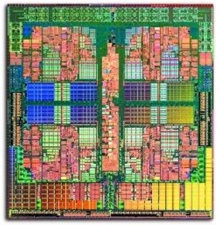


Introduction to CUDA and OpenCL

OpenCL elements

Outline

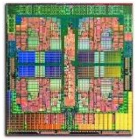
- □ A little bit about…
- □ What is the same and what is different
- □ General view on the OpenCL framework
- □ Examples, examples…
- □ Something completely different OpenACC toolkit

Tomasz Szumlak AGH-UST

Wydział Fizyki i Informatyki Stosowanej

25/11/2021

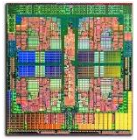
Because all is heterogeneous



- Before we start… PROJECTS!!
- We should decide the topics of projects this Thursday and next Monday

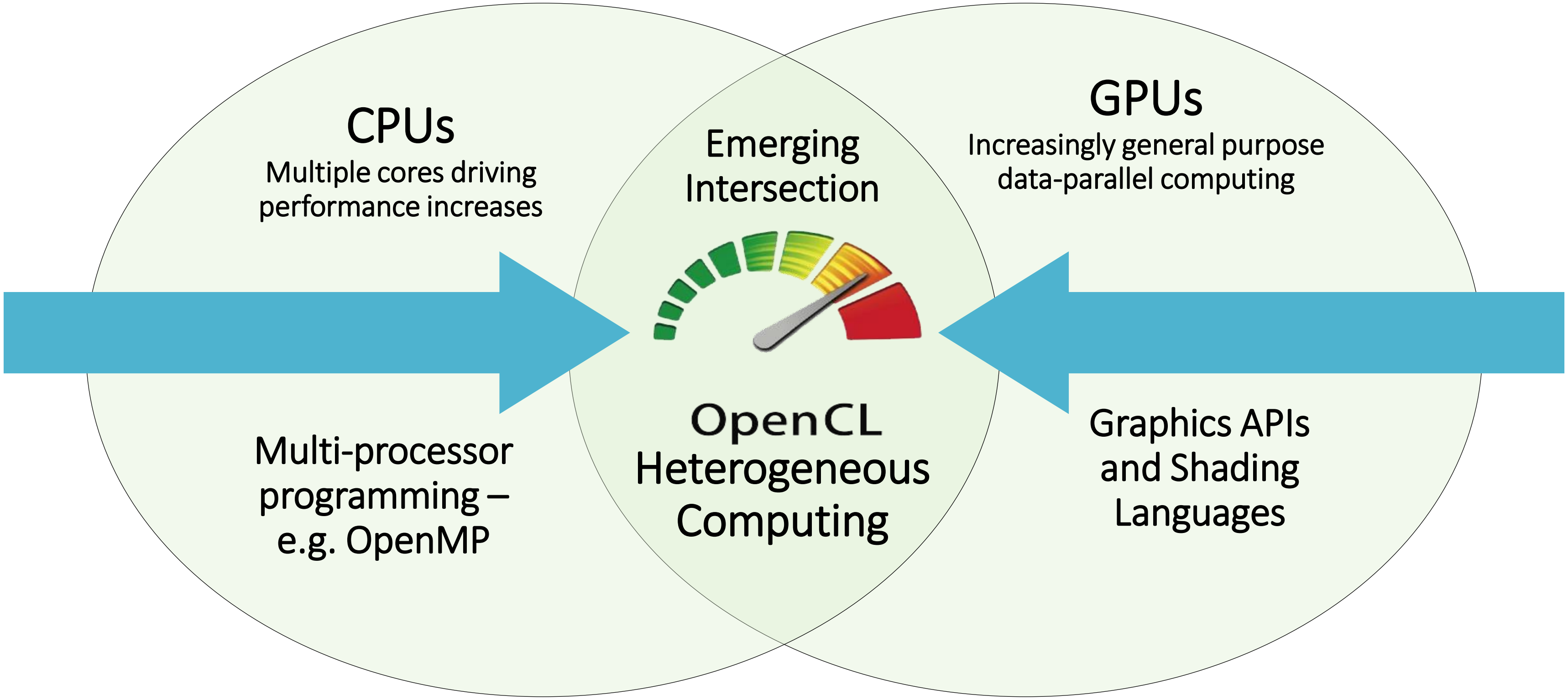
- I usually go like that: **easy** way and **hard**(er) way
- The former is just to work on a selected topic using NVIDIA developer blog and internet (e.g. delve into reduction algorithms, shared memory properties, etc.)
- Hard way is to provide a solution to a problem that is more challenging (e.g. data analysis, end-to-end project)

Because all is heterogeneous



- □ In principle all devices from mobile phones to large computing centres features **h. architecture**
- □ Even a cheap laptop now can combine up to three different processing units (P.U.): APU, CPU and GPU
- □ OpenCL (**Open Computing Language**) offers a nice way to use them all – a **portable code** that is able, in a transparent way, to **use all P.U.**

Industry Standards for Programming Heterogeneous Platforms



CPUs
Multiple cores driving performance increases

Multi-processor programming – e.g. OpenMP

Emerging Intersection

OpenCL
Heterogeneous Computing

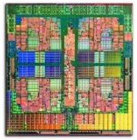
GPUs
Increasingly general purpose data-parallel computing

Graphics APIs and Shading Languages

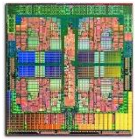
OpenCL – simply had to be invented!

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

Where to look for a kick-start



- A lot of excellent courses available on-line
- Definitively my winner is: „**Hands On OpenCL**”
    - It is a self consistent, end-to-end course
    - Hands-on examples provided via github repository
    - Very nice slides accompany the course (I borrowed a few!)
    - Extensive setting-up for various platforms provided
    - „Must see” for everybody interested in OpenCL
    - https://handsonopencl.github.io/
- NVIDIA recently integrated support for OpenCL into their software drivers package
- https://developer.nvidia.com/opencl



Hands On OpenCL

Created by
Simon McIntosh-Smith
and Tom Deakin

University of BRISTOL

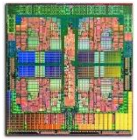
OpenCL

KITE
Khronos Initiative for Training & Education

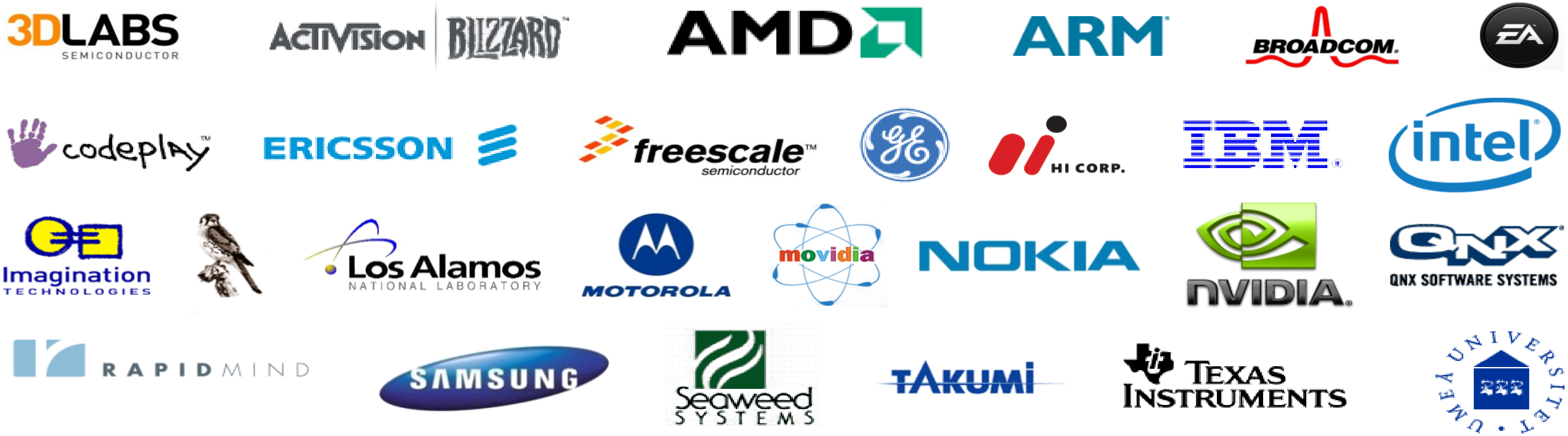
Includes contributions from:
Timothy G. Mattson (Intel) and Benedict Gaster (Qualcomm)

V 1.2 - Nov 2014

OpenCL Working Group within Khronos



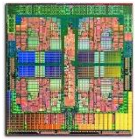
- ❑ Diverse industry participation
  - ❑ Processor vendors, system OEMs, middleware vendors, application developers.
- ❑ OpenCL became an important standard upon release by virtue of the market coverage of the companies behind it.



