Research Technical Report ICL-UT-18-03
Data Movement interfaces to support dataflow runtimes

Aurelien Bouteiller George Bosilca Thomas Herault
Jack J. Dongarra
Innovative Computing Laboratory
The University of Tennessee, Knoxville, USA

May 29, 2018

Abstract

This document presents the design study and reports on the implementation of a portable hosted accelerator device support in the PaRSEC Dataflow Tasking at Exascale runtime, undertaken as part of the ECP contract 17-SC-20-SC. The document discusses different technological approaches to transfer data to/from hosted accelerators, issues recommendations for technology providers, and presents the design of an OpenMP based accelerator support in PaRSEC.

1 Introduction

Dataflow runtimes are an emerging abstraction that has demonstrated impressive scalability and performance on distributed systems composed of hybrid/many-core nodes. Research prototypes and current state of the practice rely on ad-hoc implementations of data movement on top of long established technologies, typically CUDA and MPI. These early prototypes however indicate that supporting the dataflow on top of legacy data movement technologies is facing two different main issues: on one hand these abstractions have been designed around the idea that most programs are heavily structured with control-flow and supporting the naturally asynchronous execution of the dataflow model results in bending the two sided semantic to the limit; on the other hand, technologies that are more amenable to supporting dataflow data movement are not enjoying the same consensus that the MPI support enjoys, and one has to deal with a large ecosystem of competing technologies.

The goal of this report is to identify the challenges that data movement in dataflow runtimes brings. We discuss the data movement between node as well as between non-coherent memory (i.e. between GPU and host, or between multiple GPUs, on the same host or not). The goal is to provide recommendations to drive the efforts of emerging unification of execution support (i.e. UCX, OpenMP, Hihat, etc.) so that their new standardized constructs are adequate to support dataflow systems, and to provide recommendations as to which technologies the PaRSEC runtime should be implemented on top of, considering the existing features, the solidity of the standardization process and the size of the users' community, the portability provided by the solution, and the opportunity to impact the standardization process when needed.

2 Background

2.1 Legacy PaRSEC data movement architecture

The current implementation of the accelerator data movement engine in PaRSEC supports NVidia accelerators. When the PaRSEC scheduler encounters a task that elects for execution on an accelerator, one of the PaRSEC execution unit (i.e. an host thread) switches into accelerator support mode. As long as an accelerator is active (it has tasks scheduled onto it) that execution unit keeps organizing the execution of tasks on the accelerators. An execution unit may monitor and manage multiple devices, and will generally manage multiple streams on each device (to improve device occupancy). Two streams per devices are reserved for data transfer operations (one host-to-device stream, one device-to-host stream) while a variable number of streams are dedicated to the execution of compute kernels. Data transfers are issued using the cuda runtime API with asynchronous calls. Stream ordering and the respect of read-write ordering are enforced by using the cuda Event facility. Events are inserted in the data transfer stream to track data movement completion, force delay dependent kernel executions upon the completion of the data transfer, and delay host write operations to source memory asynchronously being transferred to an accelerator. The memory of the accelerator is managed with an LRU buffer that caches as much host memory on the accelerator as possible while evicting (i.e. updating the host copy if needed, then overwriting the content on the accelerator) least used data to make room for actual dependencies. All remote communications (i.e. toward a node across the network) from a task executing on an accelerator are first updating the host-copy of the data, which is then sent from the host memory. Similarly, all receptions from remote nodes are first allocated on the host and then copied to an accelerator if needed.

Xeon Phi accelerators are not supported by the data management engine. Instead, Xeon Phi accelerators are supported in two different modes: 1) a full PaRSEC rank is running on the accelerator, appearing as a full compute node to the PaRSEC hierarchy; each Phi core is allocated a PaRSEC execution unit which executes the PaRSEC scheduler, and a remote communication engine is started on the accelerator to issue MPI operations to communicate with both the host PaRSEC (viewed as a different PaRSEC rank) and remote nodes/accelerators. In this mode, load balancing can be challenging and has to be tweaked by hand. 2) In the "offload mode", the accelerator is not managed by PaRSEC, instead, data transfer are implicit upon calls to the computational kernel (i.e. the MKL library transfers data upon entering the routine and back when leaving the routine.) Performance in this mode is severely limited by redundant data transfer.

