VRIJE UNIVERSITEIT BRUSSEL

ULB UNIVERSITÉ LIBRE DE BRUXELLES

GPU Computing

Lesson 6: Alternative GPU Programming Methods

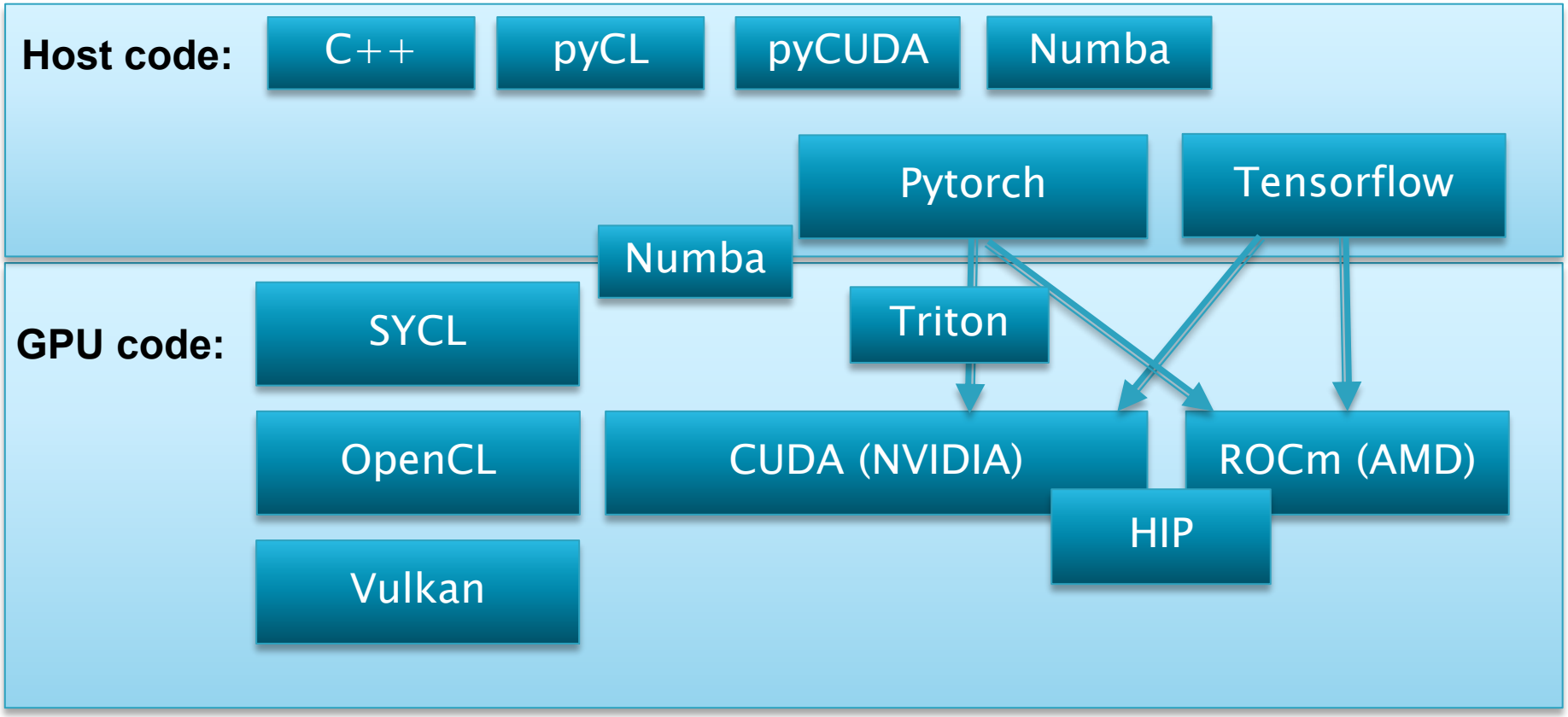
Jan Lemeire

2025-2026

http://parallel.vub.ac.be/education/gpu

VRIJE UNIVERSITEIT BRUSSEL

GPU programming Methods



CUDA

VRIJE UNIVERSITEIT BRUSSEL

CUDA <> OpenCL

- Code: almost a one-on-one mapping

VRIJE UNIVERSITEIT BRUSSEL

OpenCL	CUDA	
Work-Item	CUDA Thread	A single parallel execution instance.
Work-Group	CUDA Thread Block	Groups of threads; share local memory.
NDRange	Grid (Blocks × Threads)	Defines global execution structure.
Local Memory	Shared Memory	Fast on-chip memory per block/work-group.
Private Memory	Registers / Local Memory	Compiler-assigned private storage per thread.
Global Memory	Global Memory	Main GPU DRAM.
Constant Memory	Constant Memory	Both support cached read-only regions.
Enqueue Kernel	Kernel Launch (`kernel<<<grid, block>>>()`)	Both submit kernel execution to the device.
Events	CUDA Events	For synchronization and timing.
Barriers (`barrier()`)	`__syncthreads()`	Synchronizes threads in a work-group/block.

VRIJE UNIVERSITEIT BRUSSEL

OpenCL Concept	CUDA Equivalent	Notes
Platform	CUDA Driver/Runtime + Device Enumeration	OpenCL explicitly exposes platforms; CUDA implicitly uses NVIDIA's stack.
Device	CUDA Device (`cudaGetDevice`, etc.)	Both represent a GPU or accelerator.
Context	CUDA Context	CUDA contexts typically managed implicitly unless using driver API.
Command Queue	CUDA Stream	Both represent command submission queues.
Program	CUDA Module / NVRTC Program	OpenCL builds from source or binaries; CUDA uses PTX/Cubin or NVRTC for JIT.
Kernel	CUDA Kernel (global function)	Same concept: a function executed in parallel on the GPU.