KHRONOS GROUP

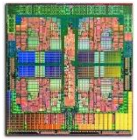
Third party names are the property of their owners.

Good news!



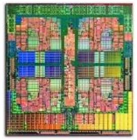
- □ If you paid attention to my lectures and did all the exercises you will hit the ground running!
- □ OpenCL **is just like CUDA** but a bit different
- □ When discussing the basic features of the OpenCL framework you will notice loads of similarities!
- □ There is not „learning from scratch” – just adjusting what you already know about CUDA
- □ Fine, let’s go!

Laying the foundation



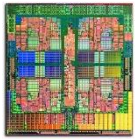
- The fundamental goal is to **use all computation units** (resources) available on a given system
- Exploits both **data** parallel (SIMD) and **task** parallel models
- You create a OpenCL code by using extension to C language (having deja vu yet…?)
- Providing abstraction of the underlying parallelism
- Different implementations (i.e., different libraries from AMD/ATI, NVIDIA, …) define platforms which in turn can enable the host system to interface with OpenCL-capable device (again – very similar to CUDA enabled devices)
- OpenCL has its own particular „structure”

Disecting OpenCL

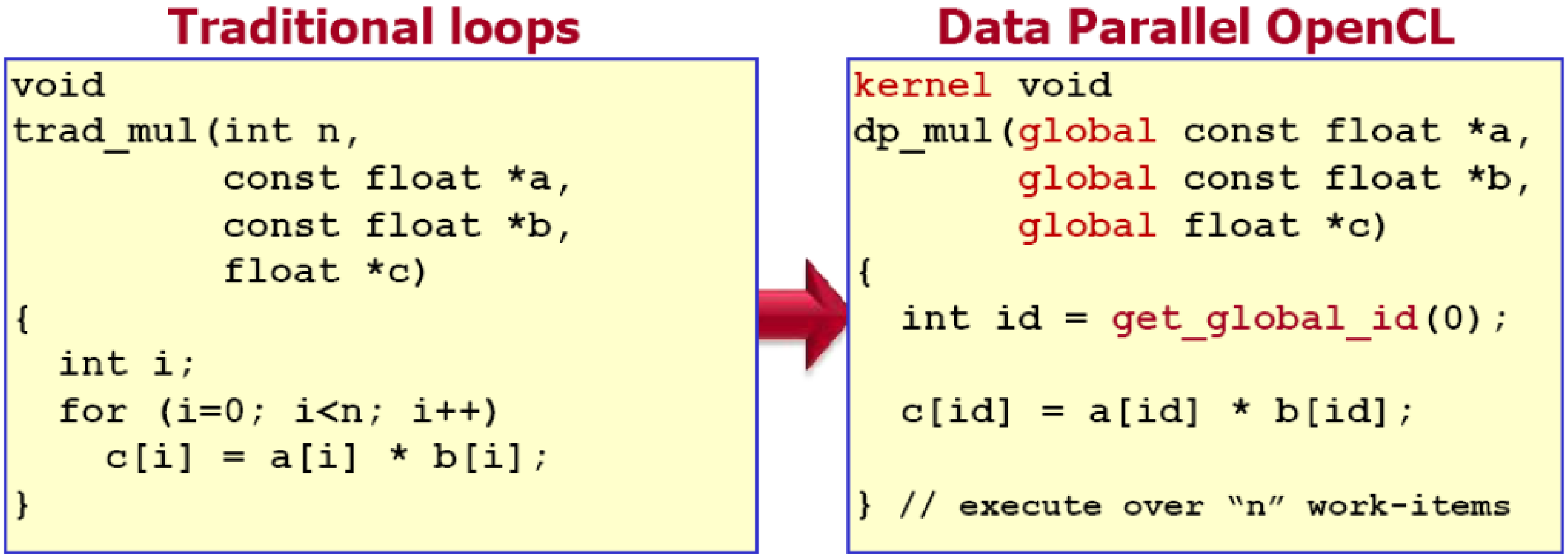


- After working with CUDA the OpenCL ecosystem structure may seem a bit complicated – but remember it is suppose to be much more generic!
- **Platform Layer API**
  - Hardware abstraction layer
  - **Query** facility, **select** and **initialize** compute devices (CD)
  - Create **compute contexts** and **task queues**
- Run-time API
  - **Execute** compute kernels
  - Scheduler to **manage the resources**: processing units and memory
- Language
  - C-based extension
  - A lot of goodies as built-in functions

Oh! It is so similar!



- ❑ When working with OpenCL we use the following hierarchy: one host + one (many) compute device(s) (here the CPU is also a C.D.!), one or more compute units and finally one or more processing elements…



