GPU Computing
Lecture 2: Programming GPUs

E. Carlinet & J.Chazalon
EPITA majeure Image 2018-2019

About next lectures / sessions

This afternoon:

- Lecture for 2 hours
- Coding for 2 hours

Friday, March 29th:

- Lecture #3 (2 hours)
- Project topics available

Friday, April 5th:

- Choose/validate project
- Work on project (3 hours)

Project

⚠ *The only grade for this course!*

- You prepare a lecture!
- 15’ presentation
  - 1. Tech.
  - 2. Algo.
  - 3. Focus on 1 issue
  - 4. Benchmark / Evaluation
- + 5’ demo
- + 10’ discussion
- Teams of 3

Logistics: questions?

Q1:
A1:

Q2:
A2:

Q3:
A3:

Today's agenda

I. Previously, in *IR/GPU Computing*
II. A tour of available technologies
III. Using OpenCL for GPU computing

I. Previously, in *IR/GPU Computing*

CPUs vs GPUs

	CPU	GPU
Optimized for:	?	?
Parallelism:	*Horiz.:* ?	*Horiz.:* ?
	Vertic.: ?	*Vertic.:* ?
Thread switching cost:	?	?
Smallest “worker” unit:	?	?

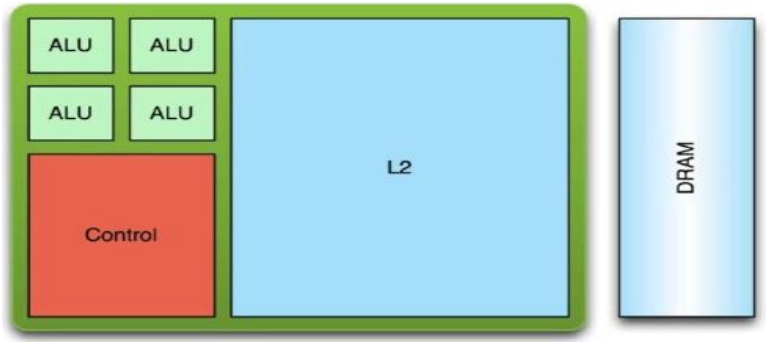
CPUs vs GPUs

In both cases, **memory throughput** generally limits computation speed

	CPU	GPU
Optimized for:	Latency	Throughput
Parallelism:	*Horiz.:* mild ILP, TLP & DLP	*Horiz.:* some ILP, massive TLP, large DLP
	Vertic.: some pipelining, predictions, etc.	*Vertic.:* large pipelining
Thread switching cost:	High	Nearly zero
Smallest "worker" unit:	Single thread	Group of thread ("warp", "wavefront", …)

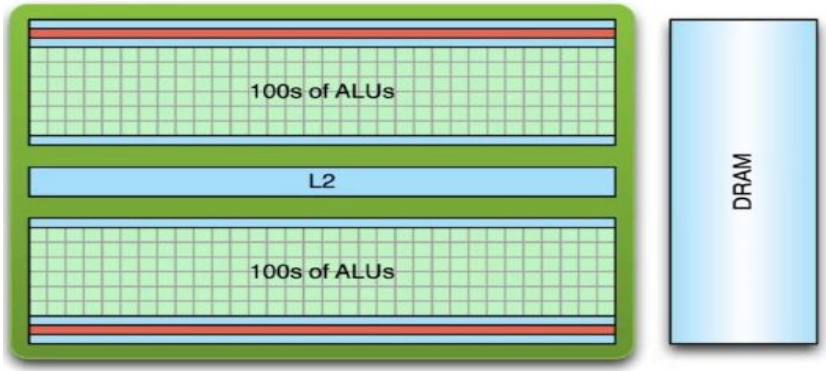
Hardware structure

CPU



- More cache / compute unit
- Lower memory latency
- Lower memory bandwidth

GPU



Device

> Group of processors (*“Stream. M.proc.”*)

