The Journal of Supercomputing (2020) 76:5117–5138
https://doi.org/10.1007/s11227-019-02829-2

dOCAL: high-level distributed programming with OpenCL and CUDA

Ari Rasch[1] · Julian Bigge[1] · Martin Wrodarczyk[1] · Richard Schulze[1] · Sergei Gorlatch[1]

Published online: 30 March 2019

Abstract

In the state-of-the-art parallel programming approaches OpenCL and CUDA, so-called host code is required for program's execution. Efficiently implementing host code is often a cumbersome task, especially when executing OpenCL and CUDA programs on systems with multiple nodes, each comprising different devices, e.g., multi-core CPU and graphics processing units; the programmer is responsible for explicitly managing node's and device's memory, synchronizing computations with data transfers between devices of potentially different nodes and for optimizing data transfers between devices' memories and nodes' main memories, e.g., by using pinned main memory for accelerating data transfers and overlapping the transfers with computations. We develop distributed OpenCL/CUDA abstraction layer (dOCAL)—a novel high-level C++ library that simplifies the development of host code. dOCAL combines major advantages over the state-of-the-art high-level approaches: (1) it simplifies implementing both OpenCL and CUDA host code by providing a simple-to-use, high-level abstraction API; (2) it supports executing arbitrary OpenCL and CUDA programs; (3) it allows conveniently targeting the devices of different nodes by automatically managing node-to-node communications; (4) it simplifies implementing data transfer optimizations by providing different, specially allocated memory regions, e.g., pinned main memory for overlapping data transfers with computations; (5) it optimizes memory management by automatically avoiding unnecessary data transfers; (6) it enables interoperability between OpenCL and CUDA host code for systems with devices from different vendors. Our experiments show that dOCAL significantly simplifies the development of host code for heterogeneous and distributed systems, with a low runtime overhead.

Keywords OpenCL · CUDA · Host code · Distributed system · Heterogenous system · Interoperability · Data transfer optimization

✉ Ari Rasch
a.rasch@wwu.de; a.rasch@uni-muenster.de

Extended author information available on the last page of the article

1 Motivation and related work

We consider modern distributed, heterogeneous systems comprising one or several nodes equipped with multi-core CPUs and accelerator devices such as graphics processing units (GPUs). The state-of-the-art approaches to programming such systems are OpenCL and CUDA. A common problem of these approaches is that they require the programmer to implement the so-called *host code* for executing OpenCL and CUDA device code (a.k.a. *kernel*).

Implementing host code is often a tedious task: boilerplate low-level commands are required, e.g., for allocating memory on the target device and for performing data transfers between the device's memory and main memory. Especially when targeting complex systems which consist of multiple nodes each equipped with different devices, e.g., two or more GPUs and CPU, host code's implementation becomes cumbersome and error-prone even for an experienced programmer: she has to manage the memories of different devices which may belong to different nodes, as well as manage the nodes' main memories, and she has to explicitly synchronize data transfers with kernel computations in different devices.

Host code development becomes additionally complex for systems with devices from different vendors: e.g., non-NVIDIA devices are usually programmed using OpenCL, while NVIDIA devices mostly rely on CUDA for performance reasons [32] and because CUDA provides better profiling and debugging tools [37]. Therefore, to program a system with both NVIDIA and non-NVIDIA devices, the programmer has to mix CUDA and OpenCL host code and explicitly program the communication between CUDA and OpenCL data structures, e.g., to combine the results of different GPUs (computed using CUDA) on a multi-core CPU using OpenCL.

To achieve high performance, the host code must be optimized: using the *pinned* and *unified memory* (a.k.a. *zero-copy buffer* in OpenCL) can accelerate, hide or even avoid data transfers between devices' memories and the main memory [23, 38]. However, using these specially optimized memory regions requires from the programmer a detailed knowledge about low-level OpenCL/CUDA host code functions and flags, thus making host code even more cumbersome.

There are several successful high-level approaches to simplify the programming process for OpenCL and CUDA host code. However, these focus on only particular host programming challenges, e.g., only data transfer optimizations or only OpenCL or CUDA, respectively, and thus, they are restricted to only specific application classes. For example, skeleton approaches [2, 5, 14, 15, 49] simplify host code programming, e.g., by managing and optimizing memory management, but they are restricted to OpenCL and CUDA programs that can be expressed via specifically provided parallel patterns (a.k.a. algorithmic skeletons [16]). Directive-based approaches such as OpenACC [55], OpenMP [8] and OpenMPC [30] automatically generate the OpenCL and/or CUDA host code, but they also automatically generate and execute the kernel code, thereby preventing the programmer from hand-optimizing the kernels as often required for

highest performance [32]. The systems built on top of OpenCL–Maat [42], ViennaCL [46], Maestro [47], Boost.Compute [51] and HPL [54]—simplify executing user-defined OpenCL kernels by providing a high-level API for host programming; unfortunately, they provide no support for CUDA. The pyOpenCL and pyCUDA approaches [28] enable implementing OpenCL/CUDA host code in the simple-to-use Python programming language, but they still require from the programmer to explicitly deal with low-level details, such as data transfers and synchronization. Multi-device controllers [33], PACXX [18], SYCL [43] and OmpSs [13] allow conveniently programming OpenCL and/or CUDA-capable devices, while StarPU [6], PEPPHER [9] and ClusterSs [52] focus on simplifying task scheduling over multi- and many-core devices. However, these approaches do not support data transfer optimizations, e.g., overlapping data transfers with computations. Moreover, the majority of the related work targets only single-node systems, thereby missing the full performance potential of modern HPC systems with multiple nodes. The SnuCL [27], rCUDA [12], dOpenCL [26] and LibWater [17] approaches target multi-node systems, but they extend the low-level OpenCL or CUDA user API, rather than providing high-level abstraction to ease host programming, e.g., by automatically performing data transfers and managing synchronization.

We develop the Distributed OpenCL/CUDA Abstraction Layer (dOCAL)—a high-level approach to OpenCL and CUDA host code programming. dOCAL is implemented as a C++ library, and it combines major advantages over the state-of-the-art approaches: (1) it simplifies implementing both OpenCL and CUDA host code by automatically managing low-level details such as data transfers and synchronization; (2) it allows executing arbitrary, user-provided OpenCL and CUDA kernels; (3) it enables conveniently targeting the devices of multi-node systems by automatically managing the node-to-node network communication; (4) it simplifies data transfer optimizations by providing different, specially allocated memory classes, e.g., pinned main memory for overlapping data transfers with computations; 5) it optimizes memory management by automatically detecting and avoiding unnecessary data transfers; 6) it enables interoperability between OpenCL and CUDA host code by automatically handling the communication between OpenCL and CUDA data structures and by automatically translating between the OpenCL and CUDA kernel programming languages.

Moreover, dOCAL is compatible with existing OpenCL and CUDA libraries, it supports interconnecting with auto-tuning systems, and it allows conveniently profiling the runtime behavior of OpenCL and CUDA programs.

The remainder of the paper is organized as follows. In Sect. 2, we illustrate the usage of our dOCAL library, using a simple single-node example. Afterward, in Sect. 3, we demonstrate dOCAL's OpenCL-CUDA interoperability feature. In Sect. 4, we show how dOCAL is used for multi-node systems, and in Sect. 5, we present dOCAL's data transfer optimizations. After presenting dOCAL's advanced features in Sect. 6, we present our experimental results in Sect. 7. Section 8 concludes our paper.