Traditional loops

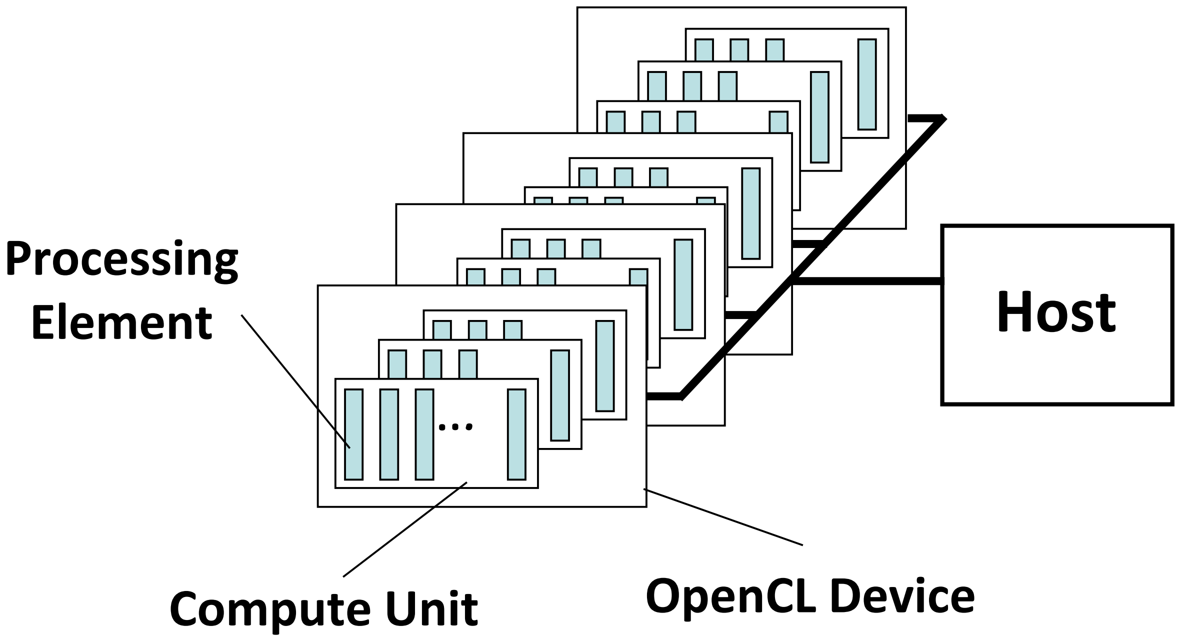
```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

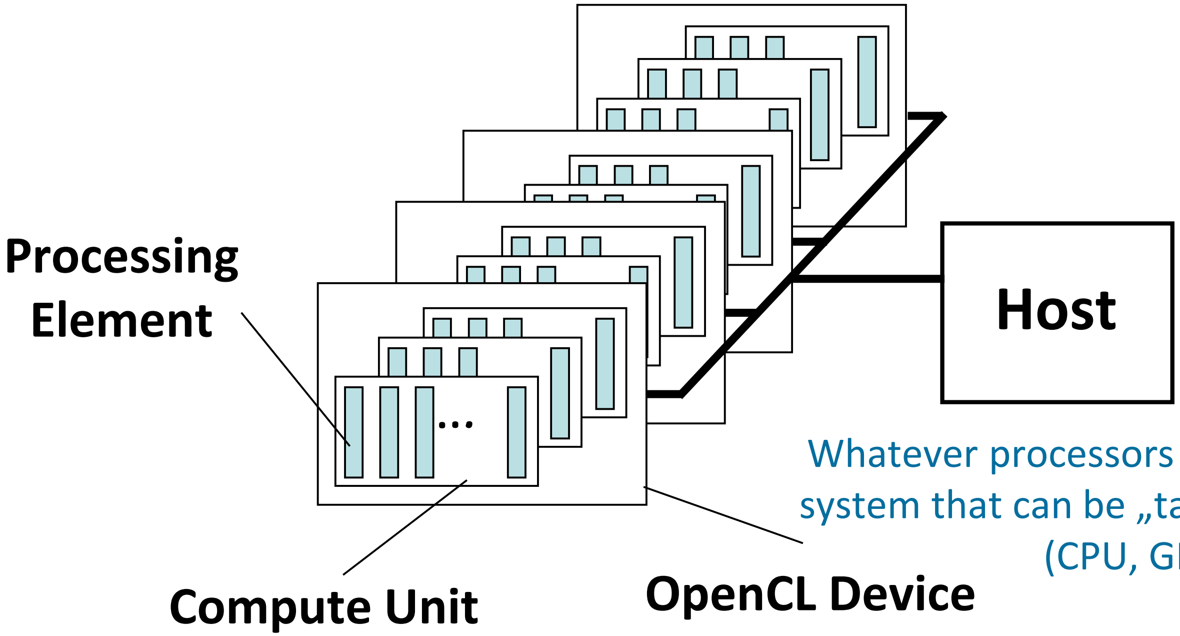
© Copyright Khronos Group

OpenCL Platform Model



- One **Host** and one or more **OpenCL Devices**
  - Each OpenCL Device is composed of one or more **Compute Units**
    - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into **host memory** and device memory

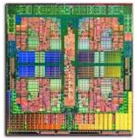
OpenCL Platform Model



Whatever processors are available in the system that can be „talked to” by OpenCL (CPU, GPU,…)

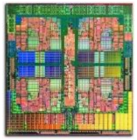
- ❑One **Host** and one or more **OpenCL Devices**
  - ❑ Each OpenCL Device is composed of one or more **Compute Units**
    - ❑ Each Compute Unit is divided into one or more ***Processing Elements***
- ❑Memory divided into **host memory** and device memory

Parlez-vous OpenCL?



- **Kernel** – the atom of execution, usually just a function (in C-language sense)
- **Host application** – one or more kernels managed via not OpenCL specific code
- **Work group**: a collection of work items, must have a unique work group ID, work item can be synchronised
- **Work item**: an instance of a kernel at run time, it must have a unique ID within the work group
- Sounds familiar…?

How does it compare to CUDA?



□ Let’s create an explicit „translation matrix”

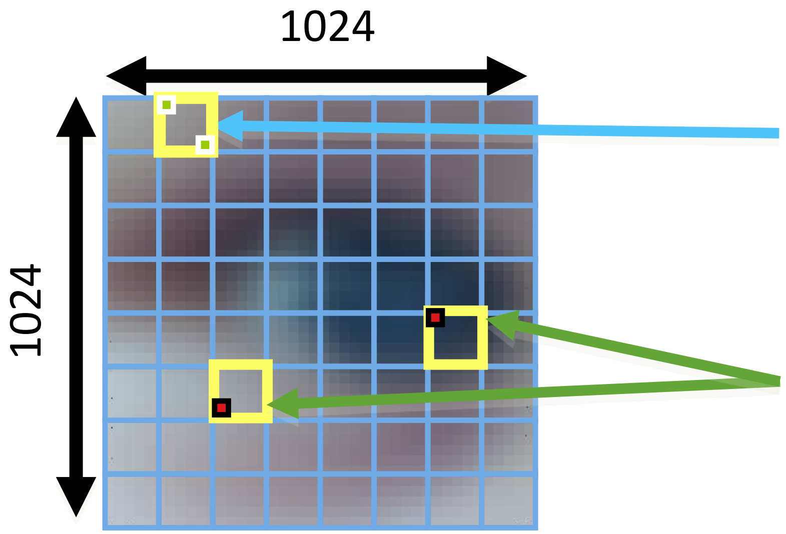
□ **OpenCL** „style”
- □ Kernel
- □ Host application
- □ NDRange
- □ Work item
- □ Work group

□ **CUDA** „style”
- □ Kernel
- □ Host application
- □ Grid
- □ Thread
- □ Block

□ Aha! Now it is really easy to understand…

An N-dimensional domain of work-items

- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 64x64 (**work-group**, executes together)

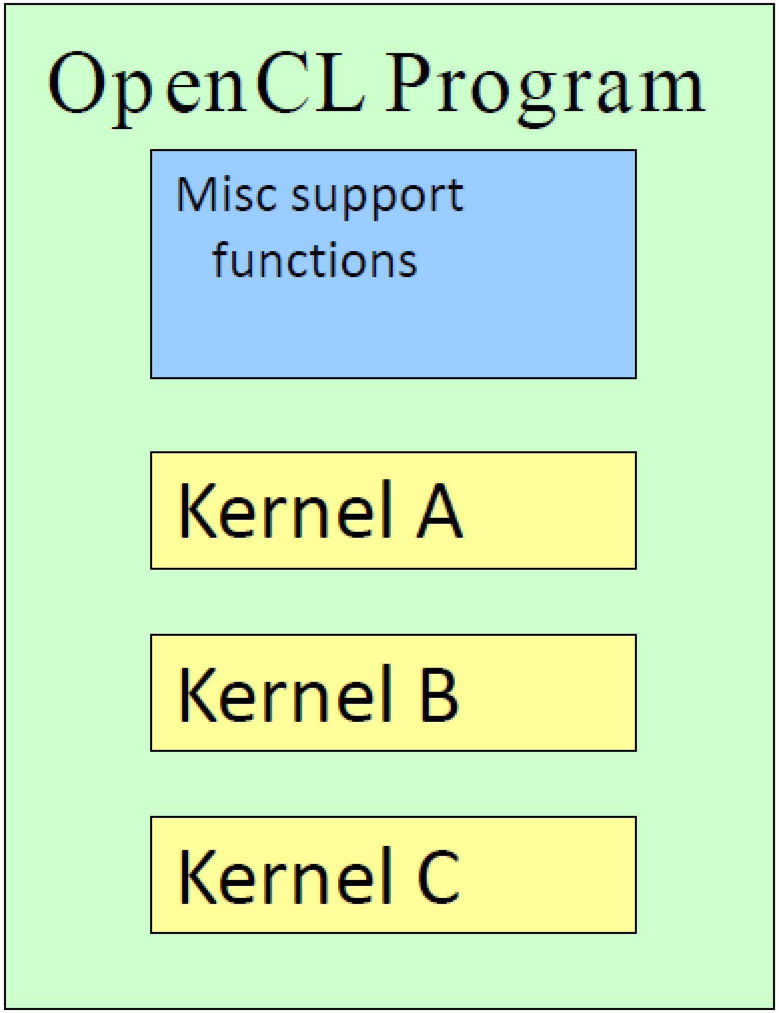


Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel

- Choose the dimensions that are “best” for your algorithm (tuning a bit more difficult)

A generic structure of an OpenCL program



OpenCL Program

Misc support functions

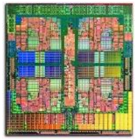
Kernel A

Kernel B

Kernel C

- ❑ Sorry for repeating myself… but a typical OpenCL program is a bit similar to its CUDA counterpart
- ❑ It has a managing (service) part and one or more kernels
- ❑ As in CUDA the kernel is just a basic atom of parallel code to be executed on the target device

The flow – vector addition example



```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
                       CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);

cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                             sizeof(cl_float)*n, NULL, NULL);

// create the program
program = clCreateProgramWithSource(context, 1,
                                &program_source, NULL, NULL);
```

```
// build the program
err = clBuildProgram(program, 0, NULL,NULL,NULL,NULL);

// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                         sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1],
                         sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                         sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
                    global_work_size, NULL,0,NULL,NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
                          CL_TRUE, 0,
                          n*sizeof(cl_float), dst,
                          0, NULL, NULL);
```

The flow – vector addition example

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,
                         CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetCo
```

Define platform and queues

```
cl_device_id[] devices = malloc(cb);
clGetContextInfo(context,CL_CONTEXT_DEVICES,cb,devices,NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context,devices[0],0,NULL);

// allocate the buffer memory objects
memobjs[0]
       CL_
```