In summary, the CUDA engine of PaRSEC is very efficient and achieves a very high fraction of GPU peak performance, insures appropriate and dynamic load balancing between host and accelerator, and can monitor multiple GPUs and streams per GPU thereby reducing the loss of host computational power. However, being heavily reliant on CUDA, it lacks portability and as a consequence support for other types of accelerators may be limited and/or requires significant engineering effort for each new type of accelerator appearing on the market.

3 Survey of GPU-Host data movement technologies

3.1 MPSS/Scif

MPSS/Scif is a low-level communication library that permits exchanging data between the host and an Intel Xeon Phi accelerator (or between Xeon Phi across multiple hosts). The Scif infrastructure provides three different programing interfaces. On one hand it emulates an Infiniband device, programmable through verbs. Although Verbs is a common network technology, its direct programming is low-level and challenging; indeed most users prefer employing higher level communication abstractions that mask most of the complexity inherent to Verbs (i.e. MPI). Not coincidentally, one of the main motivation for the Scif support is to permit the execution of MPI codes spanning multiple Xeon Phi (with host-device, and device to device and device to remote communication possible).

The second interface provided by Scif is a send-recv connected packet interface that permits sending and receiving packets. The protocol is simplistic with no matching or reordering and is best suited for sending small messages.

The third interface permits mapping the memory of a remote Scif endpoint. Then, RMA operations are provided to get and put data into the target memory location. Overall the Scif interface is rather low-level and does not provide advanced features like datatypes or events upon transfer completion. Instead one has to implement a synchronization protocol to signal message completion and flush outstanding operations. While these primitives are sufficient to perform data movement efficiently, in the context of GPU data movement within a dataflow engine, Scif's complexity entails a major engineering expenditure, while at the same time it does not offer great portability prospects, as the only type of accelerators employing it are Intel Xeon Phi.

3.2 CUDA

CUDA has long been the de-facto technology for computing on GPU accelerators. The CUDA toolkit seats at an intermediate level in the software stack, proposing a close-to-the-metal programming abstraction, with explicit data movement routines, explicit management of asynchrony, explicit management of execution streams, etc. A CUDA specific compiler produces the kernel code executing on the accelerator itself, and a runtime API permits managing the data movement and manage the GPU execution from the host. In recent iterations, CUDA has added simplified features for data movements with the Unified Virtual Memory (UVM) technology. In this model, data pointers can be used interchangeably on the host and accelerators, and the CUDA runtime takes care of rendering the target memory available at the place of access. This flexibility comes at the cost of relying on the provided page-faulting mechanism, with its black-box prefetching and caching policy. Although many workloads can enjoy acceptable performance for a decrease in complexity, a dataflow runtime has to track where the data are available in order to drive efficient scheduling decisions, meanwhile the software expense of explicit management of data movements in the dataflow runtime is amortized across multiple users and a range of applications, making the not insignificant performance boost a favorable trade-off. CUDA is extremely efficient, and provides all the necessary features to support a dataflow runtime; for example, it is one of the only models providing some support for datatypes (in its companion cuBLAS library, although limited to strided vectors and matrices with leading dimensions). The major drawback of CUDA comes in the form of its lack of portability, as only NVidia accelerators support CUDA.