2 Illustration of dOCAL

To illustrate the API design of our dOCAL library and its usage, we use a simple, demonstrative example: summing all elements of a vector (a.k.a. *reduction*) in CUDA using system's GPUs.

2.1 Using dOCAL for deploying CUDA host code

Listing 1 shows the original NVIDIA's CUDA *reduction* kernel provided in [37]. The kernel takes as input the vector `d_Input` of *N* floating point numbers (line 2), and it computes in parallel a partial sum of the vector's elements—one result per started thread (lines 4–9); the results are stored in `d_Result` (line 11) and have to be combined (summed up) to the final result in the host code after kernel's execution.

```
1  __global__ static
2  void reduceKernel(float *d_Result, float *d_Input, int N)
3  {
4      const int      tid = blockIdx.x * blockDim.x + threadIdx.x;
5      const int threadN = gridDim.x * blockDim.x;
6      float          sum = 0;
7
8      for (int pos = tid; pos < N; pos += threadN)
9          sum += d_Input[pos];
10
11     d_Result[tid] = sum;
12 }
```

Listing 1 NVIDIA's original CUDA kernel for reduction taken from [37].

```
int main(int argc, char **argv)
{
    // initialization
    int i, j, gpuBase, GPU_N;
    cudaGetDeviceCount(&GPU_N);

    /* ... prepare input data ... */

    // Allocate device and host memory
    for (i = 0; i < GPU_N; i++) {
        cudaSetDevice(i));
        cudaStreamCreate(&plan[i].stream));
        cudaMalloc((void **)&plan[i].d_Data, plan[i].dataN *
            sizeof(float));
        cudaMalloc((void **)&plan[i].d_Sum, ACCUM_N * sizeof(float
            ));
        cudaMallocHost((void **)&plan[i].h_Sum_from_device,
            ACCUM_N * sizeof(float));
        cudaMallocHost((void **)&plan[i].h_Data, plan[i].dataN *
            sizeof(float));
        for (j = 0; j < plan[i].dataN; j++)
          plan[i].h_Data[j] = (float)rand()/(float)RAND_MAX;
    }
    // Perform data transfers and start device computations
    for (i = 0; i < GPU_N; i++) {
        cudaSetDevice(i);
        cudaMemcpyAsync(plan[i].d_Data, plan[i].h_Data, plan[i].
            dataN * sizeof(float), cudaMemcpyHostToDevice, plan[i
            ].stream);
        reduceKernel<<<BLOCK_N, THREAD_N, 0, plan[i].stream>>>(
            plan[i].d_Sum, plan[i].d_Data, plan[i].dataN);
        cudaMemcpyAsync(plan[i].h_Sum_from_device, plan[i].d_Sum,
            ACCUM_N *sizeof(float), cudaMemcpyDeviceToHost, plan[i
            ].stream);
    }
    // combine GPUs' results
    for (i = 0; i < GPU_N; i++) {
        float sum;
        cudaSetDevice(i);
        cudaStreamSynchronize(plan[i].stream);
        sum = 0;
        for (j = 0; j < ACCUM_N; j++)
          sum += plan[i].h_Sum_from_device[j];
        *(plan[i].h_Sum) = (float)sum;
        cudaFreeHost(plan[i].h_Sum_from_device);
        cudaFree(plan[i].d_Sum);
        cudaFree(plan[i].d_Data);
        cudaStreamDestroy(plan[i].stream);
    }

    /* ... Compare GPU and CPU results ... */
}
```

Listing 2 Excerpt of NVIDIA's original CUDA host code taken from [37] for executing the CUDA reduction kernel (shown in Listing 1). Boilerplate low-level commands make the development of host code tedious and cumbersome.

Listing 2 shows an excerpt of the CUDA host code for executing the reduction kernel (of Listing 1) cooperatively on all of system's CUDA-capable devices; this

code is provided by NVIDIA in [37]. It comprises boilerplate low-level functions, such as `cudaMalloc` and `cudaMallocHost` for allocating device and main memory (lines 13–16), `cudaMemcpyAsync` for performing data transfers between main memory and devices' memories (lines 23 and 25), `cudaStreamCreate` for creating the so-called *CUDA streams* (line 12)—they are required to coordinate data transfers and the execution of kernels on the CUDA devices—and `cudaStreamSynchronize` for synchronization (line 31).

Listing 3 demonstrates, for the sake of comparison, the dOCAL host code that is equivalent to the NVIDIA's low-level host code in Listing 2. dOCAL is implemented as a C++ header-only library, thereby freeing the user from the burden of compiling, packaging and installing; to use dOCAL, the user only includes the corresponding header file (line 1) and implements a C++ program which performs four major steps, 1–4, in the following.

```
1  #include "docal.hpp"
2
3  int main()
4  {
5    int N = /* arbitrary chunk size */;
6
7    // 1. choose devices
8    auto devices = docal::get_all_local_devices<CUDA>();
9
10   // 2. declare kernel
11   docal::kernel reduction = cuda::source(/* reduction kernel */);
12
13   const int GS = 32, BS = 256;
14
15   // 3. prepare kernels' inputs
16   docal::buffer<float> in ( N      * devices.size() );
17   docal::buffer<float> out( GS*BS * devices.size() );
18
19   std::generate( in.begin(), in.end(), std::rand );
20
21   // 4. start device computations
22   for( auto& dev : devices )
23     dev( reduction                                    )
24        ( dim3(GS), dim3(BS)                           )
25        ( write(out.begin() + dev.id()* GS*BS, GS*BS ),
26          read (in.begin()  + dev.id()* N     , N     ),
27          N                                            );
28
29   auto res = std::accumulate( out.begin(), out.end(), std::plus<
         float>() );
30
31   std::cout << res << std::endl;
32 }
```

Listing 3 The dOCAL host code for executing the CUDA reduction kernel in Listing 1. As compared to low-level CUDA host code (Listing 2), dOCAL frees the user from using boilerplate low-level commands, thus making the host code simpler.

1. Choose devices In dOCAL, system's devices are represented as objects of the class `docal::device`; they allow the user to conveniently perform device computations, as we demonstrate later in Step 4.

In our example, we execute the reduction kernel on all of system's CUDA-capable devices. For this, we use the function `get_all_local_devices<CUDA>` (Listing 3, line 8) which constructs one `docal::device<CUDA>` object per system's CUDA device, and it returns the constructed device objects in form of a C++ vector. For constructing the device objects, dOCAL automatically interacts with the low-level CUDA API to automatically determine and manage the target devices' CUDA ids (Listing 2, lines 4–5, 11, 22, 30) and to initialize and handle the low-level CUDA streams (lines 12, 23–25, 31, 39)—per default, 32 streams per device, thus enabling simultaneously executing multiple kernels on a device and consequently a better hardware utilization (a.k.a. *Hyper-Q* in NVIDIA terminology [40]). The device id and CUDA streams are encapsulated in the dOCAL device objects to hide them from the user.

The user can also choose a specific CUDA device. For this, she initializes a `docal::device<CUDA>` object by using either (1) device's name as string, e.g., `"Tesla K20"`, (2) its numerical device id or (3) some of its device properties, e.g., the first found device with support for double precision and atomic operations.

