International Workshop On OpenCL
Vienna, April 20^{th} 2016

hiCL: An OpenCL Abstraction Layer for Scientific Computing, Application to Depth Imaging on GPU and APU

Issam Said, Pierre Fortin, Jean-Luc Lamotte, Henri Calandra

contact: isaid@uh.edu

Scientific computing

CPU platforms are the reference

Scientific applications

↓ implement

Programming languages: Fortran, C/C++, ...

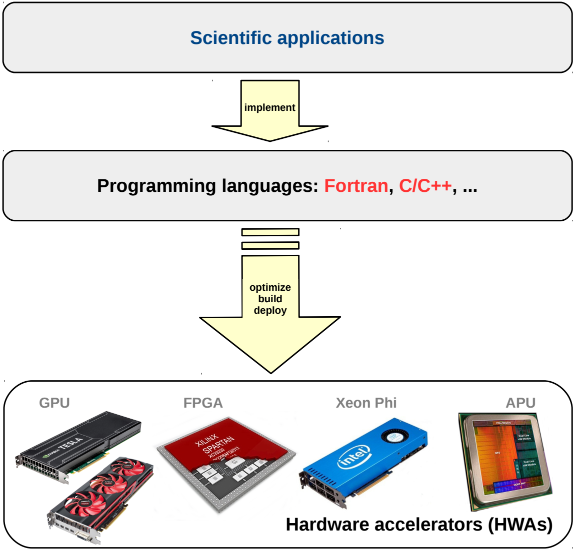
↓ optimize build deploy

OpenMP MPI

CPU platforms

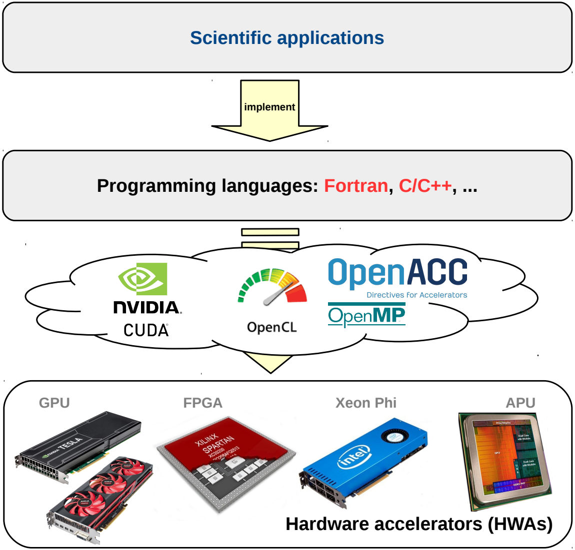
Scientific computing

Leveraging hardware accelerators (HWAs)

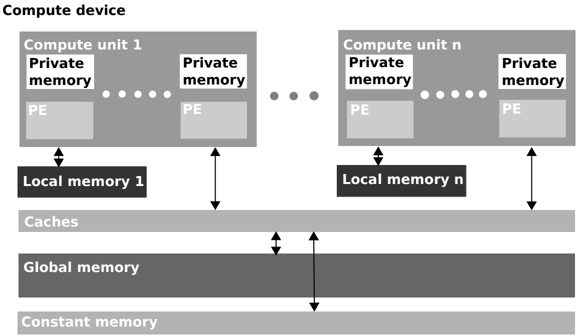


Scientific computing

Leveraging hardware accelerators (HWAs)



OpenCL: a standard for HPC



- Portable programming model (Khronos)
- Host code + kernels (compiled at runtime) executed on HWAs

OpenCL: a standard for HPC

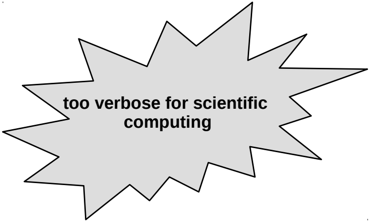
Typical programming steps

- Query the platform
- Select the devices
- Create a context
- Create command queues
- Create buffer objects
- Transfer data to device
- Create/build programs
- Extract kernels
- Launch kernels (on the device, the most important step)
- Transfer results to host
- Release buffers, kernels and the context

