2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)

OCAL: An Abstraction for Host-Code Programming with OpenCL and CUDA

Ari Rasch*, Martin Wrodarczyk*, Richard Schulze*, Sergei Gorlatch*
*University of Münster, Germany, {a.rasch, m.wrod, r.schulze, gorlatch}@uni-muenster.de

***Abstract*—The state-of-the-art parallel programming approaches OpenCL and CUDA require so-called host code for program's execution. Implementing host code is often a cumbersome task, especially when executing OpenCL and CUDA programs on systems with multiple devices, e.g., multi-core CPU and Graphics Processing Units (GPUs): the programmer is responsible for explicitly managing system's main memory and devices' memories, synchronizing computations with data transfers between main and/or devices' memories, and optimizing data transfers, e.g., by using pinned main memory for accelerating data transfers and overlapping the transfers with computations.**

In this paper, we present *OCAL (OpenCL/CUDA Abstraction Layer)* – a high-level approach to simplify the development of host code. OCAL combines five 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, uniform high-level host code abstraction API; 2) it supports executing arbitrary OpenCL and CUDA programs; 3) it simplifies implementing data-transfer optimizations by providing specially-optimized memory buffers, e.g., for conveniently using pinned main memory; 4) it optimizes memory management by automatically avoiding unnecessary data transfers; 5) it enables *interoperability* between OpenCL and CUDA host code by automatically managing the communication between OpenCL and CUDA data structures and by automatically translating between the OpenCL and CUDA programming constructs.

Our experiments demonstrate that OCAL significantly simplifies implementing host code with a low runtime overhead for abstraction.

I. Motivation and Related Work

OpenCL and CUDA are state-of-the-art approaches to programming modern heterogeneous systems equipped with multi-core CPUs and accelerator devices such as Graphics Processing Units (GPUs). Both approaches have a common problem: they demand from the programmer to implement the so-called *host code* for executing OpenCL and CUDA device code (a.k.a. *kernel*).

Implementing host code is 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 devices, e.g., two or more GPUs and CPU, OpenCL and CUDA host code's implementation becomes cumbersome and error-prone even for experienced programmers: they have to manage the memories of different devices, as well as manage system's main memory, and they have to explicitly synchronize data transfers with kernel computations in different devices.

Programming host code 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 [1] and because CUDA provides better profiling and debugging tools (e.g., CUDA-MEMCHECK for detecting out-of-bounds and misaligned memory accesses [2]). Consequently, 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 process the results of different GPUs (computed using CUDA) on a multi-core CPU using OpenCL.

For high performance, the host code is expected to 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 main memory [3], [4]. 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 [5]–[9] simplify host code programming, e.g., by managing and optimizing memory management, but are restricted to OpenCL and CUDA programs that can be expressed via specifically-provided parallel patterns (a.k.a. algorithmic skeletons [10]). Directive-based approaches such as OpenACC [11], OpenMP [12] and OpenMPC [13] automatically generate the OpenCL and/or CUDA host code from sequential program code. However, they also automatically generate and execute the kernel code, thereby preventing the programmer from hand-optimizing the kernels as often required for highest performance [1]. Maat [14], ViennaCL [15], Maestro [16], Boost.Compute [17] and HPL [18] are built on top of OpenCL and 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 [19] enable implementing OpenCL/CUDA host code in the simple-to-use Python programming language, but still require from the programmer to explicitly deal with low-level details such as data transfers and synchronization. Multi-device Controllers [20], PACXX [21], SYCL [22] and OmpSs [23] allow conveniently programming

978-1-5386-7308-9/18/$31.00 ©2018 IEEE
DOI 10.1109/ICPADS.2018.00061

OpenCL and/or CUDA-capable devices, while StarPU [24], PEPPHER [25] and ClusterSs [26] 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 via pinned main memory.

In this paper, we present *OCAL (the OpenCL/CUDA Abstraction Layer)* – a novel approach to OpenCL and CUDA host code programming. OCAL is implemented as a C++ library, and it combines five major advantages over the state-of-the-art high-level 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 simplifies data-transfer optimizations by providing to the user different, specially-optimized memory buffers, e.g., for conveniently using pinned main memory; 4) it optimizes memory management by automatically detecting and avoiding unnecessary data transfers; 5) it enables interoperability between OpenCL and CUDA host code by automatically handling the communication between the OpenCL and CUDA low-level APIs and by automatically translating between the OpenCL and CUDA programming constructs. Moreover, we demonstrate that OCAL is compatible with existing OpenCL and CUDA libraries, and that OCAL allows to conveniently profile the runtime of OpenCL and CUDA programs.