2. Declare kernels The dOCAL user declares an object of class `docal::kernel` (Listing 3, line 11) for each CUDA kernel to be executed on one of system's devices. dOCAL kernels are initialized by the kernel's source code in its string representation, using either (1) the dOCAL-provided function `cuda::source` (line 11) or (2) function `cuda::path` to use the path to kernel's source file. If the source code contains only a single kernel, dOCAL automatically extracts kernel's name using the C++ regular expression library [48]; otherwise, the user passes the target kernel's name to the dOCAL kernel. Optionally, the user can also pass CUDA compiler flags to the kernel object, e.g., `-maxrregcount` to specify the maximum number of registers to use, or `-D name=definition` to replace in kernel's code each textual occurrence of `name` by `definition`.

We enable *Just-in-time (JIT)* compilation and thus benefiting from runtime values (a.k.a. *multi-stage programming* [45]) for a better performance by passing kernels in their string representation to dOCAL. For example, the user can replace the input size `N` in kernel's code (Listing 1, line 8) by its actual value (Listing 3, line 5), thereby enabling more aggressive compiler optimizations, e.g., loop unrolling. For this replacement, the user can use the CUDA compiler flag `-D`. The dOCAL kernel class contains pre-implemented low-level code—based on NVIDIA's *Runtime Compilation Library (NVRTC)* [37]—which is automatically called by dOCAL for compiling the code. To minimize the cost for the runtime compilation, dOCAL stores the compiled kernels in the dOCAL kernel object, and also on the system's hard drive, and reuses it for further computations; this happens transparently for the user.

3. Prepare kernels' inputs CUDA kernels take as their input the values of fundamental types (e.g., `int` and `float`), vector types (e.g., `int2` and `float4`) and/or device buffers, i.e., pointers to a contiguous range of memory on a particular device (a.k.a. *device array* in CUDA). While values of fundamental and vector types are passed straightforwardly to a kernel, passing buffers requires preparation and thus programming effort from the CUDA user: the special low-level functions `cudaMalloc`/`cudaFree` (Listing 2, lines 13–14, 37–38) have to be used for allocating/de-allocating memory on the target device, and function

`cudaMemcpyAsync` (lines 23 and 25) is used for transferring data between main memory and devices' memories. The effort for programming in CUDA increases for complex applications where a buffer's content is read/written on multiple devices, e.g., the partial results of one device are combined in parallel on another device; in such cases, the programmer is in charge of explicitly managing multiple buffers—one per device—and performing the device-to-device data transfers. Moreover, synchronization is a further challenge that has to be managed by the CUDA programmer: e.g., a data transfer from main memory to a device's memory has to be completed before a kernel on that device reads the data, and the kernel has to be finished before its computed data are transferred from the device to main memory. This requires a careful management of the multiple CUDA streams (Listing 2, lines 12, 23–25, 31, 39). For complex applications where devices' computations have interdependencies, e.g., the result of one device is used as input on another device, the user has to also use and manage so-called CUDA *events* which are created as synchronization points in the different devices' streams. Events have to be carefully managed by the user to avoid race conditions, which becomes especially challenging when multiple streams are used per devices (as done in dOCAL for a better hardware utilization – see discussion before).

In order to free the user from the burdens of preparing low-level CUDA buffers for kernels' execution and explicitly managing synchronization, dOCAL provides the high-level buffer class `docal::buffer`; it represents a portion of data that can be used for kernel computations on each of system's devices. For this, dOCAL buffers encapsulate one low-level CUDA buffer per used device and a region of main memory—the buffers and main memory mirror the same data. The dOCAL buffer class automatically manages memory by: (1) allocating memory on a device when the buffer is used for kernel computations on that device (see Step 4) and by de-allocating the memory when the buffer is destructed; (2) updating an encapsulated low-level CUDA device buffer or main memory before reading or writing it by automatically performing data transfers; (3) managing synchronization across multiple streams, i.e., dOCAL ensures transparently for the user that device and/or main memory can be simultaneously read but not be simultaneously written or read and written, and dOCAL ensures correct synchronization for complex applications with interdependent device computations, by carefully using and managing CUDA events.

A dOCAL buffer (Listing 3, lines 16–17) is passed to a dOCAL device object (lines 25–26) to use the buffer's data as kernel's input, and the buffer is accessed in the host code via a convenient interface analogous to that of the C++ standard vector type [48]. dOCAL is implemented to be compatible with the C++ Standard Template Library (STL). For example, we use the STL function `std::generate` (line 19) to conveniently fill the dOCAL buffer `in` with random numbers, and we use function `std::accumulate` to combine the GPUs ' partial results on the CPU after kernels' execution (line 29). In our reduction example of Listing 3, the dOCAL buffer `in` (line 16) comprises the CUDA devices' input values—`N` random floating point numbers (line 19) per device according to the original CUDA example in [37]; the buffer `out` (line 17) is for the devices' partial results.

4. Start device computations To start computations on a device, the user chooses a dOCAL device object (this is described in Step 1) and passes to it: (i) the `docal::kernel` to be executed (declared in Step 2), (ii) the kernel's *execution configuration*—the number of thread blocks and threads per block (a.k.a. *grid* and *block size* in CUDA) and (iii) kernel's input arguments, i.e., values of fundamental/vector types such as `float` and `float4`, and/or dOCAL buffer objects which represent low-level CUDA buffers (prepared in Step 3). dOCAL then uses the pre-implemented CUDA code of the high-level dOCAL classes to automatically allocate devices' memories and main memory, perform data transfers and execute the kernel.

In the reduction example (Listing 3), we process equally sized chunks of the input cooperatively on system's CUDA-capable devices (line 22), analogously as in the NVIDIA's host code (Listing 2, lines 10, 21, 28). For this, we pass to each dOCAL device object: (1) the dOCAL reduction kernel (Listing 3, line 23), (2) the kernel's corresponding grid and block size `GS` and `BS` (line 24) which we have chosen (line 13) according to the NVIDIA sample and 3) kernel's three input arguments (lines 25–27). The input arguments are: the input buffer `in` comprising the floating point numbers to sum up, the output buffer `out` in which the kernels' partial results are stored—one per thread—and the device's input size `N`. Since each device accesses only a chunk of buffers `in` and `out`, we pass also *C++ iterators* to chunk's first element—returned by function `begin()`—summed with the corresponding offset, and the chunk size, i.e., `GS*BS` elements in case of buffer `out` (line 25) and `N` elements in case of buffer `in` (line 26). Alternatively to the chunk size, the user can use an iterator pointing to chunk's end. By setting iterators to the chunk for each device, dOCAL avoids the costly transferring of the entire buffers `in` and `out` between main memory and a device's memory and only transfers one chunk per device and buffer.

We implement functions in dOCAL as asynchronously, i.e., the control returns immediately to the main thread which only blocks when one of the kernel's output buffers is accessed in the host code. To differentiate between kernels' input and output buffers, dOCAL provides the user with three different *buffer tags*: `read`, `write` and `read_write` (Listing 3, lines 25–26); they signal to dOCAL how the kernel accesses a buffer. The tags enable dOCAL to automatically (1) coordinate device computations, e.g., a computation does not start until other computations on its input/output buffers have been finished, and (2) minimize unnecessary data transfers, e.g., dOCAL avoids a data transfer from main memory to a device's memory or between different devices' memories if a buffer is only written by the kernel or if the data have been transferred previously to the device (a.k.a. *lazy-copy* [14]), and dOCAL avoids transferring the data back after kernel's execution if buffer was only read and thus not modified by the kernel. For example, in Listing 3, analogously to the NVIDIA's hand-written low-level host code in Listing 2, the content of buffer `out` is not copied to devices' memories by dOCAL as it is tagged with `write` and as such not read by the devices, and the buffer `in` is not copied from devices' memories to main memory as it is only read by the kernel. dOCAL automatically blocks the main thread (in line 29) where kernel's output buffer `out` is accessed by function `begin()`; the computation of the main thread continues when the kernels

finish and their results are transferred by dOCAL from devices' memory to main memory, so that they become accessible for function `begin()`.

