Heterogeneous Computing (GPGPU)

INF8601 – Systèmes parallèles
Automne 2015

Francis Giraldeau
francis.giraldeau@polymtl.ca

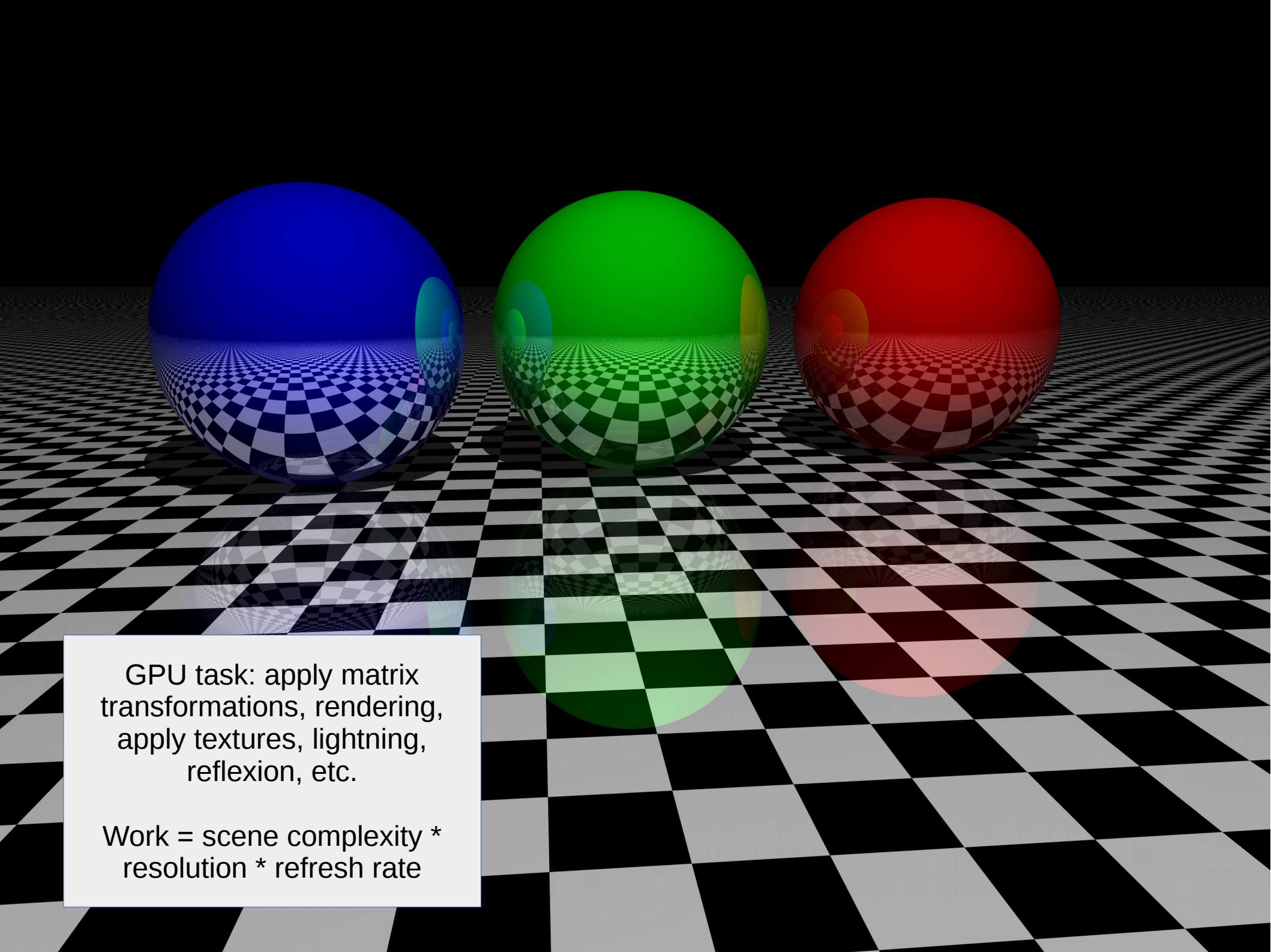
Prof. Michel Dagenais
michel.dagenais@polymtl.ca