II. Illustration of OCAL

To illustrate the API design of our OCAL approach, we use a simple, demonstrative example: summing all elements of a vector (a.k.a. *reduction*) in CUDA using system's (possibly different) GPUs.

A. Using OCAL for CUDA Host Code

Listing 1 is the original NVIDIA's CUDA *reduction* kernel provided in [27]. 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 summed 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 [27].

Listing 2 shows an excerpt of the CUDA host code for executing the reduction kernel cooperatively on all of system's CUDA-capable devices; the code is provided by NVIDIA in [27]. It comprises boilerplate low-level functions, such as `cudaMalloc` and `cudaMallocHost` for allocating device and main memory (lines 14-17), `cudaMemcpyAsync` for performing data transfers between main memory and devices' memories (lines 25 and 27), `cudaStreamCreate` for creating the so-called *CUDA streams* (line 13) – they are required to coordinate data transfers and the execution of kernels on the CUDA devices – and `cudaStreamSynchronize` for synchronization (line 34).

Listing 3 demonstrates the OCAL host code that is equivalent to the NVIDIA's low-level host code in Listing 2. OCAL is implemented as a C++ header-only library, thereby freeing the user from the burdens of compiling, packaging and installing; to use OCAL, 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) Choose Devices: In OCAL, system's devices are represented as objects of the class `ocal::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_devices<CUDA>` (line 8) which constructs one `ocal::device<CUDA>` object per CUDA device and returns the constructed device objects in form of a C++ vector. For constructing the device objects, OCAL automatically performs the interactions with the low-level CUDA API to automatically determine and mange the target devices' CUDA ids (Listing 2, lines 4-5, 12, 24, 33) and to initialize and handle the low-level CUDA streams (lines 13, 25-27, 34, 42) – per default, 32 streams per device, enabling simultaneously executing multiple kernels on a device and consequently a better hardware utilization (a.k.a. *Hyper-Q* in NVIDIA terminology [28]). The device id and CUDA streams are encapsulated in the OCAL device objects to hide them from the user.

The user can also choose a specific CUDA device. For this, he initializes an `ocal::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 OCAL user declares an object of class `ocal::kernel` (Listing 3, line 11) for each CUDA kernel to be executed on one of system's devices. OCAL kernels are initialized by the kernel's source code in its string representation, using either 1) the OCAL-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, OCAL automatically extracts kernel's name using the C++ regular expression library [29]; otherwise, the user passes the target kernel's name to the OCAL kernel. Optionally, the user can also pass CUDA compiler flags to the

```
1  int main(int argc, char **argv)
2  {
3      // initialization
4      int i, j, gpuBase, GPU_N;
5      cudaGetDeviceCount(&GPU_N);
6      //...
7
8      /* ... prepare input data ... */
9
10     // Allocate device and host memory
11     for (i = 0; i < GPU_N; i++) {
12         cudaSetDevice(i));
13         cudaStreamCreate(&plan[i].stream));
14         cudaMalloc((void **)&plan[i].d_Data, plan[i
               ].dataN * sizeof(float));
15         cudaMalloc((void **)&plan[i].d_Sum, ACCUM_N
               * sizeof(float));
16         cudaMallocHost((void **)&plan[i].
               h_Sum_from_device, ACCUM_N * sizeof(
               float));
17         cudaMallocHost((void **)&plan[i].h_Data,
               plan[i].dataN * sizeof(float));
18         for (j = 0; j < plan[i].dataN; j++)
19           plan[i].h_Data[j] = (float)rand()/(float)
                 RAND_MAX;
20     }
21
22     // Perform data transfers and start device
           computations
23     for (i = 0; i < GPU_N; i++) {
24         cudaSetDevice(i);
25         cudaMemcpyAsync(plan[i].d_Data, plan[i].
               h_Data, plan[i].dataN * sizeof(float),
               cudaMemcpyHostToDevice, plan[i].stream);
26         reduceKernel<<<BLOCK_N, THREAD_N, 0, plan[i
               ].stream>>>(plan[i].d_Sum, plan[i].
               d_Data, plan[i].dataN);
27         cudaMemcpyAsync(plan[i].h_Sum_from_device,
               plan[i].d_Sum, ACCUM_N *sizeof(float),
               cudaMemcpyDeviceToHost, plan[i].stream);
28     }
29
30     // combine GPUs' results
31     for (i = 0; i < GPU_N; i++) {
32         float sum;
33         cudaSetDevice(i);
34         cudaStreamSynchronize(plan[i].stream);
35         sum = 0;
36         for (j = 0; j < ACCUM_N; j++)
37           sum += plan[i].h_Sum_from_device[j];
38         *(plan[i].h_Sum) = (float)sum;
39         cudaFreeHost(plan[i].h_Sum_from_device);
40         cudaFree(plan[i].d_Sum);
41         cudaFree(plan[i].d_Data);
42         cudaStreamDestroy(plan[i].stream);
43     }
44
45     /* ... Compare GPU and CPU results ... */
46 }
```

