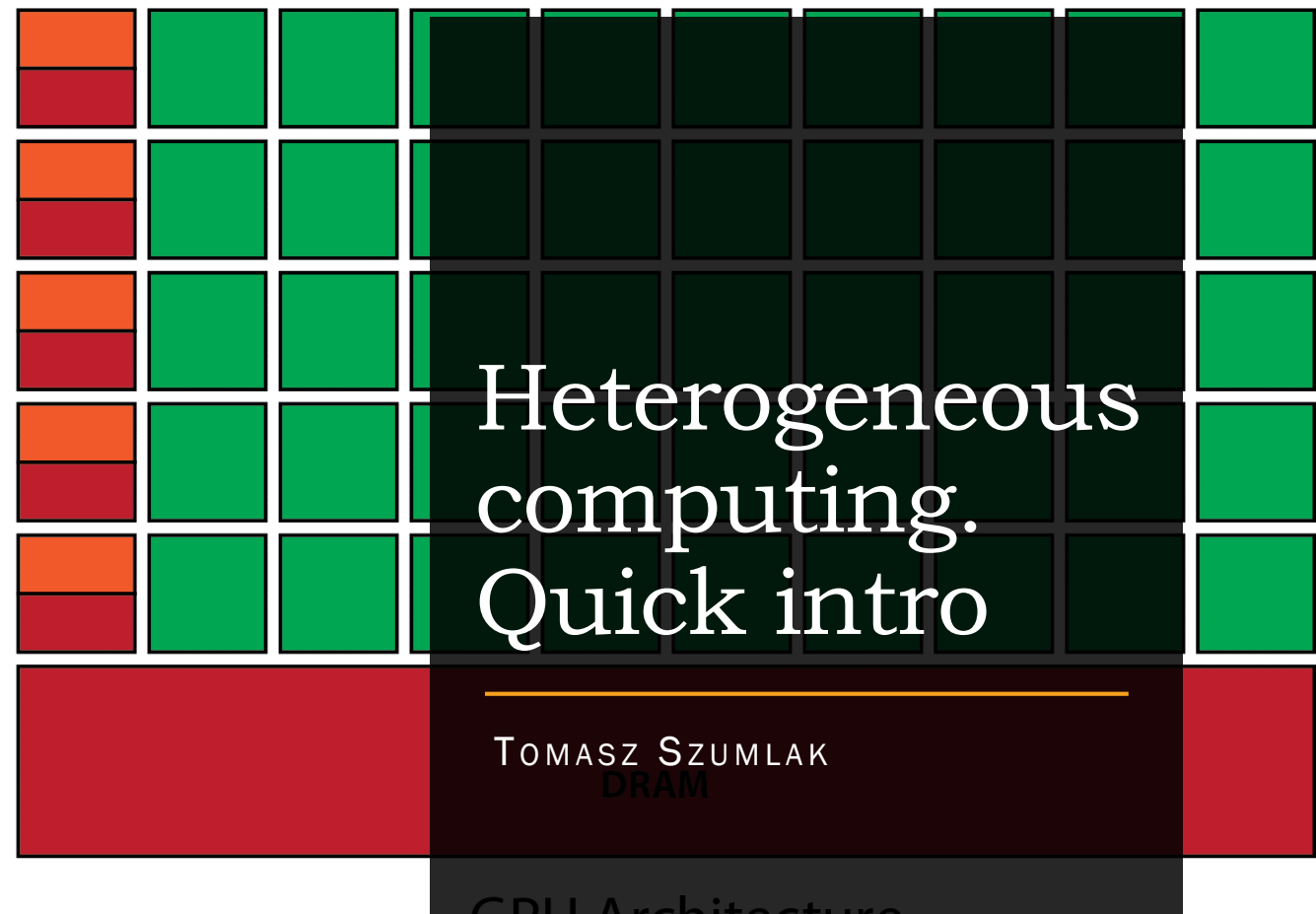


CPU Architecture



Heterogeneous
computing.
Quick intro

TOMASZ SZUMLAK
DRAM

GPU Architecture

What is heterogeneous computing?

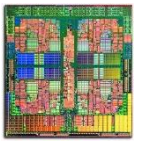
CPU, GP-GPU

TPU?

APU, DPU, SoC?

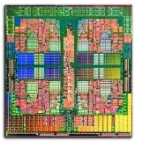
VPU, FPGA, ASIC, ARM?

Because all is heterogeneous now...



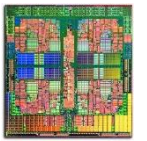
- ❑ In principle all devices from cell phones to large computing centres features **h. architecture**
- ❑ Even a cheap laptop now can combine up to three different processing units (P.U.): APU, CPU and GPU
- ❑ Formally, we define the **h.architecture** device, as one that use more than one kind of processor or cores
- ❑ These co-processors can be **ASICs, FPGA chips, GPU cards**, etc...
- ❑ The trick part is to provide **appropriate interfaces...**

What is available on the market?



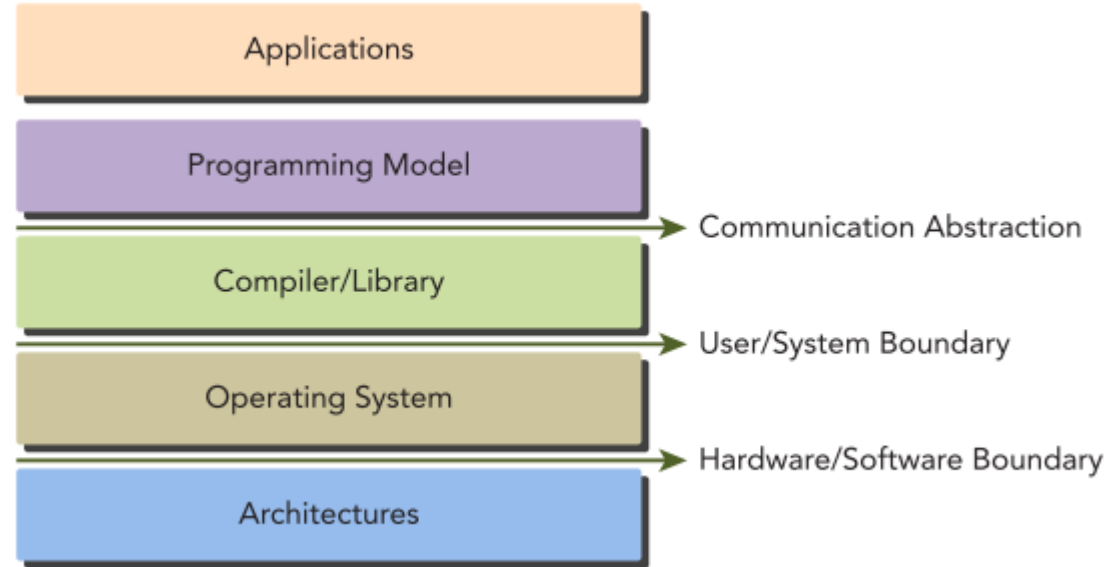
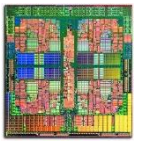
- ❑ **CUDA** – Compute Unified Device Architecture (the most popular proprietary platform for CPU-GPU systems)
- ❑ **OpenCL** – most popular open-source framework for executing code on h.architectures, very versatile and very powerful!
- ❑ **OpenACC** – a programming standard to facilitate parallel computing applications

CUDA



- In the **CUDA programming model** each computing system has:
 - It stands for: „**Compute Unified Device Architecture**“
 - **a host** – usually a ‘traditional’ CPU
 - **a device** (can be more than one actually!) – massive parallel coprocessors
 - with the coprocessor having a large number of arithmetic units
- Vital property of many algorithms is data parallelism
 - Processing operations can be safely executed on the application’s data at the same time
 - Matrix multiplication is an excellent example
- Application can be divided into parts
 - Typically sequential are run on CPU
 - These exhibiting data parallelism can be shipped to the coprocessor

