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

Ari Rasch, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch

University of Münster, Germany

Motivation

- OpenCL and CUDA are state-of-the-art approaches to programming modern multi- and many-core devices.
- OpenCL targets a broad range of devices; CUDA is for NVIDIA devices only (better performance than OpenCL).
- **Common problem:** *Host Code* is required for executing OpenCL/ CUDA programs (a.k.a. *kernel*).

OpenCL

NVIDIA
CUDA

Implementing host code is cumbersome and tedious because of:

- boilerplate low-level commands, e.g., for memory allocations and data transfers;
- explicitly managing memory and synchronization (of multiple devices);
- mixing OpenCL and CUDA host code for systems with devices from different vendors;
- data transfer optimizations, e.g., using *pinned/unified memory.*

Example: **CUDA Host Code For *Parallel Reduction***

Original NVIDIA CUDA kernel for parallel reduction:

```
__global__ static
void reduceKernel(float *d_Result, float *d_Input, int N)
{
    const int     tid = blockIdx.x * blockDim.x + threadIdx.x;
    const int threadN = gridDim.x * blockDim.x;
    float         sum = 0;

    for (int pos = tid; pos < N; pos += threadN)
        sum += d_Input[pos];

    d_Result[tid] = sum;
}
```

Example: CUDA Host Code For *Parallel Reduction*

Excerpt of original CUDA host code for executing the reduction kernel:

```
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 ... */
}
```

Example: CUDA Host Code For *Parallel Reduction*

Excerpt of original CUDA host code for executing the reduction kernel:

```
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 ... */
}
```

Boilerplate
low-level functions
for

Example: CUDA Host Code For *Parallel Reduction*

Excerpt of original CUDA host code for executing the reduction kernel:

```
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 ... */
}
```

Boilerplate low-level functions for (de)allocating device/host memory

Example: CUDA Host Code For *Parallel Reduction*

Excerpt of original CUDA host code for executing the reduction kernel:

```
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 ... */
}
```

Boilerplate low-level functions for H2D/D2H data transfers

Example: CUDA Host Code For *Parallel Reduction*

Excerpt of original CUDA host code for executing the reduction kernel:

```
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 ... */
}
```

Boilerplate low-level functions for creating/using *CUDA streams*

Example: CUDA Host Code For *Parallel Reduction*

Excerpt of original CUDA host code for executing the reduction kernel:

```
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 ... */
}
```

Boilerplate low-level functions for synchronization

Related Work

State of the art focuses on only particular host programming challenges:

1. **Skeleton Approaches** and **OpenMP, OpenACC, OpenMPC**
   + Simplify host programming by providing pre-implemented parallel patterns/directives.
   - No support for arbitrary OpenCL/CUDA kernels.
2. **ViennaCL, Maestro, Maat, Boost.Compute, HPL**
   + Simplify launching OpenCL kernels.
   - No support for CUDA.
3. **pyOpenCL, pyCUDA**
   + Enable implementing OpenCL/CUDA host code in the simple-to-use Python programming language.
   - Still require from the programmer to explicitly deal with low-level details.
4. **Multi-Device Controller, PACXX, SYCL, OmpSs** and **StarPU, PEPPHER, ClusterSs**
   + Allow conveniently programming OpenCL/CUDA-capable devices or simply scheduling tasks over such devices.
   - Do not support data transfer optimization.

What is OCAL?

OCAL (*OpenCL/CUDA Abstraction Layer*) is a **novel C++ library** for **simplifying OpenCL and CUDA host code** programming **by abstracting from low-level details**.

OCAL combines major advantages over state-of-the-art approaches:

1. simplifies implementing both OpenCL and CUDA host code;
2. allows executing arbitrary OpenCL and CUDA kernels;
3. manages host and devices’ memory;
4. enables interoperability between OpenCL and CUDA host code;
5. supports data-transfer optimizations.

