs, with little or no additional application code. We have also designed mechanisms for efficiently distributing user-authored compute kernels across multiple GPUs. User code can exercise fine-grained control over affinity. Choosing the optimal affinity being a difficult problem, we also developed a novel robust randomized algorithm that maps and allocates data over multiple devices with arbitrary affinity mappings. Our use of virtual memory manager support in CUDA allows near-optimal scaling of unmodified kernels to multiple GPUs.

Our programming model is designed such that the implementation can execute with minimal overhead compared to manually written low-level code. Moreover, our implementation adopts several advanced strategies used in manually optimized HPC code, such as pools of asynchronous streams and custom memory allocation strategies, and makes them available to any user of our programming model. Our implementation is fully integrated within the CUDA ecosystem and can orchestrate calls to existing optimized libraries (e.g., CUBLAS and CUB) along with user-provided compute kernels, possibly written using CUDASTF's own kernel-generation facilities. Natural interoperation with existing CUDA code makes incremental migration easy.

As a result of these carefully designed programming model abstractions and the use of several advanced implementation techniques, our library delivers performance that compares favorably with hand-tuned code without the need for complex manual optimization. We improve single-GPU performance of benchmarks such as miniWeather by up to 30% on a DGX A100 by leveraging CUDA Graphs. We also show that the resulting programmability and composability

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

enhancements help support writing efficient multi-GPU applications; to demonstrate this we have built the first multi-GPU implementation of the CKKS scheme for fully homomorphic encryption (FHE).

In summary, by abstracting parallelism and asynchrony and by automating low-level resource management, our library lets programmers focus on the problem at hand, such as implementing algorithms or optimizing kernels, instead of the minutiae of synchronization and memory management. It gives applications the tools necessary to efficiently use multiple GPUs within a single process, while also offering several opportunities for improving performance on single-GPU architectures.

II. Programming model

We discuss the Sequential Task Flow (STF) programming model using the simple example in Algorithm 1, which depicts a set of four operations over three variables that could be vectors, tensors, or even encrypted ciphertexts (see § VII-E). Regardless of the underlying data representation, operations O_2 and O_3 need to access X, which is the result of O_1, and O_4 needs the results of O_2 and O_3. Such a *sequence* of operations can thus be interpreted as a graph solely based on data accesses, as illustrated by the green nodes of the graph in Fig. 1. (The additional dotted nodes illustrate how instantiating such a task graph over a machine equipped with accelerators requires a variety of ancillary allocation and transfer operations.) These operations are deterministically implied by the location of computation and data, so ideally they should not be the responsibility of the programmer. The resulting graph depends neither on the underlying data representation, nor on the actual implementation of the asynchronous operations.

Algorithm 1 A sequence of operations

```
$O_1$    $X = 2X$          // scale
$O_2$    $Y = Y + X$       // add
$O_3$    $Z = Z + X$       // add
$O_4$    $Z = Z + Y$       // add
```

The STF model [1] describes programs as sequences of operations that read and/or write data, and transforms them into graphs of interdependent tasks. This is a form of dataflow that infers dependencies rather than specifying them explicitly (as in most implementations of the dataflow model), resulting in simpler code.

We implemented this model in CUDASTF, a C++17 library designed to interoperate with the CUDA environment [2] (e.g., CUBLAS, CUB, Thrust) and with the larger parallel computing ecosystem such as OpenACC [3]. An important objective has been to facilitate the gradual migration and enhancement of existing code. Leveraging the modeling power of the C++ language allows the creation of expressive and efficient abstractions without the need for custom languages or compiler extensions.

Fig. 2 shows the CUDASTF implementation of Algorithm 1. Including the `cudastf.h` header on line 1 makes the library available. (Being a header-only library, CUDASTF obviates

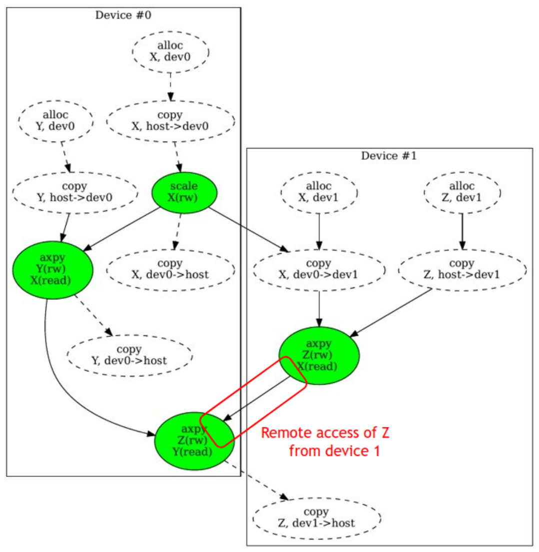

Fig. 1: Task graph obtained with code on Figure 2 when using places (high level tasks are depicted in green, ancillary tasks have dotted contours)

the need for a traditional installation process or build instructions.) The `context` object defined on line 19 serves as an entry point for API calls and a state container. (§ III-A discusses how to customize the context object, for example to use CUDA graphs internally.)

The kernels on lines 4–16 are plain CUDA implementations. However, note that they use `slice<double>` for array manipulation instead of raw `double*` pointers. `slice` is a generic type provided as an alias for `std::mdspan` [4] instantiations. The recent standardization of `std::mdspan`—a versatile component that provides a multidimensional view into existing data arrays—makes it the preferred type for interfacing across high-performance C++ libraries and applications. Within CUDASTF, `std::mdspan`'s flexibility is also harnessed to implement bounds checking—an important safety mechanism—subject to an opt-in compilation flag.

A. Data model: Logical data and shapes

Lines 21–23 of Fig. 2 define `logical_data` objects to track the C++ variables `X`, `Y`, and `Z` within `ctx`. We set out to explain this notion in depth.

Usually, data in computing systems is modeled as the content of a unique global storage. That abstraction works well for a single processor attached to a single memory, but is less adequate in modern heterogeneous systems with multiple physical memories. Such systems make manual data management difficult and error-prone. To relieve the programmer of that burden, CUDASTF defines *logical data*, a core abstraction that regards data as an abstract entity, without explicitly associating it with any specific storage. A logical data object identifies a piece of data that may have multiple coherent replica in distinct physical memories. Replicas are embodied by *data instance* objects. Each logical data object

```
1  #include "cudastf/cudastf.h"
2  using namespace cudastf;
3
4  __global__ void scale(double a, slice<double> y) {
5    int i = blockIdx.x * blockDim.x + threadIdx.x;
6    for (; i < x.size(); i += gridDim.x * blockDim.x) {
7      x(i) *= a;
8    }
9  }
10
11 __global__ void add(slice<const double> x, slice<double> y) {
12   int i = blockIdx.x * blockDim.x + threadIdx.x;
13   for (; i < x.size(); i += gridDim.x * blockDim.x) {
14     y(i) += x(i);
15   }
16 }
17
18 int main() {
19   context ctx;
20   double X[N], Y[N], Z[N]; // ... initialize X, Y and Z ...
21   logical_data<slice<double>> lX = ctx.logical_data(X);
22   auto lY = ctx.logical_data(Y);
23   auto lZ = ctx.logical_data(Z);
24
25   ctx.task(lX.rw())->*[](cudaStream_t s, slice<double> dX) {
26     scale<<<32, 128, 0, s>>>(2.0, dX);
27   };
28
29   ctx.task(lX.read(), lY.rw())->*[](auto s, auto X, auto Y) {
30      add<<<32, 128, 0, s>>>(dX, dY);
31   };
32
33   ctx.task(exec_place::device(1), lX.read(), lZ.rw())->*
34     [](auto s, auto X, auto Z) {
35       add<<<32, 128, 0, s>>>(dX, dZ);
36     };
37
38   ctx.task(lY.read(), lZ.rw(data_place::device(1)))->*
39     [](auto s, auto Y, auto Z) {
40       add<<<32, 128, 0, s>>>(dY, dZ);
41     };
42
43   ctx.finalize();
44 }
```

Fig. 2: A sequence of interdependent tasks using places

may have zero (if, for example, empty), one, or more instances in physical memory as needed. The library defines a coherency protocol to keep copies consistent and employs a specific strategy (§ VI-B) to allocate data close to computation.

Relying on logical data instead of memory addresses has a composability advantage as well. Explicit management hinders composability when the input data for an asynchronous algorithm is created by a previous algorithm, also asynchronous. Our data model is descriptive, allowing users to define the conceptual inputs and outputs of each algorithm in terms of logical data. The runtime system manages the allocation, replication, and deallocation of physical memory in coordination with the execution and completion of algorithms. This is especially helpful in complex multi-GPU systems where each GPU has its own attached memory.