3.3 OpenCL

OpenCL was an early attempt at standardizing accelerator kernel code writing. OpenCL is in many respect similar to CUDA; both share a common position in the software stack, and provide the same type of features (explicit data movements, kernel compiler, etc.) Some of the design choices in OpenCL however result in OpenCL performance generally trailing the performance of CUDA, while at the same time being slightly more complex to program. Another issue is that, despite being portable to a large range of devices, OpenCL performance is unpredictable when porting a code from device to device, and often a significant, device specific optimization phase is necessary for simply obtaining acceptable performance. As a consequence of these limitations, it appears that OpenCL has failed to gather a dominant adoption, and is a technology that is loosing traction. On one hand, CUDA provides a similar programing model with better and more predictable performance across supported devices, on the other hand, new entrants like OpenACC or OpenMP provide a significantly simpler programming model, deliver very competitive performance, and promise equal or better portability.

3.4 OpenACC

OpenACC is a directive based compiler extension that permit expressing parallelism with the aim of offloading the computation to an accelerator. Early version of OpenACC provided exclusively OpenMP-style loop directives, but as of version 2.0 of the specification (which is widely available in compilers at this time, including gcc, the Cray compiler, PGI), the OpenACC specification also feature data movement directives, and exposes as well the runtime data movement capabilities through a standardized function API. The API permits defining persistent data on the GPU memory. One limitation is that data movements are synchronous in the 2.0 API. The 2.5 API does provide asynchronous data movement routines that interact neatly with multiple computation streams on the accelerator; however, the support for the 2.5 API is preliminary or absent in most compilers at this time. For example. gcc 6.3 features almost all of the 2.0 API, but the 2.5 API is supported (partially) only in not-yet released gcc 7.x. OpenACC support for the Xeon Phi is partial at this point and not ready for production in any of the major compilers. OpenACC in version 2.5 would however provide a standardized interface that could be swapped in place of CUDA with minor code rework. One unknown is the level of support that Intel is ready to invest in OpenACC. At this point, Intel compilers do not support OpenACC, and NVidia provides a much stronger support for supporting OpenACC over NVidia GPUs in open-source tools like gcc. Meanwhile, Intel is pushing toward OpenMP for its Xeon Phi accelerators.

3.5 OpenMP

OpenMP has adopted in specification 4.0, and further in specification 4.5, a number of features for supporting GPU accelerators. In a large part, most of the features present in OpenACC 2.0 have been imported to OpenMP (sometimes in a slightly different form, or under a different nomenclature, e.g. *target, teams, tasks* versus *device, gang, async* in OpenMP and OpenACC respectively). One of the salient differences between OpenMP and OpenACC is that the former does not provide explicit data movement functions in the runtime interface. Explicit data movement and management is possible, but is entirely managed through compiler directives (e.g. *!omp target [enter] data map*). As a consequence, the rewriting effort to employ directives instead of explicit calls to move data is higher, and the performance impact of having the OpenMP runtime executing task depend clauses (which are the main way of ordering data movement tasks and dependent computation tasks) alongside the dataflow runtime core engine is uncertain. Another concern is when compiling complex applications with multiple compilers using different OpenMP backends, using simple calls to the online OpenMP runtime is expected to result in less side effects than relying on the generated code from directives running on an OpenMP runtime provided by another compiler/linker chain. An extension [**?**] proposed for OpenMP 5.0 may add runtime functions for explicit memory copies, however, full support for OpenMP 4.5 is still several month ahead of us, and it is expected that OpenMP

	OpenACC		OpenMP		
Spec	2.0	2.5	4.0	4.5	5.0
Release	Q2 2013	Q4 2015	Q2 2013	Q4 2015	TBA
Async	directives only	yes	yes (tasks)	yes	yes
RT API	yes (not async)	yes	no	no	yes
gcc 6.3	yes (Cuda)	no	yes	yes	no
icc 17	no	no	yes (Phi)	yes (Phi)	no
cce 8.5.7 (cray)	no	no	yes (Phi)	yes (Phi)	no
PGI 16.10	yes (Cuda)	yes (Cuda)	yes (Cuda)	yes (Cuda)	no

Table 1: Support for accelerator data movement technologies in compilers available on OLCF Titan

5.0 would not appear in compilers for several years. One advantage of OpenMP however is its strong position in the community, which promises better perspective in terms of portability and possibly better outlook for long term support of the technology compared to alternatives.