Define memory objects

```
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcb, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                               sizeof(cl_float)*n, NULL, NULL);

// create the
program = clCreateProgramWithSource(context, 1,
                                 &program_source, NULL, NULL);
```

Create the program

```
// build the progr
err = clBuildProgr
```

Build the program

```
// create the kernel
kernel = clCreateKernel(program, “vec_add”, NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],

err |= clSe
                         sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
                         sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;
```

Create and setup kernel

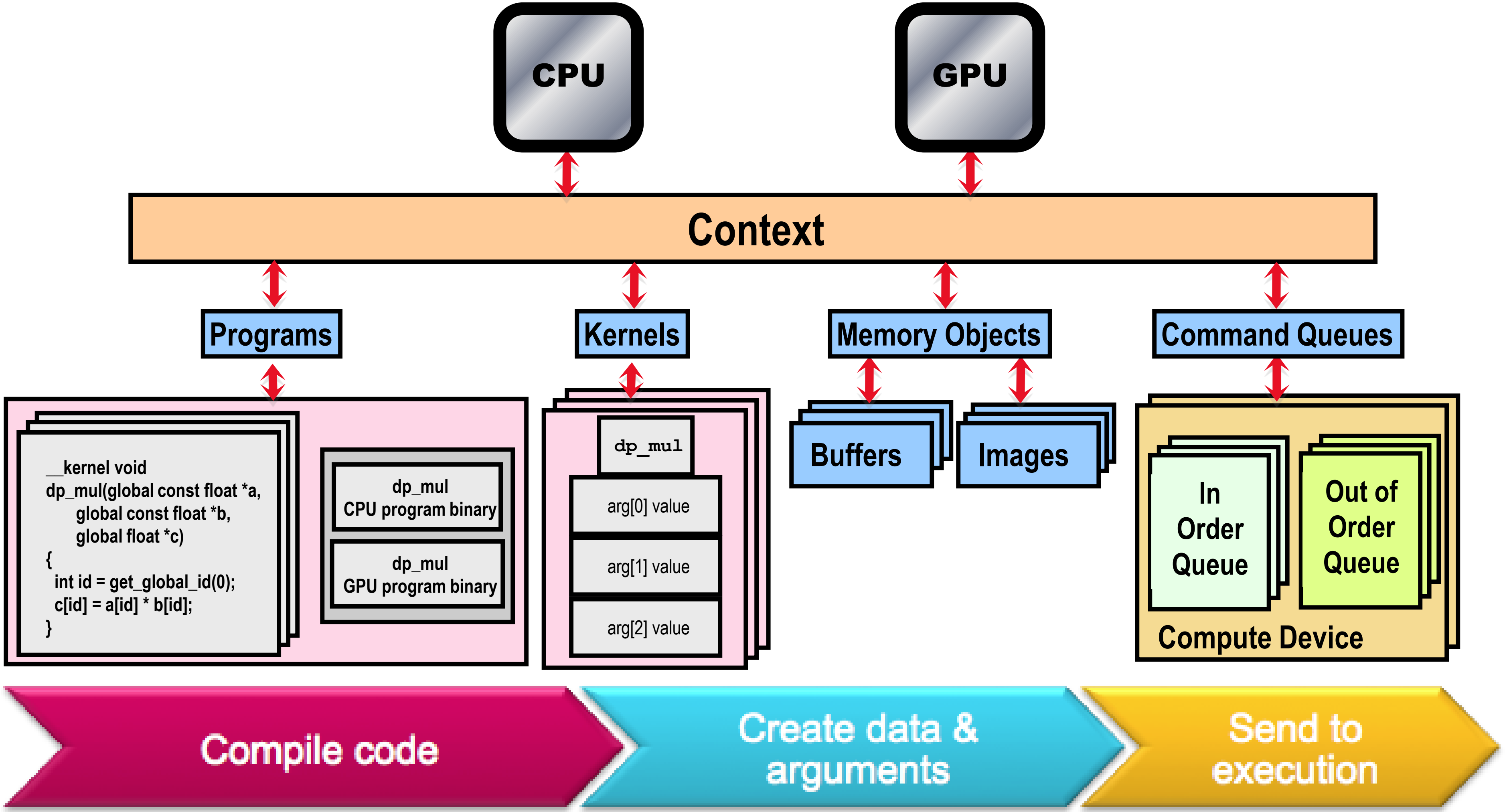
```
// execute kerne
err = clEnqueueN
                    global_work_size, NULL,0,NULL,NULL);
```

Execute the kernel

```
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],

                          0, NULL, NULL);
```

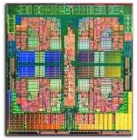
Read results on the host

A high level snapshot of what is going on



```
__kernel void
dp_mul(global const float *a,
      global const float *b,
      global float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
```

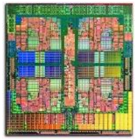
A „complete” OpenCL program



1. Select the desired devices (ex: all GPUs)
2. Create a context
3. Create command queues (per device)
4. Allocate memory on devices
5. Transfer data to devices
6. Compile programs
7. Create kernels
8. Execute
9. Transfer results back
10. Free memory on devices

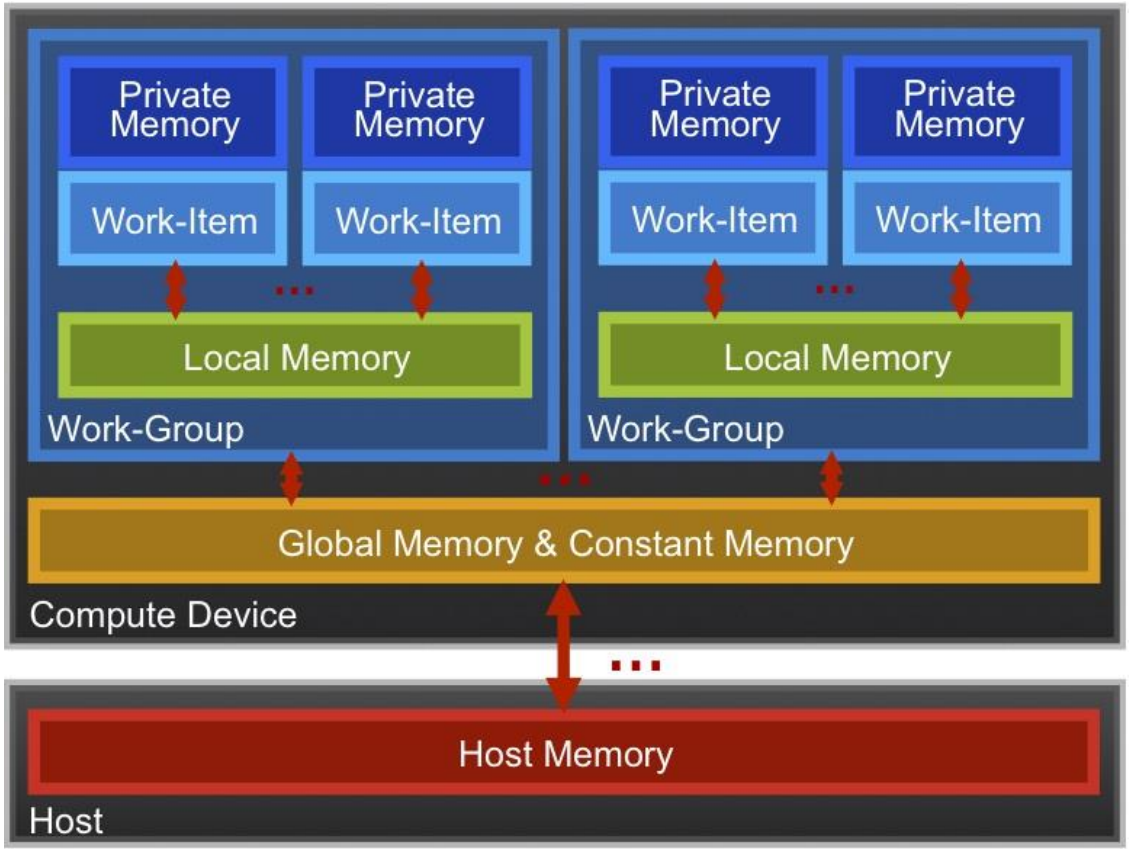
- clCreateProgramWithSource
- clBuildProgram
- clCreatKernel