OpenCL: a standard for HPC

Typical programming steps

- Query the platform
- Select the devices
- Create a context
- Create command queues
- Create buffer objects
- Transfer data to device
- Create/build programs
- Extract kernels
- Launch kernels (on the device, the most important step)
- Transfer results to host
- Release buffers, kernels and the context



OpenCL: a standard for HPC

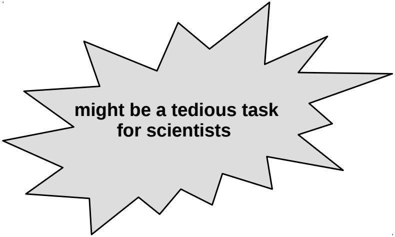
Managing memory objects

- HWAs are evolving very quickly
- Different memory subsystems are emerging:
  - Integrated HWA sharing memory with the CPU
  - Software manipulations are needed to take advantage of new designs
  - Example: the AMD Accelerated Processing Unit (APU)

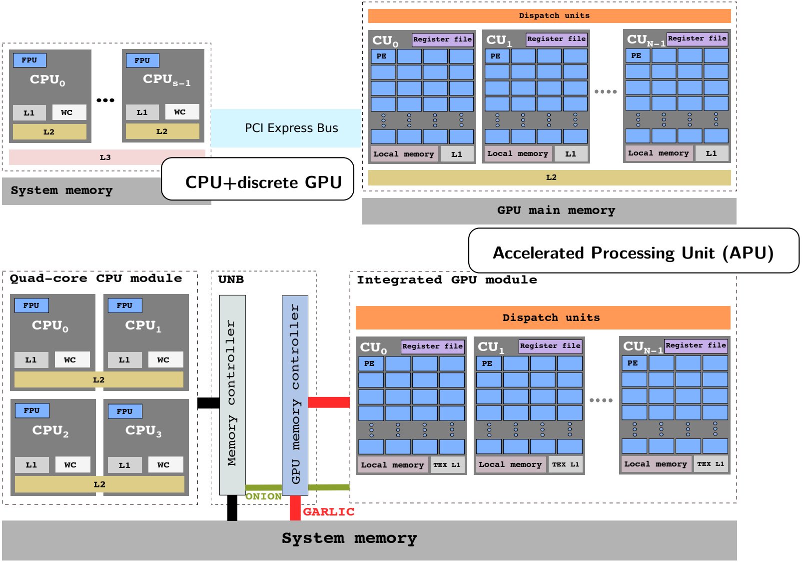
OpenCL: a standard for HPC

Managing memory objects

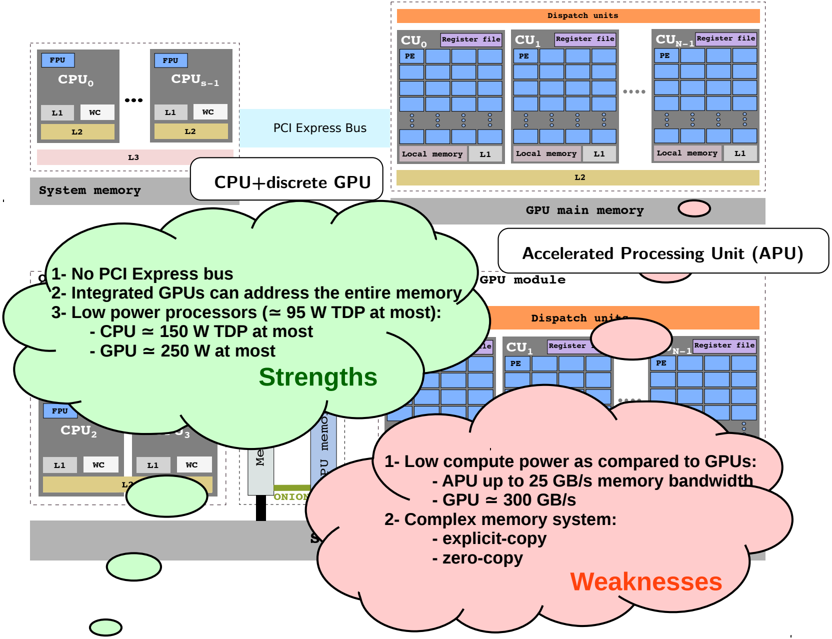
- HWAs are evolving very quickly
- Different memory subsystems are emerging:
  - Integrated HWA sharing memory with the CPU
  - Software manipulations are needed to take advantage of new designs
  - Example: the AMD Accelerated Processing Unit (APU)