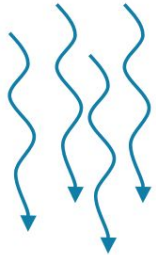
> “Processor” ~ SIMD Lane

> “Cores” ~ process. elements

Credit: NVidia

Parallel execution models

MIMD/SPMD



Multiple **independent** threads

SIMD/Vector

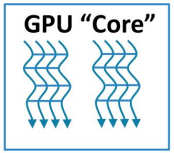
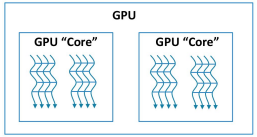
One thread with wide execution datapath

SIMT

Multiple **lockstep** threads

Credit: Derek Hower, AMD Research

Multicore Multithreaded SIMT

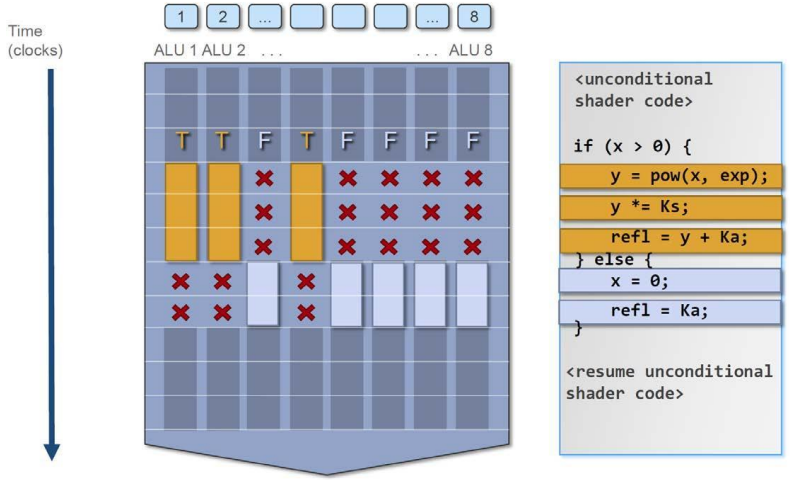
Software view **← Task / HW →** *Nvidia / OpenCL terminology*	**Nvidia/CUDA**	**AMD/OpenCL**	**CPU Analogy**
Thread, Work-item 	CUDA Processor	Processing Element	Lane
Warp, Wavefront (subgroup) 	CUDA Core	SIMD Unit	Pipeline
Block, Workgroup 	Streaming Multiprocessor	Compute Unit	Core
Grid, NDRange 	GPU Device	GPU Device	Device

Credit: Derek Hower, AMD Research

Pitfalls in GPU programming

Computation management

Divergent code paths



Work-items in wavefront run in lockstep

Memory management

Cache misses

Bank conflicts (local/shared/WG memory)