Illustration of OCAL

In the following: We demonstrate the (high-level) OCAL host code that is equivalent to the NVIDIA's (low-level) host code for parallel reduction.

```
#include "ocal.hpp"

int main()
{
  int N = /* arbitrary chunk size */;

  // 1. choose devices
  auto devices = ocal::get_all_devices<CUDA>();

  // 2. declare kernel
  ocal::kernel reduction = cuda::source(
                                      /* kernel */ );

  const int GS = 32, BS = 256;

  // 3. prepare kernels' inputs
  ocal::buffer<float> in ( N     * devices.size() );
  ocal::buffer<float> out( GS*BS * devices.size() );

  std::generate(in.begin(), in.end(), std::rand);

  // 4. start device computations
  for( auto& dev : devices )
    dev( reduction                              )
       ( dim3( GS ), dim3( BS )                 )
       ( write(out.begin()+dev.id()* GS*BS, GS*BS ),
         read (in.begin() +dev.id()* N    , N     ),
         N                                      );

  auto res = std::accumulate( out.begin(), out.end(),
                               std::plus<float>() );

  std::cout << res << std::endl;
}
```

Illustration of OCAL

In the following: We demonstrate the (high-level) OCAL host code that is equivalent to the NVIDIA's (low-level) host code for parallel reduction.

```
#include "ocal.hpp"

int main()
{
  int N = /* arbitrary chunk size */;

  // 1. choose devices
  auto devices = ocal::get_all_devices<CUDA>();

  // 2. declare kernel
  ocal::kernel reduction = cuda::source(
                                    /* kernel */ );

  const int GS = 32, BS = 256;

  // 3. prepare kernels' inputs
  ocal::buffer<float> in ( N     * devices.size() );
  ocal::buffer<float> out( GS*BS * devices.size() );

  std::generate(in.begin(), in.end(), std::rand);

  // 4. start device computations
  for( auto& dev : devices )
    dev( reduction                                 )
       ( dim3( GS ), dim3( BS )                    )
       ( write(out.begin()+dev.id()* GS*BS, GS*BS ),
         read (in.begin() +dev.id()* N    , N     ),
         N                                         );

  auto res = std::accumulate( out.begin(), out.end(),
                              std::plus<float>() );

  std::cout << res << std::endl;
}
```

1. Choose Devices:

- Function `ocal::get_all_devices<CUDA>` returns an `std::vector` comprising all of system's CUDA-capable devices.
- Devices are represented as objects of high-level class `ocal::device`.
- An `ocal::device` can also be chosen conveniently by either:
  - i) its name, e.g., *Tesla K20*,
  - ii) its CUDA device id,
  - iii) some of its device properties, e.g., support for double precision and atomic operations.

OCAL automatically performs low-level interactions with CUDA API for:

- **acquiring/setting devices,**
- **stream management,**
- **string operations,**
- **for getting/setting device properties.**

Illustration of OCAL

In the following: We demonstrate the (high-level) OCAL host code that is equivalent to the NVIDIA's (low-level) host code for parallel reduction.

```
#include "ocal.hpp"

int main()
{
  int N = /* arbitrary chunk size */;

  // 1. choose devices
  auto devices = ocal::get_all_devices<CUDA>();

  // 2. declare kernel
  ocal::kernel reduction = cuda::source(
                                    /* kernel */ );

  const int GS = 32, BS = 256;

  // 3. prepare kernels' inputs
  ocal::buffer<float> in ( N     * devices.size() );
  ocal::buffer<float> out( GS*BS * devices.size() );

  std::generate(in.begin(), in.end(), std::rand);

  // 4. start device computations
  for( auto& dev : devices )
    dev( reduction                                 )
       ( dim3( GS ), dim3( BS )                    )
       ( write(out.begin()+dev.id()* GS*BS, GS*BS ),
         read (in.begin() +dev.id()* N    , N     ),
         N                                         );

  auto res = std::accumulate( out.begin(), out.end(),
                              std::plus<float>() );

  std::cout << res << std::endl;
}
```

2. Define Kernel:

- Kernels are represented as objects of class `ocal::kernel`.
- Initialization with either `cuda::source( "…" )` or `cuda::path( "…" )`.
- CUDA compiler flags can be set, e.g., `-maxrregcount`.
- Kernel binaries are automatically saved to system's hard drive (reduces compilation overhead).

OCAL automatically performs low-level interactions with CUDA API for:

- **JIT compilation,**
- **storing/loading kernel binaries,**
- **file stream operations.**