Listing 2. Excerpt of NVIDIA's original CUDA host code taken from [27] for executing the CUDA reduction kernel shown in Listing 1. Boilerplate low-level commands make programming the host code tedious and cumbersome.

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* [30]) by passing kernels in their string representation to OCAL, leading to a better performance. 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 the replacement the user can use the CUDA compiler flag `-D`. The OCAL kernel class contains pre-implemented low-level code – based on NVIDIA's *Runtime Compilation Library (NVRTC)* [27] – which is automatically called by OCAL for compiling the CUDA kernel's code. To minimize the cost for the runtime compilation, OCAL stores the compiled kernels in the OCAL kernel object – and also on the system's hard drive – and reuses it for further computations; this happens transparently for the user.

```
1  #include "ocal.hpp"
2
3  int main()
4  {
5    int N = /* arbitrary chunk size */;
6
7    // 1. choose devices
8    auto devices = ocal::get_all_devices<CUDA>();
9
10   // 2. declare kernel
11   ocal::kernel reduction = cuda::source( /*
         reduction kernel */ );
12
13   const int GS = 32, BS = 256;
14
15   // 3. prepare kernels' inputs
16   ocal::buffer<float> in ( N     * devices.size() );
17   ocal::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 =
30       std::accumulate( out.begin(), out.end(),
31                        std::plus<float>() );
32   std::cout << res << std::endl;
33 }
```

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

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 14-15, 40-41) have to be used for allocating/de-allocating mem-

ory on the target device, and function `cudaMemcpyAsync` (lines 25 and 27) 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 should be read/written on multiple devices, e.g., the partial results of one device should be combined in parallel on another device: the programmer then is in charge of explicitly managing multiple buffers – one per device – and to perform 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 is transferred from the device to main memory. This requires a careful management of the multiple CUDA streams (lines 13, 25-27, 34, 42). Especially, for applications where devices' computations have interdependencies, e.g., the result of one device is used as input on another device, the user has to 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 become especially challenging when multiple streams are used per device (e.g., as done in OCAL for a better hardware utilization – see discussion before).

To free the user from the burdens of preparing low-level CUDA buffers and explicitly managing synchronization, OCAL provides the high-level buffer class `ocal::buffer`: it represents a portion of data that can be used for kernel computations on each of system's devices. For this, OCAL buffers encapsulate one low-level CUDA buffer per used device and a region of main memory – the CUDA buffers and main memory mirror the same data. The OCAL 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., OCAL 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 OCAL ensures correct synchronization for complex applications with interdependent device computations by carefully managing and using CUDA events.

An OCAL buffer (Listing 3, lines 16-17) is passed to an OCAL 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 [29].

OCAL 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 OCAL 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 OCAL 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 [27]; the buffer `out` (line 17) is for the devices' partial results.

4) Start Device Computations: To start device computations, the user chooses an OCAL device object (this is described in Step 1) and passes to it: i) the `ocal::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 OCAL buffer objects which represent low-level CUDA buffers (prepared in Step 3). OCAL then uses the pre-implemented CUDA code of the high-level OCAL classes to automatically allocate devices' memories and main memory, perform data transfers, and execute the kernel.

In our 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 11, 23, 31). For this, we pass to each OCAL device object: 1) the OCAL reduction kernel (Listing 3, line 23), 2) the kernel's corresponding grid and block size `GS` and `BS` (line 24) which we have chosen according to the NVIDIA sample (line 13), and 3) kernel's three input arguments (lines 25-27). The input arguments are: the input buffer `in` comprising the numbers to sum up, the output buffer `out` in which the kernels' partial results are stored, 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 the chunk for each device, OCAL 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.