The second important element of our data model is the notion of *shape*, which we use to describe data layouts and iteration spaces. The shape of a logical data object can be thought of as the full information about the layout of data in memory, but without the data itself. Examples of shapes include multidimensional arrays (`slice`s), sparse structures, hashtables, and trees. The notion of shape complements that of logical data, which is an abstract representation that does not assume a specific underlying layout or memory organization. Separating the shape from the logical data allows CUDASTF to internally use efficient methods to allocate and move data across the machine.

B. Tasking model: The Sequential Task Flow (STF) paradigm

In STF, a task represents a unit of asynchronous work. Tasks are described by the work performed and by their dependencies—i.e., the set of logical data accessed by the task. Each dependency has an access mode indicating read and/or write access. Algorithms are modeled as DAGs having tasks as vertices and dependencies between them as edges. Asynchronous execution of this graph gives the opportunity to maximize parallel execution and to hide some, if not all, of the latencies (e.g., memory movements) that we would observe in a sequential execution.

For example, line 25 of Fig. 2 submits a task for execution. The arguments passed to `ctx.task()` indicate that `lX` is being read and modified, and the result of the call is a task object initialized appropriately. That object receives the task code on the right-hand side of the `->*` operator as a lambda function that receives a CUDA stream to enqueue asynchronous work on the current device and a `slice` that is an automatically-transferred replica of `X`. The lambda function may return as soon as work is submitted; synchronization is done at context level (line 43).

It is worth noting that client code is never required to define a graph explicitly. Programming models that require declaration of explicit dependencies between tasks (e.g., Taskflow [5] or CUDA Graphs [6]) require an understanding of the graph structure and how the different parts of the algorithm are related. Our strategy derives such dependencies automatically from data dependencies, with the benefit of a dramatic reduction and simplification of client source code.

STF observes the usual concurrent access rules: multiple tasks can read the same logical data concurrently (Read-after-Read), whereas only one task can modify logical data (Read-after-Write, Write-after-Read, Write-after-Write). Our implementation applies these rules to transform sequences of tasks into DAGs. For every task T in a sequence, we introduce a dependency between T and the previous tasks that modify T's dependencies. A dependency is inserted for all previous readers of a logical data if T is modifying it. Therefore, task parallelism in conjunction with logical data provides a simple way to express parallel asynchronous algorithms. This approach is composable because we can express an algorithm regardless of how logical data have been previously used.

Finally, the `ctx.finalize()` call on line 43 ensures that all pending operations are finished, including the submitted tasks and all internal asynchronous operations (e.g., device memory deallocation). Additionally, completion of the call guarantees that all changes to logical data have propagated back to the original physical location (array `X` in the example). Notably, that write-back is performed while other computations are still taking place on the devices, in a thoroughly asynchronous manner.

Lines 34 and 39 illustrate execution and data places, respectively. Execution places let users decide where to perform computation, which by default is the current CUDA device. Data is generally fetched as close as possible to the execution place, but the user may decide to explicitly localize data in a specific data place if needed. The task on line 39 is executed on device 0, but accesses `lZ` directly from device 1. Changing the localization of computation or data can be done with minimum effort thanks to our descriptive approach; such a change would require a complete redesign within a prescriptive model.

III. Leveraging CUDA graphs

The CUDA Graph API [6] was introduced to improve the latency of workloads with repeated execution patterns and small granularity by enabling the pre-definition of entire sequences of GPU operations. However, fully tapping into the potential of this API introduces challenges (such as dependency management, data allocation and transport, and handling dynamic workload changes) that CUDASTF addresses.

A. Context backends

Contexts not only store execution state, but also allow users to choose the computational backend. The current implementation features two functionally equivalent backends that use distinct asynchrony and synchronization strategies. A default-constructed `context` object relies on CUDA streams and events for ordering asynchronous execution. For example, with that backend, memory allocations is done by using `cudaMallocAsync`, which is stream-ordered. If the context is initialized with a `graph_ctx` object instead, all asynchronous operations are lowered to the CUDA graph API, so memory would be allocated with `cudaGraphAddMemAllocNode`, which is graph-ordered. Both backends implement the same task interface, so the same code runs over CUDA streams or CUDA graphs depending only on how the context is created. Both contexts support mixing in CPU-bound workloads and can be used from multiple CPU threads.

Generating CUDA graphs from STF graphs automatically greatly improves on today's common practice of capturing CUDA workloads into CUDA graphs. The STF model automatically discovers concurrency and introduces the necessary data transfers, amenities inaccessible to a graph capture approach. It should be noted, however, that although our implementation generates multi-GPU graphs as well as single-GPU graphs, the underlying graph API is currently geared toward reducing the latency of single-GPU workloads, so that is where most gains will be noted.

A flexible user application would typically choose at runtime between the stream and the graph backend depending on the present hardware configuration and application-specific parameters such as data size. Making this choice automatically remains an interesting challenge in balancing trade-offs. On one hand, creating and and instantiating CUDA graphs adds time overhead. On the other, there are benefits for the device-side execution, which depend on task granularity and on application structure, such as repeated patterns of tasks.

Contexts can be nested, which offers further customization possibilities: for example, an application may create a graph context within a stream context and then launch the automatically generated CUDA graph into the stream associated to a task.

B. Memoization of executable graphs

There are three steps to using the CUDA Graph API. First, a `cudaGraph_t` object must be created to describe computation either explicitly or by capturing an existing workload. Second, the graph needs to be *instantiated* into an executable graph, which is the most time-intensive part of the process. Finally, the executable graph is launched. To save on the cost of instantiation, CUDA offers a means to update executable graphs using the `cudaGraphExecUpdate()` function that updates an existing executable graph with another graph object that has different parameters but an equivalent topology. Updating is an order of magnitude faster than instantiating a new executable graph, which provides strong incentive for advanced graph reuse strategies. The challenge is devising an automatic strategy that detects and exploits opportunities to reuse and update a graph, as opposed to creating a new one. To do so, we introduce the notion of *epoch*.

An epoch denotes a complete pass through an iterative workload—for example, an iterative solver would check for a stopping criteria at the end of each epoch and conditionally initiate a new one. Given that graphs do not allow loops, a new epoch would need to create, instantiate, and launch a new graph—or, ideally, reuse an existing one. In practice matters can be arbitrarily more complex, for example one epoch may use dynamic control to do different computations than the previous epoch, which makes automatic reuse of graphs difficult.

We devised a simple and effective solution to this problem. We define the non-blocking operation `ctx.fence()` to mark the end of an epoch. As a new epoch starts, the context creates its graph object and looks up a cache of previously-used graphs. An approximate match is given by means of a task summary, and an exact match is indicated by a successful call to `cudaGraphExecUpdate()`. There is no need to define a sophisticated topology comparison mechanism because `cudaGraphExecUpdate()` makes the comparison internally with full access to topology, and failed calls to it are cheap. We found this to be a simple, efficient, and robust mechanism to detect and exploit opportunities for graph reuse.

We show in § VII-D how opportunistic graph reuse improves execution speed by up to 30% for a scientific application on a single device for small problem sizes.

IV. Designing asynchronous libraries with events

CUDASTF not only implements an asynchronous runtime; its internal design also relies on the notion of an *abstract* *event*, which we use to implement a generic asynchronous infrastructure that can be adopted to design other libraries or runtime systems.

A. Abstract events

Runtimes based on asynchronous events are common [7], [8], and their benefits—maximize concurrency opportunities in a non-blocking fashion—are well-studied.

Our design relies on an abstract interface for events instead of a concrete primitive type. The entire core of our library is organized around lists of abstract events. The abstract interface allowed us to implement two backends that use the same code either on top of CUDA streams or graphs, in spite of large differences in how they define events: streams use the `cudaStreamWaitEvent()` API in the stream backend, whereas in the graph backend events are implied by dependencies between nodes. Most of the complex mechanisms implementing our event model are agnostic of the event implementation.

Runtime systems such as StarPU [9] or REALM [8] also describe asynchronous operations with internal events, and build user-facing operations upon these specific data structures with host-side synchronization. When our events are materialized as CUDA events, we can instead entirely delegate these synchronizations to hardware-based mechanisms, thus removing any host-side synchronization.

B. Composing asynchrony

Implementing a runtime system that composes many complex non-blocking operations can be daunting. To ensure that our design is extensible and maintainable, we adopt the following strategy: every asynchronous algorithm takes a list of input events and returns a list of output events that indicate completion of the operation: $l_{\text{out}} = \text{algorithm}(\ldots, l_{\text{int}})$. Event lists may also be passed as a parameter that is updated in-place: $\text{algorithm}(\ldots, l_{\text{inout}})$.

