TACL: Interoperating asynchronous device APIs with task-based programming models

David Álvarez*†, Kevin Sala*†, Vicenç Beltran*
*Barcelona Supercomputing Center, Barcelona, Spain
†Universitat Politècnica de Catalunya, Barcelona, Spain
E-mail: {david.alvarez,kevin.sala,vbeltran}@bsc.es

Keywords*—*Task-based Programming Models, OmpSS-2, Heterogeneous Computing, High-Performance Computing

I. Extended Abstract

Heterogeneous architectures have become commonplace in modern HPC systems. Eight of the world's top ten supercomputers have accelerators, and the up-and-coming MareNostrum 5 will feature accelerated partitions. However, programming these heterogeneous systems is difficult, as users have to insert data transfer operations, kernel launches and synchronizations manually from the host system to its accelerators. This is even more challenging in distributed heterogeneous systems, as programmers have to coordinate the previous activities with inter-node communications between hosts. This work presents the Task-Aware Ascend Computing Language (TACL), which interoperates with the OmpSs-2 programming model and greatly simplifies kernel execution, data transfers and synchronizations between host and accelerators by naturally leveraging the data-flow execution model of OmpSs-2.

A. OmpSs-2

OmpSs-2 [1] is a task-based parallel programming model developed by the System Tools and Advanced Runtimes (STAR) research group at BSC. OmpSs-2 programs are composed of C, C++ or Fortran code annotated with compiler directives, which can express complex computational patterns using fine-grained data dependencies between tasks.

This programming model has limited support for heterogeneous systems, as implementing device tasks requires extensive changes in the OmpSs-2 compiler and runtime. The latest release of OmpSs-2 has support for CUDA tasks only when used in conjunction with Unified Memory.

B. AscendCL

The Ascend Computing Language [2] encompasses a collection of APIs for users to leverage and operate Huawei's Ascend AI Processors [3]. The Ascend AI lineup is based on Huawei's proprietary Da Vinci architecture, intended initially to accelerate neural network training and inference workloads. However, it is possible to write custom kernels to execute directly on the accelerator hardware and thus repurpose the architecture for HPC workloads. Moreover, the AscendCL library exposes a small subset of BLAS subroutines, such as `gemm` and `gemv`, for matrix multiplication.

C. TACL

TACL is a wrapper around the AscendCL library that implements two key features to allow the interoperability of tasks and the accelerator APIs: (1) TACL transparently handles a pool of ACL streams for tasks to re-use intelligently and (2) TACL can bind the release of a task's dependencies to the completion of all operations in an ACL stream in a non-blocking manner. With these two capabilities, programmers can call ACL functions from their tasks without blocking host CPUs in synchronization functions or risking deadlocks from inter-device communication functions. The idea to deliver non-blocking synchronization by delaying the tasks' dependency release was initially pioneered by TAMPI [4], [5] for MPI communications. In this work, we explore its applicability in heterogeneous computing.

We will use the example in Listing 1 to illustrate the use of TACL in task-based applications. In this example, a producer task runs in the host, followed by an offloading step, which will copy the relevant data to the accelerator, execute a kernel, and then copy the data back to the host. This is a standard workflow for most heterogeneous HPC programs.

However, this example features two problems. First, a new device stream is created and destroyed for every offloading task. This is done because re-using the same stream for all tasks would serialize the execution, but creating streams is generally an expensive operation, which should be avoided. Moreover, in line 13, the execution of this task is suspended until all the operations pending on `stream` finish. However, in this case, the thread executing the offloading task will be blocked in this operation, while it could be busy executing other tasks. Therefore, we are wasting resources on blocking synchronizations.

We can modify the code as showcased on Listing 2 to solve the problems mentioned above. We substituted the highlighted functions with calls to the TACL library. Substitution is one-to-one and existing code can be easily adapted. After these changes, the two previous problems no longer exist. Firstly, TACL manages a pool of re-usable streams, amortizing stream creation costs. As such, `taclrtGetStream` is a much faster operation than creating a new stream from scratch. Secondly, the call to `taclrtSynchronizeStreamAsync` is a non-blocking version of the blocking `aclrtSynchronizeStream` call. The TACL call returns immediately without blocking, and the task can proceed and return. However, the dependencies of

```
#pragma oss task out (A[0;size]) label("producer")
produceDataHost(A, size);

#pragma oss task inout(A[0;size]) label("ascend offload")
{
  aclrtStream stream;
  aclrtCreateStream(&stream);

  aclrtMemcpyAsync(devA, A, ..., ACL_MEMCPY_HOST_TO_DEVICE,
      stream);
  aclopExecute(devA, ..., stream);
  aclrtMemcpyAsync(A, devA, ..., ACL_MEMCPY_DEVICE_TO_HOST,
      stream);

  aclrtSynchronizeStream(stream);
  aclrtDestroyStream(stream);
}

#pragma oss task in(A[0;size]) label("consumer")
consumeDataHost(A, size);
```

