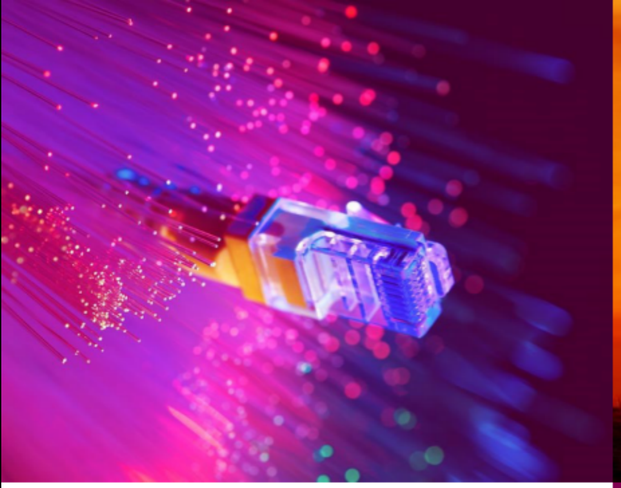

Header Only Porting: a light-weight header-only library for CUDA/HIP porting

CUG 2023

Martti Louhivuori, CSC - IT Center for Science, Finland

CSC – Finnish expertise in ICT for research, education and public administration

HIP: portable interface for GPUs

- *HIP is a C++ Runtime API and Kernel Language that allows developers to create portable applications for AMD and NVIDIA GPUs from single source code.*
- Very close to CUDA, almost a one-to-one mapping
- Needs either a ROCm or a CUDA backend, HIP is just a thin layer on top of them
- Nice tooling (hipify) available to convert existing CUDA codes to HIP

HIP from an application developer perspective

- Appealing promise of a portable low-level GPU language / API
- New kid in the block
  - uncertain how widely it will be adopted by the HPC community
- Is HIP going to be available also on NVIDIA systems? At the moment, mostly not
  - no tools to convert back to CUDA (one-way porting)

How to support both NVIDIA and AMD GPUs?

1. **Convert from CUDA to HIP at compile time** on AMD systems
2. Implement and **support both CUDA and HIP versions**
3. **Write the code in HIP** and hope for the availability of HIP on NVIDIA systems
4. **Use a higher level abstraction**, such as OpenMP, SYCL, or a HPC framework (Kokkos etc.)
5. ***Use the Header Only Porting approach***!

- If one needs to support also e.g. Intel GPUs, one is pretty much limited to using a higher level abstraction (#4)

Porting an existing CUDA/HIP code

		no code modifications		just compile and run	
approach	**source code**	**from CUDA**	**from HIP**	**on NVIDIA**	**on AMD**
CUDA (+ hipify)	CUDA	✔	✘	✔	✘
HIP only	HIP	✘	✔	✔/✘	✔
Support both	CUDA & HIP	✘	✘	✔	✔
OpenMP / SYCL / Kokkos	-	✘	✘	✔	✔
Header Only Porting	either	✔	✔	✔	✔

HOP: Header Only Porting

- Header Only Porting (HOP) is a light-weight header-only library for CUDA/HIP porting
  - for C and C++ codes (also Fortran with ISO C bindings)
  - no code modifications needed
  - just add a few extra flags at compile time to hop from CUDA to HIP or back
- Leverages the almost one-to-one mapping between CUDA and HIP
  - catches include statements
  - redefines identifiers

HOP: how does it work?

- Redefines identifiers using preprocessor directives

  `cudaMalloc` ⇔ `hipMalloc` etc.

- Catches include statements by providing alternative header files that take precedence over the original ones
  - source identifiers are redefined to target identifiers
  - target GPU backend needs to be defined (CUDA or HIP)
  - e.g. `"#include <hip/hip_runtime.h>"` will actually load a HOP header file that does a translation from HIP identifiers to CUDA identifiers

Example: redefine identifiers

translate from source (HIP):

```
#define hipMalloc                   gpuMalloc
#define hipMallocAsync              gpuMallocAsync
#define hipHostMalloc               gpuHostMalloc
#define hipHostMallocPortable       gpuHostMallocPortable
#define hipMemcpy                   gpuMemcpy
```

translate to target (CUDA):

```
#include <cuda_runtime_api.h>

#define gpuMalloc                   cudaMalloc
#define gpuMallocAsync              cudaMallocAsync
#define gpuHostMalloc               cudaHostAlloc
#define gpuHostMallocPortable       cudaHostAllocPortable
#define gpuMemcpy                   cudaMemcpy
```

hipMalloc

⇓

cudaMalloc

HOP: compile flags

`-I$HOP_ROOT`

include HOP headers

`-I$HOP_ROOT/source/cuda` OR `-I$HOP_ROOT/source/hip`

catch source code header includes

`-DHOP_TARGET_HIP` OR `-DHOP_TARGET_CUDA`

define target for translation

where `$HOP_ROOT` points to the installation path of HOP:

```
export HOP_ROOT=/path/to/hop
```

Example: compile and run

CUDA ⇒ HIP

```
export HOP_ROOT=/path/to/hop
export HOP_FLAGS=-I$HOP_ROOT -I$HOP_ROOT/source/cuda -DHOP_TARGET_HIP
CC -x hip $HOP_FLAGS hello.cu -o hello
./hello
```