Algorithm 2 Event-based task prologue

```
ready = {}
for each dependency $d_i$ in task dependencies do
    $l_i = \text{enforce\_stf}(d_i, \{\})$
    $l_i = \text{allocate}(d_i, l_i)$
    $l_i = \text{update}(d_i, l_i)$
    $\text{ready} = \text{merge}(\text{ready}, l_i)$
end for
return ready
```

Algorithm 2 illustrates this strategy by showing how we compute the events required to ensure a task is ready for execution. For every data dependency d_i, we enforce data-driven dependencies, then make sure d_i has been allocated, and finally, if needed, issue memory transfers to obtain a valid copy of d_i. The combination of all intermediate lists l_i asynchronously determines when the operation is ready for execution. Applying the algorithm to all operations results in an entirely asynchronous pipeline that never requires synchronization, regardless of the complexity of the overall algorithm.

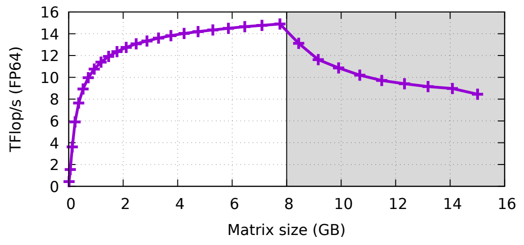

Fig. 3: Capping Cholesky decomposition at 8 GB (one A100)

In contrast, manually composing such operations (e.g., concurrent transfers or allocations) in an application or in a library would be challenging both in terms of programmability and efficiency.

Each step of an algorithm may in turn be a composition of multiple non-blocking sub-steps, which gives the library the leeway to insert additional operations seamlessly. For example, we introduced an extra operation within our data allocation that implements a non-blocking data eviction strategy when the allocation fails. In Fig. 3, a Cholesky decomposition is performed on a single A100 device where the device memory allocator was capped at 8 GB. The memory reclaiming mechanism asynchronously stages data in host memory, which permits solving problems larger than 8 GB. More experiments involving Cholesky decomposition are shown in § VII-C, but this illustrates how our event-based design enabled advanced low-level mechanisms with a modest programming effort without compromising asynchrony.

C. Asynchronous MSI coherency protocol

Fig. 1 shows how to transform a sequence of task into a graph that includes all necessary data transfers and data allocations, regardless of the underlying data layout. We implement this by using a MSI (Modified/Shared/Invalid) coherency protocol that keeps logical data coherent across different data places. Each data instance has a flag indicating whether it is *modified* (this is the only valid copy), *shared* (this is a valid copy, but at least another copy exists), or *invalid* (this is not a valid copy).

Although MSI is a traditional coherency protocol [10], [9], our implementation differs because it makes the MSI state asynchronous: a flag indicates the state the data instance *will* have in the future. For every data instance, two event lists respectively indicate when the data instance can be read or modified. This naturally fits in our design because an operation needing to access a data instance simply has to depend on one or both of these lists.

The coherency protocol is remarkably abstract—it makes no assumption about data layouts (i.e., if we are manipulating a vector vs. a tensor) or the event implementation. The core of CUDASTF will therefore ensure that logical data instances are kept coherent by introducing the necessary transfers and allocations in abstract form, leaving the lowering (copying nodes in a CUDA graph or calls to `cudaMemcpyAsync()`)

```
context ctx;
auto lA = ctx.logical_data<double>(N);
auto lB = ctx.logical_data<double>(N, N);

ctx.parallel_for(lA.shape(), lA.write())->*
  [] __device__(size_t i, auto A) {
    A(i) = 3.0*sin(i);
  };

ctx.parallel_for(lB.shape(), lA.read(), lB.write())->*
  [] __device__(size_t i, size_t j, auto A, auto B) {
    B(i, j) = A(i) * A(j);
  };

ctx.finalize();
```

Fig. 4: A sequence of two kernels using `parallel_for` to operate on 1D and 2D shapes, respectively. Each loop is transformed into a task to transparently implement interdependent kernels.

to the context implementation. Moreover, these internal asynchronous operations are transparently inserted with the overall task graph thanks to the flexible design described in § IV-B.

D. *Managing data lifetime*

CUDASTF automatically manages the lifetime of logical data by using reference counting (C++'s `std::shared_ptr`). A logical data is destroyed when there are no references left to it. During destruction, all data instances (e.g., all resources allocated on different devices) are destroyed. To avoid forcing an expensive synchronization step during deallocation, the destruction process itself is asynchronous. The events associated to the completion of these cleanup operations are stored in a per-context list of events which we denote as *dangling events*. When the context gets destroyed (or when issuing a fence at the end of an epoch), we ensure that all dangling events have completed.

Using dangling events integrates destruction within the overall asynchronous working of CUDASTF. Logical data can be dynamically created and destroyed efficiently without compromising the task submission pipeline. Moreover, memory is safely made available for new allocations as quickly as possible. This is particularly convenient for algorithms that require temporary data. In contrast, most other task-based programming systems typically require a synchronization step to properly destroy a piece of data only after all pending operations have completed.

V. Authoring structured kernels

CUDASTF gives its users the ability to author structured kernels that leverage the task paradigm by means of the `parallel_for` and `launch` primitives as shown in Fig. 4 and 5. Given that the `parallel_for` construct is commonly used in other parallel programming models and frameworks [3], [11], [12], we omit discussing it and instead focus on the `launch` operator.

Using `launch` helps algorithms with complex access patterns. In contrast with `parallel_for`, which independently

```
ctx.launch(par(), lA.read(), lB.write())->*
  [] __device__(auto th, auto A, auto B) {
    for (auto [i, j] : th.apply_partition(shape(B)) {
       B(i, j) = A(i) * A(j);
    }
  };
```

Fig. 5: Dispatching computation across the threads of a flat hierarchy using `par()`, which indicates that threads are not allowed to synchronize.

executes code for every element of a shape, `launch` dispatches code (in form of a lambda function) for collective execution by a structured *thread hierarchy*. The function body is expressed in the CUDA language, so device-side CUDA libraries such as CUB are naturally supported.

Our definition of structured kernels over computing grids leverages three core notions: thread hierarchies, shapes, and shape partitioning, which we discuss in turn below.

1) Thread hierarchies: In order to have threads coordinate within the different levels of the machine hierarchy, we rely on a first-class object to represent a thread hierarchy specification. This concept is similar to that of other programming systems [13], [14]. User code creates a thread hierarchy specification object suitable for the problem domain and passes it to `launch`, which maps the specification to the available hardware. A specification object may contain multiple levels expressed as nested calls to the functions `par()` and `con()`, for example `par(128, con<32>())` for a two-level specification. Parallel levels (introduced by calling `par()`) are groups of threads that cannot synchronize, whereas concurrent levels (introduced by calling `con()`) are allowed to synchronize within the same group. Each level width may be either set statically (e.g., `con<32>()`), chosen dynamically (e.g., `con(32 * n)`), or determined automatically (e.g., `con()`), in which case the library chooses the width to maximize device occupancy. Any combination of static and dynamic sizing is allowed, for example, `con(par(n, con<32>()))` specifies a static width of 32 threads at the innermost level, then dynamic width `n` at the middle level, and lets the library decide how many groups of threads should be created at the top level.

The library automatically maps hierarchy specification objects to the underlying hardware. For example, a two-level specification `con(128, con<32>())` may be mapped on a CUDA kernel with 128 blocks of 32 threads each. The specification object's type informs our library whether the kernel should be launched as a cooperative kernel or not, and whether blocks can be synchronized or not. If needed, user code can force mapping a level at a specific granularity (for example map a specific level exactly on one block in order to use CUDA shared memory) by using scopes (line 5 of Fig. 6). Per-level memory scratchpads are available at the appropriate level in the memory hierarchy in the hardware (shared, device, or managed memory).

Upon calling `launch`, the library transforms the specification into a typed thread hierarchy object (variable `th` on line 9 of Fig. 6). This object's type statically conveys all available information (such as depth or static sizes wherever available), which enables faster code and accurate error reporting.

2) Shapes: CUDASTF defines and uses shapes, which we briefly introduced in § II-A. Shapes are objects that contain full layout and size information about data objects, without including the data itself. At C++ level, shapes are defined generically by means of the primitives they provide. Each shape type defines, among others:

- `size()` to retrieve the full shape size;
- `rank()` to retrieve the dimensionality of the shape;
- `coords_t`, a coordinate type;
- `index_to_coords(i)` that transforms integral `i` into a `coords_t` object;
- a random iterator conforming to the homonym C++ concept.