A fierce beast – context



- ❑ We should understand context as the **environment** for managing both objects and resources in OpenCL sense
- ❑ This management is provided via **appropriate abstraction**
  - ❑ Context knows the **devices** as „something” that is capable of performing computations
  - ❑ Program objects: source that implements kernels
  - ❑ Kernels: code that can be executed on OpenCL enabled devices
  - ❑ Memory objects: data that is used by devices
  - ❑ Command queues: specialised mechanism for interacting with compute devices

Memory management



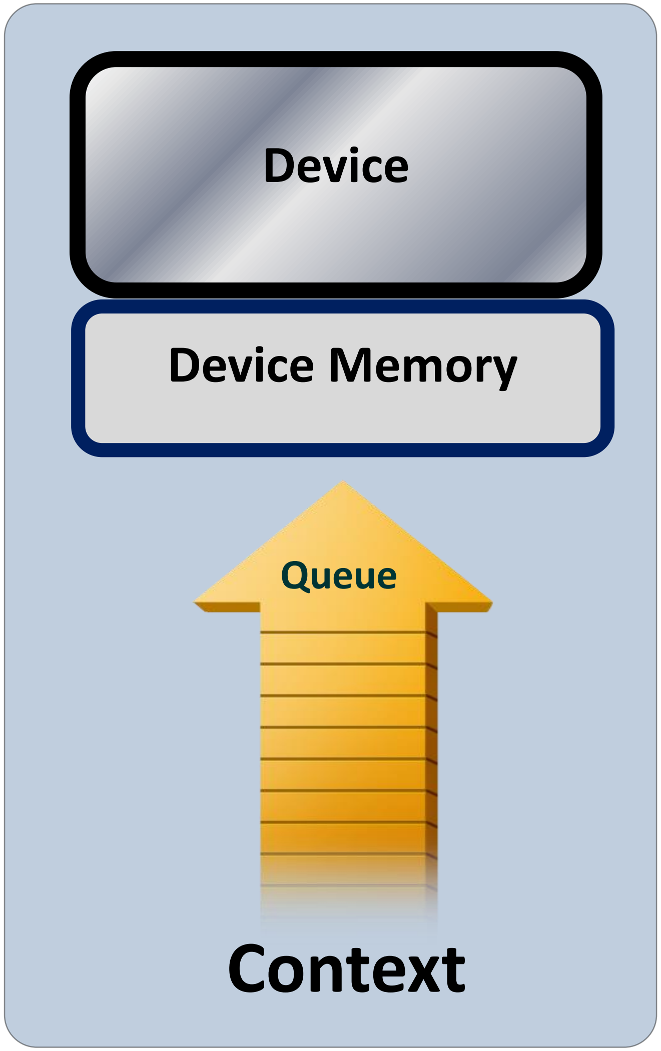
- ❑ Memory management is **explicit**:
  You are responsible for moving data from
  host → global → local *and* back

„Threads” mapping

OpenCL	CUDA
get_global_id(0)	blockIdx.x*blockDim.x+threadIdx.x
get_local_id(0)	threadIdx.x
get_global_size(0)	gridDim.x*blockDim.x
get_local_size(0)	blockDim.x

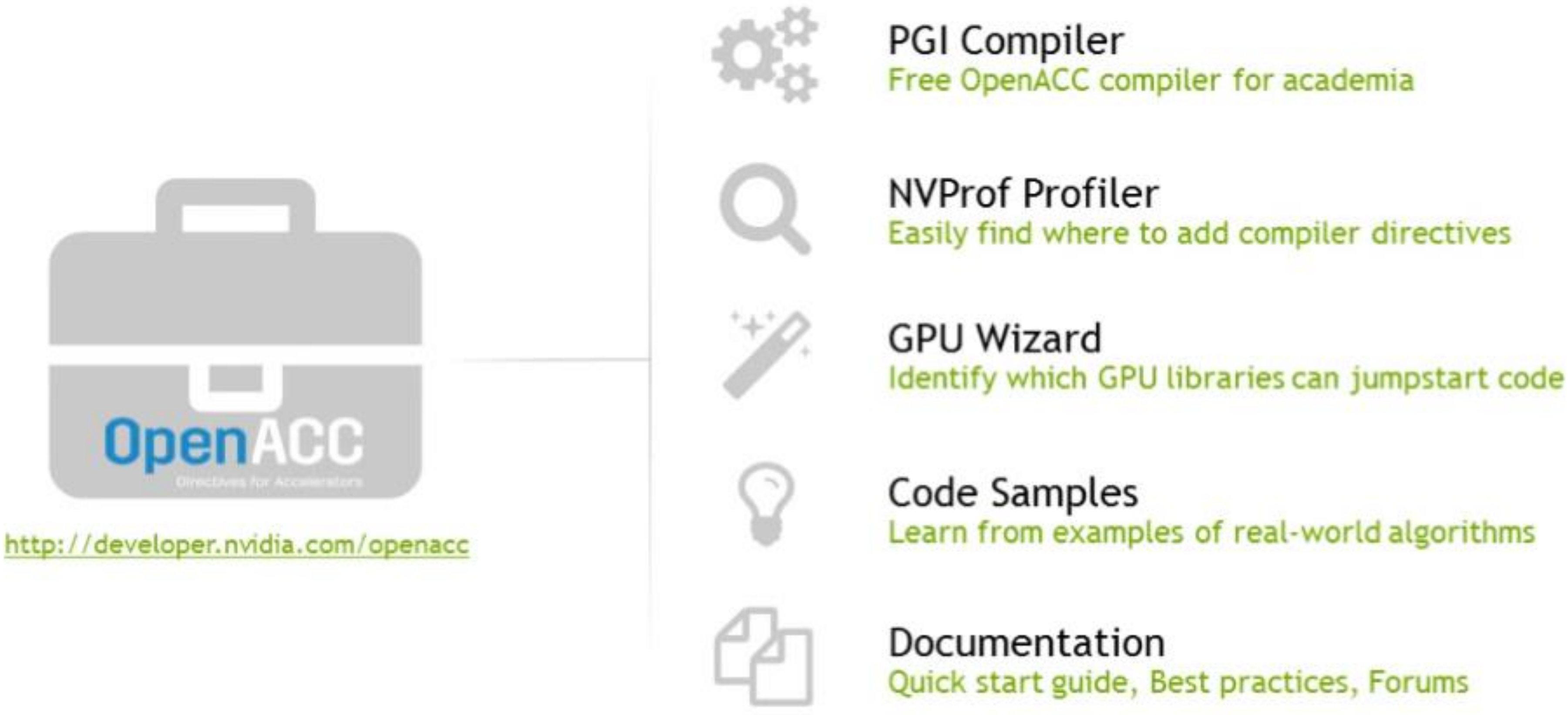
Context and Command-Queues

- ❑ **Context:**
  - ❑ The environment within which kernels execute and in which synchronization and memory management is defined.
- ❑ The **context** includes:
  - ❑ One or more devices
  - ❑ Device memory
  - ❑ One or more command-queues
- ❑ All **commands** for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a **command-queue**.
- ❑ Each **command-queue** points to a single device within a context.



This page is intentionally left blank

The toolkit



OpenACC
Directives for Accelerators

http://developer.nvidia.com/openacc

PGI Compiler
Free OpenACC compiler for academia

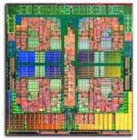
NVProf Profiler
Easily find where to add compiler directives

GPU Wizard
Identify which GPU libraries can jumpstart code

Code Samples
Learn from examples of real-world algorithms

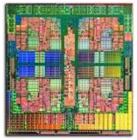
Documentation
Quick start guide, Best practices, Forums

Big picture



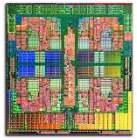
- **OpenACC** is making your computations much faster but in a completely different way…
- Minimal changes to your original code – fast to make (clear) and easy to maintain
- Hint the compiler how and where to try to make the code faster and it will obey! (almost each time that is…)
- It is somewhat in the middle of pure CUDA and OpenCL
- The source compilation will depend on the h/w resources present in your system – cool!