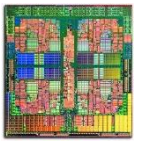
CUDA Programming Model



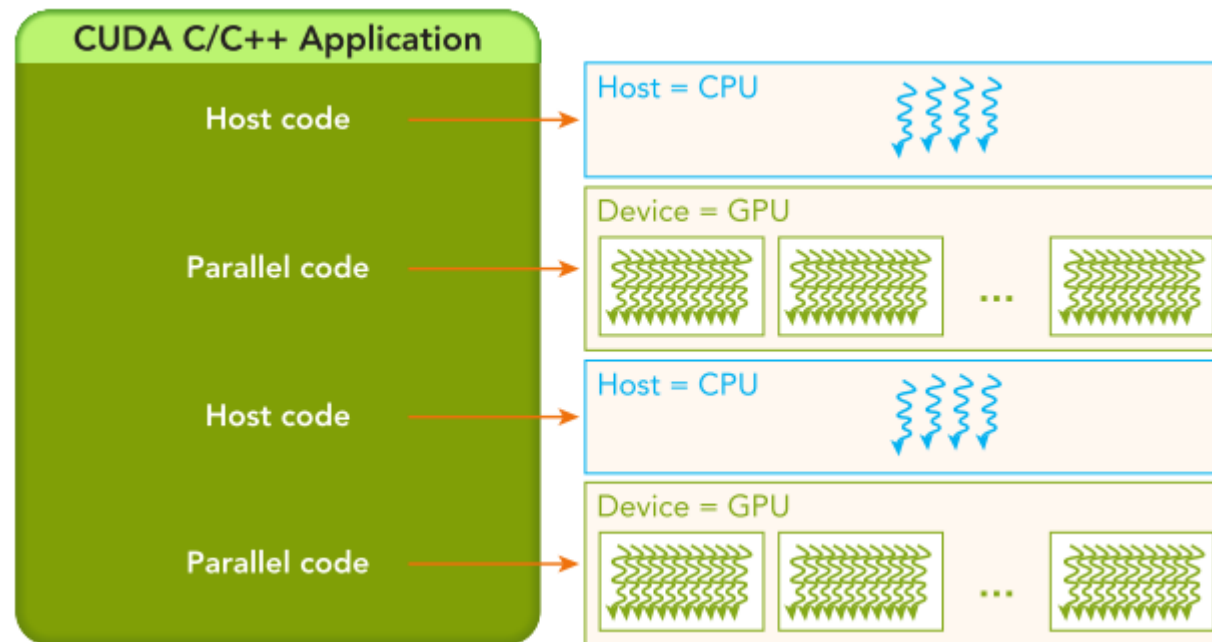
Copyright Wrox

- ❑ CUDA PM exposes to us the following:
 - ❑ a method to **organise threads** on the GPU using a hierarchy of threads
 - ❑ Tells us how to **access memory** on the GPU through a hierarchy of memory

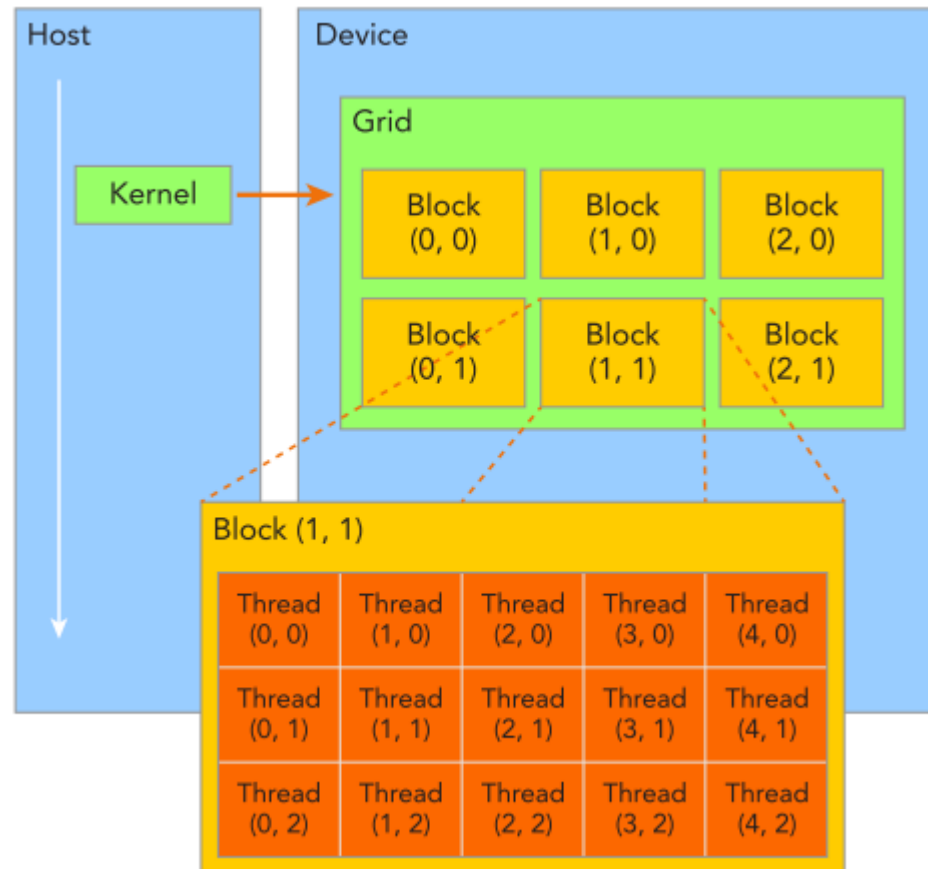
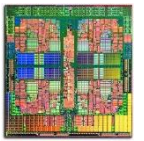
CUDA Programming Model



- ❑ And a quick reminder – CUDA Programming structure
 - ❑ **Host** and its memory
 - ❑ **Device** and its memory
 - ❑ Host → Device (processing) → Host