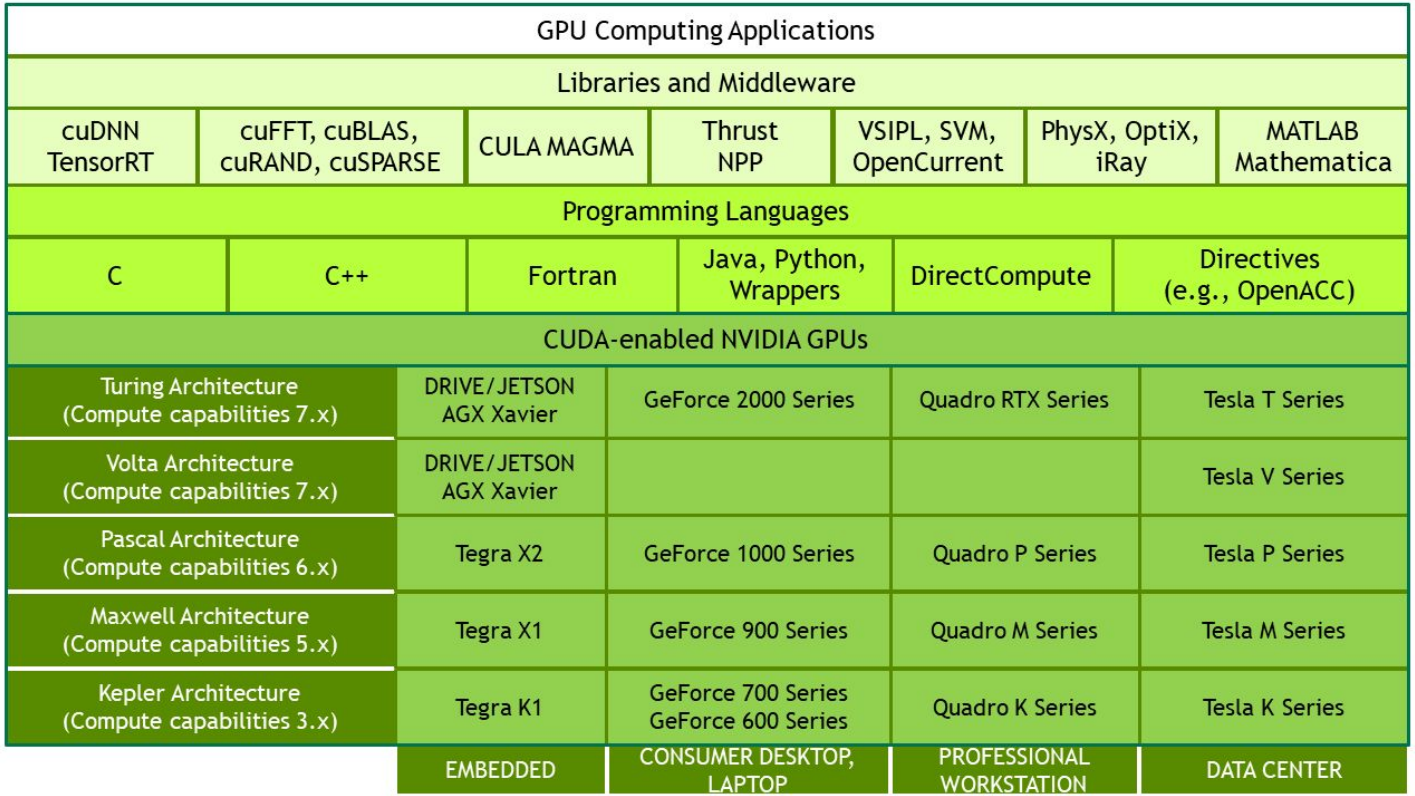
- 1 bank per running thread
- access different banks : ok
- access same word in 1 bank: ok
- access 2 words in 1 bank: conflict