3) Partitioning shapes across threads: The `apply_partition` member function of the thread hierarchy object applies a partitioning strategy to the shape and returns a sub-shape (which also conforms to the shape interface) with the subset assigned to the calling thread. Thread-level code typically uses the (sub)shape's iterator type in range-based loops (line 3 of Fig. 5).

An explicit partitioning method can be passed as an extra argument to the partition function. By default, for a specification with a single level, we apply a cyclic distribution of the shape, following the classic round-robin pattern used in many CUDA kernels that interleave indices across threads. For specifications with two or more levels, the default strategy is to compose the cyclic distribution at the innermost level with a blocked distribution per level of the hierarchy. In either case, the composition will usually ensure coalesced memory accesses.

The code in Fig. 6 creates a two-level thread hierarchy with the top level sized implicitly and the second level mapped statically to CUDA blocks of 128 threads, allowing generation of efficient code. Explicitly assigning this level to CUDA threads with `hw_scope::thread` ensures that line 16 uses statically-allocated shared memory. The `th.inner()` call on line 15 returns `th` with its outermost level stripped. The `ti.sync()` call on line 23 synchronizes threads at the corresponding level in the hierarchy (one below `th`).

VI. Execution and data places for multiple GPUs

Our library defines execution place objects that allow users to choose where to perform computations. Among these, `exec_place::device(i)` chooses the ith CUDA device (starting at 0) and `exec_place::host` directs computation to the host. A novel aspect of CUDASTF is that it also defines execution places that are collections of other execution places. For example, one can define an execution place that is a *grid* of devices. This additional abstraction is key to running unchanged user code on multiple GPUs.

A grid is a collection of data or execution places that defines a coordinate type and a dimension type. Although we use flat

```
1  context ctx;
2  double X[N]; // ... initialize ...
3
4  auto lX = ctx.logical_data(X);
5  auto spec = par(con<32>(hw_scope::thread));
6  auto where = exec_place::all_devices();
7
8  ctx.launch(spec, where, lX.read(), lsum.rw())->*
9    [=] __device__ (auto th, auto x, auto sum) {
10     double local_sum = 0.0;
11     for (auto [i] : th.apply_partition(shape(x)) {
12       local_sum += x(i);
13     }
14
15     auto ti = th.inner();
16     __shared__ double block_sum[spec.static_width<1>];
17     block_sum[ti.rank()] = local_sum;
18
19     for (size_t s = ti.size() / 2; s > 0; s /= 2) {
20       if (ti.rank() < s) {
21         block_sum[ti.rank()] += block_sum[ti.rank() + s];
22       }
23       ti.sync();
24     }
25     if (ti.rank() == 0) {
26       atomicAdd(&sum(0), block_sum[0]);
27     }
28   };
29
30 ctx.finalize();
```

Fig. 6: Multi-GPU reduction using `launch`

grids, the grid abstraction scales naturally to hierarchical grids of places with a nested coordinate type.

Fig. 6 illustrates how setting the `where` variable on line 6 to `exec_place::all_devices()` localizes the launch construct over a grid consisting of all CUDA devices installed. Similarly, in Fig. 2, replacing `exec_place::device(1)` with `exec_place::all_devices()` would dispatch computation over all devices with no other changes to client code.

A. Localizing execution over multiple places

Given a shape and the shape partitioner abstraction introduced in § V-3, divvying up execution across multiple GPUs is a relatively simple matter. Shapes passed to `parallel_for` are divided into subshapes using the `apply_partition` method of the shape partitioner, so each place in the grid is assigned a disjoint subset of the original iteration space.

The mapping of the thread hierarchy specification also implicitly maps threads over devices: in the reduction example on Fig. 6, using the `par(con<32>(hw_scope::thread))` specification automatically dispatches the outermost level as CUDA blocks across multiple CUDA kernels.

B. Localizing data over multiple places

Dispatching an iteration space over multiple devices is relatively easy, but ensuring that data is correspondingly mapped for maximum locality (proximity to the execution place) is a more challenging problem, for which we propose a novel strategy.

Our solution leverages the CUDA Virtual Memory Management (VMM) API [15], which offers the opportunity to efficiently and transparently assign logical addresses to physical addresses corresponding to all devices in a system. We first reserve a range of virtual memory addresses accessible from all devices, computed so as to cover the entire shape. Given that virtual memory is based on paging, we need to populate the reserved range with fixed-size blocks of memory, each physically allocated on the most appropriate device, chosen by means of a strategy discussed below. All devices may transparently access remote memory blocks, so choosing a bad partitioning affects performance but not correctness. The key to achieving efficiency is, of course, ensuring that non-local memory traffic is minimized.

To select the device on which to physically allocate a page, we unfold the shape in a 1D iteration space and iterate over every elements of the page to find its coordinate in the shape, and thus its affine device in the grid. We collect a histogram of how many samples were assigned to each place, and elect the place with the *most samples* to be responsible for that page.

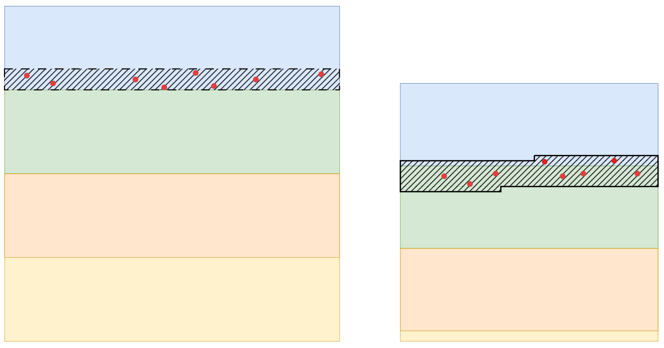
(a) $n = 128$ (b) $n = 100$

Fig. 7: Mapping of a $n \times n$ grid with 4 KB pages and tiles of 32 lines as indicated by different colors. The fourth virtual page is hatched. With $n = 128$, all 8 samples are mapped on the first device, and we observe a perfect match of the partitioner and the virtual memory mapping. With $n = 100$, 2 samples are owned by the first device and the other 6 samples are owned by the second device.

For instance, in Figure 7 we consider a grid of $n \times n$ integers and a 1D grid of P devices. With a *tiled* mapping of 32 consecutive lines of n elements per device, distributed in a round robin fashion, the `owner` method would assign coordinates (i, j) to the $\left(\frac{j}{32} \bmod P\right)^{\text{th}}$ device. With $n = 128$ and 4 KB pages, every page would contain 1024 integers, and the fourth page would span elements 3072 to 4095. Computing the `owner` 1024 times would always result in selecting the first device of the grid. With $n = 100$, the 128 elements with indices from 3072 to 3199 would be mapped on the first device, and the 896 elements from 3200 to 4095 would be mapped on the second device. Our algorithm would therefore map the fourth page on the first device with $n = 128$, and on the second one with $n = 100$.

This also illustrates that in a page-based system, strictly enforcing arbitrary mappings over a fixed granularity is not possible, even for very simple mapping policies. With $n = 100$, the first 128 elements of the fourth page would have been mapped on the first device otherwise.

All systems we have tested have a 2 MB page size for devices. This coarse granularity makes partitioning even more challenging. Electing the owner of each page would require calling the `owner` method 1024 times for a 4 KB page, or 512 K times for a 2 MB page, which is prohibitive. The core of our strategy is *random sampling*: Instead of computing the owner of each element within a page, we randomly pick samples of the coordinates associated to the indices in the page and assign pages to devices in a best-effort manner. For 2 MB pages, we have experimented with different number of samples, and have empirically found that 30 samples per page provide a sufficient accuracy. (Adaptive techniques may further optimize the sampling rate if needed.) This sampling strategy is illustrated by the 8 red dots on Figure 7 used to map each physical page. As mentioned, mismatches between the mapping obtained by sampling and the user-requested mapping impact only performance, not correctness. For all mappings that would fit page boundaries perfectly, our sampling-based strategy is optimal.

Furthermore, to minimize calls to the CUDA VMM API, consecutive pages assigned to the same place are coalesced in a single large physical memory allocation prior to mapping.

This simple strategy works for any mapping, simple or complex. The use of random sampling avoids pathological cases and statically assigns each page to a reasonably appropriate device.