What is an APU?



What is an APU?



1- No PCI Express bus
2- Integrated GPUs can address the entire memory
3- Low power processors (≃ 95 W TDP at most):
- CPU ≃ 150 W TDP at most
- GPU ≃ 250 W at most

Strengths

1- Low compute power as compared to GPUs:
- APU up to 25 GB/s memory bandwidth
- GPU ≃ 300 GB/s

2- Complex memory system:
- explicit-copy
- zero-copy

Weaknesses

Motivations and context

Quest for a tool that helps:

- Shortening the OpenCL host code
- Plugging HWAs code into legacy code (target: CPU, APU and GPU)
- Transparently manage memory objects on the different HWAs
- Programmers focus on optimizing kernels
- Spend less time on software engineering
- Spend more time on the domain of expertise

Outline

Outline

Related work

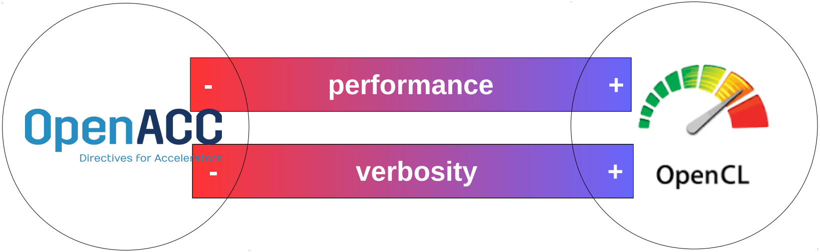
hiCL presentation

Reverse Time Migration on GPU and APU using hiCL

Conclusions and perspectives

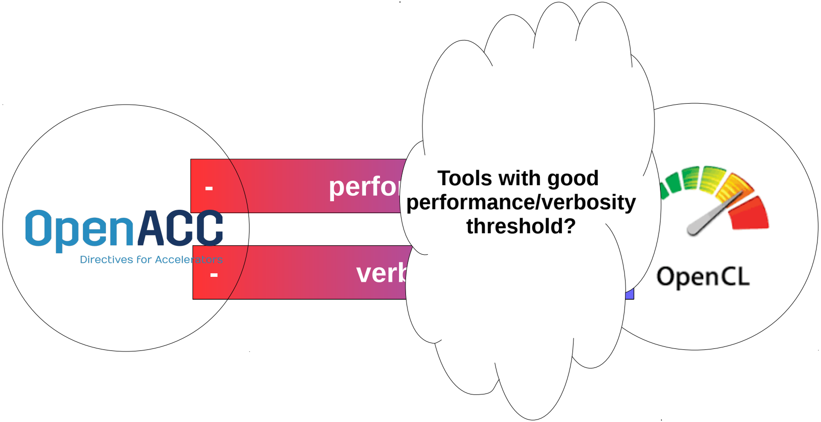
Related work

Quest for performance with less verbosity



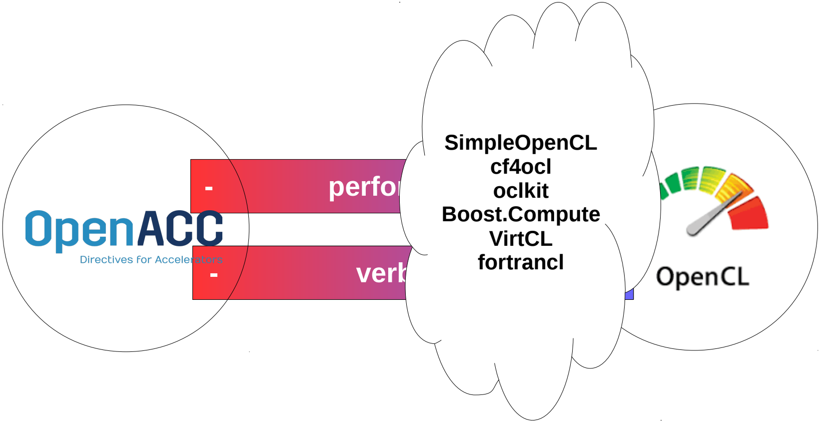
Related work

Quest for performance with less verbosity



Related work

Quest for performance with less verbosity



Outline

hiCL presentation

In a nutshell

hiCL

- Yet another OpenCL wrapper that eases scientific programming
- Abstracts the memory manipulation complexity on HWAs
- Features:
    - A simple C interface
    - C++ compatible (header guards)
    - A Fortran interface (ISO_C_BINDING Fortran 2003)

hiCL presentation

Example: matrix multiplication

```
// allocate the matrices