Functions are called in OCAL asynchronously, i.e., the control returns immediately to the main thread which only blocks when one of kernel's output buffers is accessed in the host code. To differentiate between kernels' input and output buffers, OCAL provides the user with three different *buffer tags*: `read`, `write` and `read_write` (Listing 3, lines 25-26); they signal to OCAL how the kernel accesses a buffer. The tags enable OCAL to automatically build a data dependency graph which OCAL uses transparently for the user in order to 1) coordinate device computations, e.g., a computation does not start until other computations on its input/output buffers have been finished, and 2) avoid unnecessary data transfers, e.g., OCAL 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* [6]), and OCAL 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-optimized low-level host code in Listing 2, the content of buffer `out` is not copied to devices' memories by OCAL 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. OCAL blocks the main thread according to its automatically generated data dependency graph 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 OCAL from devices' memory to main memory, so that they become accessible for function `begin()`.

B. Using OCAL for OpenCL Host Code

OCAL provides in addition to its high-level host code interface for CUDA (as described in Section II-A) a further, analogous high-level interface to simplify programming OpenCL host code. For example, for executing the OpenCL reduction kernel in [31] (which is equivalent to the CUDA kernel in Listing 1), the user only has to slightly modify the CUDA example of Listing 3, as follows: 1) replace function `get_all_devices<CUDA>` in line 8 by function `get_all_devices<OpenCL>` to acquire all OpenCL-compatible devices from OCAL, and 2) set the OCAL kernel object in line 11 to the OpenCL kernel's source code using the OCAL-function `opencl::source`. OCAL 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 GPU. All OCAL optimizations for CUDA host code are also provided by OCAL for OpenCL, e.g., avoiding unnecessary data transfers, caching kernel binaries for reducing JIT-compilation overhead, and using multiple streams (a.k.a. *command queue* in OpenCL terminology) for a better hardware utilization.

III. OpenCL-CUDA Interoperability in OCAL

OCAL allows to arbitrarily combine OCAL host code for OpenCL and CUDA in the same program (we call this *OpenCL-CUDA interoperability*). For example, an OCAL buffer with the results of a CUDA computation can be passed to an OpenCL device object to be further processed in parallel on system's multi-core CPU. Moreover, OCAL allows executing a CUDA kernel on an OpenCL device for portability reasons [32], e.g., to perform a CUDA kernel on an Intel multi-core CPU, and OCAL also allows to execute an OpenCL kernel on a CUDA device for higher performance – the CUDA compiler often generates more efficient machine code for NVIDIA devices than OpenCL's compiler [1]. For this, OCAL 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 [29] and has some limitations: advanced C++ features such as automatic type deduction and template meta programming are not supported yet.

Listing 4 demonstrates how OCAL is used to utilize system's multi-core CPU in our OCAL 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 OCAL 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 to the OCAL OpenCL device object: 1) the reduction kernel (line 8) – it comprises CUDA device code which is automatically translated by OCAL to the equivalent OpenCL code so that it can be executed 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) the OCAL buffer `out` (Listing 3, line 17), ii) the buffer `cpu_res` for CPU's partial results (Listing 4, line 6), and iii) the input size, i.e, the number of floating numbers that are comprised by 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 that is internally managed by buffer `out`. OCAL copies the results, according to its interoperability feature, transparently from the user to an OpenCL data structure so that it can be passed to the OpenCL reduction kernel.

Note that in Listing 4, we set the execution configuration, 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 OCAL by using the OCAL function `nd_range`, rather than `dim3`. OCAL allows the user to arbitrarily choose between setting the execution configuration as either grid and

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

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

block size (using OCAL'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 OCAL's source-to-source translation feature in the reverse direction – from OpenCL to CUDA – contributes to a better kernel performance due to the usually higher efficiency of CUDA for NVIDIA devices as compared to OpenCL [33].

Figure 1 demonstrates the speedups of the OpenCL GEMM kernel (GEneral Matrix Multiplication) of the popular OpenCL BLAS library CLBlast [34] on an NVIDIA Tesla K20 GPU; the bars show the speedup of the kernel when translated by OCAL to CUDA over their initial OpenCL implementation. We show the results for 20 input sizes that are heavily used in the deep learning framework Caffe [35]; as concrete neural network, we use Caffe's *siamese* sample for handwriting recognition [36]. We observe that using an equivalent CUDA kernel for CLBlast's OpenCL GEMM kernel leads to speedups of up to 2. This is because CUDA generates and executes more efficient NVIDIA machine code as compared to OpenCL [1]. The overhead for the translation — 250ms on our system — is negligible because once the GEMM kernel is translated to CUDA, it is automatically stored by OCAL and reused for each new call – in the siamese sample, GEMM is called over $> 10^6$ times on each of the input sizes in Figure 1.