One work-item stalls → entire wavefront must stall

Credit: K. Fatahalian & T. Mowry, Carnegie Mellon University

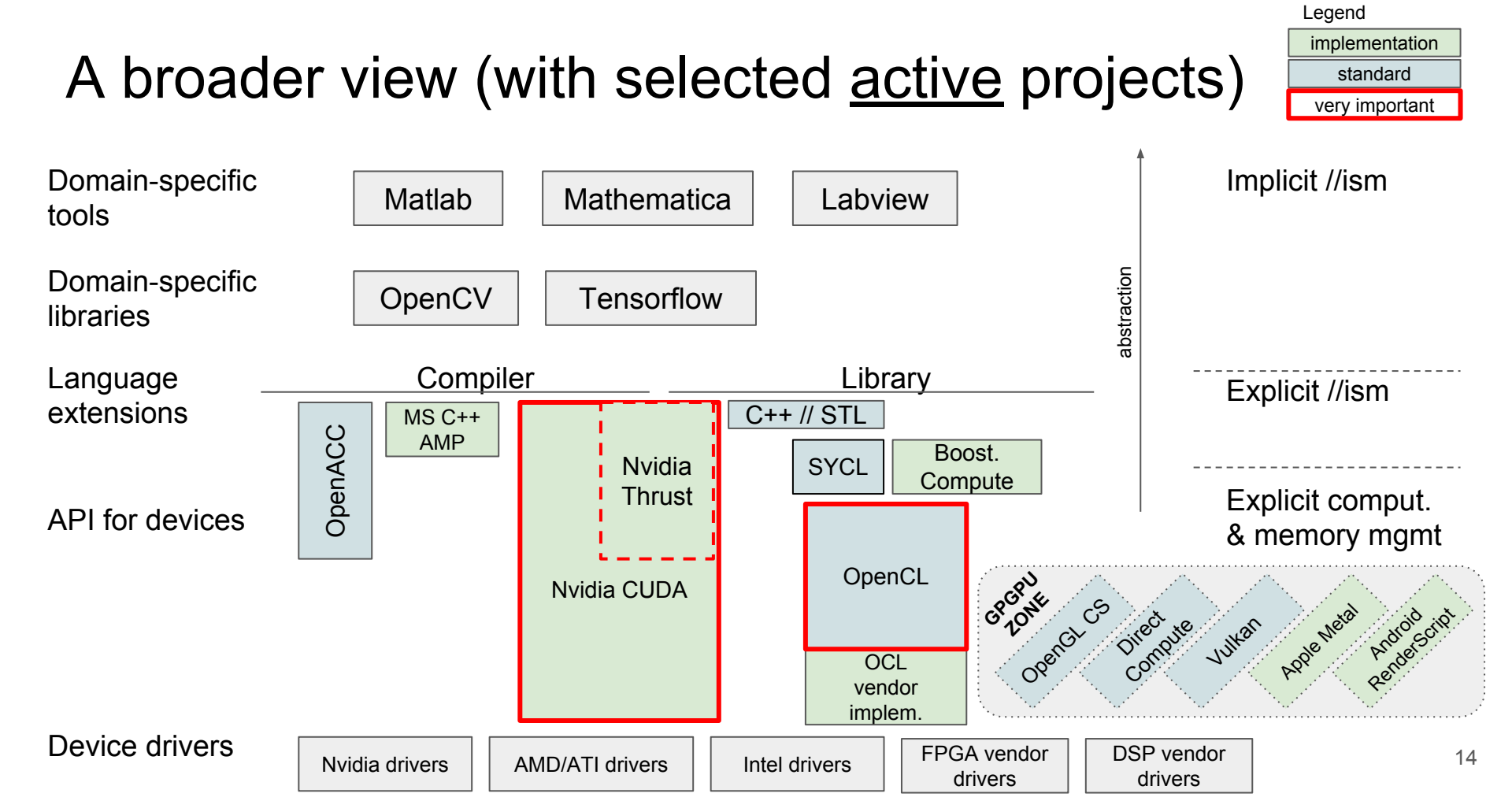
II. A tour of available technologies

A glimpse at the CUDA ecosystem



Credit: NVidia

A broader view (with selected active projects)



Single- vs multiple-source models

Do not mix with possible situations for **device code** compilation:
- JIT (from source or bytecode)
- static/cross build (frozen binarie(s))

Single-source

Source composed of only 1 language

Build with a single compiler
- Standard (C++ parallel STL)
- Custom (OpenACC)

Runtime compilation still is possible (for underlying device code)

Important questions:
1. How many languages do I need to know?
2. How and when is the device code generated?
3. What is the availability of the tools I need?

Multiple-source

Source separates clearly (different languages)
- Host code
- Device code

Build with:
- 1 custom compiler (CUDA)
- Two compilers
  - usually a dynamic compilation at runtime for device code — OpenCL
  - With an optional precompilation in bytecode at build time — SPIR-V for OpenCL, PTX for NVidia

High-level tools and libraries

Tools: Matlab, Mathematica, Labview…

⇒ *Just code your algorithms using provided constructs (hopefully parallelized)*

Libraries: OpenCV, Tensorflow…

⇒ *Like for high-level tools, hope for parallelized implementation*

⇒ *Libraries like Tensorflow specify a graph of operations, then runs it optimized (sort of TBB for GPUs specialized for tensor manipulation)*

Why wouldn't you use them?