3.6 UCX

Unified Communication X (UCX) is a communication substrate intended to unify the variety of communication hardware and their driver/access libraries (i.e. GNI, Verbs, etc.) under a single flexible, standardized API. In the context of accelerator data movement, one of the interesting feature of UCX is that it can target accelerator memory as an endpoint. UCX provides one sided operations, registration and atomic operations that can be employed to efficiently transfer memory from the host to a device. In the current implementation, UCX provides a Verbs transport (i.e. for Infiniband networks and Scif Xeon Phi hosted cards). The UCX implementors plan on providing a Cuda transport (i.e. for Nvidia cards access). Open UCX, the open source implementation of UCX, is strongly supported by vendors from both the networking and accelerator communities (Mellanox, Nvidia, IBM, etc.) Although the engineering complexity of using UCX is similar to using Scif, the portability of Open UCX is excellent, and its design facilitates accelerator to accelerator on a single host transfers, as well as remote host (or remote accelerator) access in a natural way (the communication primitives are similar).

3.7 Current level of support

Table 1 summarizes the level of support for OpenMP/OpenACC currently found on a typical DOE environment, the OLCF Titan. The machine permits changing compilers and programming environment by swapping *environment modules*. The only compiler currently supporting OpenACC 2.5 asynchronous data movement functions is the PGI compiler. the GCC compiler version 7.1, soon to be released will support OpenACC 2.5 for Nvidia devices. Most commonly available support for OpenACC at this point is specification 2.0, which does not feature the asynchronous data movement runtime API functions. Support for OpenMP, even modern features of OpenMP is a lot more common across compilers. However, some compilers can generate CUDA specific code while some other can generate Phi specific code. None of the OpenMP specifications available today supports runtime API functions for data movement (only directives).

4 Recommendations for the design of data movement libraries

4.1 asynchrony

Providing asynchronous primitives is of crucial importance for achieving high performance. The dataflow engine of PaRSEC needs to be able to overlap the cost of communicating with the accelerator with 1) concurrent work on the accelerator on independent data already staged in (i.e. multi-stream abstraction), and 2) work and/or scheduling activities on the host processor. It is generally undesirable that the host thread blocks when data transfer occur, because that thread is also responsible for monitoring the progress of concurrent GPU tasks in other streams and/or devices, reporting completion to CPU tasks and schedule remote data transfers (i.e. toward distributed nodes across the network), and scheduling followup tasks for completed GPU tasks on other streams.

Some of the programming models (e.g. OpenMP) provide asynchrony without a fine control of termination events and in the form of directives only. It is generally preferable that the interface provides a library call to issue data movements as this eases issues with program scoping that are inherent to directives.

4.2 datatypes

Complex datatype support is inexistent or limited in almost all frameworks that permit accelerator-host data movement. In the best case, rudimentary datatypes like strided vectors are supported. This entails that in most cases the data movement of complex data layout must pass through a stage-in area on the host memory, where it gets packed, or the host must issue multiple (potentially numerous) data movement orders to transfer individual small contiguous blocks. Better datatype support, with an interface to express complex composite datatypes (like in MPI) could be of interest. At a minimum, support for triangular and trapezoidal 2D matrices is very beneficial.

4.3 accelerator issued communication

In many cases, upon completion of a tasks updating a data on an accelerator, the host is not executing tasks directly dependent on the updated data. It is therefore important to have the capability of updating the copy on other accelerators without passing through host staging (which reduces the available bandwidth and may miss on the opportunity of using special channels like NVlinks connecting two accelerators). CUDA has a good track record of providing accelerator-to-accelerator data transfer, but the case for directive based frameworks is less clear as there is no special operation to issue such a transfer (one has to assume that the underlying runtime can detect that type of data transfer and use the appropriate transfer method). Similarly, remote hosts and accelerators should be able to be targeted directly from the accelerator in order to avoid doubling the amount of data transiting on the bus. This is a feature that is available in hardware centric data movement systems (i.e. Scif, UCX, Cuda with GPUDirect) but is absent directive based systems.