Listing 5 demonstrates that using OCAL, 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 an OCAL CUDA device object (declared in line 3); OCAL then automatically translates the OpenCL code to CUDA, and uses the CUDA framework for executing the translated kernel.

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

Listing 5. Using OCAL for conveniently executing the CLBlast's OpenCL GEMM kernel in the CUDA framework for higher performance.

IV. Advanced OCAL usage

A. Data Transfer Optimizations

OCAL provides in addition to its standard buffer type `ocal::buffer` (introduced in Section II), two further buffer types: 1) `ocal::pinned_buffer`, and 2) `ocal::unified_buffer`; both are used analogously to OCAL's standard buffer type. As compared to an OCAL standard buffer, OCAL's pinned buffer uses internally *pinned main memory* which enables fast data transfers between main memory and devices' memories [4], and pinned memory is also required for overlapping data transfers with device computations [37]. However, since pinned memory has a high allocation time, it should only be used if many data transfers are performed. OCAL's unified buffer type uses *unified memory* which is beneficial when kernels access main memory sparsely and the target device provides a hardware support for unified memory [38]. Especially when targeting CPUs, using unified memory (a.k.a. *zero-copy buffer* in OpenCL [3]) avoids data transfers since CPUs' device memories coincides with system's main memory [3].

The OpenCL and CUDA documents [3], [4] recommend the programmer to straightforwardly test which allocation type – naive, pinned or unified – suits best for their applications, dependent on the target hardware. However, testing the special allocation types – pinned and unified – requires a significant effort from the programmer. For example, for using pinned memory in standard OpenCL, the user has to initialize an OpenCL-specific `cl_mem` object using the special flag `CL_MEM_ALLOC_HOST_PTR`, and he 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 the user has to use multiple OpenCL command queues to enable overlapping data transfers with computations [37].

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

Figure 2 (left) shows the runtime of Intel's hand-optimized OpenCL `ZeroCopy` benchmark [39] for evaluating unified memory on an Intel Xeon E5 CPU, compared to the runtime of the equivalent OCAL program which uses OCAL'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 OCAL achieves a competitive runtime with the low-level OpenCL code. This is because OCAL'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` [37] with OCAL 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. OCAL shows the same performance as the low-level CUDA code: OCAL's pinned buffers use internally pinned main memory, analogously to the NVIDIA's

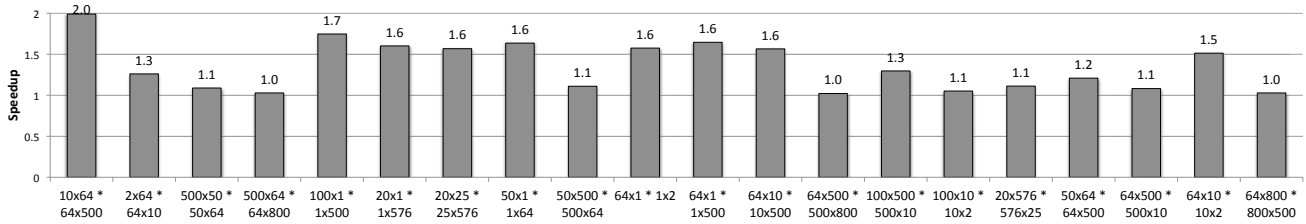

Fig. 1. Speedup of CLBlast's OpenCL GEMM kernel [34] (higher is better) when translated with OCAL to CUDA as compared to its original OpenCL implementation on an NVIDIA Tesla K20 GPU for 20 input sizes that are heavily used in the deep learning framework Caffe [35].

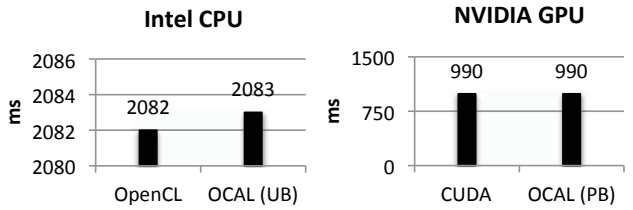

Fig. 2. Runtime comparison of OCAL (lower is better) with low-level OpenCL and CUDA host code for benchmarking the unified memory on Intel CPU (left) and pinned memory on NVIDIA GPU (right). OCAL achieves competitive performance with the hand-optimized low-level code.

benchmark, thus enabling fast data transfers and overlapping the transfers with computations.