- You absolutely need a parallelized version of your algorithm
- And your needs are not covered by domain-specific tools and libraries
- Or you want to support more hardware than they do

Microsoft C++ AMP

Sum 2 arrays element-wise

```
#include <amp.h>
#include <iostream>
using namespace concurrency;

const int size = 5;

void CppAmpMethod() {
    int aCPP[] = {1, 2, 3, 4, 5};
    int bCPP[] = {6, 7, 8, 9, 10};
    int sumCPP[size];

    // Create C++ AMP objects.
    array_view<const int, 1> a(size, aCPP);
    array_view<const int, 1> b(size, bCPP);
    array_view<int, 1> sum(size, sumCPP);
    sum.discard_data();

    parallel_for_each(
        // Define the compute domain, which is the set of threads that are created.
        sum.extent,
        // Define the code to run on each thread on the accelerator.
        [=](index<1> idx) restrict(amp) {
            sum[idx] = a[idx] + b[idx];
        }
    );

    // Print the results. The expected output is "7, 9, 11, 13, 15".
    for (int i = 0; i < size; i++) {
        std::cout << sum[i] << "\n";
    }
}
```

Microsoft C++ with extensions

- Library
- Language (“restrict”) keyword

Develop with Microsoft tools

Build with Visual Compiler

Implementation based on DirectX 11 (good GPU support)

Support for Windows platforms, but some ports of the “standard” to other ones

⇒ **Microsoft Windows only**

OpenACC

Windowed matrix mapping with non-local access and reduction

```
#pragma acc data copy(A) create(Anew)
while ( error > tol  &&  iter  <  iter_max )  {
  error = 0.0;
#pragma acc kernels {
#pragma acc loop independent collapse(2)
  for (  int  j = 1; j < n-1;  j++ )  {
    for (  int  i = 1; i < m-1; i++ )  {
      Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                              A [j-1] [i] + A [j+1] [i]);
      error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
     }
    }
  }
}
```

The OpenACC board include individuals from the following organisations:

- *Manufacturers of supercomputers, AMD and Nvidia*
- *A handful of national laboratories doing simulations (weather, physics, automotive…) and Total*
- *Academics doing HPC research*

Open standard (C++ and Fortran) by the OpenACC consortium

Directive-based ⇒ requires a custom compiler
Heavy influence from OpenMP

Several compilers available, both commercial and open-source, for several platforms

Notable Nvidia support with CUDA backend. Also some OpenCL backends.

Need dedicated tools for development and debugging.

⇒ HPC/simulation oriented
⇒ Your should know the target hardware

Nvidia Thrust

Generate random numbers on the host and transfer them to the device where they are sorted

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per second on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Nvidia C++ template library, featured with every CUDA installation.

Should build with any compiler, but in practice Thrust header rely on nvcc tolerance / Nvidia's own interpretation of C++ standard (plus you can include CUDA kernel code).

Links / depends on CUDA, meaning you build for a range of Nvidia devices.

Develop with many tools, but debug/profile with Nvidia tools (CUDA toolkit).

⇒ Speedup for simple map, reduce, sort op.
⇒ Nvidia lockdown (hardware, support…)

C++ Parallel STL

Since C++ 17:

```
std::transform(std::execution::par,
               vec.begin(), vec.end(), out.begin(),
               [](double v) { return v * 2.0; }
);
```

Execution policies: `seq`, `par`, `par_vec`

Parallel version of various STL algorithms: `all_of`, `find`, `move`, `sort`...

New algorithms: `for_each`, `reduce`, `transform_reduce`...

Recent extension of the standard.