HIP ⇒ CUDA

```
export HOP_ROOT=/path/to/hop
export HOP_FLAGS=-I$HOP_ROOT -I$HOP_ROOT/source/hip -DHOP_TARGET_CUDA
CC -x cu $HOP_FLAGS hello.cpp -o hello
./hello
```

HOP in code development

- HOP uses generic identifiers as intermediates in the translation
  - `gpuMalloc`, `gpuMemcpyHostToDevice`, ...
- One can use these generic identifiers directly in code
  - no CUDA/HIP identifiers, just generic identifiers that are then mapped to the correct target identifiers
- HOP headers are named and organised similar to HIP headers
  - if code uses only generic identifiers and includes the appropriate HOP headers, no need for `-I$HOP_ROOT/source/..`
- HOP headers may also be embedded in end-user code
  - MIT license

Header Only Porting as a general approach

- Use generic identifiers (`gpuMalloc` ...)
  - easy to swap between GPU backends (single header change)
  - allows one to also implement more complex wrapper functions if and when needed
- Strong preference for features that are supported by both CUDA and HIP
  - if needed, wrapper functions can be used to write backend-specific implementations
- Use standard compliant C/C++
  - avoid implicit header includes (`nvcc`, we are looking at you!)
  - kernel launch with `<<<...>>>()` works, but better to use `gpuLaunchKernel()` that can be mapped to whatever is needed by the target GPU backend

LUMI porting using HOP

- GPAW (https://wiki.fysik.dtu.dk/gpaw/)
  - DFT code based on the projector-augmented wave method
  - Python + C
- TurboGAP (https://github.com/mcaroba/turbogap) ~5-6h
  - MD using machine-learned force fields
  - Fortran, ISO C bindings

  incl. adding minimal support for FFT libs
- CloverLeaf (https://uk-mac.github.io/CloverLeaf/) ~1h
  - mini-app, Euler equations on a Cartesian grid
- MiniFMM (https://github.com/UoB-HPC/minifmm/) ~2-3h
  - mini-app, Fast Multipole Method

  incl. adding support for no header includes

Case: GPAW

```
__global__ void Zgpu(bmgs_cut_kernel)(
        const Tgpu* a, const int3 c_sizea, Tgpu* b, const int3 c_sizeb,
#ifdef GPU_USE_COMPLEX
        gpuDoubleComplex phase,
#endif
        int blocks, int xdiv)
{
    int xx = gridDim.x / xdiv;
    int yy = gridDim.y / blocks;
    ...
```

```
extern "C" void Zgpu(bmgs_cut_gpu)(...) {
    ...
    gpuLaunchKernel(Zgpu(bmgs_cut_kernel), dimGrid, dimBlock, 0, stream,
                    (Tgpu*) a, hc_sizea, (Tgpu*) b, hc_sizeb,
#ifdef GPU_USE_COMPLEX
                    phase,
#endif
                    blocks, xdiv);
    gpuCheckLastError();
```

- generic identifiers
- standard compliant C++

cut.cpp

Case: GPAW

```
#include <hip/hip_runtime.h>
#include <hipblas.h>

#define gpuMemcpyKind            hipMemcpyKind
#define gpuMemcpyDeviceToHost    hipMemcpyDeviceToHost

#define gpuSetDevice(id)         gpuSafeCall(hipSetDevice(id))
#define gpuGetDevice(dev)        gpuSafeCall(hipGetDevice(dev))
#define gpuGetDeviceProperties(prop, dev) \
        gpuSafeCall(hipGetDeviceProperties(prop, dev))
#define gpuDeviceSynchronize()   gpuSafeCall(hipDeviceSynchronize())

#define gpuFree(p)               if ((p) != NULL) gpuSafeCall(hipFree(p))
#define gpuFreeHost(p)           if ((p) != NULL) gpuSafeCall(hipHostFree(p))
```

```
#ifdef GPAW_CUDA
#include "cuda.h"
#endif
#ifdef GPAW_HIP
#include "hip.h"
#endif
```

- customised HOP headers embedded

- Switch of a single header enough to hop between CUDA and HIP

Case: TurboGAP

```
# TurboGAP makefile for LUMI

F90=ftn
CU=CC -x hip
PP=-e Z -D _MPIF90
F90_MOD_DIR_OPT=-J

# Header Only Porting
HOP_ROOT=$(HOME)/sandbox/hop
HOP_OPTS= -I$(HOP_ROOT) -I$(HOP_ROOT)/source/cuda -DHOP_TARGET_HIP

F90_OPTS=-fPIC -O3 -lhipblas
CUDA_OPTS=$(HOP_OPTS)

LIBS=-lsci_cray
```

- few extra compile flags
- code is untouched (Fortran + CUDA)

HOP: benefits and drawbacks

Pros:

- Easy porting between CUDA and HIP
  - no code modifications
  - works also from HIP to CUDA!
- No code duplication
  - one can use generic identifiers, HIP, or CUDA
- Flexible and simple
  - transparent one-to-one mappings
  - trivial to add hardware specific implementations if and when needed

Cons:

- Mapping limited to features supported by both HIP and CUDA
- Not aimed at other GPU backends

HOP: how to get started?

- Code available at: https://github.com/mlouhivu/hop
  - working proof of concept implementation
  - most common runtime identifiers included
  - rudimentary support for BLAS and FFT libraries
- Future outlook:
  - expand coverage of identifiers (WIP)
  - add support for other libraries
  - better documentation :)

Thanks!

martti.louhivuori@csc.fi
https://github.com/mlouhivu/hop