Illustration of OCAL

In the following: We demonstrate the (high-level) OCAL host code that is equivalent to the NVIDIA's (low-level) host code for parallel reduction.

```
#include "ocal.hpp"

int main()
{
  int N = /* arbitrary chunk size */;

  // 1. choose devices
  auto devices = ocal::get_all_devices<CUDA>();

  // 2. declare kernel
  ocal::kernel reduction = cuda::source(
                                      /* kernel */ );

  const int GS = 32, BS = 256;

  // 3. prepare kernels' inputs
  ocal::buffer<float> in ( N     * devices.size() );
  ocal::buffer<float> out( GS*BS * devices.size() );

  std::generate(in.begin(), in.end(), std::rand);

  // 4. start device computations
  for( auto& dev : devices )
    dev( reduction                                )
       ( dim3( GS ), dim3( BS )                   )
       ( write(out.begin()+dev.id()* GS*BS, GS*BS ),
         read (in.begin() +dev.id()* N    , N     ),
         N                                        );

  auto res = std::accumulate( out.begin(), out.end(),
                              std::plus<float>() );

  std::cout << res << std::endl;
}
```

3. Prepare Kernel's Input:

- Fundamental and vector types are prepared straightforwardly.
- Input/output buffers require preparation → OCAL provides high-level buffer class `ocal::buffer`.
- Automatically mirror data in host/devices memories by performing data transfers when necessary and carefully performing synchronization.
- OCAL buffers are compatible with the *C++ Standard Template Library (STL)*.

OCAL automatically performs low-level interactions with CUDA API for:

- **host and device memory allocations/deallocations,**
- **data transfers,**
- **synchronization.**

Illustration of OCAL

In the following: We demonstrate the (high-level) OCAL host code that is equivalent to the NVIDIA’s (low-level) host code for parallel reduction.

```
#include "ocal.hpp"

int main()
{
  int N = /* arbitrary chunk size */;

  // 1. choose devices
  auto devices = ocal::get_all_devices<CUDA>();

  // 2. declare kernel
  ocal::kernel reduction = cuda::source(
                                      /* kernel */ );

  const int GS = 32, BS = 256;

  // 3. prepare kernels' inputs
  ocal::buffer<float> in ( N     * devices.size() );
  ocal::buffer<float> out( GS*BS * devices.size() );

  std::generate(in.begin(), in.end(), std::rand);

  // 4. start device computations
  for( auto& dev : devices )
    dev( reduction                                )
       ( dim3( GS ), dim3( BS )                   )
       ( write(out.begin()+dev.id()* GS*BS, GS*BS ),
         read (in.begin() +dev.id()* N    , N     ),
         N                                        );

  auto res = std::accumulate( out.begin(), out.end(),
                               std::plus<float>() );

  std::cout << res << std::endl;
}
```

4. Start Device Computations:

Device computations are started by passing to an OCAL device object:

1. an `ocal::kernel` object;
2. the *execution configuration*: number of thread blocks (GS) and threads (BS);
3. Kernel arguments:
   - scalars,
   - vector types (e.g., `float4`),
   - `ocal::buffers`

OCAL automatically performs low-level interactions with CUDA API for:

- **setting kernel arguments,**
- **starting kernel,**
- **synchronization.**

OCAL for OpenCL Host Code

OCAL can also be used the same for OpenCL host code:

```
#include "ocal.hpp"

int main()
{
  int N = /* arbitrary chunk size */;

  // 1. choose devices
  auto devices = ocal::get_all_devices<CUDA>();

  // 2. declare kernel
  ocal::kernel reduction = cuda::source(
                                    /* kernel */ );

  const int GS = 32, BS = 256;

  // 3. prepare kernels' inputs
  ocal::buffer<float> in ( N     * devices.size() );
  ocal::buffer<float> out( GS*BS * devices.size() );

  std::generate(in.begin(), in.end(), std::rand);

  // 4. start device computations
  for( auto& dev : devices )
    dev( reduction                                )
       ( dim3( GS ), dim3( BS )                   )
       ( write(out.begin()+dev.id()* GS*BS, GS*BS ),
         read (in.begin() +dev.id()* N    , N     ),
         N                                        );

  auto res = std::accumulate( out.begin(), out.end(),
                               std::plus<float>() );

  std::cout << res << std::endl;
}
```

The OCAL host code for CUDA reduction has to by only slightly modified for OpenCL:

OCAL for OpenCL Host Code

OCAL can also be used the same for OpenCL host code:

```
#include "ocal.hpp"

int main()
{
  int N = /* arbitrary chunk size */;

  // 1. choose devices
  auto devices = ocal::get_all_devices<CUDA>();

  // 2. declare kernel
  ocal::kernel reduction = cuda::source(
                                    /* kernel */ );

  const int GS = 32, BS = 256;

  // 3. prepare kernels' inputs
  ocal::buffer<float> in ( N     * devices.size() );
  ocal::buffer<float> out( GS*BS * devices.size() );

  std::generate(in.begin(), in.end(), std::rand);

  // 4. start device computations
  for( auto& dev : devices )
    dev( reduction                             )
       ( dim3( GS ), dim3( BS )                )
       ( write(out.begin()+dev.id()* GS*BS, GS*BS ),
         read (in.begin() +dev.id()* N    , N     ),
         N                                     );

  auto res = std::accumulate( out.begin(), out.end(),
                              std::plus<float>() );

  std::cout << res << std::endl;
}
```