5 Recommendations for the future design of PaRSEC

We recommend thrust in two complimentary directions. One direction is to substitute OpenMP/OpenACC to CUDA, with the expectation that this will improve portability without incurring a significant performance penalty. OpenACC substitution is simpler than OpenMP substitution (because OpenACC has a streaming asynchronous model which is very similar to the CUDA stream abstraction currently in use), but arguably OpenMP is poised to enjoy a larger market penetration, thereby a better claim for portability (to different accelerators and using a wider variety of compilers). In both cases, the initial evaluation with a simplistic proxy-app representing the PaRSEC runtime have pointed that achieving asynchronous data transfer in a non-scoped context (i.e. when the target tasks are not enclosed in the data regions) is non-trivial and requires additional mechanisms to delegate the transfers and/or to track their completion.

The second direction is to further investigate the use of OpenUCX as an host-accelerator communication mechanism. One of the enticing effects would be to bring uniformity to all types of communication in the PaRSEC engine, as the MPI engine is also being transitioned to UCX simultaneously. An impediment to this strategy is the current unavailability of the CUDA channel in the release of OpenUCX.

6 OpenMP device design and implementation in PaRSEC

Based on these recommendations, we have designed an OpenMP target device in the PaRSEC runtime [1]. The goal of the design is to enable the execution of OpenMP target tasks, declared in PaRSEC task bodies (similarly to

[1] `https://bitbucket.org/icldistcomp/parsec/pull-requests/199/add-an-open-mp-target-device`

the CUDA device), with implicit (from the users' perspective) data movement back and forth the hosted accelerator. Thus, in order to avoid duplicate data movement, the PaRSEC device pre-position and tracks the access mode of data on the hosted accelerator. The end-user is simply responsible for declaring an "OPENMP" body for the task class, that body executes on the host and submits asynchronous OpenMP target tasks (typically with `pragma omp target async` directives). No further tracking of data availability is required from the end-user, as when the OPENMP task body executes, the data is already prepositioned on the accelerator.

When the PaRSEC scheduler schedules a task to the hosted accelerator, the necessary data are verified for availability in the target device memory. The same infrastructure is shared between the CUDA and OpenMP devices to track the access mode of the data and their status with respect to the host copy (shared, exclusive, owned, invalid). If the data is not available (or obsoleted by a more recent version), data transfer are scheduled. In order to retain the asynchronous nature of data transfer between the host and the device, the OpenMP device instantiates virtual streams, represented by OpenMP threads in a team. Each transfer is then carried by the OpenMP API memcpy to/from the device which is delegated to one of the team's thread by issuing an enclosing asynchronous OpenMP task. The progress of each stream is tracked by a per-stream variable on which OpenMP depend clauses are inserted by the PaRSEC runtime. When all necessary transfers are complete, the computational, user provided task is inserted by the PaRSEC runtime, and similarly, the completion of the target region is tracked by a depend clause on the per-stream variable.

This general design permits reusing a large portion of the CUDA code, as the overall task lifecycle is thus very similar, and retains the capability of executing data transfer asynchronously with respect to task execution on the target devices.

7 Conclusion

Accelerators are an important part of the HPC ecosystem. Although a trend toward self-hosted accelerators is picking traction, notably with later iterations of the Xeon Phi, in the foreseeable future, some accelerators will continue to employ the host-accelerator model. Some processors (like the ShenWei processor) have some SOC host-accelerator features that can benefit from a careful examination of the memory movement.

Historically, CUDA has provided very efficient data movement for host-accelerator transfers, but has lacked portability. Mature, standardized programing abstractions like OpenMP, OpenACC start to feature the necessary constructs to support efficiently data transfer in a portable way. However, at this point, implementation and availability of these emerging features is still spotty and incomplete, which diminishes the abstract claim of portability. It is however our assessment that the strong community behind these programming models entails that these features will receive satisfactory support in the future.