2.2 Using dOCAL for deploying OpenCL host code

In addition to its high-level host code interface for CUDA (as described in Sect. 2.1), dOCAL provides an analogous high-level interface to simplify programming OpenCL host code. For example, for executing the OpenCL reduction kernel provided by NVIDIA in [36] (which is equivalent to the CUDA kernel in Listing 1), the user only has to slightly modify the dOCAL code in Listing 3 (for CUDA), as follows: (1) replace function `get_all_local_devices<CUDA>` (in line 8) by function `get_all_local_devices<OpenCL>` to acquire all OpenCL-compatible devices from dOCAL and (2) set the dOCAL kernel object (in line 11) to the OpenCL kernel's source code using the dOCAL-function `opencl::source`. dOCAL then automatically performs the low-level OpenCL commands for executing the OpenCL reduction kernel on all of system's OpenCL-capable devices which may be of different vendors, e.g., Intel multi-core CPU and NVIDIA/AMD GPU. All dOCAL optimizations for CUDA host code, e.g., using multiple streams (a.k.a. *command queue* in OpenCL terminology) for better hardware utilization, avoiding unnecessary data transfers and caching kernel binaries for reducing the overhead of JIT compilation, are also provided by dOCAL for OpenCL.

3 OpenCL-CUDA interoperability in dOCAL

The dOCAL library supports developing host code for programs that use both OpenCL and CUDA kernels, by allowing to arbitrarily combine dOCAL host code for OpenCL and CUDA in the same program. (We call this *OpenCL-CUDA interoperability*) For example, a dOCAL buffer with the results of a CUDA kernel can be passed to an OpenCL device object to be further processed in parallel on system's multi-core CPU. Furthermore, dOCAL allows executing a CUDA kernel on an OpenCL device to achieve portability [11], e.g., to perform a CUDA kernel on an Intel multi-core CPU. dOCAL also allows for executing an OpenCL kernel on a CUDA device for higher performance—CUDA compilers often generate more efficient machine code for NVIDIA devices than OpenCL compilers [32]. For this, dOCAL automatically performs source-to-source translation between the OpenCL and CUDA kernel programming languages. Our translation engine is currently a proof-of-concept implementation that is based on the C++ regular expression library [48] and has some limitations: advanced C++ features such as automatic type deduction and template meta programming are not supported.

Listing 4 demonstrates how dOCAL is used to utilize system's multi-core CPU in our reduction example of Listing 3: we use OpenCL to further sum the GPUs' partial results (obtained with CUDA) in parallel on system's multi-core CPU, rather than summing them only sequentially as done in Listing 3 (and also in the original CUDA host code in Listing 2). For this, we replace line 29 of our dOCAL

program (Listing 3) by the code in Listing 4. In this optimized code, we use system's multi-core CPU (line 1), and we declare buffer `cpu_res` (line 6) for CPU's partial results. We then start parallel computations on the CPU by passing the following to the dOCAL OpenCL device object: (1) the reduction kernel (line 8)—it comprises the CUDA device code (in Listing 1) which is automatically translated by dOCAL to the equivalent OpenCL code to be executable on the multi-core CPU via OpenCL; (2) the execution configuration (line 9) which we choose as one thread group per CPU's core, and we choose the thread group size as CPU's SIMD vector length (lines 3–4); (3) the kernel's input arguments (line 10). The input arguments are: (i) dOCAL buffer `out` (Listing 3, line 17), (ii) buffer `cpu_res` for CPU's partial results (Listing 4, line 6) and (iii) input size, i.e, the number of floating numbers in buffer `out`. Buffer `out` contains the GPUs ' partial results that are obtained with CUDA (Listing 3, line 25) and thus reside in a low-level CUDA data structure which is internally managed by buffer `out`. dOCAL copies the results, according to its interoperability feature (transparently for the user) to an OpenCL data structure so that it can be passed to the OpenCL reduction kernel.

```
1  docal::device<OpenCL_CPU> cpu;
2
3  int NUM_CORES = /* number of CPU's cores    */;
4  int VL        = /* CPU's SIMD vector length */;
5
6  docal::buffer cpu_res( NUM_CORES*VL );
7
8  cpu( reduction                                  )
9     ( dim3( NUM_CORES ) , dim3( VL )             )
10    ( write( cpu_res ), read( out ), out.size() );
11
12 auto res = std::accumulate( cpu_res.begin(), cpu_res.end(), std::
       plus<float>() );
```

Listing 4 Improved excerpt for dOCAL host code from Listing 3, line 29: the OpenCL-CUDA interoperability in dOCAL allows summing GPU's partial results obtained with CUDA in parallel on the multi-core CPU using OpenCL.

Note that in Listing 4, we set the execution configuration (Listing 4, line 9), analogously to before (Listing 3, line 24), according to CUDA 's approach as grid and block size using function `dim3`. In OpenCL, the execution configuration (a.k.a. *NDRange* in OpenCL terminology) is usually set as *global* and *local size*—the total number of threads and thread group size—which can be done in dOCAL by using the dOCAL function `nd_range`, rather than `dim3`. dOCAL allows the user to arbitrarily choose weather setting the execution configuration as grid and block size (using dOCAL's function `dim3`) or as global and local size (using function `nd_range`) for both OpenCL and CUDA device objects.

In the following, we demonstrate that dOCAL's source-to-source translation feature—from OpenCL to CUDA—contributes to a better kernel performance due to the usually higher efficiency of CUDA on NVIDIA devices as compared to OpenCL [25].

Figure 1 shows the measured speedups of the OpenCL GEMM kernel (general matrix multiplication) of the popular OpenCL BLAS library CLBlast [34] on an

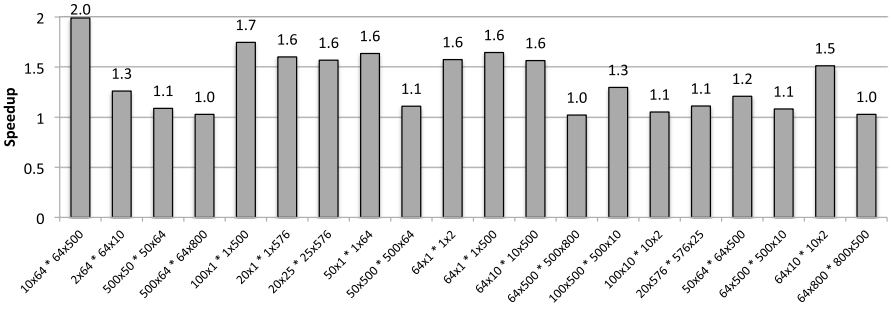

Fig. 1 Speedup (higher is better) of CLBlast's OpenCL GEMM kernel [34] when translated with dOCAL to CUDA as compared to its original OpenCL implementation on an NVIDIA Tesla K20 GPU for 20 input sizes that are heavily used in the deep learning framework Caffe [24]

NVIDIA Tesla K20 GPU; the bars show the speedup of the kernel when translated by dOCAL to CUDA over their initial OpenCL implementation (higher is better). We show the results for 20 input sizes that are heavily used in the deep learning framework Caffe [24]; as concrete neural network, we use Caffe's *siamese* sample for handwriting recognition [29]. We observe that using an equivalent CUDA kernel for CLBlast 's OpenCL GEMM kernel leads to speedups of up to 2, because CUDA generates more efficient NVIDIA machine code as compared to OpenCL [32]. The overhead for the translation (not included in our measurements in Fig. 1)—250ms on our system—is negligible because once the GEMM kernel is translated from OpenCL to CUDA, it is automatically stored by dOCAL on the system and reused for each new call—in the siamese sample, GEMM is called over $> 10^6$ times on each input size in Fig. 1, requiring > 6 total computation time on our system.