The OCAL host code for CUDA reduction has to by only slightly modified for OpenCL:

1. `get_all_devices<CUDA>()`
   → `get_all_devices<OCL>()`

OCAL for OpenCL Host Code

<u>OCAL can also be used the same for OpenCL host code:</u>

```
#include "ocal.hpp"

int main()
{
  int N = /* arbitrary chunk size */;

  // 1. choose devices
  auto devices = ocal::get_all_devices<CUDA>();

  // 2. declare kernel
  ocal::kernel reduction = cuda::source(
                                      /* kernel */ );

  const int GS = 32, BS = 256;

  // 3. prepare kernels' inputs
  ocal::buffer<float> in ( N     * devices.size() );
  ocal::buffer<float> out( GS*BS * devices.size() );

  std::generate(in.begin(), in.end(), std::rand);

  // 4. start device computations
  for( auto& dev : devices )
    dev( reduction                                )
       ( dim3( GS ), dim3( BS )                   )
       ( write(out.begin()+dev.id()* GS*BS, GS*BS ),
         read (in.begin() +dev.id()* N    , N     ),
         N                                        );

  auto res = std::accumulate( out.begin(), out.end(),
                               std::plus<float>() );

  std::cout << res << std::endl;
}
```

The OCAL host code for CUDA reduction has to by only slightly modified for OpenCL:

1. `get_all_devices<CUDA>()`
   ➞ `get_all_devices<OCL>()`

2. `cuda::source( “…” )`
   ➞ `ocl::source( “…” )`

OCAL for OpenCL Host Code

OCAL can also be used the same for OpenCL host code:

```
#include "ocal.hpp"

int main()
{
  int N = /* arbitrary chunk size */;

  // 1. choose devices
  auto devices = ocal::get_all_devices<CUDA>();

  // 2. declare kernel
  ocal::kernel reduction = cuda::source(
                                      /* kernel */ );

  const int GS = 32, BS = 256;

  // 3. prepare kernels' inputs
  ocal::buffer<float> in ( N     * devices.size() );
  ocal::buffer<float> out( GS*BS * devices.size() );

  std::generate(in.begin(), in.end(), std::rand);

  // 4. start device computations
  for( auto& dev : devices )
    dev( reduction                                 )
       ( dim3( GS ), dim3( BS )                    )
       ( write(out.begin()+dev.id()* GS*BS, GS*BS ),
         read (in.begin() +dev.id()* N    , N     ),
         N                                        );

  auto res = std::accumulate( out.begin(), out.end(),
                              std::plus<float>() );

  std::cout << res << std::endl;
}
```

The OCAL host code for CUDA reduction has to by only slightly modified for OpenCL:

1. `get_all_devices<CUDA>()`
   → `get_all_devices<OCL>()`

2. `cuda::source( “…” )`
   → `ocl::source( “…” )`

OpenCL is more challenging than CUDA, because of managing also:

- ***platforms* of different vendors,**
- **so-called *OpenCL contexts*,**
- **multiple kernel binaries — one per platform.**

OpenCL-CUDA Interoperability

OCAL allows mixing OpenCL and CUDA host code in the same program:

- OCAL kernels can be instantiated with both OpenCL or CUDA kernel source code.
  - → kernel code is automatically translated between OpenCL and CUDA.
- OCAL buffers can be passed to both OpenCL and CUDA device.
  - → data is automatically transferred between OpenCL and CUDA data structures.
- The user can arbitrarily choose between setting the execution configuration as either:

  i) *grid and block size* (as in CUDA) via function `dim3(...);`

  ii) *global and local size* (as in OpenCL) via function `nd_range(…);`

Reduction Example:

OCAL allows easily utilizing system's multi-core CPU to combine GPUs' partial results

```
ocal::device<OpenCL_CPU> cpu;

ocal::buffer cpu_res( NUM_CORES*VL );

cpu( reduction                                    )
   ( dim3( NUM_CORES ) , dim3( VL )               )
   ( write( cpu_res ), read( out ), out.size() );

auto res = std::accumulate( cpu_res.begin(),
cpu_res.end(), std::plus<float>() );
```