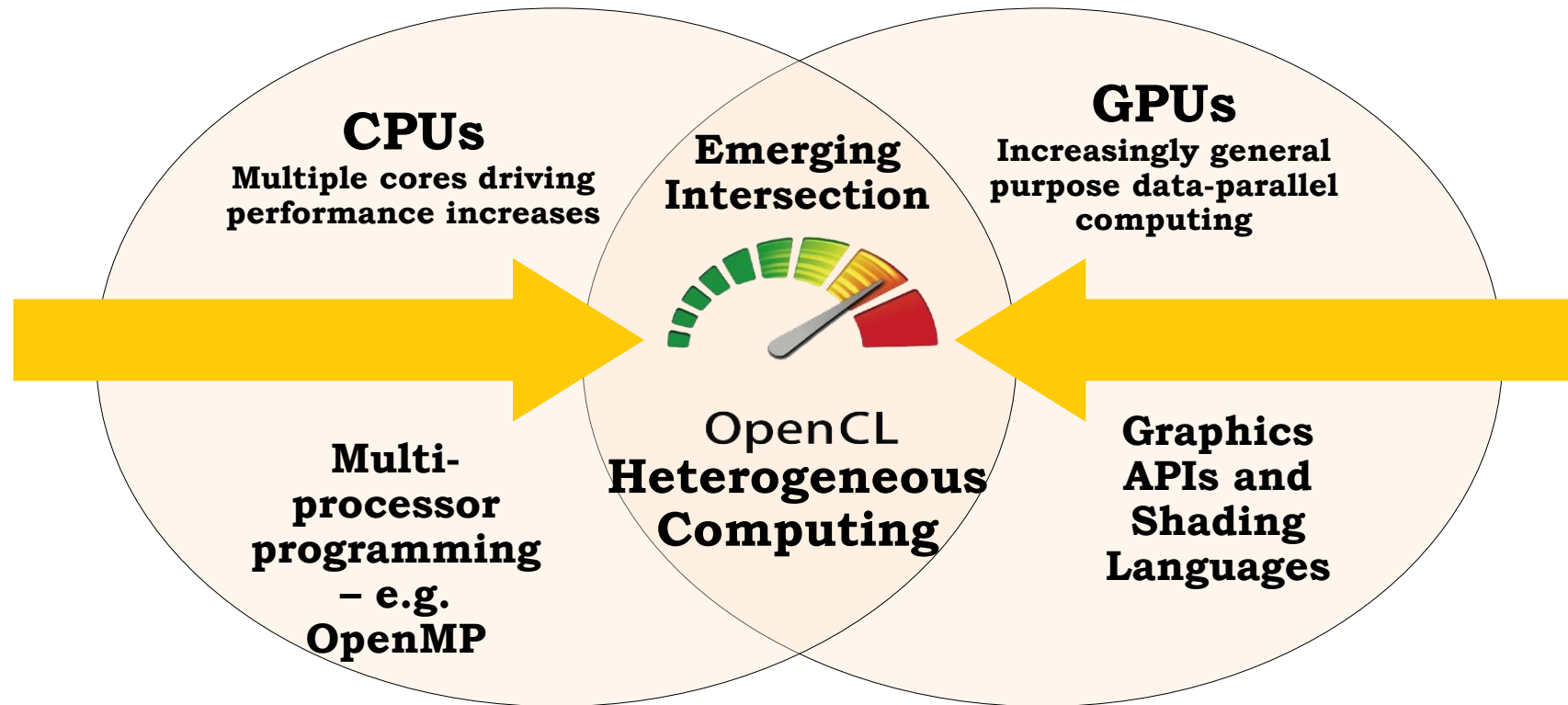
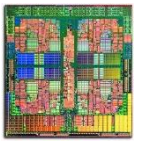
Managing the threads



Copyright by Wrox

- ❑ All threads ,spawn' by a **single** kernel is called a **grid**
- ❑ All threads in a grid **share** the same **global memory**
- ❑ A grid can be **split** into **blocks** of threads
- ❑ For each such block threads **can cooperate** with each other:
 - ❑ block-based synchronization
 - ❑ block level shared memory
- ❑ NOTE! Threads from **different** blocks **cannot cooperate!**
- ❑ All of this is done with a language that is an extension to C... (Saturday...)

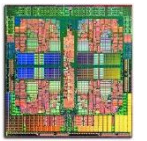
Open-source solution



OpenCL – simply had to be invented!

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, ASICs, FPGAs,....

Where to look for a kick-start



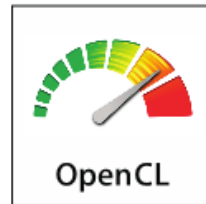
- ❑ A lot of excellent courses available on-line
- ❑ Definitely my winner is: „**Hands On OpenCL**”
 - ❑ It is a self consistent, end-to-end course
 - ❑ Hands-on examples provided via github repository
 - ❑ Very nice slides accompany the course (I borrowed a few!)
 - ❑ Extensive setting-up for various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

OpenCL Hands-on



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

🏠 UL HPC Tutorials

Search docs

- Home
- Latest HPC School
- SETUP
- Overview
- Pre-Requisites
- Clone and setup ULHPC Tutorials
- FIRST INTERACTIONS WITH ULHPC
- The UNIX/Linux Shell
- SSH and OpenOnDemand
- GETTING STARTED
- Overview
- Job scheduling with SLURM
- Using software modules
- TOOLS / SOFTWARE MANAGEMENT
- Easybuild
- Data Management
- SCHEDULING
- Advanced job scheduling with SLURM

Docs » GPU Programming » Introduction to OpenCL Programming

by **ULHPC** license **GPL-3.0** issues **0 open** slides **PDF** sources **github** docs **passing**  Star **81**

Introduction to OpenCL Programming (C/C++)

Copyright (c) T. Carneiro, L. Koutsantonis, 2021 UL HPC Team <hpc-team@uni.lu>



Uni.lu HPC School 2021

PS10b: Introduction to OpenCL Programming

Uni.lu High Performance Computing (HPC) Team
T. Carneiro L. Koutsantonis

University of Luxembourg (UL), Luxembourg

<https://hpc.uni.lu/>

<https://ulhpc-tutorials.readthedocs.io/en/latest/gpu/opencl/>

OpenCL Working Group within Khronos

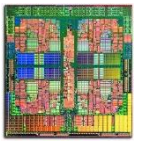


- ❑ Diverse industry participation
 - ❑ Processor vendors, system OEMs, middleware vendors, application developers.
- ❑ OpenCL became an important standard upon release by virtue of the market coverage of the companies behind it.



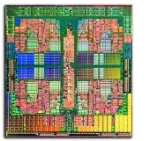
Third party names are the property of their owners.

Laying the foundation



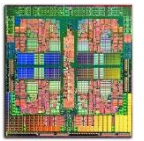
- ❑ The fundamental goal is to **use all computation units** (resources) available on a given system
- ❑ Exploits both **data** parallel (SIMD) and **task** parallel models
- ❑ **You create a OpenCL code by using extension to C** language (hmm, sounds similar to something you heard today...?)
- ❑ Providing 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 a bit 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

A hello word in OpenCL



- ❑ 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];
}
```



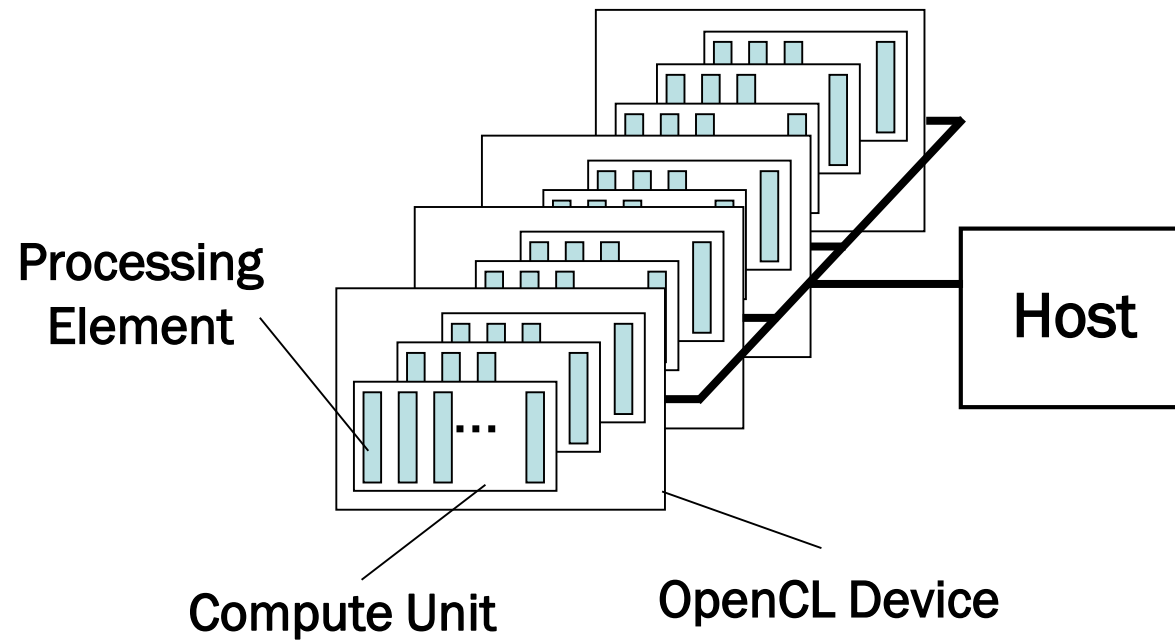
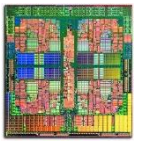
Data Parallel OpenCL