Listing 5 demonstrates that using dOCAL, the CLBlast 's OpenCL GEMM kernel can be easily translated and executed in the CUDA programming framework. As shown in line 5, the user only passes the kernel's OpenCL code (line 1) to a dOCAL CUDA device object (declared in line 3); dOCAL then automatically translates the OpenCL code to CUDA, and uses the CUDA framework for executing the translated kernel.

```
1  docal::kernel gemm = opencl::source( /* GEMM's OpenCL code */ )
2
3  docal::device<CUDA> gpu( "Tesla K20" );
4
5  gpu( gemm       )
6     ( /* ... */ )
7     ( /* ... */ );
```

Listing 5 Using dOCAL for conveniently executing the CLBlast's GEMM kernel (originally written in OpenCL) in CUDA for higher performance.

4 Using dOCAL for distributed systems

In a distributed system (a.k.a. cluster) with several nodes, our dOCAL library enables conveniently executing OpenCL and CUDA kernels on nodes that are connected via TCP/IP. For this, the user starts a dOCAL daemon process on the target

nodes; dOCAL then automatically handles node-to-node data transfers and starts kernel computations on the nodes' devices, using the Boost.Asio C++ networking library [4].

Our example in Listing 3 which uses the devices of a single node can be easily extended to use the devices of all nodes: the user only replaces the function `docal::get_all_local_devices<CUDA>()` in line 8 by function `docal::get_all_devices<CUDA>()`; dOCAL then automatically acquires the devices of different nodes, transfers the devices' input and output data over the TCP/IP network and synchronizes the different nodes' computations.

The user can also target specific remote devices. For this, a `docal::device` object is initialized additionally with the target node's name, rather than with only the device name, device id or device properties. For example, the user uses `docal::device<CUDA>("gpu_node", 0)` to get the CUDA device with id 0 on the node with name `gpu_node`. Alternatively to the node's name, the user can use the node's IP address.

5 Data transfer optimizations

In addition to its standard buffer type `docal::buffer` (introduced in Sect. 2), dOCAL provides two further buffer types: (1) `docal::pinned_buffer` and (2) `docal::unified_buffer`; both are used analogously to dOCAL's standard buffer type. As compared to a dOCAL standard buffer, dOCAL's pinned buffer uses internally *pinned main memory* [38] which enables fast data transfers between a node's main memory and its devices' memories, and pinned memory is also required for overlapping data transfers with device computations [39]. However, since pinned memory has a high allocation time, it should only be used if many data transfers are performed. dOCAL's unified buffer type uses *unified memory* [41] which is beneficial when kernels access main memory sparsely and when the target device provides hardware support for unified memory. Especially when targeting CPUs, using unified memory (a.k.a. *zero-copy buffer* in OpenCL [23]) avoids data transfers between devices' memory and main memory because for CPUs ' device memories and main memory coincide [23].

The OpenCL and CUDA documents [23, 38] recommend the programmer to empirically test which allocation type—naive, pinned or unified—suits best for their applications, dependent on the target hardware. However, testing these special allocation types—pinned and unified—requires a significant effort from the programmer. For example, for using pinned memory in low-level OpenCL, the user has to initialize an OpenCL-specific `cl_mem` object using the special flag `CL_MEM_ALLOC_HOST_PTR`, and she has to use the special function `clEnqueueMapBuffer` to get access to the pinned memory region comprised by the `cl_mem` object. Moreover, the user is in charge of explicitly synchronizing the buffer (e.g., before it is read by a kernel), using the function `clEnqueueUnmapMemObject`, and user has to use multiple *command queues*—the OpenCL equivalent to CUDA streams—to enable overlapping data transfers with computations [39].

The two optimized dOCAL buffer types automatically handle the inconvenient low-level interactions with the OpenCL and CUDA API for allocating and using these special memory regions. Moreover, the user can easily switch between different allocation types by only changing the dOCAL buffer type, e.g., from `docal::buffer` to `docal::pinned_buffer` to use pinned memory instead of naively allocated memory.

Figure 2 (left) shows the runtime of Intel's OpenCL `ZeroCopy` benchmark [20]—for evaluating unified memory—on an Intel Xeon E5 CPU, compared to the runtime of an equivalent dOCAL program which uses dOCAL's unified buffer type—the Intel benchmark computes *Ambient Occlusion* which is popular in the field of visual computing. According to the benchmark's implementation, we measure the runtime for data transfers and the kernel's execution, i.e., we ignore the runtimes for initializing OpenCL, compiling the kernel, etc. We observe that dOCAL achieves competitive runtime with the low-level OpenCL code. This is because dOCAL's unified buffers use, analogously to the Intel's benchmark, unified memory which enables avoiding data transfers when targeting CPU architectures (as discussed above).

Figure 2 (right) shows the runtime comparison of NVIDIA's benchmark `overlap-data-transfers` [39] with dOCAL using its pinned buffer type; the benchmark computes trigonometric functions to evaluate the performance of pinned main memory. We perform experiments on an NVIDIA Tesla K20 GPU. Analogously to before, we measure only the runtime for data transfers and the kernel executions, according to our reference benchmark. dOCAL achieves the same performance as the low-level CUDA code: dOCAL's pinned buffers use internally pinned main memory, analogously to the NVIDIA's benchmark, thus enabling fast data transfers and overlapping the transfers with computations.

6 Advanced dOCAL usage

6.1 dOCAL compatibility with existing OpenCL/CUDA libraries

There is a broad range of expert-implemented OpenCL/CUDA libraries, such as the OpenCL linear algebra library CLBlast [34] and the CUDA library cuDNN

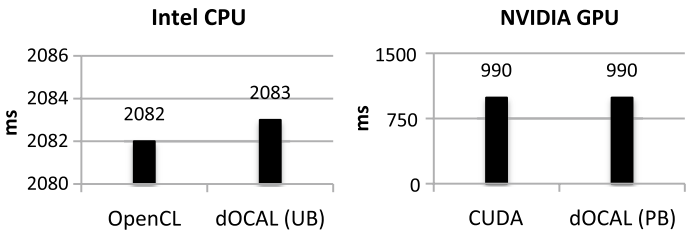

Fig. 2 Runtime comparison (lower is better) of dOCAL with low-level OpenCL and CUDA host code for benchmarking the unified memory on Intel CPU (left) and pinned memory on NVIDIA GPU (right). dOCAL achieves competitive performance with the low-level code

for Deep Learning applications [22]. To enable compatibility between dOCAL and such libraries, dOCAL's three buffer types (discussed in Sects. 2 and 5) can be cast to the native buffer representation of OpenCL and CUDA: `cl_mem` in case of OpenCL and `void*` in case of CUDA. This cast happens either automatically in dOCAL—then, the OpenCL/CUDA buffer is returned that belongs to the most recently used device—or, alternatively, the user can use the dOCAL buffers' function `get_cuda_buffer(dev)` to get the CUDA buffer for a specific device `dev`. Here, `dev` is either a dOCAL device object, the device's name as string, or device's numerical CUDA device id. For OpenCL, dOCAL provides the analogous member function `get_opencl_buffer`.