OpenCL-CUDA Interoperability

OCAL allows mixing OpenCL and CUDA host code in the same program:

- OCAL kernels can be instantiated with both OpenCL or CUDA kernel source code
  - → kernel code is automatically translated between OpenCL and CUDA.
- OCAL buffers can be passed to both OpenCL and CUDA kernels
  - → data is automatically transferred between OpenCL and CUDA data structures.
- The user can arbitrarily choose between setting the execution configuration as either:
  - i) *grid and block size* (as in CUDA) via function `dim3(...);`
  - ii) *global and local size* (as in OpenCL) via function `nd_range(…);`

Reduction Example:

OCAL allows easily utilizing system's multi-core CPU to combine GPUs' partial results

```
ocal::device<OCL_CPU> cpu;

ocal::buffer cpu_res( NUM_CORES*VL );

cpu( reduction                                   )
   ( dim3( NUM_CORES ) , dim3( VL )              )
   ( write( cpu_res ), read( out ), out.size() );

auto res = std::accumulate( cpu_res.begin(),
cpu_res.end(), std::plus<float>() );
```

Declare an OCAL OpenCL device object to target system's CPU

OpenCL-CUDA Interoperability

OCAL allows mixing OpenCL and CUDA host code in the same program:

- OCAL kernels can be instantiated with both OpenCL or CUDA kernel source code
  - → kernel code is automatically translated between OpenCL and CUDA.
- OCAL buffers can be passed to both OpenCL and CUDA kernels
  - → data is automatically transferred between OpenCL and CUDA data structures.
- The user can arbitrarily choose between setting the execution configuration as either:
  - i) *grid and block size* (as in CUDA) via function `dim3(...);`
  - ii) *global and local size* (as in OpenCL) via function `nd_range(…);`

Reduction Example:

OCAL allows easily utilizing system's multi-core CPU to combine GPUs' partial results

```
ocal::device<OCL_CPU> cpu;

ocal::buffer cpu_res( NUM_CORES*VL );

cpu( reduction                                 )
   ( dim3( NUM_CORES ) , dim3( VL )            )
   ( write( cpu_res ), read( out ), out.size() );

auto res = std::accumulate( cpu_res.begin(),
cpu_res.end(), std::plus<float>() );
```

Pass CUDA reduction kernel to CPU device (→ automatically translated to OpenCL)

OpenCL-CUDA Interoperability

OCAL allows mixing OpenCL and CUDA host code in the same program:

- OCAL kernels can be instantiated with both OpenCL or CUDA kernel source code
  - → kernel code is automatically translated between OpenCL and CUDA.
- OCAL buffers can be passed to both OpenCL and CUDA kernels
  - → data is automatically transferred between OpenCL and CUDA data structures.
- The user can arbitrarily choose between setting the execution configuration as either:
  - i) *grid and block size* (as in CUDA) via function `dim3(...);`
  - ii) *global and local size* (as in OpenCL) via function `nd_range(…);`

Reduction Example:

OCAL allows easily utilizing system's multi-core CPU to combine GPUs' partial results

```
ocal::device<OCL_CPU> cpu;

ocal::buffer cpu_res( NUM_CORES*VL );

cpu( reduction                                 )
   ( dim3( NUM_CORES ) , dim3( VL )            )
   ( write( cpu_res ), read( out ), out.size() );

auto res = std::accumulate( cpu_res.begin(),
cpu_res.end(), std::plus<float>() );
```

Use the OCAL buffer `out` which holds the GPUs' partial results (→ results are automatically copied from low-level CUDA to OpenCL data structure)

OpenCL-CUDA Interoperability

OCAL allows mixing OpenCL and CUDA host code in the same program:

- OCAL kernels can be instantiated with both OpenCL or CUDA kernel source code
  - → kernel code is automatically translated between OpenCL and CUDA.
- OCAL buffers can be passed to both OpenCL and CUDA kernels
  - → data is automatically transferred between OpenCL and CUDA data structures.
- The user can arbitrarily choose between setting the execution configuration as either:
  - i) *grid and block size* (as in CUDA) via function `dim3(...);`
  - ii) *global and local size* (as in OpenCL) via function `nd_range(…);`

Reduction Example:

OCAL allows easily utilizing system’s multi-core CPU to combine GPUs’ partial results

```
ocal::device<OCL_CPU> cpu;

ocal::buffer cpu_res( NUM_CORES*VL );

cpu( reduction                                )
   ( dim3( NUM_CORES ) , dim3( VL )           )
   ( write( cpu_res ), read( out ), out.size() );

auto res = std::accumulate( cpu_res.begin(),
cpu_res.end(), std::plus<float>() );
```

Each used CPU core produces a new partial sum in output buffer `cpu_res`

OpenCL-CUDA Interoperability

OCAL allows mixing OpenCL and CUDA host code in the same program:

- OCAL kernels can be instantiated with both OpenCL or CUDA kernel source code
  - → kernel code is automatically translated between OpenCL and CUDA.
- OCAL buffers can be passed to both OpenCL and CUDA kernels
  - → data is automatically transferred between OpenCL and CUDA data structures.
- The user can arbitrarily choose between setting the execution configuration as either:
  - i) *grid and block size* (as in CUDA) via function `dim3(...);`
  - ii) *global and local size* (as in OpenCL) via function `nd_range(…);`

Reduction Example:

OCAL allows easily utilizing system's multi-core CPU to combine GPUs' partial results

```
ocal::device<OCL_CPU> cpu;

ocal::buffer cpu_res( NUM_CORES*VL );

cpu( reduction                                 )
   ( dim3( NUM_CORES ) , dim3( VL )            )
   ( write( cpu_res ), read( out ), out.size() );

auto res = std::accumulate( cpu_res.begin(), 
cpu_res.end(), std::plus<float>() );
```

The new partial results are accumulated conveniently using STL function `std::accumulate`

Data-Transfer Optimization

Data-transfer optimizations are performed in OpenCL/CUDA via specially-allocated memory:

- **Pinned Main Memory:**
  - \+ enables fast data accesses and overlapping data transfers with device computation
  - \- high allocation time (→ beneficial when many data transfers are performed)
- **Unified memory:**
  - \+ beneficial in case of sparse data accesses
  - \- requires hardware support for high performance

OpenCL and CUDA optimization guides recommend naively test which allocation type suits best, specifically for target system
→ requires significant implementation effort in standard OpenCL/CUDA.

- OCAL provides two specially-optimized buffer types:
  - `ocal::pinned_buffer` → uses internally pinnend main memory;
  - `ocal::unified_buffer` → uses internally unified memory.
- The user can use both conveniently the same as `ocal::buffer`.

Experimental Results

- We compare OCAL to low-level OpenCL and CUDA host code in terms of: i) **code complexity**, and ii) **performance**.