Listing 1. Pseudocode of an application offloading computation to an Ascend accelerator

```
#pragma oss task out (A[0;size]) label("producer")
produceDataHost(A, size);

#pragma oss task inout(A[0;size]) label("ascend offload")
{
  aclrtStream stream;
  taclrtGetStream(&stream);

  aclrtMemcpyAsync(devA, A, ..., ACL_MEMCPY_HOST_TO_DEVICE,
      stream);
  aclopExecute(devA, ..., stream);
  aclrtMemcpyAsync(A, devA, ..., ACL_MEMCPY_DEVICE_TO_HOST,
      stream);

  taclrtSynchronizeStreamAsync(stream);
  taclrtReturnStream(stream);
}

#pragma oss task in(A[0;size]) label("consumer")
consumeDataHost(A, size);
```

Listing 2. Code from Listing 1 adapted with TACL

the offloaded task (the `inout(A[0;size])`) will not be released until all pending operations in the stream finish. This allows the OmpSs-2 runtime to re-use the thread to execute other tasks instead of wasting resources waiting for blocking operations.

D. TACL Architecture

The architecture of the TACL library has two main components: the stream pool, managing device streams, and the TACL polling service, as illustrated in Figure 1.

When a program using TACL asks for a stream, one is returned from the pool. Then, upon calling to `taclrtSynchronizeStreamAsync`, the library will use the OmpSs-2 external event API to block task dependency release until all outstanding operations are completed. An event is registered on the device stream to check for the completion of enqueued operations. This event will be completed once all previous stream operations have finished, and thus it enables TACL to check their status. Both the device event and the OmpSs-2 task handle are stored inside the library.

The polling service component periodically checks the completion of events. In OmpSs-2, it is possible to have a recurrent task that has a configurable deadline. Each time the deadline passes, the task will be scheduled at the next possible opportunity. This is how the polling service is implemented

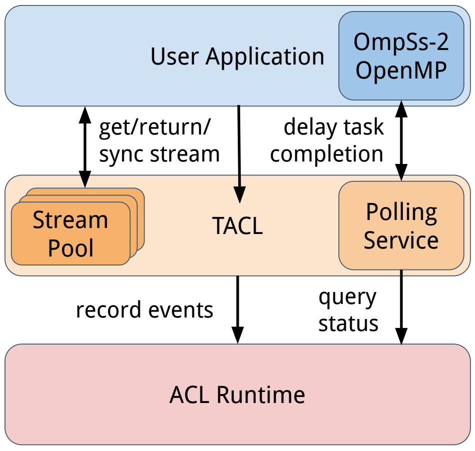

Fig. 1. Architectural diagram of TACL

in TACL. Internally, this polling service queries the status of every outstanding device event. Once an event is completed, TACL uses the OmpSs-2 external event API to notify that the task that initially requested synchronization should no longer wait for its offloaded operations to release its dependencies.

E. Conclusions and Future Work

We have shown how TACL can be leveraged to prevent common problems when writing heterogeneous programs using task-based programming models. Future work is centered around porting existing accelerated applications such as HPL-AI to TACL and evaluating their performance.

II. Acknowledgment

This work has been financially supported by the Huawei-BSC collaboration project.

References

[1] BSC STAR Research Group, "Ompss-2 specification." [Online]. Available: https://pm.bsc.es/ftp/ompss-2/doc/spec

[2] Huawei, "AscendCL Overview." [Online]. Available: https://support.huawei.com/enterprise/en/doc/EDOC1100155021/d63e3d89/ascendcl-overview

[3] ——, "Ascend 910 AI Processor." [Online]. Available: https://e.huawei.com/en/products/cloud-computing-dc/atlas/ascend-910

[4] K. Sala *et al.*, "Improving the interoperability between mpi and task-based programming models," in *Proceedings of the 25th European MPI Users' Group Meeting*, ser. EuroMPI'18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3236367.3236382

[5] K. Sala, S. Macià, and V. Beltran, "Combining one-sided communications with task-based programming models," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 528–541.

David Álvarez received his BSc degree in Informatics Engineering from UPC in 2019 and his MSc degree in Research and Innovation in Informatics at UPC in 2021. Since 2019, he has been with the System Tools and Advanced Runtimes (STAR) group of Barcelona Supercomputing Center (BSC). He is currently a PhD Student and an Associate Part-time Professor in the Computer Architecture Department at UPC.