6.2 Auto-tuning support

dOCAL supports the user in the cumbersome task of finding a kernel's good-performing values of performance-critical parameters, e.g., cache/thread block sizes and loop unrolling factors. For this, dOCAL allows conveniently interconnecting with an *auto-tuning system*—they use advanced search heuristics and/or machine learning techniques to automatically explore the search space of a kernel's performance-critical parameters; the determined values are then used to build an optimized kernel [1].

Auto-tuning systems for OpenCL and CUDA can be conveniently generated by using the auto-tuning framework (ATF) [1]: the user annotates the kernel code with *tuning directives* which specify its performance-critical parameters by their: (1) types (e.g., `int` or `float`), (2) ranges of possible values, and (3) possible interdependencies (e.g., a parameter has to evenly divide another parameter). ATF then automatically generates the corresponding auto-tuner that optimizes the kernel for a target hardware.

For connecting dOCAL with an auto-tuner, the user provides to dOCAL the concrete auto-tuner for its kernel, e.g., generated with ATF, by storing it to a corresponding path on the hard drive. dOCAL then manages transparently from the user the cumbersome tasks of (1) calling the auto-tuner for each device on which the kernel is executed, (2) storing on the hard drive the auto-tuned kernel that is obtained by the auto-tuner, and (3) reusing the auto-tuned version of the kernel in each following kernel execution.

For high-quality tuning results, auto-tuning has to be performed depending also on runtime values (e.g., input size), and not only depending on the target device [53]. For this, the user generates the corresponding auto-tuner—this is described in detail in [1]—and passes to the dOCAL kernel object the concrete runtime values using dOCAL's `tuning` function. For example, to auto-tune the reduction kernel (shown in Listing 1) also for the input size `N` (Listing 3, line 5), the user: (1) provides the input-aware auto-tuner for the kernel [1], and (2) initializes the dOCAL kernel (in line 11) with the input size N using the `tuning` function, i.e., `ocal::kernel reduction = { cuda::source(/*...*/), tuning(N) };`

6.3 Profiling OpenCL/CUDA programs with dOCAL

dOCAL enables convenient profiling of OpenCL and CUDA programs, i.e., without requiring the use of low-level profiling functions, such as `cudaEventRecord` and `cudaEventSynchronize` (for CUDA), or `clGetEventProfilingInfo` and `clWaitForEvents` (for OpenCL). To enable profiling in dOCAL, the user only defines the C preprocessor macro `dOCAL_ENABLE_PROFILING`; dOCAL then automatically measures and outputs the runtimes for initializing OpenCL/CUDA, performing data transfers, executing kernels, and compiling the kernels. Additionally, dOCAL stores the measured runtimes in a JSON file—a popular file format for human-readable name-value pairs.

7 Experimental evaluation

We experimentally prove that dOCAL simplifies implementing host code for both OpenCL and CUDA, with a low runtime overhead for abstraction. After describing our experimental setup in Sect. 7.1, we report experimental results for a single-node system (Sect. 7.2) and a multi-node system (Sect. 7.3).

7.1 Experimental setup

For the runtime evaluation, we use a system with two nodes, each equipped with two Intel Xeon E5-2640 v2 8-core CPUs, clocked at 2GHz with 128GB main memory and hyper-threading enabled, as well as two NVIDIA Tesla K20m GPUs; the two nodes are connected via an InfiniBand FDR network. We perform experiments using both the CPUs and GPUs as OpenCL devices. A node's two CPUs are represented in OpenCL as a single device with 32 compute units, corresponding to the overall 2×16 logical cores in the node. For runtime measurements, we use the unix `time` command. As C++ compiler, we use `clang` version 3.8.1 with its `-O3` optimization flag enabled on the CentOS operating system version 7.4.

7.2 Single-node experiments

We perform our single-node experiments by comparing to all of the expert-implemented, real-world, multi-device code samples provided by Intel and NVIDIA in [21] and [37] for OpenCL and CUDA, against equivalent dOCAL programs. The Intel samples are: (1) `intel_ocl_multidevice_basic` for computing scaled dot product and (2) `intel_ocl_tone_mapping_multidevice` for high dynamic range tone mapping. For CUDA, we use the three NVIDIA's samples: (1) `simpleMultiGPU` for reduction, (2) `MonteCarloMultiGPU` for a Monte Carlo experiment and (3) `nbody` for N-body simulation. We compare each sample against the equivalent dOCAL program in terms of both code complexity and runtime.

We measure the code complexity using four classical metrics for development effort: (1) lines of code (LOC), excluding blank lines and comments, (2) COCOMO development effort (DE) in person-months [3], (3) McCabe's cyclomatic complexity (CC) [31] and (4) the Halstead development effort (HDE) [19]. McCabe's cyclomatic complexity is the number of linearly independent paths through the source code, while the Halstead development effort metric is based on the number of operators and operands in the source code. Low cyclomatic complexity and Halstead development effort imply that code is simpler to develop and debug. We measure the metrics LOC and CC with the tool provided in [50], the DE with [10] and HDE with [44].

Figure 3 compares the code complexity of the original OpenCL and CUDA samples from the vendors with their dOCAL counterparts. The kernel code is excluded in our measurements, because dOCAL and the OpenCL/CUDA samples use the same kernel codes. We observe that dOCAL programs are significantly simpler; on average they (1) require 2.72× fewer lines of code (LOC) in case of OpenCL and 1.85× lines in case of CUDA, (2) require a 2.8× less development effort (DE) in case of OpenCL and 1.9× in case of CUDA, (3) have a cyclomatic complexity (CC) that is reduced by a factor of 2.73× for OpenCL and 1.7× for CUDA, and 4) their Halstead development effort (HDE) is reduced by the factor 2.78× (OpenCL) and 1.79× (CUDA). Even for simple applications, e.g., *scaled dot product* and *reduction*, dOCAL programs are significantly simpler than their low-level OpenCL/CUDA equivalents, because of the boilerplate code required by the low-level approaches, e.g., for initializing OpenCL/CUDA and for performing data transfers. We observe that dOCAL programs achieve more reduction in complexity for OpenCL than for CUDA, because OpenCL requires boilerplate commands for devices of different vendors while CUDA targets NVIDIA devices only.