B. OCAL Compatibility with existing OpenCL/CUDA Libraries

There is a broad range of existing, expert-implemented OpenCL/CUDA libraries such as the OpenCL linear algebra library CLBlast [34] and the CUDA library cuFFT for Fast Fourier Transforms [27]. To enable compatibility between OCAL and such libraries, OCAL's three buffer types (discussed in Section II and IV-A) can be cast to the native buffer representation of OpenCL and CUDA: `cl_mem` in case of OpenCL and `void*` in case of CUDA. The cast happens either automatically in OCAL – then, the OpenCL/CUDA buffer is returned that belongs to the most recently used device – or, alternatively, the user can use the OCAL buffers' function `get_cuda_buffer(dev)` to get the CUDA buffer for a specific device `dev`. Here, `dev` is either an OCAL device object, the device's name as string, or device's numerical CUDA device id. For OpenCL, the three OCAL buffer types provide the analogous member function `get_opencl_buffer`.

C. Profiling OpenCL/CUDA Programs with OCAL

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

V. Experimental Evaluation

We experimentally prove that OCAL simplifies implementing both OpenCL and CUDA host code, with a low runtime overhead for abstraction.

For the runtime evaluation, we use a system 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. We perform experiments using both the CPUs and GPUs as OpenCL devices. System's two CPUs are represented in OpenCL as a single device with 32 compute units, corresponding to the overall 2×16 logical cores. 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.

We perform our experiments by comparing the expert-implemented, real-world, multi-device code samples provided by Intel [40] and NVIDIA [27] for OpenCL and CUDA with equivalent OCAL 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 following 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 with the equivalent OCAL program in terms of code complexity and runtime.

We measure the code complexity using four classical metrics for the overall programming effort: 1) Lines of Code, excluding blank lines and comments, 2) COCOMO development effort in person months [41], 3) McCabe's cyclomatic complexity [42], and 4) the Halstead development effort [43]. 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 [44], the DE with [45], and HDE with [46].

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

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

Figure 3 compares the code complexity of the OpenCL and CUDA samples with their OCAL counterparts. The kernel code is excluded in our measurements because OCAL and the OpenCL/CUDA samples use the same codes. We observe that OCAL programs are significantly more simple: on average they 1) require $2.72\times$ fewer lines of code (LOC) in case of OpenCL and $1.85\times$ lines in case of CUDA, 2) require a $2.8\times$ less development effort (DE) in case of OpenCL and $1.9\times$ in case of CUDA, 3) have a cyclomatic complexity (CC) that is reduced by a factor of $2.73\times$ for OpenCL and $1.7\times$ for CUDA, and 4) their Halstead development effort (HDE) is reduced by the factor $2.78\times$ (OpenCL) and $1.79\times$ (CUDA). Even for simple applications, e.g., `Scaled-Dot-Product` and `Reduction`, OCAL programs are significantly more simple than their corresponding 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. OCAL programs for OpenCL are more simple than OCAL programs for CUDA when comparing them to low-level code because OpenCL requires boilerplate commands for devices of different vendors while CUDA targets NVIDIA devices only, making programming OpenCL more complex than CUDA.

Figures 4 and 5 demonstrate the speedups (or slowdowns if < 1) of our high-level OCAL programs as compared to their corresponding low-level samples in OpenCL and CUDA. We present results for each of OCAL's three buffer types – buffer (B), pinned buffer (PB), and unified buffer (UB) – for which the OpenCL and CUDA documents [3], [4] recommend to naively test which type suits best for a concrete combination of target application and hardware architecture. The low-level samples all use pinned main memory which corresponds to using OCAL's pinned buffer type (the corresponding bars are filled dark grey for clarification). We run the Intel's

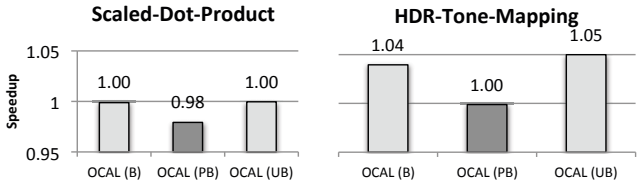

Fig. 4. Speedup/slowdown of OCAL (higher is better) over Intel's OpenCL samples on two Intel Xeon E5 CPUs for each of OCAL's three buffer types: buffer (B), pinned buffer (PB), and unified buffer (UB). The buffer type that corresponds to the memory used in the reference implementations is filled dark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

OpenCL samples on system's Intel CPUs and the NVIDIA CUDA samples on the system's NVIDIA GPUs, according to our reference implementations.