Based on this initial technological assessment, we were able to design and implement an OpenMP device for the PaRSEC runtime that carries the transfer of data to-from an OpenMP target device. Hence, the PaRSEC runtime is capable of executing OpenMP target tasks on hosted accelerators without imparting a large performance penalty for duplicate data transfer, and without imparting a cumbersome manual management of memory from the end-user task code.

Acknowledgements

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

A Appendices

This appendix presents the "PaRSEC runtime mockup" code employed to evaluate the matching between OpenMP, OpenACC features and the operating modes of the PaRSEC runtime. The code employs directive and API based access to OpenMP and OpenACC data movement routines and mimics the task synchronization behavior of the PaRSEC runtime in a smaller setting.

The real implementation of the OpenMP PaRSEC device can be found in the public pull request 199 [2]

A.1 CUDA to OpenACC and OpenMP substitution in a "PaRSEC engine" proxy-app

```c
/*
 * Copyright (c) 2017-2018 The University of Tennessee and The University
 *                         of Tennessee Research Foundation.  All rights
 *                         reserved.
 */


#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <omp.h>
#include <openacc.h>
#include <cuda.h>
```

[2] https://bitbucket.org/icldistcomp/parsec/pull-requests/199/add-an-open-mp-target-device

```c
#include <cuda_runtime_api.h>
#include <cublas_v2.h>

typedef enum { copy_cublas, copy_acc, copy_omp, copy_none } copy_transport_t;
static char* pcopy_transport[] = { "cublas", "acc", "omp", "none" };
typedef union { cudaStream_t cuda; int acc; int omp; } copy_stream_t;
typedef union { cudaEvent_t cuda; int acc; int omp; } copy_sync_t;

int main(void) {
    int cuda_version;
    double start, cpy, end;
    double *A=NULL,
           *B=NULL,
           *C=NULL,
           *dA=NULL,
           *dB=NULL,
           *dC=NULL;
    int m=2000,
        n=2000,
        k=2000,
        i, j, r;
    copy_transport_t copy_transport = copy_cublas;
    copy_stream_t copy_stream;
    copy_sync_t copy_sync;


    // always using cublas to run the compute, so this init unconditional
    cublasStatus_t st;
    cublasHandle_t handle;
    cudaError_t ce;
    st = cublasCreate(&handle);
    assert( CUBLAS_STATUS_SUCCESS == st );

    // get the cublas stream to post synchronizes
    cudaStream_t cublas_stream;
    st = cublasGetStream(handle, &cublas_stream);
    assert( CUBLAS_STATUS_SUCCESS == st /* cublasGetStream */ );

    st = cudaRuntimeGetVersion(&cuda_version);
    assert( cudaSuccess == st /* cudaRuntimeGetVersion */ );
    printf("OpenACC version %d ~~ CUDA version %d\n\n", _OPENACC, cuda_version);

    // host allocation and filling
    A = malloc( sizeof(double)*m*k );
    B = malloc( sizeof(double)*k*n );
    C = malloc( sizeof(double)*m*n );

    for( i = 0; i < m; i++ ) for( j = 0; j < k; j++ )
        A[i+j*m] = (i+j)/(m+n+1.);
    for( i = 0; i < k; i++ ) for( j = 0; j < n; j++ )
        B[i+j*m] = (i+j)/(m+n+1.);
    for( i = 0; i < m; i++ ) for( j = 0; j < n; j++ )
        C[i+j*m] = (i+j)/(m+n+1.);

  for( copy_transport = copy_cublas; copy_transport != copy_none; copy_transport++ ) for( r
    printf("Starting a %s copy to feed a cuBLAS...\n", pcopy_transport[copy_transport]);
    start = omp_get_wtime();
    // device allocation
    switch( copy_transport ) {
    case copy_cublas:
        ce = cudaStreamCreate(&copy_stream.cuda);
        assert( cudaSuccess == ce /* cudaStreamCreate */ );
        ce = cudaMalloc(&dA, sizeof(double)*m*k);
        assert( cudaSuccess == ce /* cudaMalloc */ );
        ce = cudaMalloc(&dB, sizeof(double)*k*n);
        assert( cudaSuccess == ce /* cudaMalloc */ );
        ce = cudaMalloc(&dC, sizeof(double)*m*n);
        assert( cudaSuccess == ce /* cudaMalloc */ );
        break;
    case copy_acc:
#if _OPENACC > 0
        dA = acc_malloc( sizeof(double)*m*k );
        dB = acc_malloc( sizeof(double)*k*n );
        dC = acc_malloc( sizeof(double)*m*n );
#endif
        assert( NULL != dA );
        assert( NULL != dB );
        assert( NULL != dC );
        break;
#if USE_OMP_DIRECTIVES
    case copy_omp: {
#pragma omp target enter data map(alloc: A[:m*k], B[:k*n], C[m*n])
        break; }
#else
    case copy_omp:
        dA = omp_target_alloc( sizeof(double)*m*l );
        dB = omp_target_alloc( sizeof(double)*k*n );
        dC = omp_target_alloc( sizeof(double)*m*n );
        assert( NULL != dA );
        assert( NULL != dB );
        assert( NULL != dC );
        break;
```

```
#endif /*USE_OMP_DIRECTIVES*/
    default:
        assert( 0 /* invalid transport */ );
    }

    // copy A to dA
    switch( copy_transport ) {
    case copy_cublas:
        cublasSetMatrixAsync(m, k, sizeof(*A), A, m, dA, m, copy_stream.cuda);
        cublasSetMatrixAsync(k, n, sizeof(*B), B, k, dA, k, copy_stream.cuda);
        cublasSetMatrixAsync(m, n, sizeof(*C), C, m, dC, m, copy_stream.cuda);
        ce = cudaEventCreate(&copy_sync.cuda);
        assert( cudaSuccess == ce /* cudaEventCreate */ );
        ce = cudaEventRecord(copy_sync.cuda, copy_stream.cuda);
        assert( cudaSuccess == ce /* cudaRecordEvent */ );
        break;
    case copy_acc:
        copy_stream.acc = 1;
#if _OPENACC >= 201510
        acc_memcpy_to_device_async( dA, A, sizeof(*A)*m*k, copy_stream.acc );
        acc_memcpy_to_device_async( dB, B, sizeof(*A)*k*n, copy_stream.acc );
        acc_memcpy_to_device_async( dC, C, sizeof(*A)*m*n, copy_stream.acc );
#else
#warning "acc_memcpy_to_device_async is not available"
        acc_memcpy_to_device( dA, A, sizeof(*A)*m*k );
        acc_memcpy_to_device( dA, B, sizeof(*A)*k*n );
        acc_memcpy_to_device( dC, C, sizeof(*C)*m*n );
#endif
        break;
#if USE_OMP_DIRECTIVES
    case copy_omp: {
#pragma omp target update to(A[:m*k], B[k*n], C[m*n]) nowait depend(out:A, B, C)
        break; }
#else
    case copy_omp:
        copy_stream.omp = 0;
#pragma omp task shared(copy_stream.omp)
{
        omp_target_memcpy( dA, A, sizeof(*A)*m*k );
        omp_target_memcpy( dA, B, sizeof(*A)*k*n );
        omp_target_memcpy( dC, C, sizeof(*C)*m*n );
#pragma omp atomic
        copy_stream.omp = 1;
}
        break;
#endif /*USE_OMP_DIRECTIVES*/
```

```
    default:
        assert( 0 /* invalid transport */ );
    }

progress_loop:
    // compute on host - run the parsec event loop, etc

    // wait for copy to dA to finish
    switch( copy_transport ) {
    case copy_cublas:
        ce = cudaStreamWaitEvent(cublas_stream, copy_sync.cuda, 0);
        assert( cudaSuccess == ce /* cudaStreamWaitEvent */ );
        break;
    case copy_acc:
        // Still need to know the final kernel is Cuda based if we want to
        // transfer dependency resolution to the device. Note we do not do
        // what the cublas code above shows, in PaRSEC: we do something