Several implementations now exist: Microsoft, Intel using TBB, and even some prototype ones using SYCL/OpenCL to target CPUs, GPUs, DSP…

⇒ Portable alternative to Nvidia Thrust
⇒ Need complete implementations

SYCL

Write work-item global id to output buffer

```
#include <CL/sycl.hpp>
#include <iostream>
int main() {
  using namespace cl::sycl;
  int data[1024]; // initialize data to be worked on
  // By including all the SYCL work in a {} block, we ensure
  // all SYCL tasks must complete before exiting the block
  {
    // create a queue to enqueue work to
    queue myQueue;
    // wrap our data variable in a buffer
    buffer<int, 1> resultBuf(data, range<1>(1024));
    // create a command_group to issue commands to the queue
    myQueue.submit([&](handler& cgh) {
      // request access to the buffer
      auto writeResult = resultBuf.get_access<access::write>(cgh);
      // enqueue a prallel_for task
      cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
        writeResult[idx] = static_cast<int>(idx[0]);
        }); // end of the kernel function
    });       // end of our commands for this queue
  }           // end of scope, so we wait for the queued work to complete
}
```

⇒ Very promising

⇒ As long as

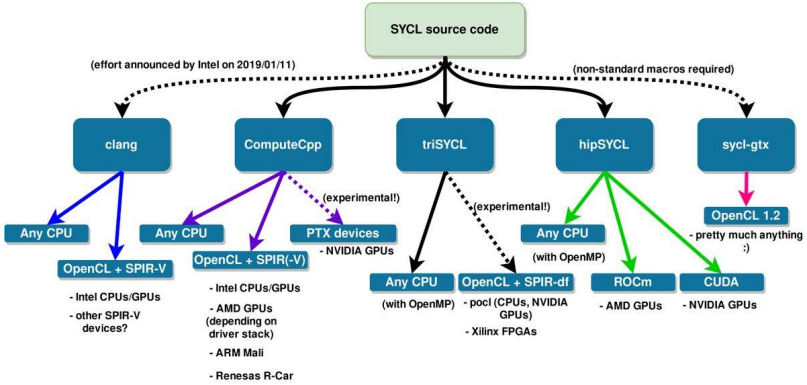
1. Implementations are maintained
2. OpenCL goes on

Royalty-free, cross-platform C++ abstraction layer that builds on top of OpenCL

Single-source development, use traditional host compilers to produce standard C++ code.

Possible to use advanced OpenCL concepts

Several implementations



Boost.Compute

```cpp
#include <vector>
#include <algorithm>
#include <boost/compute.hpp>

namespace compute = boost::compute;

int main()
{
    // get the default compute device
    compute::device gpu = compute::system::default_device();

    // create a compute context and command queue
    compute::context ctx(gpu);
    compute::command_queue queue(ctx, gpu);

    // generate random numbers on the host
    std::vector<float> host_vector(1000000);
    std::generate(host_vector.begin(), host_vector.end(), rand);

    // create vector on the device
    compute::vector<float> device_vector(1000000, ctx);

    // copy data to the device
    compute::copy(
        host_vector.begin(), host_vector.end(), device_vector.begin(), queue
    );

    // sort data on the device
    compute::sort(
        device_vector.begin(), device_vector.end(), queue
    );

    // copy data back to the host
    compute::copy(
        device_vector.begin(), device_vector.end(), host_vector.begin(), queue
    );

    return 0;
}
```

Sort random floats on GPU

GPU/parallel-computing library for C++ based on OpenCL

Header only library

Compile with

```
g++ -I/path/to/compute/include sort.cpp -lOpenCL
```

Run with any OpenCL driver

⇒ **Boost convenience**
⇒ **More GitHub ★ than any SYCL implem.**

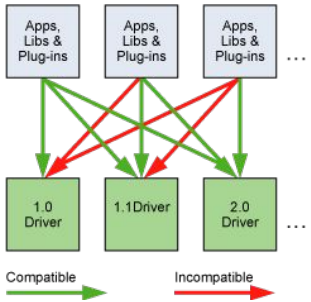
CUDA

```
// simple doubling kernel
__global__ void myKernel(int *result)
{
  int i = threadIdx.x;
  result[i] = 2*i;
}

//use only one block
dim3 blocksPerGrid(1,1,1);
//use N threads in the block
dim3 threadsPerBlock(N,1,1);
// launch kernel
myKernel<<<blocksPerGrid, threadsPerBlock>>>(result);
```