Sample	Code	LOC	DE	CC	HDE
Scaled-Dot-Product	**OpenCL**	293	0,68	21	57.523
	dOCAL	54	0,12	8	10.729
HDR-Tone-Mapping	**OpenCL**	523	1,25	88	290.102
	dOCAL	246	0,57	32	114.451
Reduction	**CUDA**	110	0,26	14	19.980
	dOCAL	56	0,12	13	11.974
Monte-Carlo	**CUDA**	336	0,82	32	131.259
	dOCAL	190	0,45	24	76.337
N-body	**CUDA**	812	1,96	80	412.182
	dOCAL	434	1,03	37	226.962

Fig. 3 Code complexity of the OpenCL and CUDA samples as compared to their dOCAL counterparts using the classical metrics: (1) lines of code (LOC), (2) COCOMO development effort (DE) in person months, (3) McCabe's cyclomatic complexity (CC) and (4) Halstead development effort (HDE). The metrics indicate that dOCAL code is significantly simpler than low-level OpenCL and CUDA host code

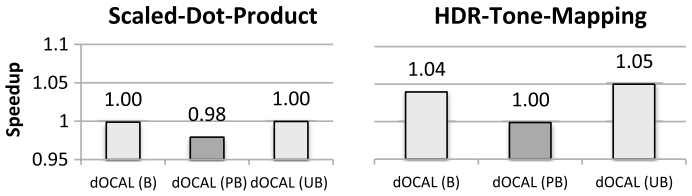

Fig. 4 Speedup/slowdown (higher is better) of dOCAL over Intel's OpenCL samples on two Intel Xeon E5 CPUs for each of dOCAL's three buffer types: buffer (B), pinned buffer (PB) and unified buffer (UB). The buffer type that corresponds to the memory used in the low-level samples is filled dark gray. Speedups are computed using the median runtime of 30 runs. We observe that dOCAL's performance is competitive to low-level OpenCL host code

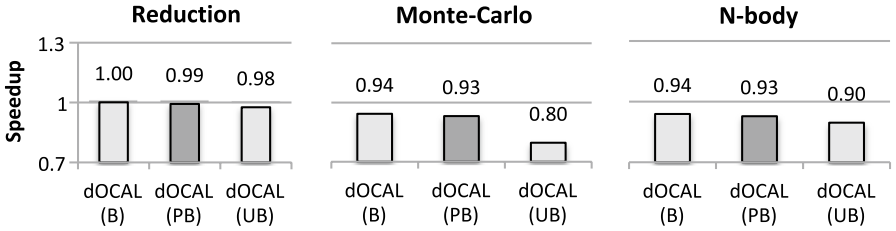

Fig. 5 Speedup/slowdown (higher is better) of dOCAL over NVIDIA's CUDA samples on two NVIDIA Tesla K20m GPUs using dOCAL's three buffer types: buffer (B), pinned buffer (PB) and unified buffer (UB). The buffer type that corresponds to the memory used in the low-level samples is filled dark gray. Speedups are computed using the median runtime of 30 runs. dOCAL's performance is competitive to low-level CUDA host code

Figures 4 and 5 demonstrate the speedups (or slowdowns if < 1) of our high-level dOCAL programs as compared to their corresponding low-level samples in OpenCL and CUDA. We present results for each of dOCAL's three buffer types—buffer (B), pinned buffer (PB) and unified buffer (UB)—for which the OpenCL and CUDA documents recommend to naively test which type suits best for a particular combination of target application and hardware architecture [23, 38]. The low-level samples all use pinned main memory which corresponds to using dOCAL's pinned buffer type (the corresponding bars are filled dark gray for clarification). The Intel's OpenCL samples run on a node's two Intel CPUs, and the NVIDIA CUDA samples run on the node's two NVIDIA GPUs.

We observe that dOCAL's high-level approach causes a quite low runtime overhead of $< 2\%$ in comparison with OpenCL and $< 7\%$ in comparison with CUDA when using pinned memory (dark gray bars) as in the low-level samples. This is due to modern compilers efficiency—in our case, the `clang` compiler—which significantly optimize dOCAL's abstraction overhead, e.g., by performing optimizations such as inline expansion [7]. For the two further dOCAL's buffer types—buffer and unified buffer—we observe the same or sometimes even slightly better performance of dOCAL as compared to the low-level samples. This is caused by the high allocation time for pinned memory which is used by the samples. In contrast, dOCAL's

buffer (B) and unified buffer (UB) types use straightforwardly allocated memory or unified memory, correspondingly, causing a lower allocation time (as discussed in Sect. 5). The better performance of dOCAL for OpenCL as compared to CUDA is because the OpenCL samples implement and use several helper functions, e.g., for selecting the OpenCL platform, which causes runtime overhead.

7.3 Multi-node experiment

We use the example of general matrix multiplication (GEMM) to demonstrate dOCAL's efficiency on multi-node systems. For this, we use the OpenCL GEMM kernel provided by NVIDIA in [35].

Figure 6 shows GEMM 's runtime on 16384 × 16384 matrices of single precision floating point numbers (`float`) when executed (1) on a single local GPU, (2) on two local GPUs, and (3) on the four GPUs of two nodes, i.e., two local GPUs (first node) and two remote GPUs (second node). We observe that switching from a single local GPU to two local GPUs increases performance by a factor of 1.6; when using the second node's two remote GPUs as well (i.e., four GPUs in total), performance is increased further by a factor of 1.3. Performance increases more significantly when doubling the number of local GPUs, rather than when doubling the number of remote GPUs, because using remote GPUs requires communication between different nodes. For example, in case of GEMM, chunks of the input matrices have to be transferred over the network from the local node to the remote node, making nearly 5 seconds of runtime. If excluding this overhead, we would achieve again a speedup of nearly 1.6× (instead of a speedup of only 1.3), i.e., the overhead for using the remote GPUs is mainly caused by the (inherent) node-to-node data transfers over the InfiniBand network.

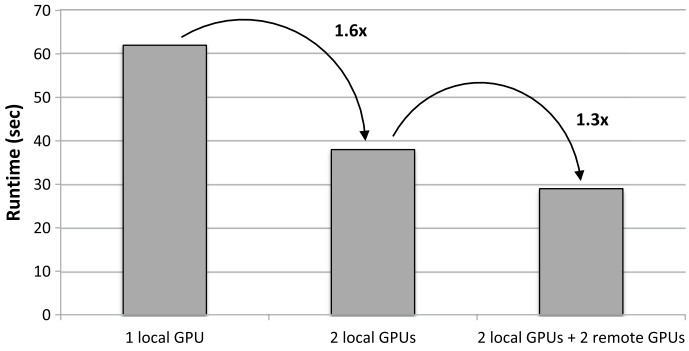

Fig. 6 Runtime comparison (lower is better) of NVIDIA's general matrix multiplication (GEMM) in OpenCL when executed (1) on a single local GPU, (2) on two local GPUs, and (3) on two local GPUs and two remote GPUs. Doubling the number of local GPUs speeds up performance by a factor of 1.6; using in addition two remote GPUs increases performance further by a factor of 1.3

8 Conclusion

We present dOCAL—a high-level C++ library for conveniently implementing OpenCL and CUDA host code. dOCAL allows easily executing arbitrary OpenCL and CUDA kernels on the devices of different nodes by automatically managing different nodes' main memories and their devices' memories, performing node-to-node communication, handling synchronization, minimizing data transfers and supporting data transfer optimization between device and main memory. Furthermore, dOCAL allows interoperability between OpenCL and CUDA host code by automatically moving data between OpenCL and CUDA data structures and by performing source-to-source translation between the OpenCL and CUDA kernel languages. Our experimental evaluation on real-world samples from Intel and NVIDIA shows that dOCAL arguably simplifies host code as compared to standard OpenCL and CUDA, with a low runtime overhead for abstraction.

In future work, we will demonstrate dOCAL's efficiency for a broad range of applications. Furthermore, we aim to improve our OpenCL-to-CUDA/CUDA–OpenCL translation engine, e.g., by supporting advanced C++ features such as automatic type deduction and template meta programming.