C. *Composite data places*

Combining grids of places with a partitioner object, we define a distinct kind of data place: the *composite data place*. This abstraction allows the cache coherency protocol to work the same over data allocated on one device, or scattered across multiple devices—an important reduction in code size and complexity. Two accesses to a logical data object in the same composite data place (same grid of places and same partitioner) will result in a cache hit and no transfer. Even though more sophisticated strategies exist, we implemented switching from one mapping to another by simply performing a memory transfer between two pieces of contiguous virtual memory, letting the memory management system take care of accessing the appropriate physical pages.

VII. Experimental validation

We now analyze how CUDASTF can be used to fully tap into the potential of GPU-accelerated hardware against a variety of workloads.

The experiments presented here were carried out on two different machine configurations: (a) DGX-A100, a NVIDIA DGX machine equipped with eight A100s and two AMD Epyc 7742 chips (64 cores, 2.25 GHz); and (b) DGX-H100, a NVIDIA DGX machine with eight H100 GPUs and two Intel

Xeon 8480C PCIe Gen5 CPUs with 56 cores at 2.0 GHz. The A100 and H100 GPUs have 80 GB of memory each. All machines are running Ubuntu 22.04 and use NVHPC 24.1.

A. *Task overhead*

The first experiment measures the overhead introduced by CUDASTF when creating tasks and enforcing data dependencies. We evaluated that by setting up dependencies as demanded by a variety of real-world tasks, but leaving all created tasks empty.

Table I shows the average time to submit a task on top of the CUDA stream backend. For each graph topology that corresponds to a typical parallel computing pattern, we report the average time and corresponding standard deviation to execute a task when repeating each experiment 5000 times.

Graph Topology (avg. dependency count)	Avg. task submission time of 5000 runs (μs) DGX-A100	DGX-H100
TRIVIAL (0)	1.64 ± 0.031	1.18 ± 0.020
TREE (0.95)	2.40 ± 0.017	1.83 ± 0.023
FFT (1.4)	2.40 ± 0.030	1.83 ± 0.014
SWEEP (1.5)	2.62 ± 0.018	2.00 ± 0.027
RANDOM (1.75)	2.78 ± 0.015	2.15 ± 0.025
STENCIL (2.4)	2.99 ± 0.010	2.32 ± 0.013

TABLE I: Task cost for different graph topologies

Most of the overhead is related to the underlying CUDA API calls (mainly manipulating events and streams). This explains the lower overhead of H100 compared to A100: the newer architecture has been optimized to reduce such latencies. The TRIVIAL testcase corresponds to independent tasks, so it shows the base overhead. The additional cost of per-data dependencies varies with the topology (refer to the average number of dependencies parenthesized in the leftmost column); we note that the additional overhead is typically lower than the overhead of launching a single CUDA kernel. The low overhead indicates that our tasking system is efficient on tasks with a granularity as small as tens of microseconds. Moreover, CUDA graphs significantly improve the efficiency of fine-grained workloads, and their adoption is simplified by CUDASTF (see §VII-D).

B. *Multi-GPU reduction*

Table II shows the scalability of the reduction algorithm implemented in Figure 6 with the help of the `launch()` function introduced in § V.

On a single device, our implementation achieves 90% of the performance of the highly optimized CUB library with 1608 GB/s against 1796 GB/s. As we increase the GPU count up to 8, with no change to the implementation, we achieve good strong scaling; there is no equivalent multi-GPU implementation in CUB or Thrust.

GPU count	Bandwidth (GB/s)	Speedup
1	1608 ± 1.2	1.00x
2	3240 ± 0.6	2.00x
4	6353 ± 2.8	3.95x
8	11590 ± 7.1	7.21x

TABLE II: Strong Scalability of Sum Reduction by using `launch()` on 1–8 A100 GPUs

C. *Dense Linear Algebra kernels*

Task-based programming models are often used to implement dense linear algebra kernels [1], [16]. Cholesky decomposition is easily split into tasks, whereas manually enforcing efficient asynchronous execution with *look-ahead* techniques would be an algorithmic and implementation burden. We have therefore implemented a Tiled Cholesky decomposition [17]. The implementation is straightforward and consists only of creating one logical data object per tile and calling cuBLAS and cuSOLVER [18] kernels within tasks, leaving all coordination, memory management, and synchronization to our library. To demonstrate the efficiency of this approach, we compare our measurements with cuSolverMg [18], which implements LAPACK kernels over multiple GPUs for a single node. That library is part the NVIDIA CUDA toolkit, so we consider it a well-optimized implementation of Cholesky decomposition.

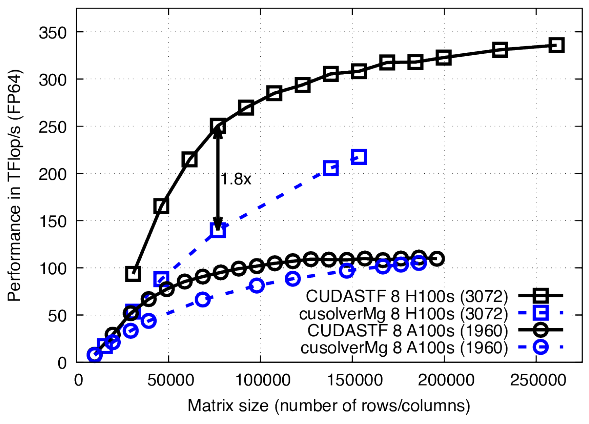

Fig. 8: Cholesky decomposition over 8 GPUs

Fig. 8 depicts the performance of the Cholesky decomposition algorithms over 8 GPUs for real numbers in double precision. We consider both A100 and H100 devices, using block sizes of 1960 and 3072, respectively. On both architectures, we perform better than cuSolverMg. As we rely on compute kernels from cuBLAS and cuSOLVER too, this suggests that the 1D block-cyclic algorithm used in cuSolverMg is less efficient and probably does not implement look-ahead mechanisms, which are automatic with CUDASTF. It is also important to note that no changes or other platform-specific optimizations were made when comparing results.

We measured the contribution of automatic stream pooling and found that it has a marked positive impact on performance. On eight A100s, with a problem of 58800 unknowns, entirely disabling our stream pool results in a 15% performance degradation. Simply using a stream for computation and another one for data transfers is still 8% slower than our implementation.

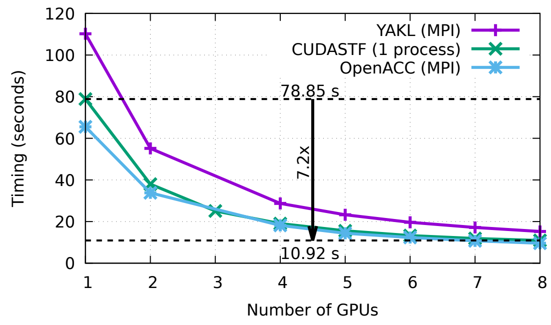

Fig. 9: Strong Scalability of miniWeather (simulation of 10000×5000 elements over 10 seconds). A 7.2x speedup is measured on 8 A100s.

Pooling is also useful in a single-device setup, as we also measured a 5% degradation when using a single stream on one A100 for a problem of 19600 unknowns.

D. miniWeather

To illustrate how CUDASTF can improve the scaling of scientific applications, we used the miniWeather code [19], which performs a simplified simulation of a weather model. It solves 2D Euler equations using a finite volume model on a regular Cartesian grid using a Runge-Kutta time stepping. The original code is only 500 lines long, but it is representative of many scientific applications and the optimization and scaling challenges they face.

Our test code naturally describes all the fields used in the simulation as logical data. The simulation is composed of several dozens nested loops, which we transformed into `parallel_for` constructs operating over these fields.

We also took the opportunity to illustrate how sharing work across GPU and CPU can be effected. The miniWeather program can generate a NetCDF file with the temporal evolution of parameters, operation performed synchronously in the original code. Our library made it easy to move file operations to a task localized on the host, so that I/O is performed in parallel with the computation.

Fig. 9 shows the scalability of the miniWeather benchmark over multiple devices, comparing our implementation with the reference OpenACC+MPI version. It also plots the performance of the YAKL library which, similarly to CUDASTF, provides a `parallel_for` construct that maps C++ lambda functions over CUDA kernels [20]. YAKL is a portability layer that translates nested loops into kernels, but does not handle kernel dependencies or multiple devices. Contrary to OpenACC which is based on a compiler, library-based solutions like CUDASTF and YAKL cannot leverage static code analysis. We respectively measure 65.51, 78.85, and 110.21 seconds for 1 GPU, and 9.59, 10.92, and 15.25 seconds for 8 GPUs. Both OpenACC and YAKL rely on a manually tuned MPI implementation, whereas our kernels are transparently scaled across multiple devices. CUDASTF thus makes it easy to obtain good scaling at the node level.