Missing boilerplate code for device and buffer management

Code forward compatible →
not backward compatible



Actually many ways to use CUDA, not only C++

Huge adoption

Fine control over hardware optimizations

Low-level SIMT programming

Non-standard “single-source” C++ (actually kernel code and host code), compile using `nvcc`

Device bytecode (PTX) compiled at runtime by CUDArt, but need to generate generation-specific code at build time

⇒ Leading platform, safe industrial choice
⇒ Nvidia lockdown

OpenCL

We'll see C++ code in a few slides

Increment elements in buffer, complete C code

Available on YOUR computer.

```c
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

#include <CL/cl.h>

int main() {
    cl_command_queue command_queue;
    cl_context context;
    cl_device_id device;
    cl_int input = 1;
    cl_int kernel_result = 0;
    cl_kernel kernel;
    cl_mem buffer;
    cl_platform_id platform;
    cl_program program;
    const char *source = "__kernel void increment(int in, __global int* out) { out[0] = in + 1; }";

    clGetPlatformIDs(1, &platform, NULL);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, NULL);
    context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
    command_queue = clCreateCommandQueue(context, device, 0, NULL);
    buffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR, sizeof(cl_int), NULL, NULL);
    program = clCreateProgramWithSource(context, 1, &source, NULL, NULL);
    clBuildProgram(program, 1, &device, "", NULL, NULL);
    kernel = clCreateKernel(program, "increment", NULL);
    clSetKernelArg(kernel, 0, sizeof(cl_int), &input);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &buffer);
    clEnqueueTask(command_queue, kernel, 0, NULL, NULL);
    clFlush(command_queue);
    clFinish(command_queue);
    clEnqueueReadBuffer(command_queue, buffer, CL_TRUE, 0, sizeof (cl_int), &kernel_result, 0, NULL, NULL);

    assert(kernel_result == 2);
    return EXIT_SUCCESS;
}
```

Standard from OpenGL makers
Inspired by CUDA

C/C++ API, Standard C++

```
g++ prog.cpp -lOpenCL -o prog
```

Many implementations on CPU, GPU, DSP, FPGA…

Can use several "platforms" simultaneously, and several "devices" from those platforms

Compilation for specific HW or JIT compilation of device code by platform driver. Obfuscated IR possible (SPIR-V bytecode)

OpenCL future

1999: NVidia launches the first programmable GPU *(GeForce 256)*

2006: NVidia launches CUDA

2008: DirectX has DirectCompute

2009: First OpenCL specification

2012: OpenGL 4.3 has OpenGL CS *(Compute Shaders)*

2014: Apple launches Metal, merging features from OpenGL and OpenCL

2015: Khronos group announces Vulkan

2018: Khronos group announces that OpenCL would be merging into Vulkan

III. Using OpenCL for GPU computing

Installation / Build modes

To find out all possible (known) properties of the OpenCL platform and devices available on the system, install and run `clinfo`.

2 build/run modes:

1. **Direct**: Build directly for some hardware (direct link to vendor lib.)
   Prefered for embedded
2. **Indirect:** Use ICD loader to select implementation(s), platform(s) and device(s) at runtime
   Prefered in all other cases
   Implies JIT compile device code

Installable Client Drivers (ICDs) `cl_khr_icd` extension

ICD Loader (Vendor agnostic) > ICD (Vendor dependent) > Driver(s) & Device(s)

OpenCL ICD loader (libOpenCL.so)

- platform-agnostic library, proxy to device-specific drivers through the OpenCL API
- Several open source implementations

OpenCL Runtimes: hardware-specific runtimes, need to be installed, expose OCL API, talk to device driver, listed under `/etc/OpenCL/vendors`