```
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];

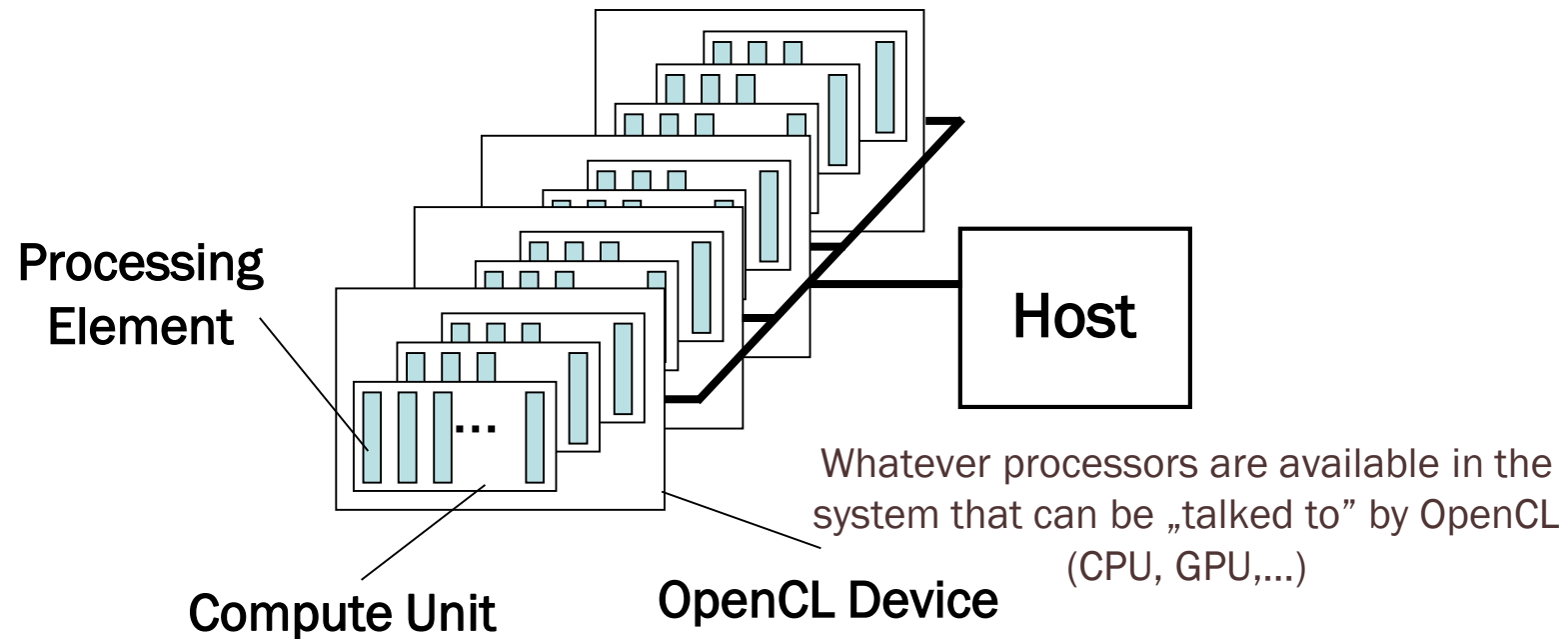
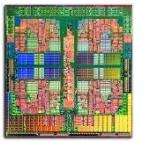
} // execute over "n" work-items
```

OpenCL Platform Model



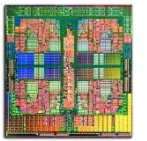
- ❑ One **Host** and one or more **OpenCL Devices**
 - ❑ Each OpenCL Device is composed of one or more **Compute Units**
 - ❑ Each Compute Unit is divided into one or more **Processing Elements**
- ❑ Memory divided into **host memory** and **device memory**

OpenCL Platform Model



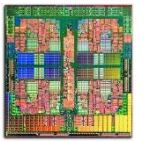
- ❑ One **Host** and one or more **OpenCL Devices**
 - ❑ Each OpenCL Device is composed of one or more **Compute Units**
 - ❑ Each Compute Unit is divided into one or more *Processing Elements*
- ❑ Memory divided into **host memory** and **device memory**

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”

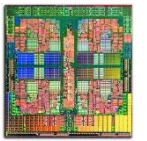
- ❑ Kernel
- ❑ Host application
- ❑ NDRange
- ❑ Work item
- ❑ Work group

- ❑ **CUDA** „style”

- ❑ Kernel
- ❑ Host application
- ❑ Grid
- ❑ Thread
- ❑ Block

- ❑ Aha! **If you know one, you know both of them!**

An N-dimensional domain of work-items

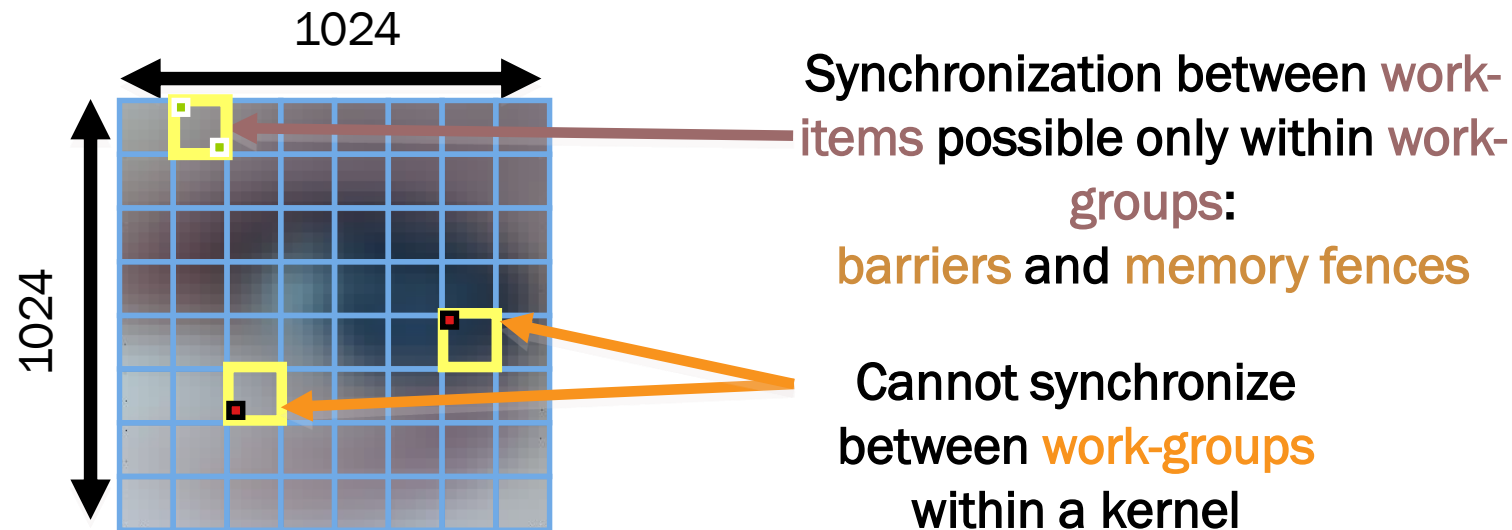


❑ Global Dimensions:

- ❑ 1024x1024 (whole problem space)

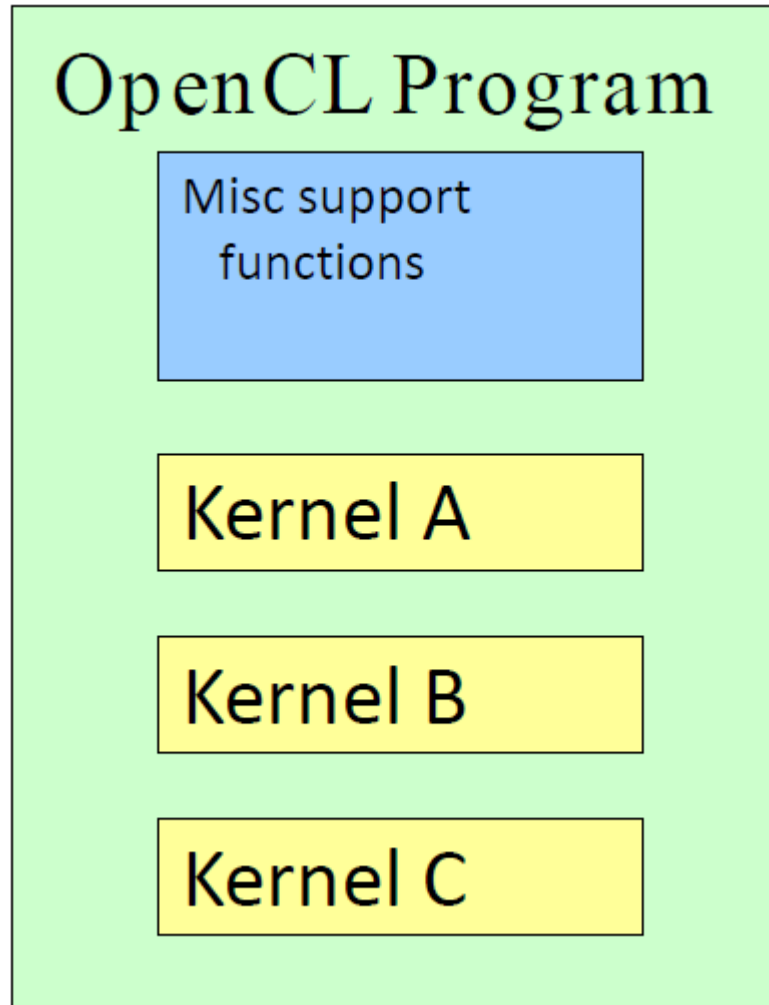
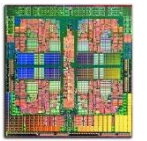
❑ Local Dimensions:

- ❑ 64x64 (**work-group**, executes together)



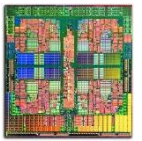
- ❑ Choose the dimensions that are “best” for your algorithm (tuning a bit more difficult)

A generic structure of an OpenCL program



- ❑ 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); // read output array

// 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);

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
```

```
clGetContextInfo(context, CL_DEVICE_LIST, 0, NULL, NULL);

cl_device_id devices[1];
clGetContextInfo(context, CL_DEVICE_LIST, 1, &devices, NULL);
```