        // similar to that acc code here.
        // One may use acc_cuda_get_stream and friends to transmit the event
        // to cublas, if one knows that the kernel will execute on the cublas
        // stream.
        //
        // Also note that if the progress loop keeps queuing more work, we
        // will try to check their completion as well, there are no event
        // markers in acc to avoid this caveat. An option is to use multiple
        // async to post transfers and test them to see if the dependent
        // task can start. One may have to wait that an async becomes
        // available from the limited pool of available async before
        // enqueuing transfers, so that's a complication.
#if _OPENACC >= 201510
        if( !acc_async_test(copy_stream.acc) ) {
            goto progress_loop;
        }
#endif
        break;
#if USE_OMP_DIRECTIVES
    case copy_omp: {
        // Delegate the resolution of dependencies to the target device with
        // depend clauses on the actual input. This is not similar to the
        // PaRSEC runtime behavior as this will trigger the issue of
        // accelerator tasks before their input data are available, while
        // in PaRSEC a task is scheduled only when the data is pre-staged.
#pragma omp target nowait depend(in: A, B, C)
        { /* just a sync */ }
        break; }
#else
```

```
    case copy_omp:
        // Use pseudo-events (implemented with atomic ops on a control variable)
        // to track memcpy to target completion
        while(!copy_stream.omp) {
#pragma omp taskyield
        }
        break;
#endif /*USE_OMP_DIRECTIVES*/
    default:
        assert( 0 /* invalid transport */ );
    }
    cpy=omp_get_wtime();
    printf("  Copy is done %s, took %g\n", pcopy_transport[copy_transport], cpy-start);
    // run a cublas
    double alpha=1.0,
           beta=1.0;
    //TODO: Do a LLT/QR so that results can be verified
    cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, m, n, k, &alpha, dA, m, dB, k, &beta, dC,
    cudaStreamSynchronize(cublas_stream);
    end = omp_get_wtime();
    printf("  Cublas executed (copy %s), took %g total, %g since copy\n", pcopy_transport[co

    // retreive dC->C when ready. Should look
    // similar to above, except clearly one has to know what async to look in
    // acc.
    //
    switch( copy_transport ) {
    case copy_cublas:
        ce = cudaFree(dA); dA=NULL;
        assert( cudaSuccess == ce /* cudaFree */ );
        ce = cudaFree(dB); dB=NULL;
        assert( cudaSuccess == ce /* cudaFree */ );
        ce = cudaFree(dC); dC=NULL;
        assert( cudaSuccess == ce /* cudaFree */ );
        ce = cudaStreamDestroy(copy_stream.cuda);
        assert( cudaSuccess == ce /* cudaStreamDestroy */ );
        break;
    case copy_acc:
#if _OPENACC > 0
        acc_free(dA); dA=NULL;
        acc_free(dB); dB=NULL;
        acc_free(dC); dC=NULL;
#endif
        break;
#if USE_OMP_DIRECTIVES
    case copy_omp: {
```

```
#pragma omp target exit data map(delete: A[:m*k], B[:k*n], C[m*n])
        break; }
#else
    case copy_omp:
        omp_target_free(dA); dA=NULL;
        omp_target_free(dB); dB=NULL;
        omp_target_free(dC); dC=NULL;
        break;
#endif /*USE_OMP_DIRECTIVES*/
    default:
        assert( 0 /* invalid transport */ );
    }
  } /*for copy_transport*/

    // cleanup
    // todo: copy_stream and copy_sync cleanup
    st = cublasDestroy(handle);
    assert( CUBLAS_STATUS_SUCCESS == st );

    return 0;
}
```