Big picture

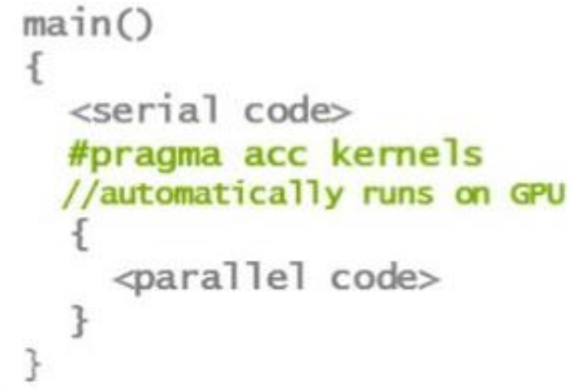


- The main motivation behind providing yet another way of accelerating stuff was to make it more accessible for scientist that do not like to do computing… (there are people like that!)
- In a way it is much more transparent and do not require people to attend CUDA lectures…
- The changes are made by introducing directives into the code
- However, if one wants to go deeper, as usual, extensive effort is needed – no pain no gain!

Memory hierarchy (II)

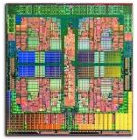


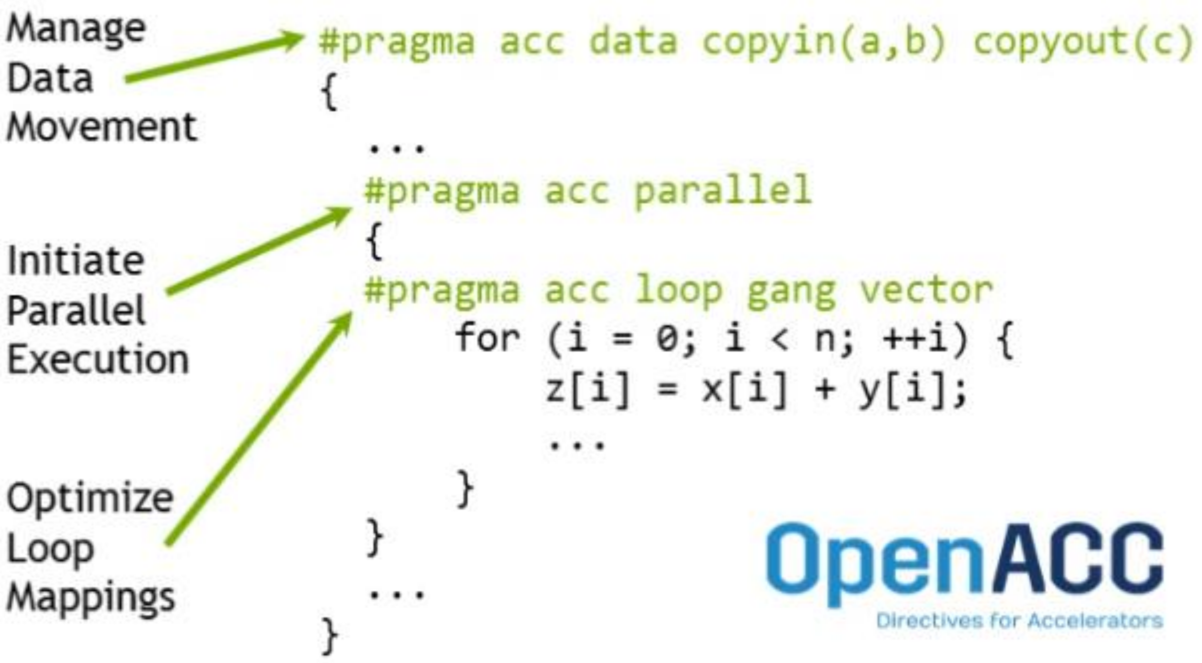
- The main paradigms of OpenACC
  - **Minimal intrusion** (just a few percent of code changes may bring a huge speed-ups)
  - **Use pragmas** (compiler hints)
  - **Portability** – do not limit your code to a given OS or h/w – one code to run everywhere



```
main()
{
   <serial code>
   #pragma acc kernels
   //automatically runs on GPU
   {
      <parallel code>
   }
}
```

A first view





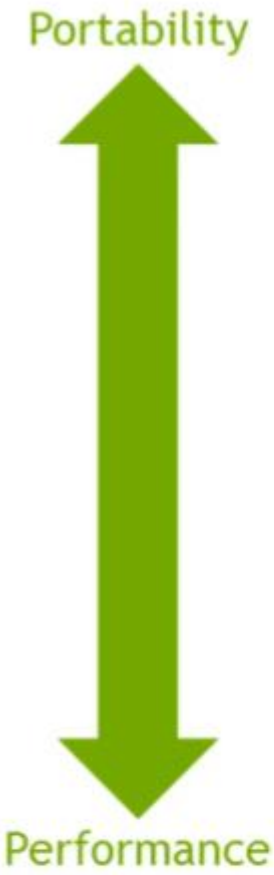
Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(a,b) copyout(c)
{
  ...
  #pragma acc parallel
  {
  #pragma acc loop gang vector
      for (i = 0; i < n; ++i) {
          z[i] = x[i] + y[i];
          ...
      }
  }
  ...
}
```

Compromises



Portability

Performance

Accelerated Libraries

High performance with little or no code change

Limited by what libraries are available

Compiler Directives

High Level: Based on existing languages; simple, familiar, portable

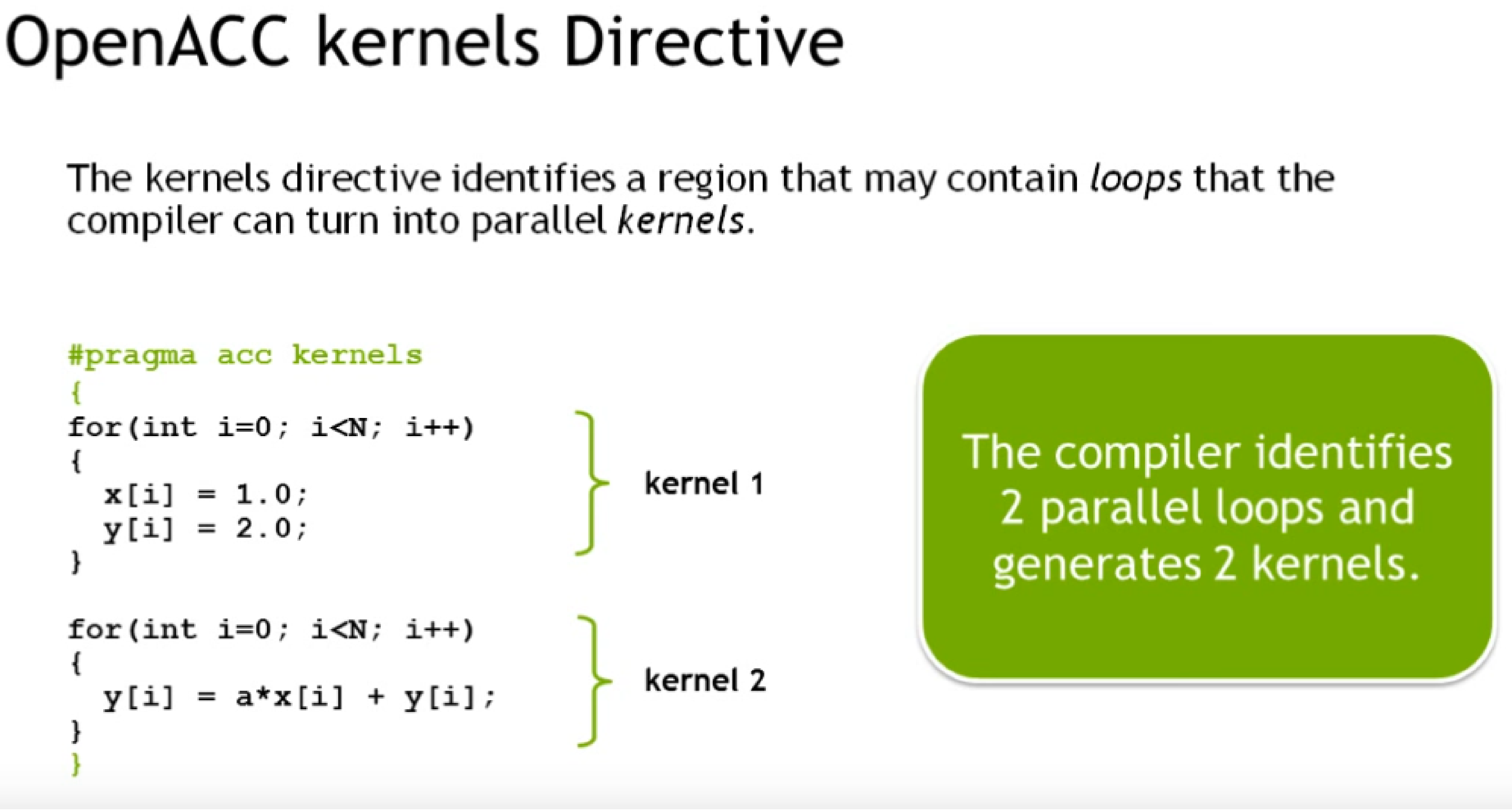
High Level: Performance may not be optimal

Parallel Language Extensions

Greater flexibility and control for maximum performance

Often less portable and more time consuming to implement

Kernel directives



OpenACC kernels Directive

The kernels directive identifies a region that may contain *loops* that the compiler can turn into parallel *kernels*.

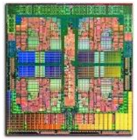
```
#pragma acc kernels
{
for(int i=0; i<N; i++)
{
  x[i] = 1.0;
  y[i] = 2.0;
}