- AMD/ATI: opencl-mesa (GPU), opencl-amd (GPU closed src), amdgpu-pro-opencl, opencl-catalyst, amdapp-sdk (CPU)
- NVIDIA: opencl-nvidia
- Intel: compute-runtime (NEO runtime, Intel HD Graphics GPU on Gen8/Broadwell+), beignet (for older HW), intel-opencl (for older HW), intel-opencl-runtim (Intel Core and Xeon processors. Also supports non-Intel CPUs)
- Other: pocl (LLVM-based OpenCL implementation)

Headers / Compilation

Either download headers for the appropriate version on Khronos' website

Or use the ones provided with your distribution

Compilation using the indirect linking is straightforward:

```
g++ prog.cpp -lOpenCL -o prog
```

Indirect linking / ICD proxy model

OpenCL implementations that implement OpenCL ICD interfaces will return `cl_khr_icd` in their `CL_PLATFORM_EXTENSIONS` string.

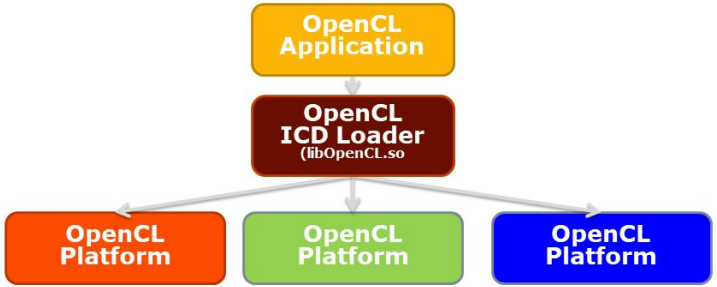
The only function that an OpenCL implementation must export to work with the OpenCL ICD loader is `clGetExtensionFunctionAddress`.

```
$ nm -D --defined-only libAnOpenCLImplementation.so
000000000002b500 T clGetExtensionFunctionAddress
```

The ICD loader is almost always named `libOpenCL.so`.

The ICD loader exports all of the OpenCL API functions.

```
$ nm -D --defined-only libOpenCL.so.1.2
00000000000191c0 T clBuildProgram
000000000003ec10 T clCloneKernel
00000000000199a0 T clCompileProgram
0000000000011c50 T clCreateBuffer
...
```



Credit: Ben Ashbaugh

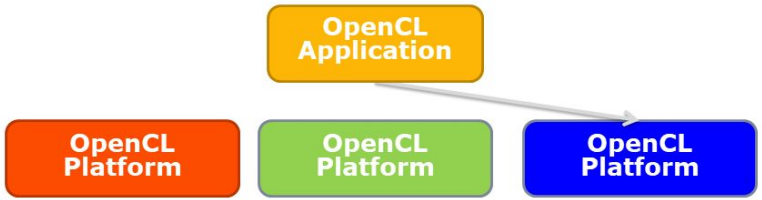
Direct Linking

The *Direct Linking* method is less common, but is occasionally used in mobile or embedded applications that target a specific hardware configuration.

When using the *Direct Linking* method, applications link against a specific OpenCL implementation, rather than a generic OpenCL loader.

To support *Direct Linking*, the OpenCL implementation must export all of the APIs required by the application:

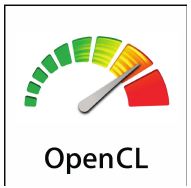
```
$ nm -D --defined-only libAnOpenCLImplementation.so
000000000004bd50 T clBuildProgram
000000000007cd20 T clCloneKernel
0000000000044720 T clCompileProgram
00000000000699b0 T clCreateBuffer
....
```



Credit: Ben Ashbaugh

Hands On OpenCL

Created by
Simon McIntosh-Smith
and Tom Deakin



Includes contributions from:
Timothy G. Mattson (Intel) and Benedict Gaster (Qualcomm)

V 1.2 – Nov 2014