Memory Buffer (`clCreateBuffer`)	cudaMalloc / cudaMallocManaged	Core memory allocation.

VRIJE UNIVERSITEIT BRUSSEL

Example

OpenCL

```
__kernel void vector_add(__global const float*
A,
                         __global const float* B,
                         __global float* C)
{
    int i = get_global_id(0);
    C[i] = A[i] + B[i];
}
```

CUDA

```
 __global__ void vector_add(const float* A,
                            const float* B,
                            float* C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Host code

```
__global__ void vector_add(const float* A, const float* B, float* C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
int main() {
    // 1. Allocate GPU memory
    cudaMalloc(&dA, sizeof(A));
    cudaMalloc(&dB, sizeof(B));
    cudaMalloc(&dC, sizeof(C));
    // 2. Copy inputs
    cudaMemcpy(dA, A, sizeof(A), cudaMemcpyHostToDevice);
    cudaMemcpy(dB, B, sizeof(B), cudaMemcpyHostToDevice);
    // 3. Launch kernel
    int blockSize = 256;
    int gridSize = (N + blockSize - 1) / blockSize;
    vector_add<<<gridSize, blockSize>>>(dA, dB, dC);
    // 4. Copy result back
    cudaMemcpy(C, dC, sizeof(C), cudaMemcpyDeviceToHost);

    // 5. Free
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
}
```

Kernel code in same file

Example 2: matrix multiplication

```
__kernel void matmul(__global const float* A, __global const float* B,
__global float* C, const int N){
    int row = get_global_id(1);   // y dimension
    int col = get_global_id(0);   // x dimension
    float sum = 0.0f;
    for (int k = 0; k < N; k++)
       sum += A[row * N + k] * B[k * N + col];
   C[row * N + col] = sum;
}
```

OpenCL