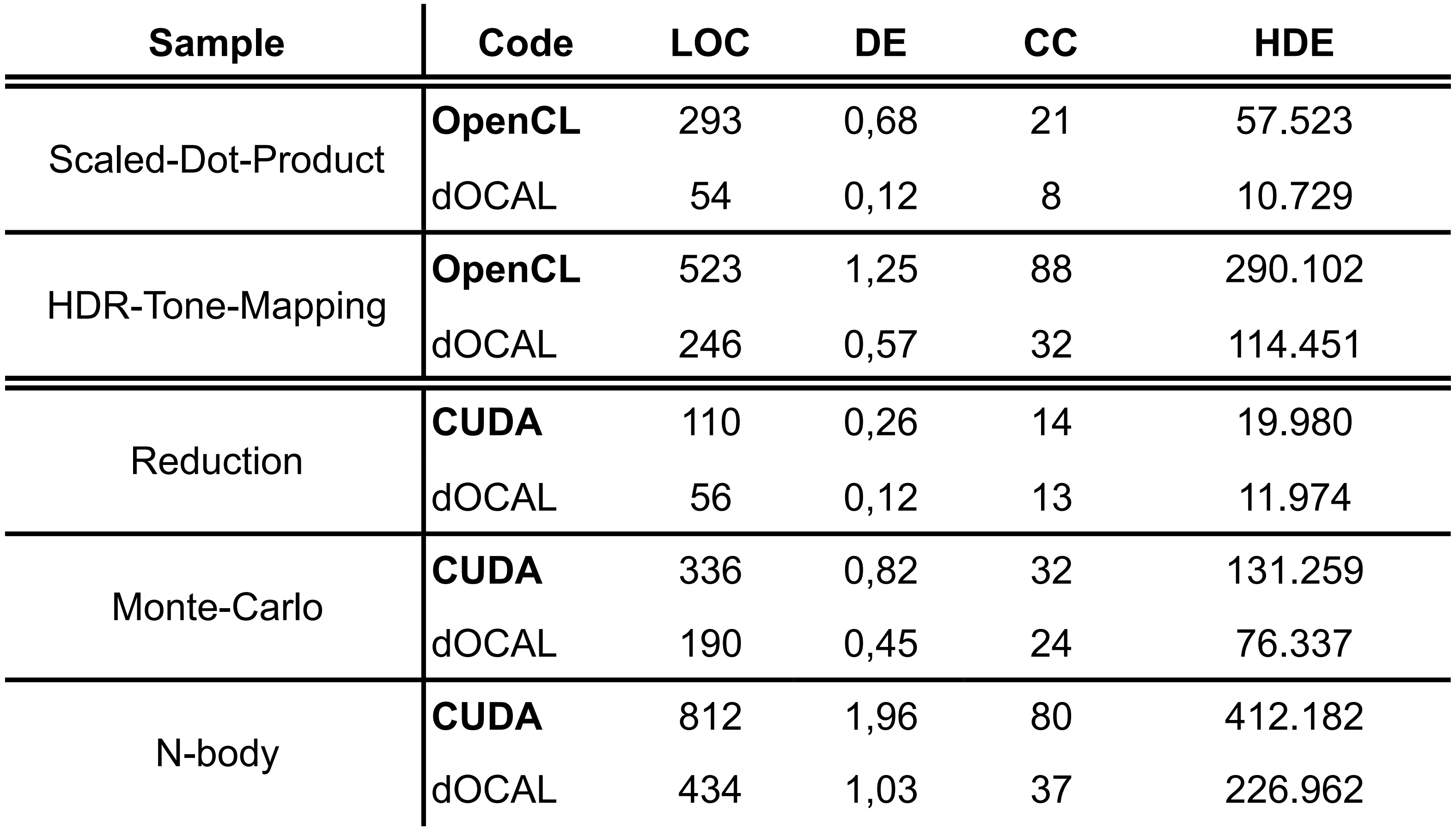
- Experimental setup: two Intel Xeon E5-2640 CPUs; two NVIDIA Tesla K20m GPUs.

- We compare to the hand-optimized **Intel OpenCL samples**:
  i) scaled dot product,
  ii) range tone mapping.

- We compare to the hand-optimized **NVIDIA CUDA samples**:
  i) parallel reduction,
  ii) Monte Carlo simulation,
  iii) N-Body simulation.

Experimental Results

Code complexity of the OpenCL and CUDA samples as compared to their OCAL counterparts using four classical metrics:



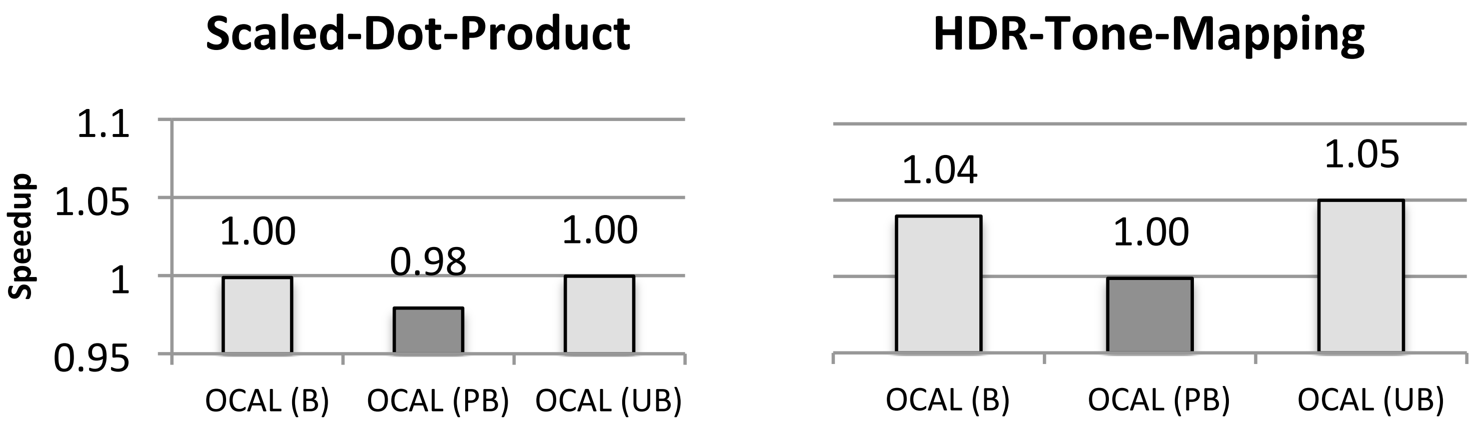
Sample	Code	LOC	DE	CC	HDE
Scaled-Dot-Product	**OpenCL**	293	0,68	21	57.523
	dOCAL	54	0,12	8	10.729
HDR-Tone-Mapping	**OpenCL**	523	1,25	88	290.102
	dOCAL	246	0,57	32	114.451
Reduction	**CUDA**	110	0,26	14	19.980
	dOCAL	56	0,12	13	11.974
Monte-Carlo	**CUDA**	336	0,82	32	131.259
	dOCAL	190	0,45	24	76.337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