References

1. Rasch A, Gorlatch S (2018) ATF: a generic, directive-based auto-tuning framework. In: CCPE, pp 1–16. https://doi.org/10.1002/cpe.4423
2. Aldinucci M et al (2015) The loop-of-stencil-reduce paradigm. In: IEEE Trustcom/BigDataSE/ISPA, pp 172–177
3. Boehm B et al (1995) Cost models for future software life cycle processes: COCOMO 2.0. In: Annals of software engineering, pp 57–94
4. Boost: Boost.Asio (2018). http://www.boost.org/doc/libs/1_66_0/doc/html/boost_asio.html
5. Castro D et al (2016) Farms, pipes, streams and reforestation: reasoning about structured parallel processes using types and hylomorphisms. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP, pp 4–17
6. Cedric A et al (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: Concurrency and computation: practice and experience, pp 187–198
7. Chang PP et al (1989) Inline function expansion for compiling C programs. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp 246–257
8. Dagum L et al (1998) OpenMP: an industry-standard api for shared-memory programming. In: IEEE computational science and engineering, pp 46–55
9. Dastgeer U et al (2014) The PEPPHER composition tool: performance-aware dynamic composition of applications for GPU-based systems. In: Computing, pp 1195–1211
10. Wheeler David A (2018) SLOCCount. https://www.dwheeler.com/sloccount/
11. Du P et al (2012) From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming. In: Parallel computing, pp 391 – 407
12. Duato J et al (2010) rCUDA: reducing the number of GPU-based accelerators in high performance clusters. In: International Conference on High Performance Computing Simulation, pp 224–231
13. Duran A et al (2011) OmpSs: a proposal for programming heterogeneous multi-core architectures. In: Parallel processing letters, pp 173–193
14. Enmyren J et al (2010) SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: HLPP, pp 5–14
15. Ernsting S et al (2011) Data parallel skeletons for GPU clusters and multi-GPU systems. In: PARCO, pp 509–518

16. Gorlatch S, Cole M (2011) Parallel skeletons. In: Encyclopedia of parallel computing, pp 1417–1422
17. Grasso I et al (2013) LibWater: heterogeneous distributed computing made easy. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS, pp 161–172
18. Haidl M, Gorlatch S (2014) PACXX: towards a unified programming model for programming accelerators using C++14. In: LLVM compiler infrastructure in HPC, pp 1–11
19. Halstead MH (1977) Elements of software science. Elsevier computer science library: operational programming systems series
20. Intel: Ambient Occlusion Benchmark (AOBench) (2014). http://code.google.com/p/aobench
21. Intel: Code Samples (2018). https://software.intel.com/en-us/intel-opencl-support/code-samples
22. Intel: CUDA Deep Neural Network Library (2018). https://developer.nvidia.com/cudnn
23. Intel: how to increase performance by minimizing buffer copies on intel processor graphics (2018). https://software.intel.com/en-us/articles/getting-the-most-from-opencl-12-how-to-increase-performance-by-minimizing-buffer-copies-on-intel-processor-graphics
24. Jia Y et al (2014) Caffe: convolutional architecture for fast feature embedding. In: arXiv preprint arXiv:1408.5093
25. Karimi K et al (2010) A performance comparison of CUDA and OpenCL. In: CoRR
26. Kegel P et al (2012) dOpenCL: towards a uniform programming approach for distributed heterogeneous multi-/many-core systems. In: IEEE 26th international parallel and distributed processing symposium workshops PhD forum, pp 174–186
27. Kim J et al (2012) SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS, pp 341–352
28. Klöckner A et al (2012) PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. In: Parallel computing, pp 157 – 174
29. Koch G et al (2015) Siamese neural networks for one-shot image recognition. In: ICML deep learning workshop
30. Lee S et al (2010) OpenMPC: extended OpenMP programming and tuning for GPUs. In: ACM/IEEE International Conference for high Performance Computing, Networking, Storage and Analysis, pp 1–11
31. McCabe T.J (1976) A complexity measure. In: IEEE transactions on software engineering, pp 308–320
32. Memeti S et al (2017) Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In: Workshop on adaptive resource management and scheduling for cloud computing, pp 1–6
33. Moreton-Fernandez A et al (2017) Multi-device controllers: a library to simplify parallel heterogeneous programming. Int J Parallel Program 47(1):94–113
34. Nugteren C (2016) CLBlast: a tuned OpenCL BLAS library. In: CoRR
35. NVIDIA: nvidia-opencl-examples. https://github.com/sschaetz/nvidia-opencl-examples (2012)
36. NVIDIA: OpenCL samples (2012). https://github.com/sschaetz/nvidia-opencl-examples/
37. NVIDIA: CUDA Toolkit 9.1 (2018). https://developer.nvidia.com/cuda-toolkit
38. NVIDIA: how to optimize data transfers in CUDA C/C++ (2018). https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/
39. NVIDIA: how to overlap data transfers in CUDA C/C++ (2018). https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/
40. NVIDIA: hyper-Q (2018). http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf
41. NVIDIA: unified memory for CUDA beginners (2018). https://devblogs.nvidia.com/unified-memory-cuda-beginners/
42. Pérez B et al (2016) Simplifying programming and load balancing of data parallel applications on heterogeneous systems. In: GPGPU, pp 42–51
43. Reyes R et al (2015) SYCL: single-source C++ accelerator programming. In: PARCO, pp 673–682
44. rharish100193: halstead metrics tool (2016). https://sourceforge.net/projects/halsteadmetricstool/
45. Rompf T et al (2015) Go meta! A case for generative programming and DSLs in performance critical systems. In: LIPIcs–Leibniz international proceedings in informatics, pp 238–261
46. Rupp K et al (2010) Automatic performance optimization in ViennaCL for GPUs. In: POOSC, pp 1–6
47. Spafford K et al (2010) Maestro: data orchestration and tuning for OpenCL devices. In: Euro-Par–parallel processing. Springer, Berlin, pp 275–286

48. Standard C++ foundation foundation members: ISO C++ (2018). https://isocpp.org
49. Steuwer M et al (2011) SkelCL—a portable skeleton library for high-level GPU programming. In: IEEE IPDPS workshops, pp 1176–1182
50. Steve Arnold: CCCC project documentation (2005). http://sarnold.github.io/cccc/
51. Szuppe J (2016) Boost.Compute: a parallel computing library for C++ based on OpenCL. In: IWOCL, pp 1–39
52. Tejedor E et al (2011) ClusterSs: a task-based programming model for clusters. In: Proceedings of the 20th international symposium on high performance distributed computing, HPDC, pp 267–268
53. Tillet P, Cox D (2017) Input-aware auto-tuning of compute-bound HPC kernels. In: SC, pp 1–12
54. Vinas M et al (2015) Improving OpenCL programmability with the heterogeneous programming library. In: International Conference on Computational Science, ICCS, pp 110 – 119
55. Wienke S et al (2012) OpenACC—first experiences with real-world applications. In: Euro-Par parallel processing, pp 859–870

Affiliations

Ari Rasch[1] · Julian Bigge[1] · Martin Wrodarczyk[1] · Richard Schulze[1] · Sergei Gorlatch[1]

Julian Bigge
j.bigge@uni-muenster.de

Martin Wrodarczyk
m.wrod@uni-muenster.de

Richard Schulze
r.schulze@uni-muenster.de

Sergei Gorlatch
gorlatch@uni-muenster.de

[1] Department of Mathematics and Computer Science, University of Münster, Münster, Germany