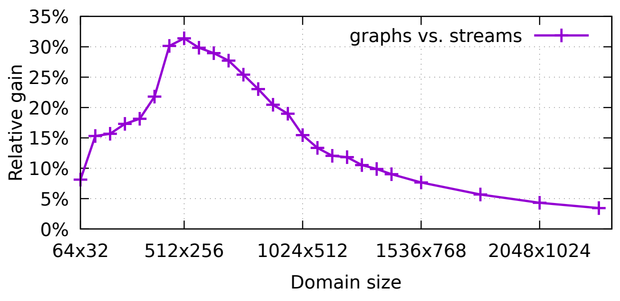

Fig. 10: Performance gains resulting from the use of CUDA graphs on small miniWeather problem sizes on one A100

Kernel launch and synchronization overhead become non-negligible for small problem sizes. CUDA graphs minimize the latency of such back-to-back interdependent kernels. By selecting the graph backend and using the epochs introduced in § III, CUDASTF automatically builds such CUDA graphs, fuses multiple iterations to obtain sufficient granularity, and caches previously instantiated graphs to reduce the overhead of graph API on the host side—all on the same user code as the stream backend.

Fig. 10 shows the performance improvements obtained with the graph backend over the stream backend on a single A100. On very small problems where each iteration only lasts a few milliseconds (domains smaller than 512×256), the management of the CUDA graph becomes non-negligible and the gain is more limited, but as the problem size grows the gains peak at around 30% around domains of size 2048×1024, after which the relative overhead of launching kernels drops progressively.

For small problem sizes, one may expect a parallelized implementation on CPUs to be faster than using GPUs, so we measured that baseline as well. Using the reference OpenMP implementation, to simulate 500×250 elements over 1000 seconds, one CPU core takes 348 seconds and 32 CPU cores take 32.6 seconds. One A100 is still much faster, with 2.33±0.009 seconds for OpenACC, 1.85±0.06 seconds for YAKL, 2.03 seconds using our stream backend, and only 1.39 seconds with the graph backend. YAKL benefits from its simplicity as it leaves dependency analysis to the user, and OpenACC's optimized kernels are penalized by suboptimal asynchrony management and large inter-kernel gaps.

These experiments illustrate the benefits of CUDASTF for scientific applications and show that our methodology to write kernels that span multiple devices is competitive with hand-tuned MPI implementations, in spite of its simplicity.

E. Fully Homomorphic Encryption

Fully homomorphic encryption (FHE) techniques perform computations directly on encrypted data without decryption of any intermediate value, thereby offering maximal privacy protection [21]. Because FHE represents all values (e.g., integers) as high-degree polynomials, it requires huge computing resources, transforming even simple computations (such

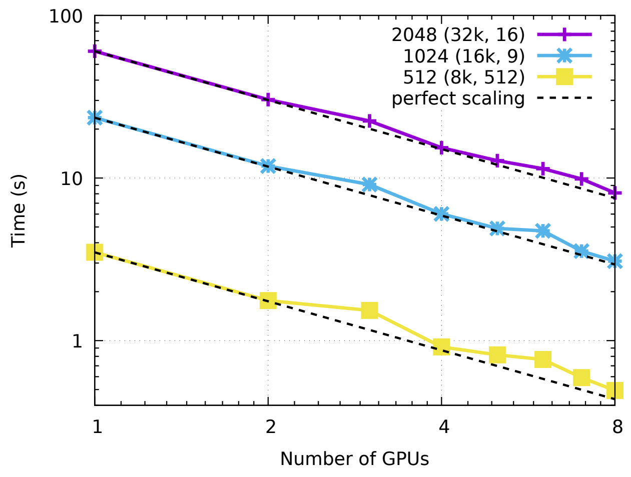

Fig. 11: Strong Scalability of a dot product using the CKKS scheme for different configurations. Each configuration is the vector size, and a pair with polynomial size and moduli count.

as vector dot product) into HPC problems when encrypted. We have used CUDASTF to implement a multi-GPU CKKS scheme [22] using the existing C++ SEAL interface [23], a feat not previously accomplished to our knowledge.

The FHE application is a good illustration of an algorithm combining many pieces of data in complex ways, which can explain the absence of a prior multi-GPU implementation. Manual resource management in this setting would be particularly difficult, as the CKKS scheme combines many operators that affect data size and routinely consume and create temporary buffers. Furthermore, the SEAL interface was not designed with the needs of asynchrony or heterogeneity in mind. CUDASTF automates synchronization and memory management, plus it resolves data-level dependencies automatically, all without changing the SEAL interface (besides a fence mechanism to wait for asynchronously submitted work). Due to its relatively fine task granularity, we used multiple CPU threads to inject tasks over multiple devices in a scalable manner.

Although a dot product algorithm is trivial over unencrypted data and would take a few microseconds for vectors of 2048 elements, its encrypted counterpart generates 475K tasks and takes 60.2s on a A100 (configuration 32K, 16). Fig. 11 shows the scalability of this encrypted algorithm up to 8 A100s. On three configurations, the plot is close to the dashed lines that indicate perfect scaling in this log-log scale, even for small problems.

VIII. Related Work

Task-based runtime systems have been extensively studied on accelerators [24]. Legion [25] is a distributed runtime system that automatically detects complex data access patterns within logical regions to dispatch computation at a large scale. Our charter (single-node multi-GPU) is different as our name-based dependencies do not permit distributing computation over multiple nodes, but have much less overhead. Unlike our work, the REALM [8] runtime system used by Legion requires explicit allocations and transfers. PARSEC [16], [26] is a distributed runtime system that infers data management and dependencies based on data accesses. Its static graph description (PTG) has a limited applicability outside dense linear algebra, but its dynamic task discovery (DTD) interface matches the STF programming model with a C language API. DTD is then used to implement higher-level domain-specific interface such as SLATE [27]. StarPU [9] has scheduling capabilities, but tasks are required to block CUDA streams upon returning, which leaves less opportunity for intra-device concurrency. It cannot be used to generate CUDA graphs.

OpenMP tasks and OmpSs [28] are systems that automatically derive dependencies based on `in/out` data access annotations. In contrast to our approach, they require compiler customization and may not be extended easily for computations that do not fit exactly into directive-based solutions (e.g., when calling into stateful libraries), which limits migration and interoperability. OpenACC's data directives improve code composability because the underlying library detects pieces of data already loaded on a device, but it does not automatically resolve dependencies and does not ensure coherency over multiple devices. Recent work however explores how OpenACC constructs can be orchestrated from OmpSs-2, resulting in a better model composability [29].

Several C++ libraries for GPU programming are available. Kokkos [30] is a portability layer that provides efficient algorithms such as `parallel_for` and `parallel_reduce`. Kokkos allows describing explicit graphs of algorithms with explicit dependencies (e.g., using `when_all`). CUDASTF and Kokkos are similar insofar as they both embrace CUDA's threading model, but also bear significant differences. The `mdspan` artifact is not specific to Kokkos and it has been approved in the C++23 standard (with additions slated for C++26), so we believe it is an important abstraction to build upon. Kokkos predates `mdspan` and corresponding features in the C++ standard, and deviates from them in design. CUDASTF aims to fully integrate with the C++ standard library. Similarly, HPX [31] implements asynchronous tasks based on C++ futures and explicit dependency constraints. Taskflow [5] is a C++ library that allows users to describe a graph of asynchronous operations with a simplified API (compared to CUDA or SYCL). The programmer must still decide which data transfers to include in the graphs and identify dependencies, both of which require a deep understanding of the algorithms. We believe that making these operations explicit hinders composability because a generic algorithm may not know in advance the location of valid data copies. Another difference is that CUDASTF builds on the STF model, which fundamentally infers the dependency graph, whereas in Kokkos and Taskflow graph construction is explicit. Taro [32] is a prescriptive framework for asynchrony in C++ that supports CUDA explicit API calls but does not automate them. SYCL's `buffer` and `accessor` interfaces enable implicit data-driven dependencies with implicit data transfers [7], like the STF paradigm, but they are not yet implemented in a multi-GPU environment to the best of our knowledge.

Phalanx [13] and Agency [14] author structured kernels based on hierarchical descriptions. Integrating our thread hierarchy in our tasking system further allows to asynchronously manage memory resources across different hierarchy levels.