Define platform and queues

```
// 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, srcb, NULL);
```

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

Create the program

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

Build the program

```
// create the kernel
```

```
kernel = clCreateKernel(program, "vec_add", NULL);
```

```
// set the args values
```

```
err = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
```

```
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1],
                       sizeof(cl_mem));
```

Create and setup kernel

```
// set work-item dimensions
```

```
global_work_size[0] = n;
```

```
// execute kernel
```

```
err = clEnqueueNDRangeKernel(kernel, 1, global_work_size, NULL, NULL, cmd_queue);
```

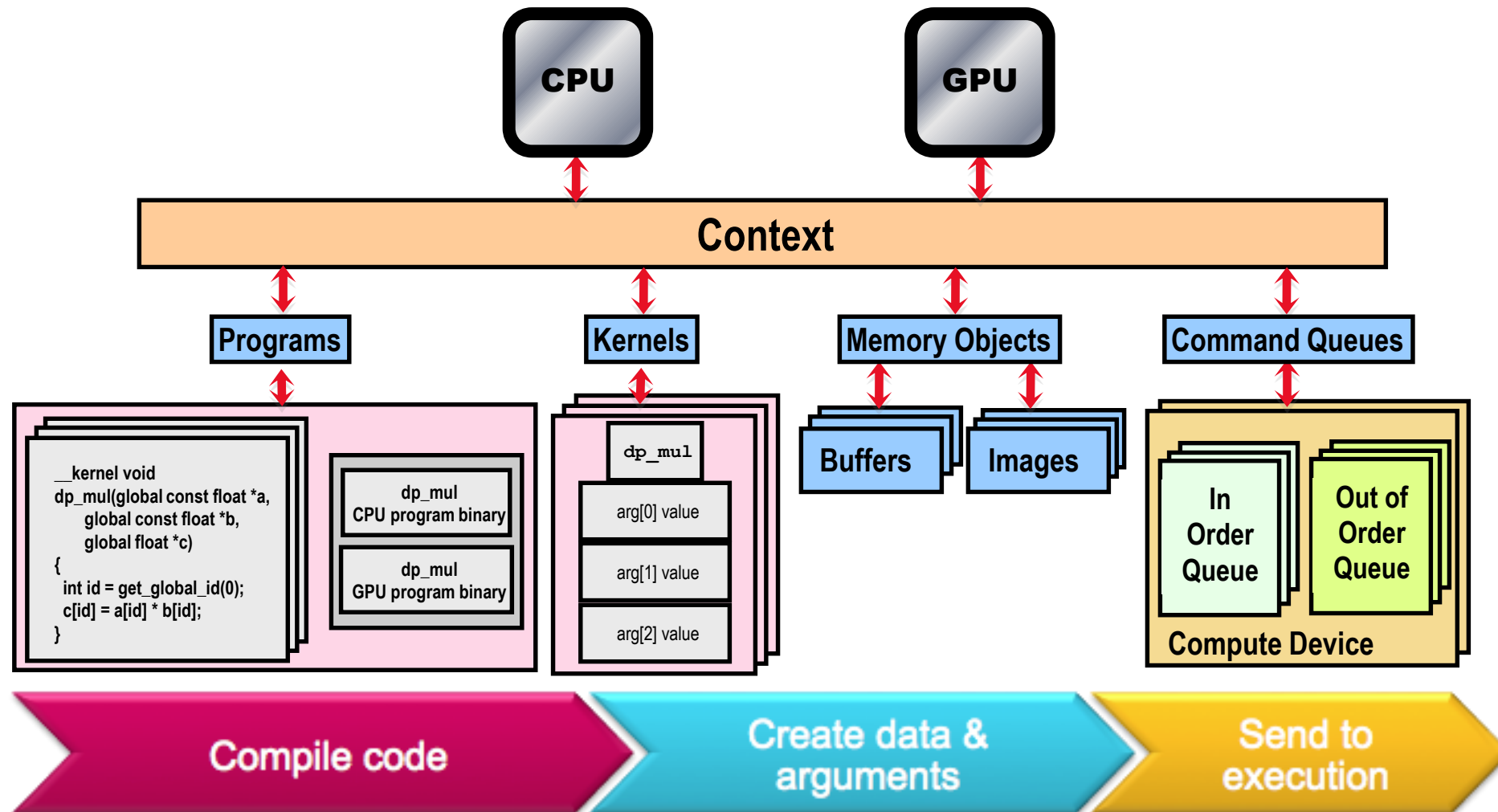
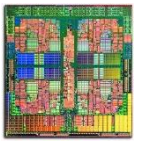
Execute the kernel

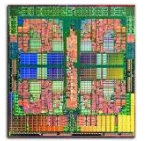
```
// read output array
```

```
err = clEnqueueReadBuffer(cmd_queue, memobjs[2], 0, 0, n, NULL, 0);
```

Read results on the host

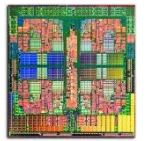
A high level snapshot of what is going on