float *a=(float)malloc(N*N*sizeof(float));
float *b=(float)malloc(N*N*sizeof(float));
float *c=(float)malloc(N*N*sizeof(float));
// initialize matrices a, b and c
init(a, b, c);
...
...
...
...
...
...
...
// run the matrix multiplication c+=a*b
sgemm(a, b, c, N);
...
...
...
// delete matrices a, b, and c
free(a, b, c);
```

hiCL presentation

Example: matrix multiplication

```c
// allocate the matrices
float *a=(float)malloc(N*N*sizeof(float));
float *b=(float)malloc(N*N*sizeof(float));
float *c=(float)malloc(N*N*sizeof(float));
// initialize matrices a, b and c
init(a, b, c);
...
...
...
...
...
...
...
// run the matrix multiplication c+=a*b
sgemm(a, b, c, N);
...
...
...
// delete matrices a, b, and c
free(a, b, c);
```

```c
// allocate the matrices
float *a=(float)malloc(N*N*sizeof(float));
float *b=(float)malloc(N*N*sizeof(float));
float *c=(float)malloc(N*N*sizeof(float));
// initialize matrices a, b and c
init(a, b, c);
...
dev gpu1 = hicl_init(GPU | FIRST);
hicl_load("sgemm.cl", NULL);
hicl_mem_wrap(gpu1, a, N*N, READ_ONLY  | HWA);
hicl_mem_wrap(gpu1, b, N*N, READ_ONLY  | HWA);
hicl_mem_wrap(gpu1, c, N*N, READ_WRITE | HWA);
...
// run the matrix multiplication c+=a*b
hicl_run("sgemm", gpu1, a, b, c, N);
hicl_mem_update(c, READ_ONLY);
hicl_release();
...
// delete matrices a, b, and c
free(a, b, c);
```

hiCL presentation

Example: matrix multiplication

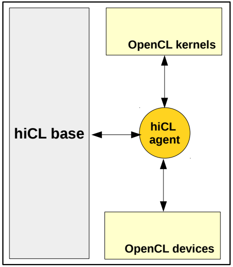
	standalone OpenCL	with hiCL
Lines of code [1]	525	280
Execution time[2]	0.479 s	0.491 s