Several strategies exist to dispatch work over multiple GPUs. The AMGE runtime system [33], [34] implements C++ multidimensional arrays to decompose work and distribute data. MAPS-multi [35] proposes a specific semantic expressive enough to author multi-GPU kernels and to map local segments of memory over the different devices. Lightning [36] extends the CUDA kernel API with annotations that allow dispatching kernels over a grid of multiple devices and even multiple nodes. CODA [37] performs static code analysis based on LLVM to estimate data access patterns. Kim et al. [38] apply compilation techniques to generate a runtime analysis step on the host to sample access patterns in an OpenCL kernel. This relates to our sampling-based strategy to allocate localized memory with CUDA VMM. Their compiler-based approach is however complex to implement and supports only simple access patterns and limited control flow.

The IRIS runtime system and its polyhedral compiler can partition problems into multiple tasks to exploit heterogeneous architectures. CUDASTF could also be used as a backend for such polyhedral compilers so that they can concentrate on efficient code generation [39].

IX. Conclusion and Future Work

We have introduced a task-based programming model that automatically infers asynchrony and concurrency based on data accesses, and implemented it over CUDA APIs in the CUDASTF C++ library. It enables efficient execution over single- and multi-GPU machines while automating memory allocation, data coherency, and CUDA graph generation and use (on demand). Careful placement of both data and execution combined with a sampling-based allocation strategy allow dispatching unmodified kernels over multiple GPUs.

Data coherency is currently enforced at the scope of the whole data structure. We are investigating introducing a new partitioning API to manage data subsets independently.

As discussed in § III-A, having the library choose automatically between the stream and the graph backend depending on the available hardware, task, and data at hand is an open problem.

This work concentrates on single-node performance, but we believe it will combine perfectly with asynchronous communication libraries such as the Task-Aware MPI [40] or the NCCL [41] library. Our initial results with the automatic scheduling of kernels using the HEFT strategy [9] are promising. We plan to experiment with more advanced control flows to leverage the recently-introduced conditional CUDA graph nodes. Our multi-GPU placement could be extended on NUMA machines. Though our work does not require any language or compiler customization, it could be used to improve asynchrony in OpenACC implementations. Combined with the MLIR framework, we could introduce powerful optimization such as task fusion. Composable asynchrony is a prominent concern addressed by proposals such as C++ sender-receivers: we believe our methodology could enrich these models to also derive concurrency from data accesses, and ensure data availability beyond the sole use of managed memory.

We recognize that significant open problems and challenges lie ahead in developing performance models and in effectively scheduling tasks for concurrent execution on the same device as well as on multiple devices; we hope that CUDASTF is a step forward and also a stepping stone toward future work.

References

[1] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[2] "NVIDIA CUDA toolkit," [Accessed: 2024-03-31]. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[3] R. Farber, *Parallel programming with OpenACC*. Newnes, 2016.

[4] C. Trott, D. Hollman, D. Lebrun-Grandie, M. Hoemmen, D. Sunderland, H. C. Edwards, B. A. Lelbach, M. Bianco, B. Sander, A. Iliopoulos *et al.*, "P0009r18: MDSPAN," *Programming Language C++ LWG*, July 2022.

[5] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1303–1320, 2022.

[6] "Getting Started with CUDA Graphs," [Accessed: 2024-03-31]. [Online]. Available: https://developer.nvidia.com/blog/cuda-graphs/

[7] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL*. Springer Nature, 2021.

[8] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 263–276.

[9] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15*. Springer, 2009, pp. 863–874.

[10] Y. Solihin, *Fundamentals of parallel multicore architecture*. CRC Press, 2015.

[11] R. Chandra, *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[12] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731514001257

[13] M. Garland, M. Kudlur, and Y. Zheng, "Designing a unified programming model for heterogeneous machines," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. USA: IEEE Computer Society, 2012, p. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2012.48

[14] J. Hoberock and M. Garland, "Agency: A framework for low-level parallelism," 2017, version 0.1.0. [Online]. Available: http://agency-library.github.io/

[15] C. Perry and N. Sakharnykh, "Introducing low-level gpu virtual memory management," 2020, [Accessed: 2024-03-31]. [Online]. Available: https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/

[16] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[17] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel computing*, vol. 35, no. 1, pp. 38–53, 2009.

[18] "cuSOLVER," [Accessed: 2024-03-31]. [Online]. Available: https://docs.nvidia.com/cuda/cusolver/index.html

[19] M. R. Norman and USDOE, "miniWeather," 3 2020. [Online]. Available: https://www.osti.gov/biblio/1631691

[20] M. Norman, I. Lyngaas, A. Bagusetty, and M. Berrill, "Portable C++ code that can look and feel like Fortran code with yet another kernel launcher (YAKL)," *Int. J. Parallel Program.*, vol. 51, no. 4–5, p. 209–230, dec 2022. [Online]. Available: https://doi.org/10.1007/s10766-022-00739-0

[21] C. Gouert, V. Joseph, S. Dalton, C. Augonnet, M. Garland, and N. Tsoutsos, "Accelerated encrypted execution of general-purpose applications," *IACR Cryptol. ePrint Arch.*, vol. 2023, p. 641, 2023.

[22] R. Agrawal and A. Joshi, "The CKKS FHE scheme," in *On Architecting Fully Homomorphic Encryption-based Computing Systems*. Springer, 2023, pp. 19–48.

[23] M. Research, "Microsoft SEAL," 2020, easy-to-use homomorphic encryption library. [Online]. Available: https://github.com/microsoft/SEAL

[24] R. Hoque and P. Shamis, "Distributed task-based runtime systems - current state and micro-benchmark performance," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2018, pp. 934–941.

[25] M. Bauer and M. Garland, "Legate NumPy: Accelerated and distributed array computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356175

[26] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in PaRSEC: a data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3148226.3148233

[27] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "Slate: Design of a modern distributed and accelerated linear algebra library," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–18.

[28] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures." *Parallel Process. Lett.*, vol. 21, no. 2, pp. 173–193, 2011. [Online]. Available: http://dblp.uni-trier.de/db/journals/ppl/ppl21.html#DuranABLMMP11

[29] O. Korakitis, S. G. De Gonzalo, N. Guidotti, J. a. P. Barreto, J. C. Monteiro, and A. J. Peña, "Towards OmpSs-2 and OpenACC interoperation," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 433–434. [Online]. Available: https://doi.org/10.1145/3503221.3508401

[30] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[31] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, 2014, pp. 1–11.

[32] The Taro Developers, "Taro: Task graph-based asynchronous programming system," 2024, accessed: 2024-03-31. [Online]. Available: https://github.com/dian-lun-lin/taro

[33] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W.-m. W. Hwu, "Automatic parallelization of kernels in shared-memory multi-GPU nodes," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 3–13.

[34] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W.-m. Hwu, "Automatic execution of single-GPU computations across multiple GPUs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 467–468. [Online]. Available: https://doi.org/10.1145/2628071.2628109

[35] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, "Memory access patterns: the missing piece of the multi-GPU puzzle," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807611

[36] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, and R. V. van Nieuwpoort, "Lightning: Scaling the GPU programming model beyond a single GPU," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, Jun 2022, pp. 492–503. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS53621.2022.00054

[37] H. Kim, R. Hadidi, L. Nai, H. Kim, N. Jayasena, Y. Eckert, O. Kayiran, and G. Loh, "CODA: Enabling co-location of computation and data for multiple GPU systems," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, sep 2018. [Online]. Available: https://doi.org/10.1145/3232521

[38] J. Kim, H. Kim, J. H. Lee, and J. Lee, “Achieving a single compute device image in opencl for multiple GPUs,” *SIGPLAN Not.*, vol. 46, no. 8, p. 277–288, feb 2011. [Online]. Available: https://doi.org/10.1145/2038037.1941591

[39] J. Kim, S. Lee, B. Johnston, and J. S. Vetter, “Iris: A portable runtime system exploiting multiple heterogeneous programming systems,” in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–8.

[40] K. Sala, J. Labarta, and V. Beltran, “Task-aware MPI (TAMPI) and OpenMP,” in *JLESC Workshop*, 2019.

[41] NVIDIA Corporation, “NVIDIA collective communications library (NCCL),” 2023, [Accessed: 2024-03-31]. [Online]. Available: https://developer.nvidia.com/nccl

Appendix: Artifact Description

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. *Paper's Main Contributions*

This paper introduces a task-based programming model that is based on the Sequential Task Flow programming model over CUDA APIs. It is implemented as a new C++ header-only library named CUDASTF.

- C_1 A new programming model with abstractions for managing asynchrony and placement of work and data;
- C_2 Mechanisms for efficiently distributing user-authored compute kernels across multiple GPUs;
- C_3 A novel robust randomized algorithm that maps and allocates data with arbitrary affinity mappings over multiple devices;
- C_4 Seamless adoption of CUDA graphs;
- C_5 Advanced strategies such as pools of asynchronous streams and custom memory allocation strategies;
- C_6 The first multi-GPU implementation of the CKKS scheme for fully homomorphic encryption (FHE).