Average improvements when OCAL is used for OpenCL/CUDA:

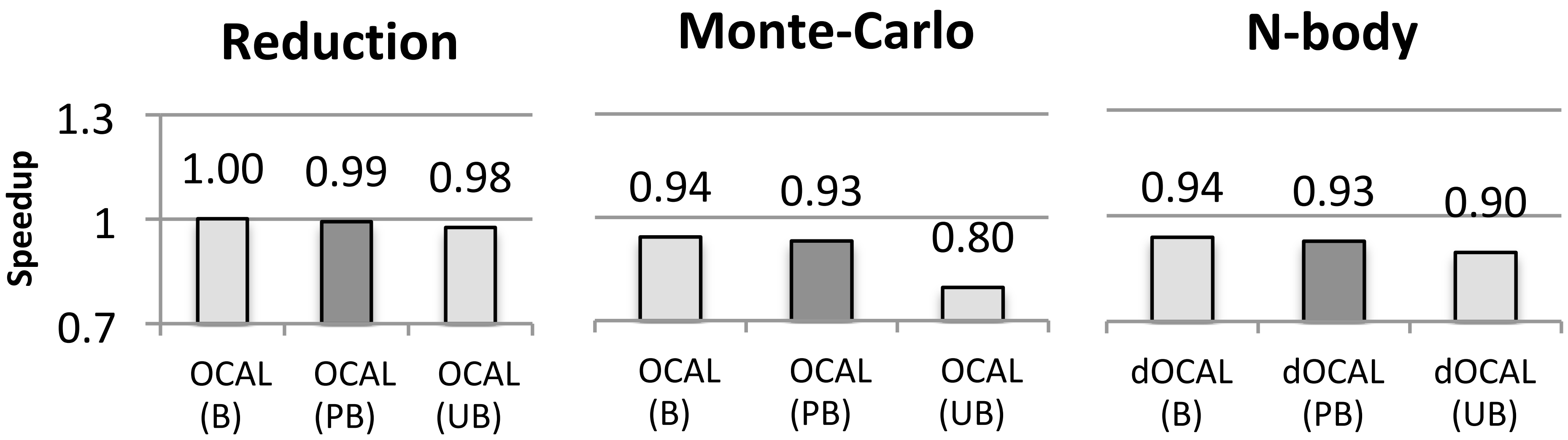
- 2.72x/1.82x fewer *Lines Of Code (LOC)*;
- 2.80x/1.90x fewer *COCOMO Development Effort (DE)*;
- 2.73x/1.70x fewer *Cyclomatic Complexity (CC)*;
- 2.78x/1.79x fewer Halstead Development Effort (HDE).

Experimental Results

We observe that OCAL code is competitive to low-level OpenCL/CUDA host code:



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).



Speedup/slowdown of OCAL (higher is better) over NVIDIA's CUDA samples on two NVIDIA Tesla K20 GPUs for each of OCAL's three buffer types: Buffer (B), Pinned Buffer (PB), and Unified Buffer (UB).

Conclusion

We have seen:

- OCAL simplifies programming OpenCL and CUDA host code.
- OCAL supports mixing OpenCL and CUDA host code (interoperability).
- OCAL supports data-transfer optimizations.
- OCAL causes a quiet low runtime overhead.

Moreover:

- OCAL is compatible with OpenCL/CUDA libraries (e.g., cuDNN).
- OCAL allows conveniently profiling OpenCL/CUDA programs.

Future work:

- OCAL extended for distributed systems (*dOCAL*).
- OCAL supporting interconnecting with *auto-tuning systems*.

Questions?