A „complete” OpenCL program

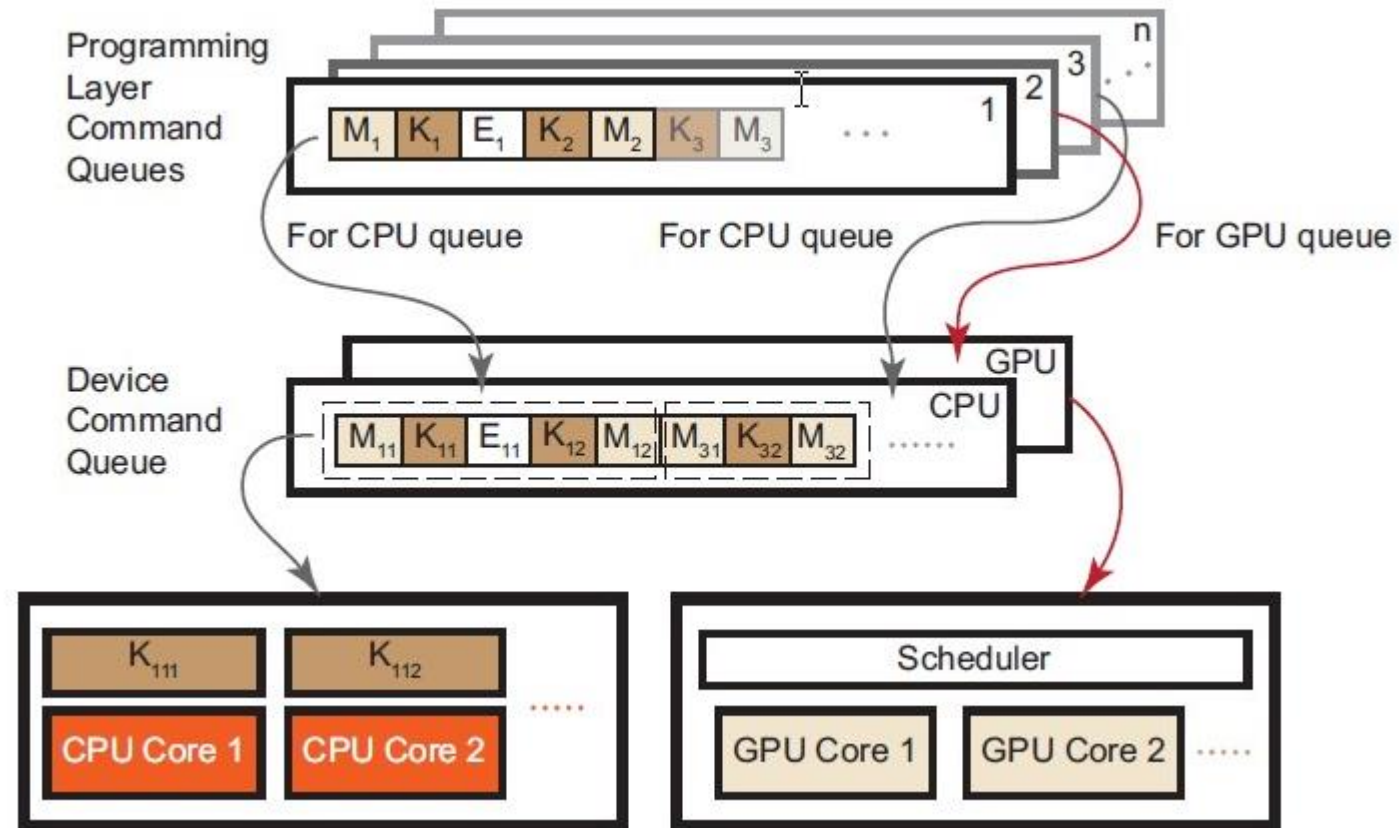
1. Select the desired devices (ex: all GPUs)
 2. Create a context
 3. Create command queues (per device)
 4. Allocate memory on devices
 5. Transfer data to devices
 6. Compile programs
 7. Create kernels
 8. Execute
 9. Transfer results back
 10. Free memory on devices
- `clCreateProgramWithSource`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



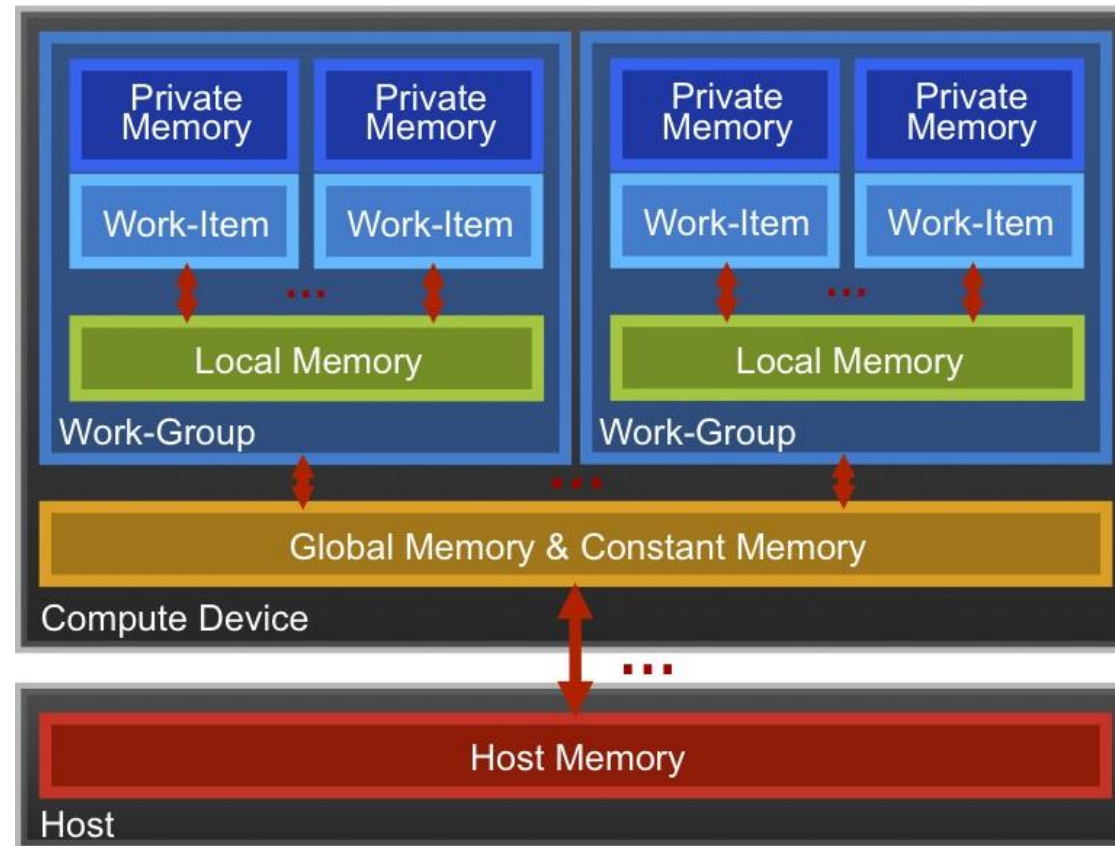
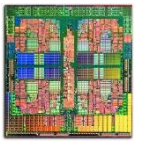
A fierce beast – context

- ❑ 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

Command queue

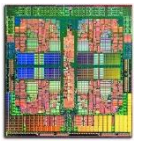


Memory management



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

„Threads” mapping

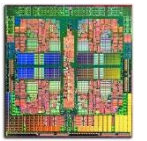


OpenCL

- `get_global_id(0)`
- `get_local_id(0)`
- `get_global_size(0)`
- `get_local_size(0)`