```
__global__ void matmul(const float* A, const float* B, float* C, int N)
{
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   int row = blockIdx.y * blockDim.y + threadIdx.y;

   float sum = 0.0f;
   for (int k = 0; k < N; k++)
          sum += A[row * N + k] * B[k * N + col];
   C[row * N + col] = sum;
}
```

CUDA

VRIJE UNIVERSITEIT BRUSSEL

Simple reduction with shared memory

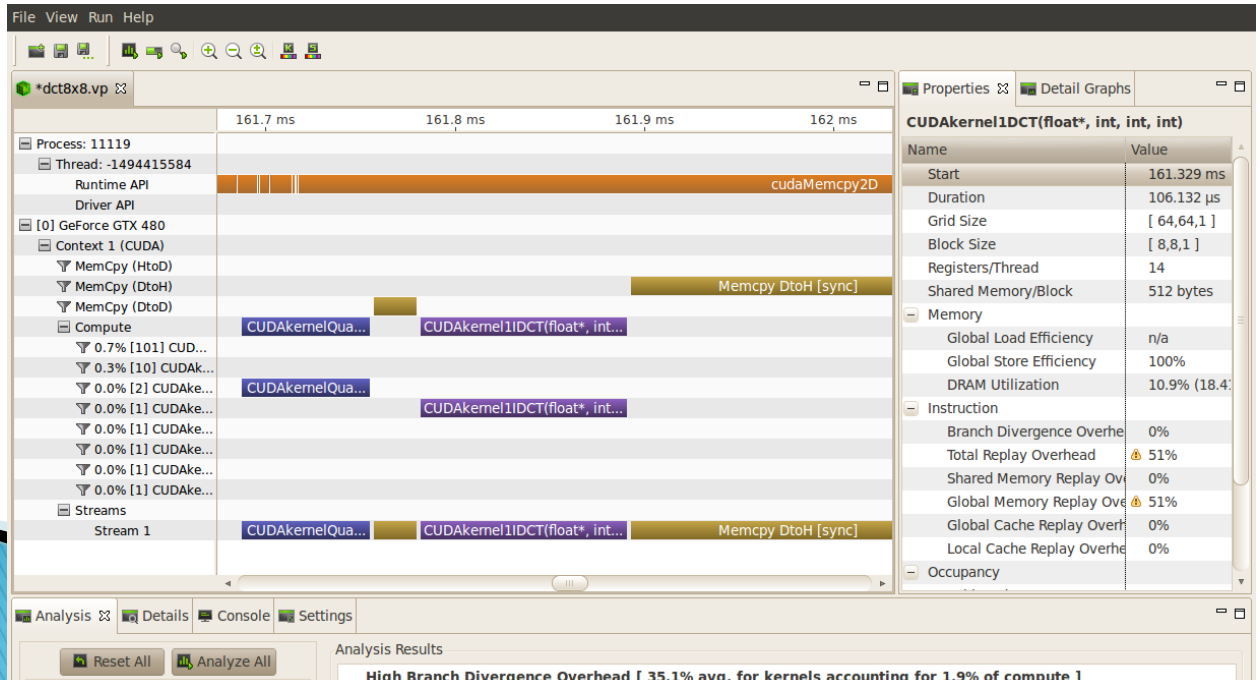
```
__global__ void block_reduce(const float* input, float* output) {
    __shared__ float sdata[256]; // Shared memory buffer for this block

    int tid  = threadIdx.x;
    int idx  = blockIdx.x * blockDim.x + threadIdx.x;

    sdata[tid] = input[idx]; // Load data into shared memory
    __syncthreads(); // Barrier

   for (int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
              if (tid < stride)
          sdata[tid] += sdata[tid + stride];
       __syncthreads();  // Barrier
    }

    if (tid == 0)
       output[blockIdx.x] = sdata[0];
}
```

CUDA additional capabilities

- Direct access to Tensor Cores, accelerate AI training and inference.
- Warp-level primitives (warp shuffles, warp ballots) for fine-tuned thread coordination.
- Shared memory optimizations tightly matched to NVIDIA's SM architecture.
- highly-optimized GPU libraries tied to CUDA:
  - cuDNN (deep learning primitives)
  - TensorRT (inference engine)
  - cuBLAS, cuFFT, cuSPARSE, cuSOLVER, etc.
- CUDA's advantage in "warp-level primitives, shared memory tricks, and specialized kernels," which OpenCL cannot match due to its vendor-neutral design.

NVIDIA Tools exclusive to CUDA

- Nsight Compute (kernel profiler)
- Nsight Systems (system-level profiler)
- CUDA-GDB, CUDA-MEMCHECK
- Extremely mature compiler toolchain (NVCC)



SYCL

VRIJE UNIVERSITEIT BRUSSEL

SYCL

- SYCL = modern C++ abstraction layer over heterogeneous hardware
- Standardized by the Khronos Group (OpenCL, OpenGL, ...)
- Device code is specified in source code with advanced C++ constructs (Lambdas)
- Which SYCL implementation to use: DPC++

VRIJE UNIVERSITEIT BRUSSEL

Example SYCL program