[1] Includes hiCL code and the kernel code

[2] N=4096, gpu1=AMD HD7970

hiCL presentation

A simplified OpenCL compute model



- Are exposed to the user:
  - OpenCL kernels
  - Selected OpenCL devices
- hiCL base:
  - Encompasses the typical OpenCL work-flow
- hiCL agent:
  - Lists of the used memory objects
  - Devices/Kernels/Memory interactions

hiCL presentation

Reducing the OpenCL verbosity

- `hicl_init(flags)`
  - Only one call to initialize the OpenCL environment
  - Only one context is supported
  - One or multiple devices can be selected depending on `flags`
  - Each device has a pre-defined number of command queues
  - `flags` determine the user choices
    - Default platform with default device: `DEFAULT`
    - Choose the vendor: `NVIDIA`, `AMD`, ...
    - Choose the device type: `NVIDIA | GPU`
    - Even more: `NVIDIA | GPU | FIRST`
    - **Rule: what is not specified is default**
- `hicl_release()`
  - Releases the OpenCL context
  - Automatically releases the registered memory objects and kernels
- `hicl_info()`
  - Returns informations about the selected OpenCL resources

hiCL presentation

Loading kernels

- `hicl_load(file, options)`
  - Load ".cl" files, compile OpenCL programs, extract kernels
  - The hiCL agent register them for clean release afterwards
  - `options` are passed to the OpenCL compiler

hiCL presentation

Data consistency

- `hicl_mem_wrap(hwa_name, ptr, size, flags)`
  - `ptr` is a regular pointer allocated by the user
  - **an OpenCL buffer is created and registered behind the curtains**
    - **the buffer is associated to** `ptr`
  - `size` is the size of the buffer in number of elements
  - `flags` determine where and how the OpenCL objects are created

hiCL presentation

Data consistency

`flags` can combine:

hiCL memory flags	description
CPU	allocate the data on the system main memory if not already allocated
HWA	allocate the data on the HWA memory and copy it from the CPU memory
ZERO_COPY	the data is shared between the CPU and the HWA
READ_ONLY	the data is read-only
WRITE_ONLY	the data is write-only
READ_WRITE	the data is read-write
FLOAT, DOUBLE, INT ...	determine the data type

DEFAULT = HWA | READ_WRITE | FLOAT

hiCL presentation

Data consistency

In order to ensure data consistence between the host and the HWA:

- `hicl_mem_update(ptr, flag)`
- Prior to altering any hiCL memory (positions a dirty bit)
- Keep track of the changes issued by the host on the data
- `flag` can be:
  - `READ_ONLY`: the host reads only the data
  - `WRITE_ONLY`: the host modifies the data
  - `READ_WRITE`: the host reads and then modifies the data
- The dirty bit is positioned if the flag is `WRITE_ONLY` or `READ_WRITE`
- If the bit is already positioned the data is updated from the HWA

hiCL presentation

Running kernels

- `hicl_run("kernel name", hwa_name, arg1, arg2, arg3 ...)`
  - Run `"kernel name"` on the device `hwa_name`
  - C Variadic functions help passing arguments to the OpenCL kernels
    - Not yet possible in the Fortran hiCL interface
  - Related memory objects are:
    - Automatically updated from the host if they are dirty
    - Positioned dirty by the HWA if they are `WRITE_ONLY` or `READ_WRITE`

hiCL presentation

Example: 3D finite difference stencil

```
// allocate the buffers
float *u=(float)malloc(N*N*N*sizeof(float));
float *v=(float)malloc(N*N*N*sizeof(float));
// initialize the buffer u
init(u);
...
...
...
...
...
...
// run the stencil 10 times
for(int i; i<10; i++)
    fd_stencil(u, v, N, i);
...
...
// perform a snapshot (save to disk)
snapshot(v)
...
...
// delete matrices u, v
free(u, v);
```