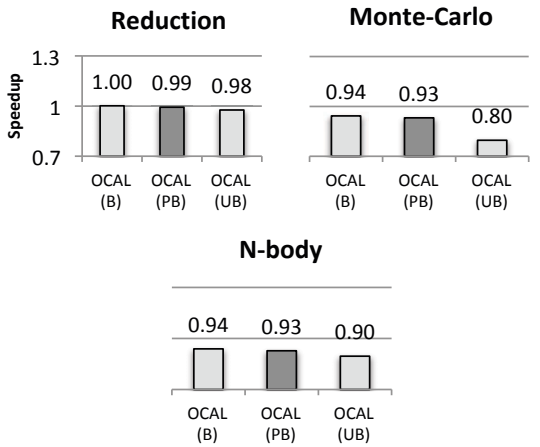

Fig. 5. Speedup/slowdown of OCAL (higher is better) over NVIDIA's CUDA samples on two NVIDIA Tesla K20m GPUs using OCAL's three buffer types: buffer (B), pinned buffer (PB) and unified buffer (UB). The buffer type that corresponds to the memory used in the reference implementations is filled dark grey.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

We observe that OCAL's high-level approach causes a quite low runtime overhead of $< 2\%$ in case of OpenCL and $< 7\%$ in case of CUDA when using pinned memory (dark grey bars) which is the memory type that is also used by the reference implementations. This is due to modern compilers' efficiency – in our case, the clang compiler – which significantly optimize OCALs abstraction overhead, e.g., by performing optimizations such as inline expansion [47]. For the two further OCALs buffer types – buffer and unified buffer – we sometimes observe the same or even slightly better performance of OCAL as compared to the reference implementations. This is because the references use pinned memory which causes a high allocation time for these samples on our system. In contrast, OCAL's buffer and unified buffer types use straightforwardly allocated or unified memory, respectively, causing a lower allocation time (as discussed in Section IV-A). The better performance of OCAL 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, analogously to OCAL, cause slight runtime overhead.

VI. Conclusion

We present OCAL – a high-level approach for conveniently developing OpenCL and CUDA host code. OCAL allows easily executing arbitrary, user-provided OpenCL and CUDA kernels by automatically managing main memory and devices' memories, handling synchronization, minimizing data transfers, and supporting data transfer optimization between device and main memory through high-level buffer classes. Furthermore, OCAL 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 OCAL significantly simplifies host code as compared to standard OpenCL and CUDA, with a low runtime overhead for abstraction.

References

[1] S. Memeti *et al.*, "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption," in *Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, 2017, pp. 1–6.

[2] NVIDIA, "CUDA-MEMCHECK," 2018. [Online]. Available: https://developer.nvidia.com/cuda-memcheck

[3] Intel, "How to Increase Performance by Minimizing Buffer Copies on Intel Processor Graphics," 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, 2018.

[4] NVIDIA, "How to Optimize Data Transfers in CUDA C/C++," https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/, 2018.

[5] M. Aldinucci *et al.*, "The Loop-of-Stencil-Reduce Paradigm," in *IEEE Trustcom/BigDataSE/ISPA*, 2015, pp. 172–177.

[6] J. Enmyren *et al.*, "SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems," in *High-Level Parallel Programming and Applications*, 2010, pp. 5–14.

[7] M. Steuwer *et al.*, "SkelCL - A Portable Skeleton Library for High-Level GPU Programming," in *IEEE IPDPS Workshops*, 2011, pp. 1176–1182.

[8] D. Castro *et al.*, "Farms, Pipes, Streams and Reforestation: Reasoning About Structured Parallel Processes Using Types and Hylomorphisms," in *The International Conference on Functional Programming*, 2016, pp. 4–17.

[9] S. Ernsting *et al.*, "Data Parallel Skeletons for GPU Clusters and Multi-GPU Systems," in *PARCO*, 2011, pp. 509–518.

[10] S. Gorlatch *et al.*, "Parallel Skeletons," 2011, pp. 1417–1422.

[11] S. Wienke *et al.*, "OpenACC — First Experiences with Real-World Applications", booktitle="Euro-Par Parallel Processing," 2012, pp. 859–870.

[12] L. Dagum *et al.*, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, pp. 46–55, 1998.

[13] S. Lee *et al.*, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.