```
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
    queue q;

    q.submit([&](handler& h){
        h.single_task([=](){
            printf("Hello from device\n");
        });
    }).wait();
}
```

VRIJE UNIVERSITEIT BRUSSEL

Device Info Query

```
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;

int main(){
    for (auto dev : device::get_devices()) {
        std::cout << dev.get_info<info::device::name>() << "\n";
        std::cout << "Compute Units: "
                  << dev.get_info<info::device::max_compute_units>() << "\n";
        std::cout << "Global mem: "
                  << dev.get_info<info::device::global_mem_size>()/1e6
                  << " MB\n\n";
    }
}
```

VRIJE UNIVERSITEIT BRUSSEL

Vector Add Kernel

```
#include <sycl/sycl.hpp>
using namespace sycl;

int main(){
    const int N = 1024;
    queue q;

    float *a = malloc_shared<float>(N,q);
    float *b = malloc_shared<float>(N,q);
    float *c = malloc_shared<float>(N,q);

    for(int i=0;i<N;i++){
        a[i]=i;
        b[i]=2*i;
    }

    q.parallel_for(N,[=](id<1> i){
        c[i]=a[i]+b[i];
    }).wait();
}
```

VRIJE UNIVERSITEIT BRUSSEL

Matrix Multiplication

```
q.parallel_for(nd_range<2>({N,N},{T,T}),
[=](nd_item<2> it){

    int row = it.get_global_id(0);
    int col = it.get_global_id(1);

    float sum = 0;

    for(int k=0;k<N;k++)
        sum += A[row*N+k] * B[k*N+col];

    C[row*N+col] = sum;
});
```

Numba

VRIJE UNIVERSITEIT BRUSSEL

Numba

- Host & GPU code in Python
- Based on CUDA

Example

```
import numpy as np
from numba import cuda

# 1. Define a CUDA kernel
@cuda.jit
def vector_add(a, b, c):
    idx = cuda.grid(1)
     # 1D index for current thread
    if idx < a.size:
        c[idx] = a[idx] + b[idx]

# 2. Prepare input data on the host
N = 32
a = np.arange(N, dtype=np.float32)
b = np.arange(N, dtype=np.float32)
c = np.zeros(N, dtype=np.float32)

# 3. Transfer data to the GPU
d_a = cuda.to_device(a)
d_b = cuda.to_device(b)
d_c = cuda.device_array_like(c)
```

```
# 4. Configure the kernel launch parameters
threads_per_block = 16
// threads_per_block:
blocks_per_grid = (N + threads_per_block - 1)