hiCL presentation

Example: 3D finite difference stencil

```c
// allocate the buffers
float *u=(float)malloc(N*N*N*sizeof(float));
float *v=(float)malloc(N*N*N*sizeof(float));
// initialize the buffer u
init(u);
...
...
...
...
...
...
// run the stencil 10 times
for(int i; i<10; i++)
    fd_stencil(u, v, N, i);
...
...
// perform a snapshot (save to disk)
snapshot(v)
...
...
// delete matrices u, v
free(u, v);
```

```c
// allocate the buffers
float *u=(float)malloc(N*N*N*sizeof(float));
float *v=(float)malloc(N*N*N*sizeof(float));
// initialize the buffer u
init(u);
...
dev gpu1 = hicl_init(GPU | FIRST);
hicl_load("fd_stencil.cl", NULL);
hicl_mem_wrap(gpu1, u, N*N*N, READ_WRITE  | HWA);
hicl_mem_wrap(gpu1, v, N*N*N, READ_WRITE  | HWA);
...
// run the stencil 10 times
for(int i; i<10; i++)
    hicl_run("fd_stencil", gpu1, u, v, N, i);
// only here a HWA-CPU memory transfer takes place
hicl_mem_update(v, READ_ONLY);
// perform a snapshot (save to disk)
snapshot(v)
hicl_release();
...
// delete the buffers
free(u, v);
```

hiCL presentation

Example: 3D finite difference stencil

	standalone OpenCL	with hiCL
Lines of code[3]	638	328
Execution time[4]	1.571 s	1.582 s

[3] Includes hiCL code and the kernel code

[4] 320×320×320 with 100 iterations on an AMD HD7970 GPU

hiCL presentation

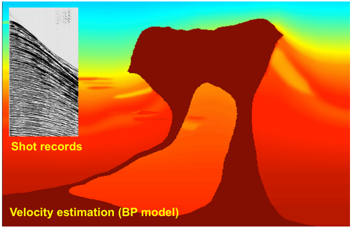
Overhead and performance

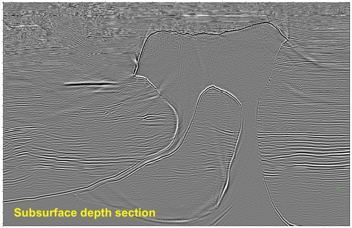
- Use red-black trees to index:
  - the hiCL memory objects by the memory addresses (pointers)
  - the hiCL kernels by names
  - the hiCL devices by `cl_device_id`
- Enhance the memory objects and kernel lookups

Outline

Reverse Time Migration (RTM)

- The reference imaging algorithm in the Oil and Gas industry
- Repositions seismic events into their true location in the subsurface





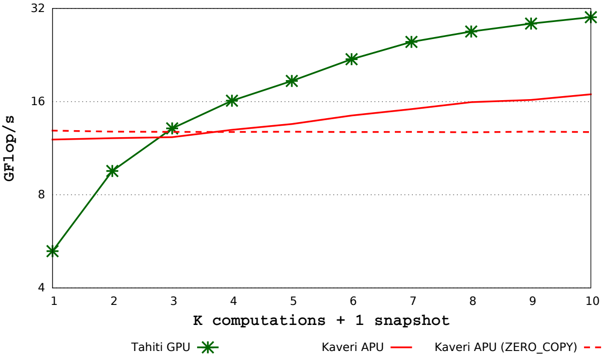
- Sub-salt and steep dips imaging
- Accurate (two-way wave equation)
- Requires massive compute resources (compute and storage)

Implementing RTM using hiCL

- Use the 3D finite difference stencil kernel to solve the wave equation
- Use the `HWA` flag to run on the GPU and on the APU (explicit-copy)
- Use the `HWA | ZERO_COPY` flags to run on the APU (zero-copy)
- Use the hiCL Fortran interface (initial code is in Fortran)

Implementing RTM using hiCL

Performance results



- Run the same host code while changing the memory flags
- The APU is more efficient than the GPU:
  - Only for high frequencies of data retrieval ($K < 3$)
  - The zero-copy feature enhances the performance for $K < 3$

Outline

Conclusions and perspectives

Conclusions

- hiCL is a scientific programming friendly OpenCL wrapper
- Helps integrate OpenCL kernels into existing industrial codes
- Comes with C/C++ and Fortran interfaces
- Its main focus is to simplify the memory management
- Targets cutting-edge accelerators
- Release date (in few weeks on github):
  - for release announcement please subscribe on https://groups.google.com/d/forum/hicl

Perspectives

- Compliance with OpenCL 2.0
- Performance enhancement and overhead reduction
- Support Intel integrated GPU
- Support OpenCL images