for(int i=0; i<N; i++)
{
  y[i] = a*x[i] + y[i];
}
}
```

kernel 1

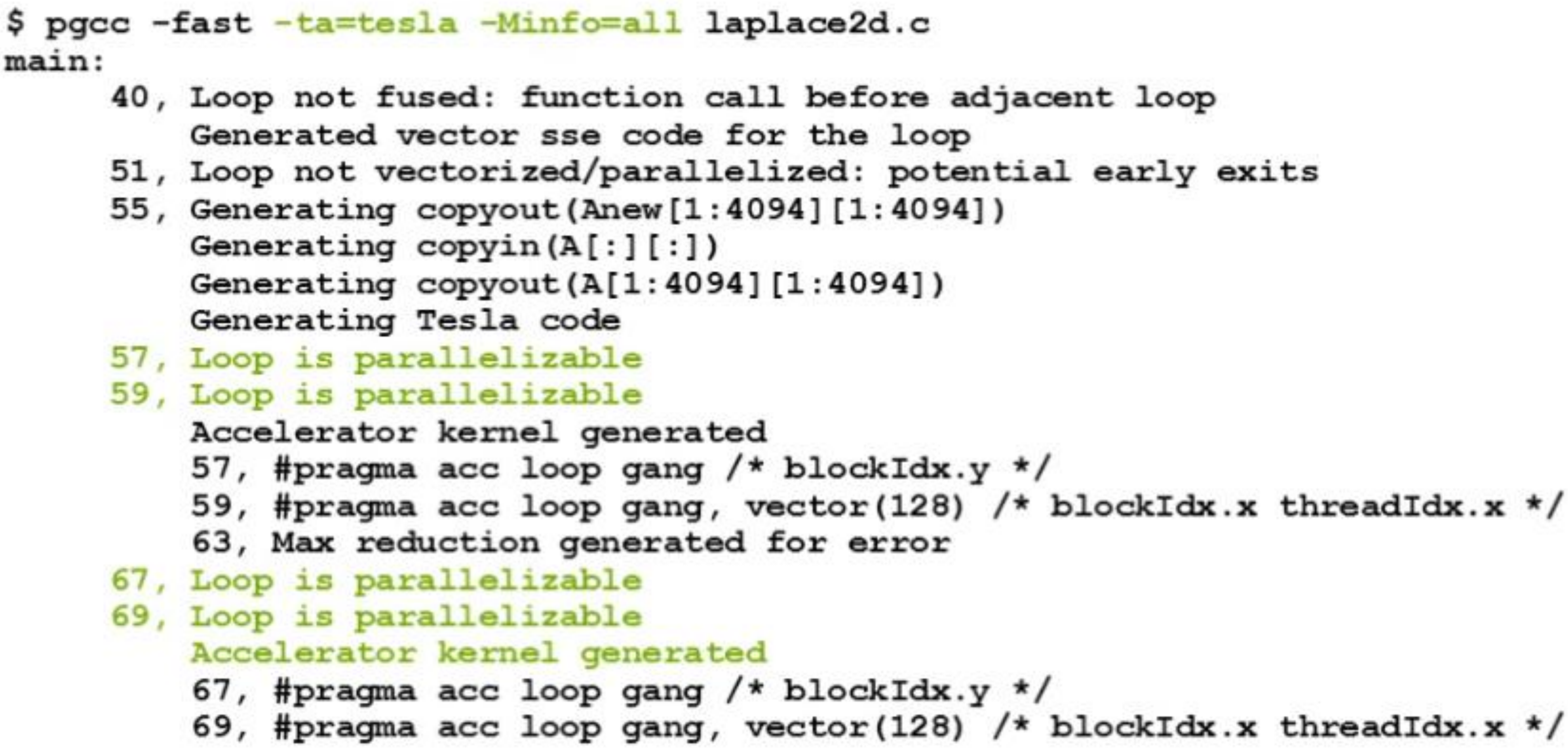
kernel 2

The compiler identifies 2 parallel loops and generates 2 kernels.

Compiling



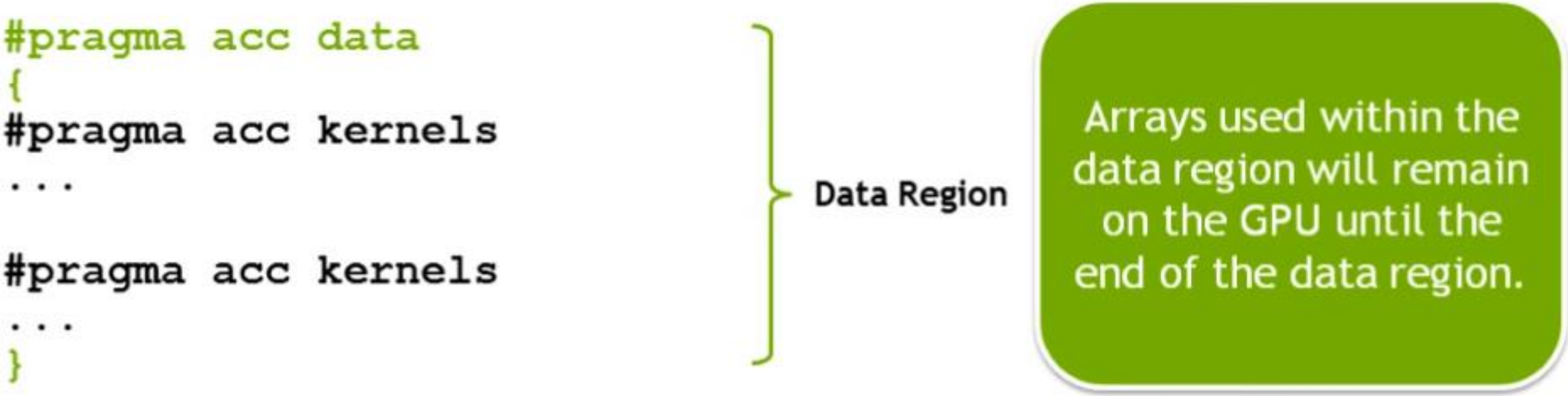
PGI comiler
(OpenACC toolkit)



```
$ pgcc -fast -ta=tesla -Minfo=all laplace2d.c
main:
     40, Loop not fused: function call before adjacent loop
         Generated vector sse code for the loop
     51, Loop not vectorized/parallelized: potential early exits
     55, Generating copyout(Anew[1:4094][1:4094])
         Generating copyin(A[:][:])
         Generating copyout(A[1:4094][1:4094])
         Generating Tesla code
     57, Loop is parallelizable
     59, Loop is parallelizable
         Accelerator kernel generated
         57, #pragma acc loop gang /* blockIdx.y */
         59, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
         63, Max reduction generated for error
     67, Loop is parallelizable
     69, Loop is parallelizable
         Accelerator kernel generated
         67, #pragma acc loop gang /* blockIdx.y */
         69, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Data movement… yes!