CUDA

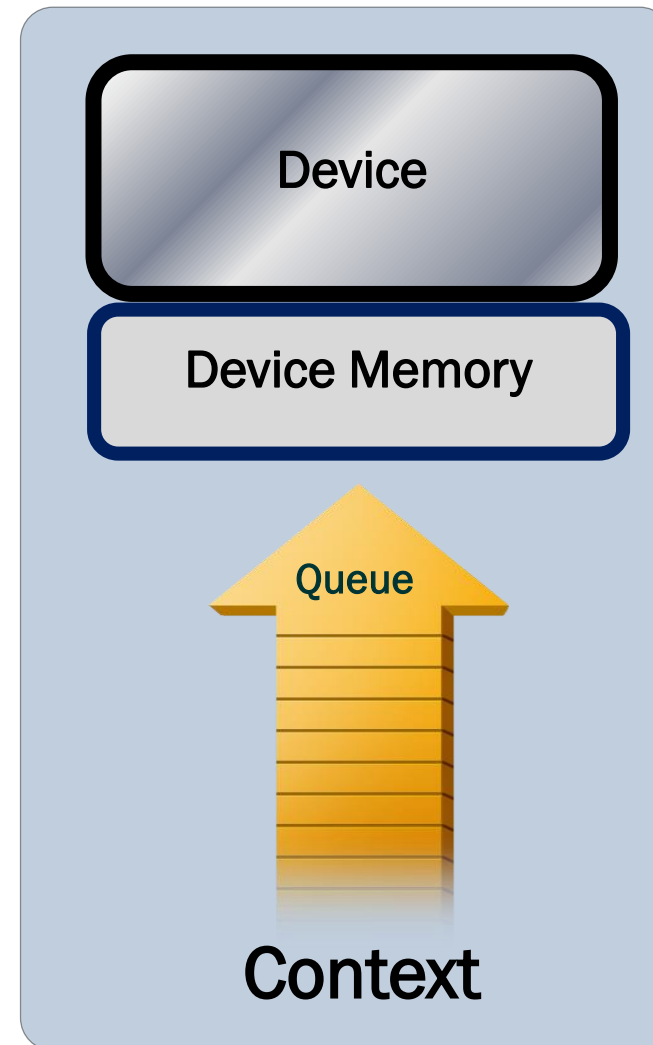
- `blockIdx.x*blockDim.x+threadIdx.x`
- `threadIdx.x`
- `gridDim.x*blockDim.x`
- `blockDim.x`