# 5. Launch the kernel
vector_add[blocks_per_grid, threads_per_block](d_a, d_b, d_c)

# 6. Copy result back to host
c = d_c.copy_to_host()
```

Vulkan

VRIJE UNIVERSITEIT BRUSSEL

Vulkan Compute

- A general-purpose GPU compute model built on top of Vulkan's shader programming via SPIR-V.
  - **Low-level**: gives you explicit control over memory, synchronization, and hardware queues.
  - **Cross-vendor**: runs on NVIDIA, AMD, Intel, Qualcomm, Apple (via MoltenVK), etc.
  - **Cross-platform**: Windows, Linux, Android, macOS (via MoltenVK).
  - **Unified**: compute and graphics in the same API.
- Based on the same concepts as OpenCL/CUDA: work-item -> Invocation, Workgroup, ...

VRIJE UNIVERSITEIT BRUSSEL

Vulkan Compute

- ✅ Gives explicit control over the GPU
- ✅ Is vendor-neutral and cross-platform
- ✅ Integrates compute + graphics seamlessly
- ✅ Is lower-level than both CUDA and OpenCL
- ✅ Requires more boilerplate but offers maximum flexibility
- Much more boilerplate
- Memory management is manual
- Pipelines require more setup than CUDA kernels
- Debugging is harder
- Compute-only tasks might be simpler in CUDA/OpenCL

Tensorflow & Pytorch for AI

VRIJE UNIVERSITEIT BRUSSEL

PyTorch

- open-source machine learning and deep learning framework
  - build, train, and deploy neural networks using a Python-first, user-friendly approach
  - core data structure is the **tensor**, similar to NumPy arrays but with built-in GPU acceleration.
  - its dynamic computation graph ("define-by-run"), which makes model building intuitive and highly flexible
  - Rich Neural Network Library (torch.nn)
- developed by Meta, now Linux Foundation

VRIJE UNIVERSITEIT BRUSSEL

Tensorflow (from Google)

- Structured, optimized for large-scale deployment
- Excels in large-scale distributed training, especially with TPUs.

- Use PyTorch if you prioritize flexibility, research speed, debugging ease, or experimental AI.
- Use TensorFlow if you need scalable deployment, mature production tooling, multi-platform support, or enterprise-grade pipelines.

VRIJE UNIVERSITEIT BRUSSEL

Keras

- The official high-level API of TensorFlow, accessible as tf.keras.
- Gives developers the full power of TensorFlow (TPUs, GPUs, distributed training) while working with a simpler interface
- Keras 3 supports multiple backends:
  - TensorFlow
  - PyTorch
  - JAX

Triton

VRIJE UNIVERSITEIT BRUSSEL

Custom Neural network kernel

In Pytorch, options:

1. CUDA with a C++ wrapper
2. Triton
3. CuPy / Numba

VRIJE UNIVERSITEIT BRUSSEL

Example with CUDA

- Step 1 — Write the CUDA kernel (relu_kernel.cu)
- Step 2 — Create a C++ wrapper (relu.cpp)
- Step 3 — Python binding & build (build.py)
- Step 4 — Use it in PyTorch

See code in “GPU Computing – Lesson 6 – Alternative Programming Methods – example code.pdf”

VRIJE UNIVERSITEIT BRUSSEL

Example with Triton

1. ReLU in Triton
2. Python wrapper
3. Usage

See code in “GPU Computing – Lesson 6 – Alternative Programming Methods – example code.pdf”

ROCm & HIP

VRIJE UNIVERSITEIT BRUSSEL

ROCm

- ROCm was created because AMD needed:
  - a more modern, CUDA-competitive stack,
  - better performance,
  - deeper control over GPU kernels and memory,
  - better compatibility with ML frameworks (PyTorch, TensorFlow).
- ROCm promotes HIP, a CUDA-like API designed for high-performance GPU programming

HIP

- HIP provides performance-oriented GPU kernel programming.
- It enables CUDA portability, letting developers port high-performance CUDA kernels to AMD GPUs with minimal changes.
- ROCm focuses on HIP as its main programming model, whereas OpenCL is only one optional part of the stack.

VRIJE UNIVERSITEIT BRUSSEL

ROCm for AI

- ROCm includes specialized **high-performance libraries** for machine learning, linear algebra, vision
- OpenCL does not provide these libraries. It is a thin, generic compute API.
- This means ROCm workloads use optimized kernels tuned for AMD GPUs, whereas OpenCL users must hand-optimize kernels or rely on vendor-specific extensions.
- PyTorch now ships official ROCm builds and supports AMD GPUs on consumer cards in 2026.

VRIJE UNIVERSITEIT BRUSSEL

Conclusions

⇒ **"OpenCL is dead"...**

- OpenCL failed to provide a widely accepted generic approach for accelerators

Explanations:

- A generic approach cannot achieve optimal performance on highly advanced parallel processors...

  => Specific hardware-specific languages win (CUDA, ROCm)
- The need for highly optimized libraries
  - NVIDIA invested a lot in this since the very beginning of General Purpose GPUs
- Companies protect their commercial interests
- ...

Luckily, all programming methods rely on the same concepts taught in this course