The data directive defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

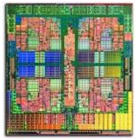


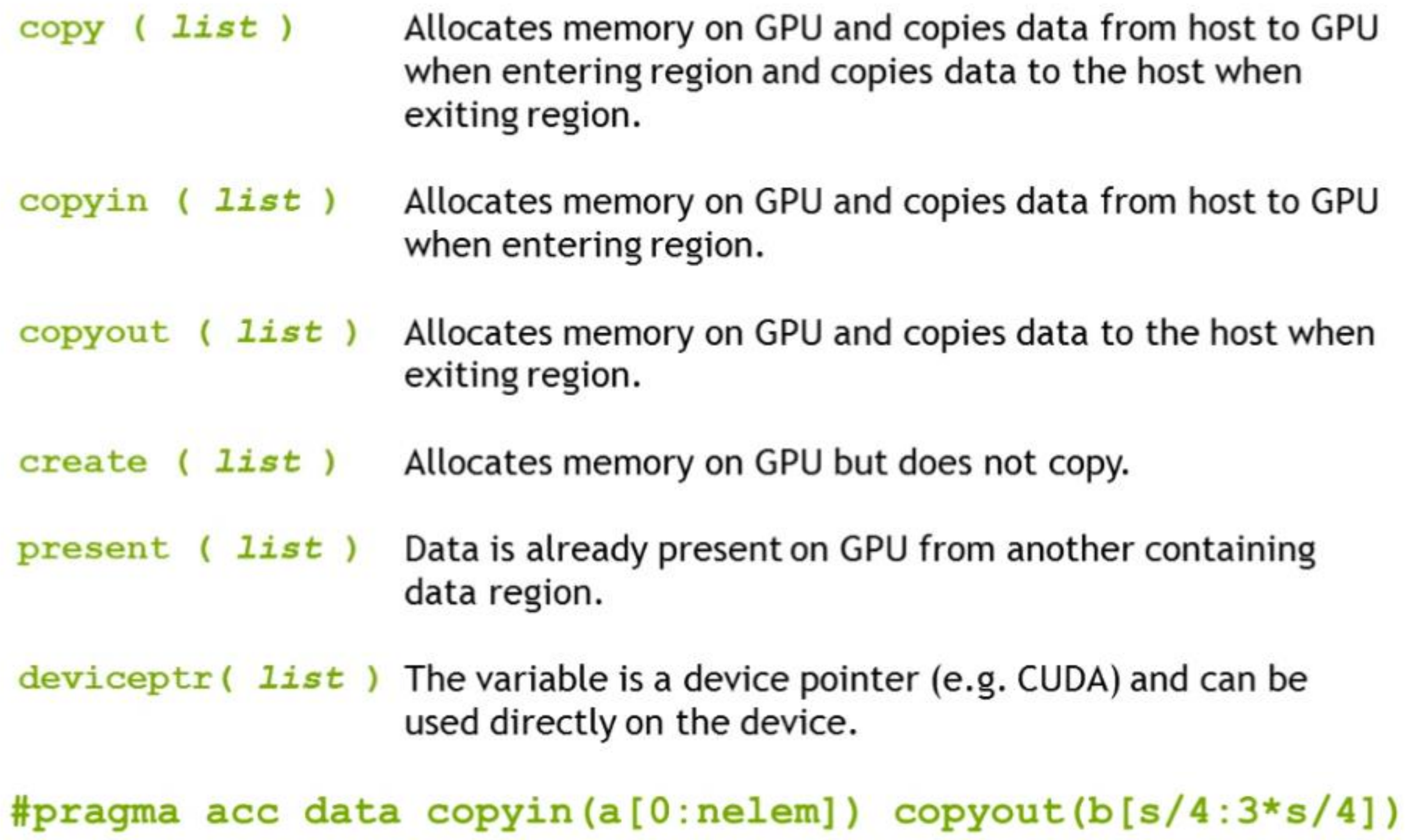
```
#pragma acc data
{
#pragma acc kernels
...

#pragma acc kernels
...
}
```

Arrays used within the data region will remain on the GPU until the end of the data region.

More hints to the compiler…

Data clauses



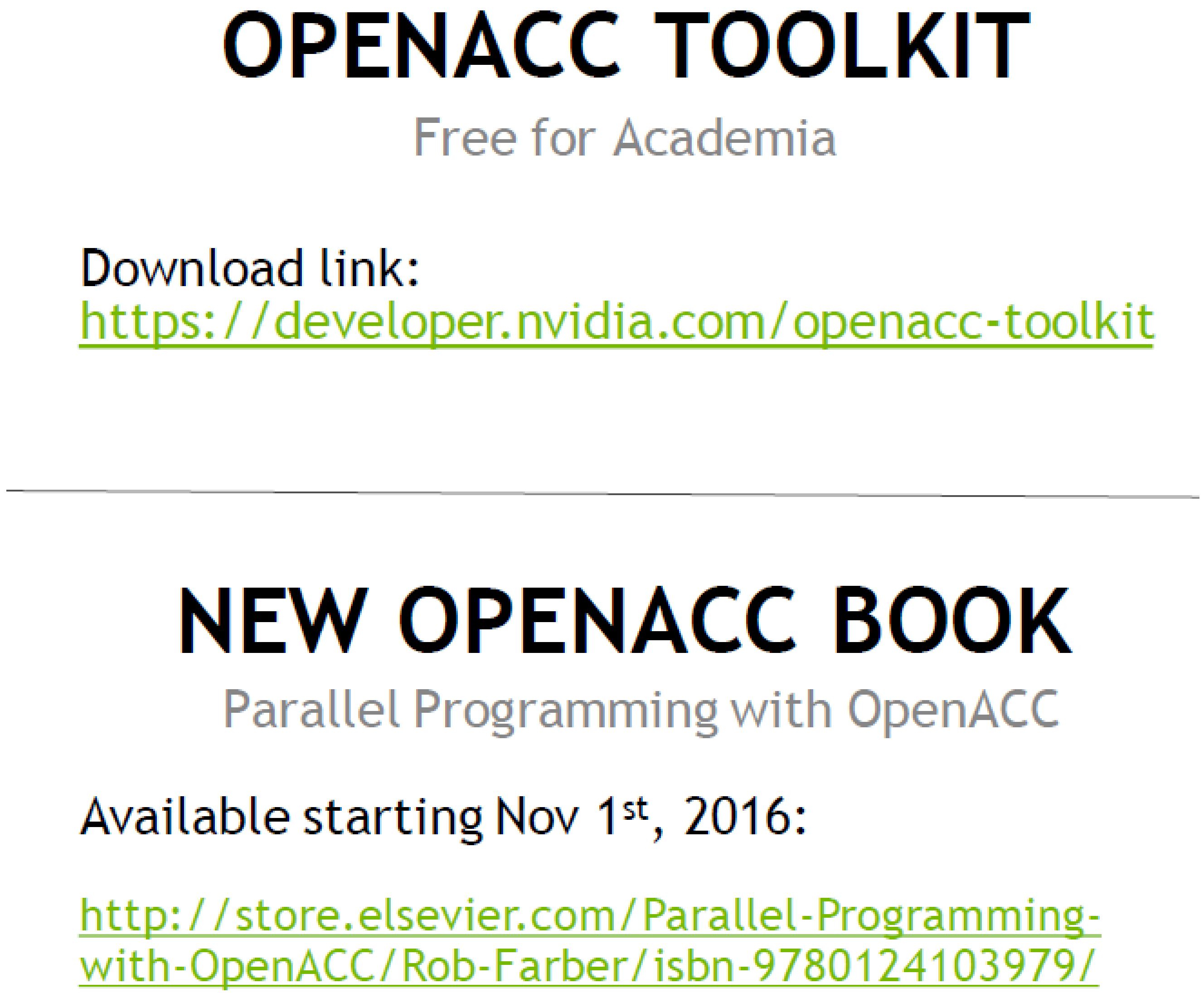


`copy ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
`copyin ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region.
`copyout ( list )`	Allocates memory on GPU and copies data to the host when exiting region.
`create ( list )`	Allocates memory on GPU but does not copy.
`present ( list )`	Data is already present on GPU from another containing data region.
`deviceptr( list )`	The variable is a device pointer (e.g. CUDA) and can be used directly on the device.

```
#pragma acc data copyin(a[0:nelem]) copyout(b[s/4:3*s/4])
```

Explicit shaping

Resources



OPENACC TOOLKIT

Free for Academia

Download link:
https://developer.nvidia.com/openacc-toolkit

NEW OPENACC BOOK

Parallel Programming with OpenACC

Available starting Nov 1st, 2016:

http://store.elsevier.com/Parallel-Programming-with-OpenACC/Rob-Farber/isbn-9780124103979/