[14] B. Pérez *et al.*, "Simplifying Programming and Load Balancing of Data Parallel Applications on Heterogeneous Systems," in *Workshop on General Purpose Processing on Graphics Processing Units*, 2016, pp. 42–51.

[15] K. Rupp *et al.*, "Automatic Performance Optimization in ViennaCL for GPUs," in *Parallel/High-Performance Object-Oriented Scientific Computing*, 2010, pp. 1–6.

[16] K. Spafford *et al.*, "Maestro: Data Orchestration and Tuning for OpenCL Devices," in *Euro-Par - Parallel Processing*, 2010.

[17] J. Szuppe, "Boost.Compute: A Parallel Computing Library for C++ Based on OpenCL," in *International Workshop on OpenCL*, 2016, pp. 1–39.

[18] "Improving OpenCL Programmability with the Heterogeneous Programming Library," *Procedia Computer Science*, pp. 110 – 119, 2015.

[19] A. Klöckner *et al.*, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, pp. 157–174, 2012.

[20] A. Moreton-Fernandez *et al.*, "Multi-device Controllers: A Library to Simplify Parallel Heterogeneous Programming," *International Journal of Parallel Programming*, pp. 1–20, 2017.

[21] M. Haidl *et al.*, "PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14," in *LLVM Compiler Infrastructure in HPC*, 2014, pp. 1–11.

[22] R. Reyes *et al.*, "SYCL: Single-source C++ accelerator programming." in *PARCO*, 2015, pp. 673–682.

[23] A. Duran *et al.*, "OmpSs: A Proposal For Programming Heterogeneous Multi-Core Architectures," *Parallel Processing Letters*, pp. 173–193, 2011.

[24] A. Cedric *et al.*, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, pp. 187–198, 2011.

[25] U. Dastgeer *et al.*, "The PEPPHER composition tool: performance-aware composition for GPU-based systems," *Computing*, pp. 1195–1211, 2014.

[26] E. Tejedor *et al.*, "ClusterSs: A Task-based Programming Model for Clusters," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC, 2011, pp. 267–268.

[27] NVIDIA, "CUDA Toolkit 9.1," https://developer.nvidia.com/cuda-toolkit, 2018.

[28] NVIDIA, "Hyper-Q," http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf, 2018.

[29] Standard C++ Foundation Foundation Members, "ISO C++," https://isocpp.org, 2018.

[30] T. Rompf *et al.*, "Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems," in *LIPIcs-Leibniz International Proceedings in Informatics*, 2015, pp. 238–261.

[31] NVIDIA, "OpenCL Samples," https://github.com/sschaetz/nvidia-opencl-examples/, 2012.

[32] P. Du *et al.*, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, pp. 391 – 407, 2012.

[33] K. Karimi *et al.*, "A Performance Comparison of CUDA and OpenCL," *CoRR*, pp. 1–12, 2010.

[34] C. Nugteren, "CLBlast: A Tuned OpenCL BLAS Library," *CoRR*, pp. 1–10, 2017.

[35] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, pp. 675–678, 2014.

[36] G. Koch *et al.*, "Siamese Neural Networks for One-shot Image Recognition," in *ICML Deep Learning Workshop*, 2015.

[37] NVIDIA, "How to Overlap Data Transfers in CUDA C/C++," https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/, 2018.

[38] NVIDIA, "Unified Memory for CUDA Beginners," 2018. [Online]. Available: https://devblogs.nvidia.com/unified-memory-cuda-beginners/

[39] Intel, "Ambient Occlusion Benchmark (AOBench)," 2014. [Online]. Available: http://code.google.com/p/aobench

[40] Intel, "Code Samples," https://software.intel.com/en-us/intel-opencl-support/code-samples, 2018.

[41] B. Boehm *et al.*, "Cost models for future software life cycle processes: COCOMO 2.0"," *Annals of Software Engineering*, pp. 57–94, 1995.

[42] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, pp. 308–320, 1976.

[43] M. H. Halstead, *Elements of software science*, 1977.

[44] Steve Arnold, "CCCC Project Documentation," http://sarnold.github.io/cccc/, 2005.

[45] David A. Wheeler, "SLOCCount," https://www.dwheeler.com/sloccount/, 2018.

[46] rharish100193, "Halstead Metrics Tool," https://sourceforge.net/projects/halsteadmetricstool/, 2016.

[47] P. P. Chang *et al.*, "Inline Function Expansion for Compiling C Programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1989, pp. 246–257.