The experiments were carried out on two different machine configurations: (a) DGX-A100 is a NVIDIA DGX machine equipped with eight A100s and two AMD Epyc 7742 chips (64 cores, 2.25 GHz); and (b) DGX-H100 is a NVIDIA DGX machine with eight H100 GPUs and two Intel Xeon 8480C PCIe Gen5 CPUs with 56 cores at 2.0 GHz. The A100 and H100 GPUs have 80 GB of memory each. All machines are running Ubuntu 22.04 and NVHPC 24.1.

All experiments were done at least 5 to 10 times to report the standard deviation. When not shown, standard deviation is negligible.

B. *Computational Artifacts*

- A_1 Task overhead evaluation
- A_2 Multi-GPU reduction
- A_3 Cholesky decomposition (CUSOLVERMG, CUDASTF)
- A_4 MiniWeather (OpenMP, YAKL, CUDASTF)
- A_5 Multi-GPU FHE encryption with CKKS scheme

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Table 1
A_2	C_1, C_2, C_3	Table 2
A_3	C_1, C_5	Figure 7
A_4	C_1, C_2, C_3, C_4	Figures 8-9
A_5	C_1, C_6	Figures 8-9

II. Artifact Identification

Even if we envision opening it eventually, the CUDASTF library and its test suite are currently closed source and cannot be disclosed yet.

A. *Computational Artifact A_1*

Relation To Contributions

We considered graph topologies inspired by TaskBench (`https://github.com/StanfordLegion/task-bench`), and implemented a CUDASTF application that generates graphs of empty tasks. This shows that our library (C_1) has a limited overhead.

Expected Results

Task submission time should be around a couple microseconds, showing an overhead typically lower than that of a CUDA kernel submission.

Expected Reproduction Time (in Minutes)

The expected duration of the tests is less than 30 minutes for all tests in Table 1.

Artifact Setup (incl. Inputs)

Hardware: DGX-A100-80GB, DGX-H100-80GB
Software: Ubuntu 22.04 and NVHPC 24.1
Datasets/Inputs: none
Installation and Deployment: This is part of the CUDASTF test suite which is currently closed-source.

Artifact Execution

This is compiled automatically with the test suite of CUDASTF. A binary file is created and simply has to be executed on a machine with a NVIDIA GPU.

Artifact Analysis (incl. Outputs)

For each graph topology, mean time and standard deviations are printed by the application.

B. *Computational Artifact A_2*

Relation To Contributions

We have implemented a multi-gpu algorithm that computes the sum of every elements in a vector. This demonstrates how CUDASTF (C_1) gives the opportunity to author structured kernels (C_2) while transparently taking care of dispatching data across devices (C_3).

Expected Results

We demonstrate a good scalability up to 8 A100 GPUs, and competitive performance on a single GPU with the NVIDIA CUB library which is only available for a single device (CUB is part of NVHPC 24.1).

Expected Reproduction Time (in Minutes)

The expected duration of the tests is less than 30 minutes for all tests in Table 2.

Artifact Setup (incl. Inputs)

Hardware: DGX-A100-80GB
Software: Ubuntu 22.04 and NVHPC 24.1

Datasets/Inputs: none

Installation and Deployment: This is part of the CUDASTF test suite which is currently closed-source.

Artifact Execution

This is compiled automatically when compiling the test suite of CUDASTF. A binary file is created and simply has to be executed on a machine with a NVIDIA GPU.

Artifact Analysis (incl. Outputs)

A shell script is provided in the test suite to run this binary over different numbers of GPUs. It repeats every experiments 10 times to evaluate variability.

C. Computational Artifact A_3

Relation To Contributions

We have implemented a Tiled Cholesky decomposition over multiple devices thanks to CUDASTF (C_1), using CUSOLVER and CUBLAS. We have optionally disabled the stream pools (C_5) to demonstrate their advantage in the last paragraph of VII.C. We compare against the CUSOLVERMG library, which is shipped in NVHPC.

Expected Results

We measure efficiency of our method by reporting synthetic FLOP/s, and show that we significantly outperform the CUSOLVERMG library.

By disabling our stream pools, we report up to 15% of performance loss on 8 GPUs, and 5% on a single device, which shows these mechanisms are necessary.

Expected Reproduction Time (in Minutes)

The expected duration of the tests is less than 10 hours.

Artifact Setup (incl. Inputs)

Hardware: DGX-A100-80GB, DGX-H100-80GB

Software: Ubuntu 22.04 and NVHPC 24.1

Datasets/Inputs: none

Installation and Deployment: This is part of the CUDASTF test suite which is currently closed-source.

Artifact Execution

This is compiled automatically with the test suite of CUDASTF. A binary file is created and simply has to be executed on a machine with a NVIDIA GPU.

Artifact Analysis (incl. Outputs)

A shell script is provided in the test suite to run this binary over different numbers of GPUs. It repeats every experiments 10 times to evaluate variability.

The evaluation of the CUSOLVERMG library is part of another closed-source benchmark that cannot be disclosed, but which makes a straightforward use of CUSOLVERMG.

D. Computational Artifact A_4

Relation To Contributions

We have adapted miniWeather (`https://github.com/mrnorman/miniWeather`) to demonstrate the applicability of CUDASTF (C_1) on scientific applications. It transparently dispatches computations and data over multiple devices (C_2, C_3). We have compared this with reference implementations based on MPI combined with OpenACC or the YAKL C++ library.

For small problem-sizes, we further show that using our CUDA graph backends leads to an improved execution time (C_4).

Expected Results

We compare the strong scalability up to 8 A100s on a problem with 10000x5000 elements, simulating 10s, with the "injection" testcase (Fig. 8). We observe a good strong scalability, and are competitive with the OpenACC compiler-based code generation, and outperform YAKL which is another C++ library-based solution. Both implementations are shipped with miniWeather.

We compare the performance using either CUDA streams or CUDA graphs on smaller problems sizes and report performance gains on a A100 GPU for problems which otherwise suffer from overheads related to a small granularity.

Expected Reproduction Time (in Minutes)

The expected duration of the tests is less than 3 hours.

Artifact Setup (incl. Inputs)

Hardware: DGX-A100-80GB

Software: Ubuntu 22.04 and NVHPC 24.1. OpenMPI version shipped within NVHPC (intra-node execution). miniWeather (main git branch).

Datasets/Inputs: injection testcase

Installation and Deployment: The CUDASTF implementation is part of the CUDASTF test suite which is currently closed-source.

OpenACC and YAKL are compiled using the Makefile provided in miniWeather repository for NVHPC.

Artifact Execution

This is compiled automatically when compiling the test suite of CUDASTF. A binary file is created and simply has to be executed on a machine with a NVIDIA GPU.

Artifact Analysis (incl. Outputs)

All binaries take as argument the domain size, and the number of seconds to simulate. They report execution times. The CUDASTF version accepts an additional command line argument to select the backend. A closed-source shell script was written to repeat these measurements on different number of devices, or for different problem sizes.

E. Computational Artifact A_5

Relation To Contributions

We have used CUDASTF (C_1) to accelerate Microsoft's SEAL library (`https://github.com/microsoft/SEAL`) that implements the CKKS FHE encyption scheme. This complex application involves the composition of numerous complex operations, so that there was no prior multi-GPU implementation (C_6).

Expected Results

We have implemented an algorithm that performs an encrypted DOT product over multiple devices. We apply this algorithm on different cryptographic configurations, and on different number of devices. This should demonstrate the ability to design complex applications with a good scaling over multiple devices, as illustrated on Figure 10.

Expected Reproduction Time (in Minutes)

The expected duration of the tests is less than 3 hours.

Artifact Setup (incl. Inputs)

Hardware: DGX-A100-80GB

Software: Modified version of the SEAL library. Ubuntu 22.04 and NVHPC 24.1.

Datasets/Inputs: none

Installation and Deployment: The sources of this application are closed, and the application itself is not available at the moment either.

Artifact Execution

CUDASTF was added as a submodule of the SEAL project, and new files were added into the original code. The DOT benchmark was added to SEAL examples, and are compiled using the existing cmake facilities. A binary is generated for this DOT benchmark, and takes the cryptographic configuration and the problem size as command-line parameters.

Artifact Analysis (incl. Outputs)

A closed-source shell script was written to execute this DOT benchmark binary over different cryptographic configurations.