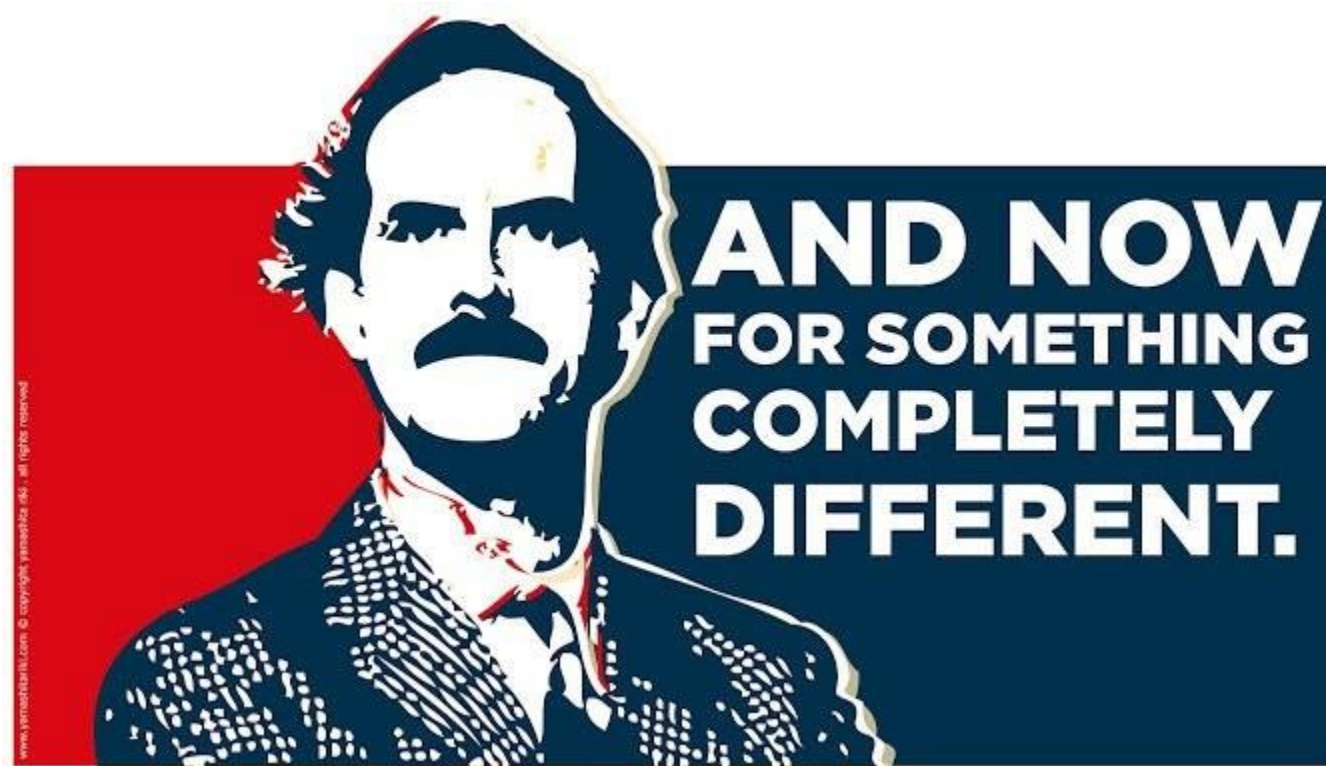
Context and Command-Queues

❑ 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
(almost) blank



The toolkit



<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

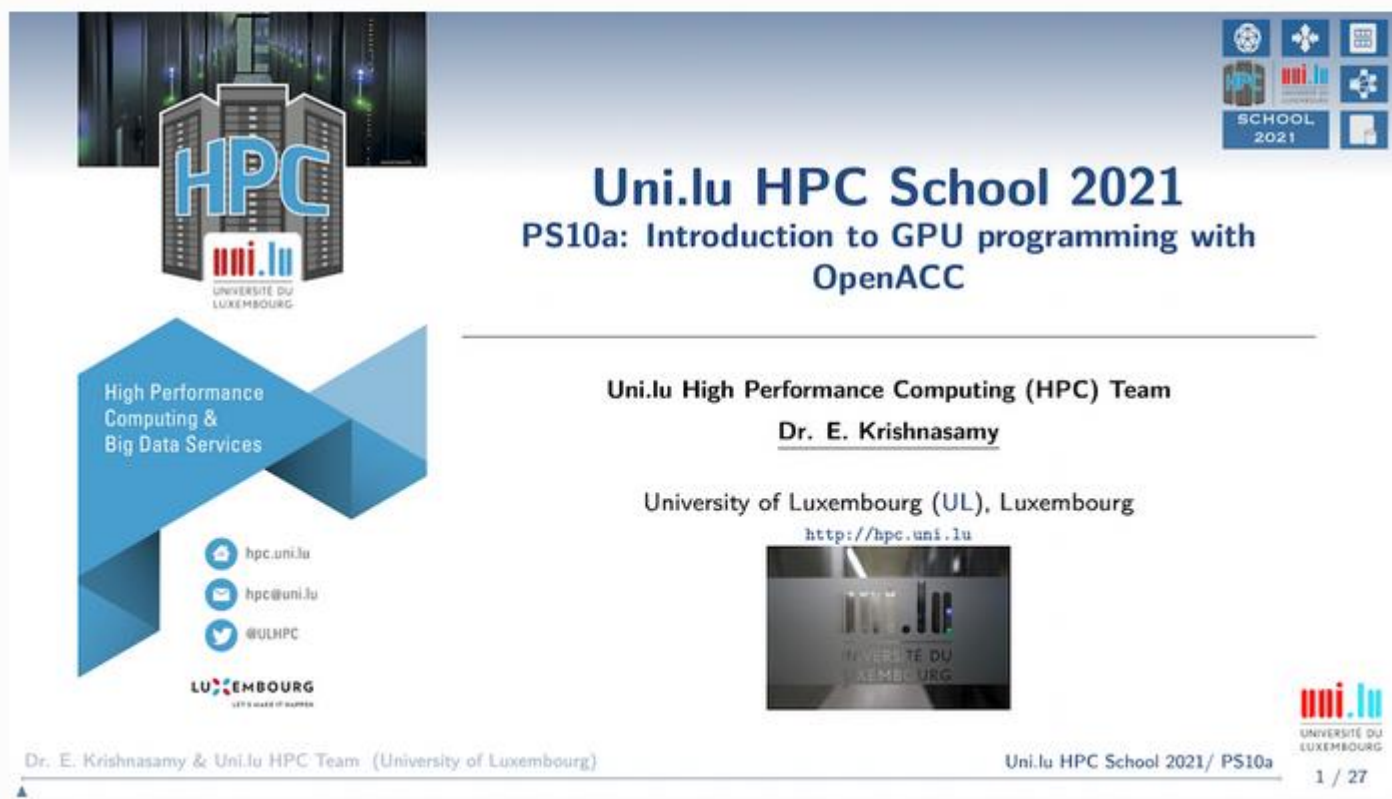


Documentation

Quick start guide, Best practices, Forums

Introduction to OpenACC Programming Model (C/C++ and Fortran)

Copyright (c) E. Krishnasamy, 2013-2021 UL HPC Team <hpc-sysadmins@uni.lu>



Uni.lu HPC School 2021
PS10a: Introduction to GPU programming with OpenACC

Uni.lu High Performance Computing (HPC) Team
Dr. E. Krishnasamy

University of Luxembourg (UL), Luxembourg
<http://hpc.uni.lu>

High Performance Computing & Big Data Services

hpc.uni.lu
hpc@uni.lu
@ULHPC

UNIVERSITÉ DU LUXEMBOURG

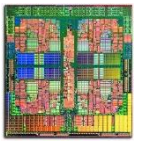
Dr. E. Krishnasamy & Uni.lu HPC Team (University of Luxembourg)

Uni.lu HPC School 2021/ PS10a

1 / 27

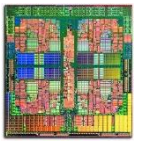
<https://ulhpc-tutorials.readthedocs.io/en/latest/gpu/openacc/basics/>

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!

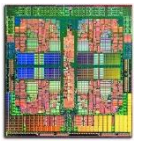
Big picture



- ❑ 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!



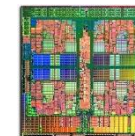
Main ideas



- ❑ 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>
    }
}
```

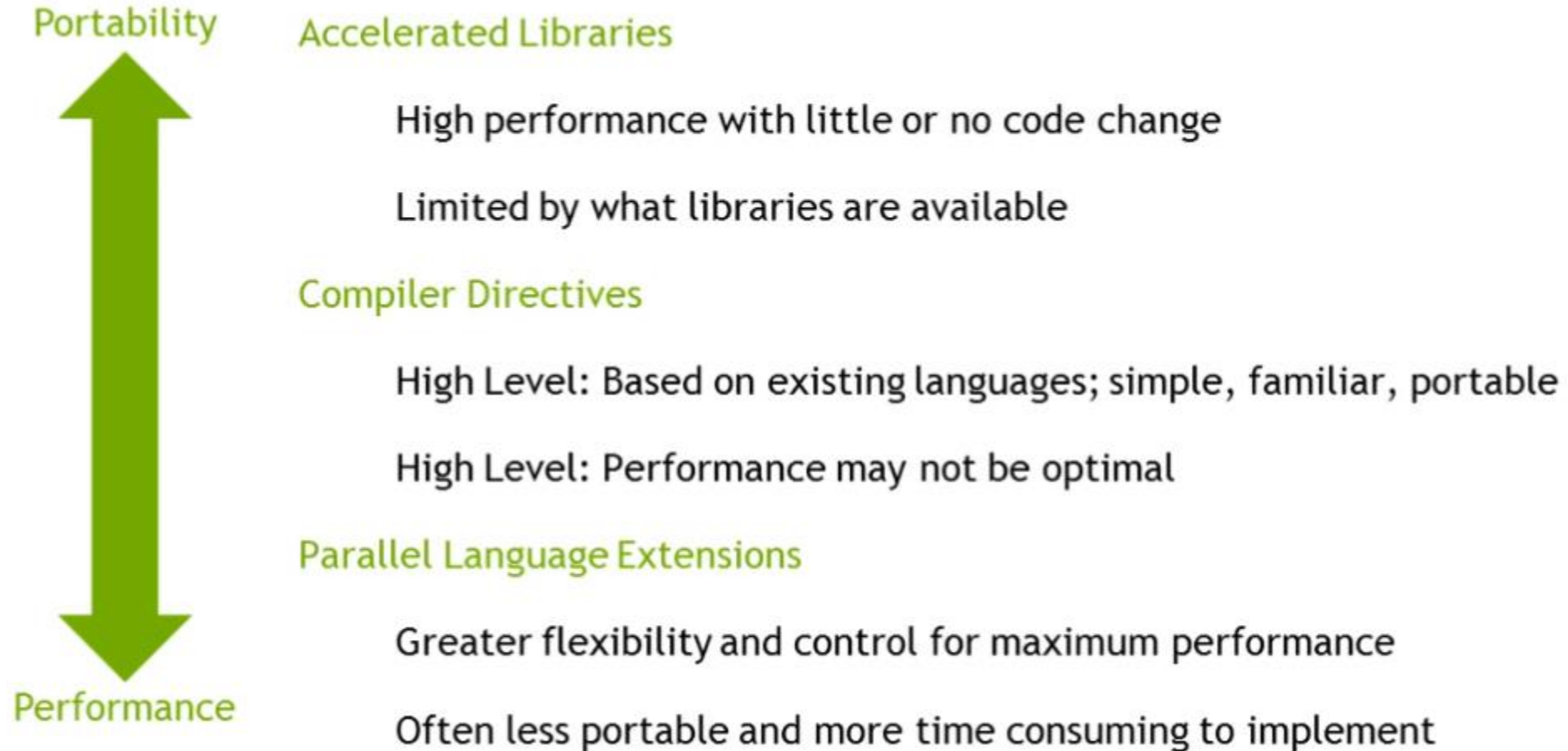
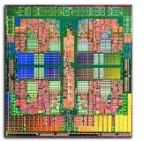
A first view



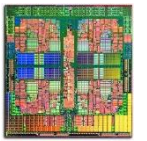
Manage
Data
Movement → `#pragma acc data copyin(a,b) copyout(c)`
`{`
`...`
`#pragma acc parallel`
`{`
`#pragma acc loop gang vector`
Initiate
Parallel
Execution → `for (i = 0; i < n; ++i) {`
`z[i] = x[i] + y[i];`
`...`
`}`
Optimize
Loop
Mappings → `}`
`...`
`}`

OpenACC
Directives for Accelerators

Compromises



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];
}
}
```

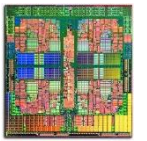
} kernel 1

} kernel 2

The compiler identifies
2 parallel loops and
generates 2 kernels.

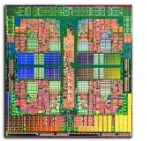
Compiling

PGI compiler
(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
...
}
```

} Data Region

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

More hints to the compiler...

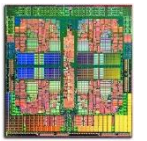
Data clauses



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

Explicit shaping

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



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/>