École Polytechnique de Montréal

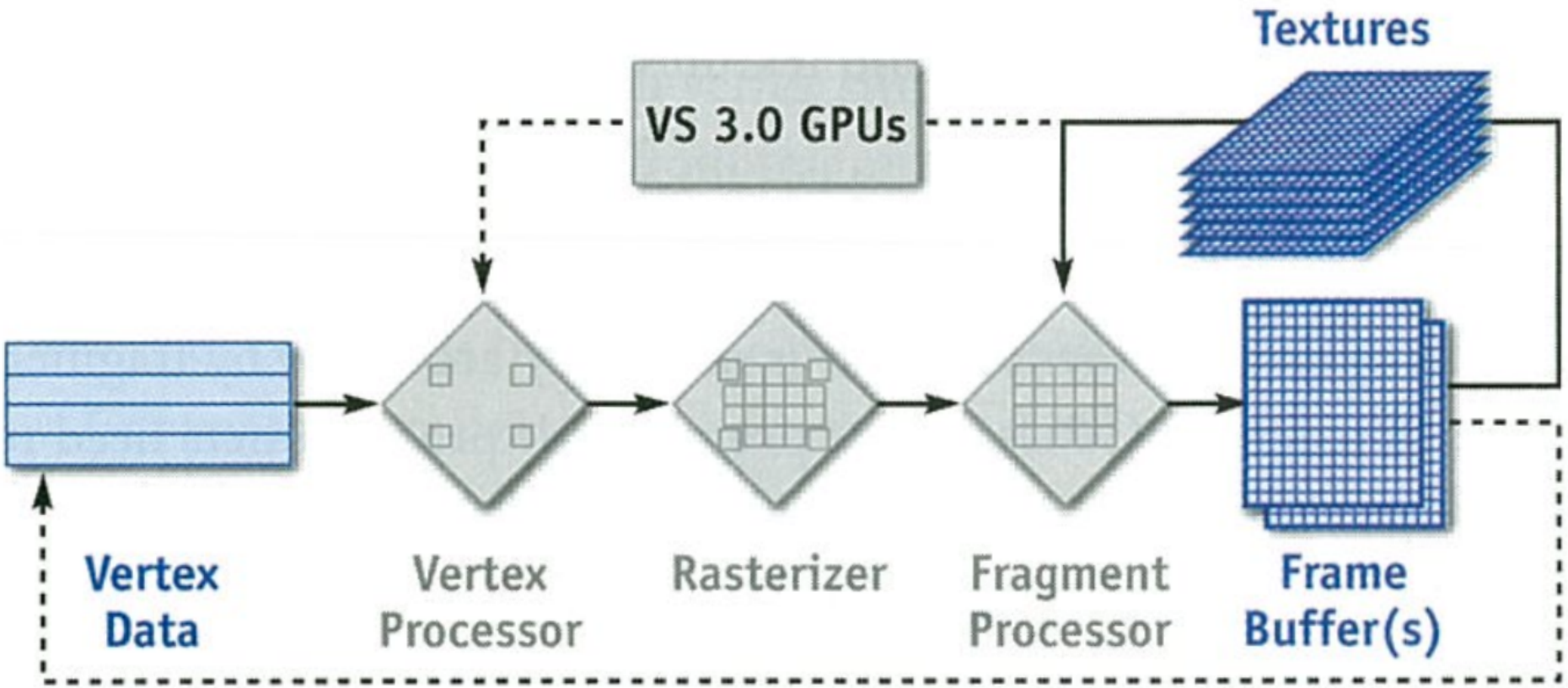
POLYTECHNIQUE MONTRÉAL

Plan

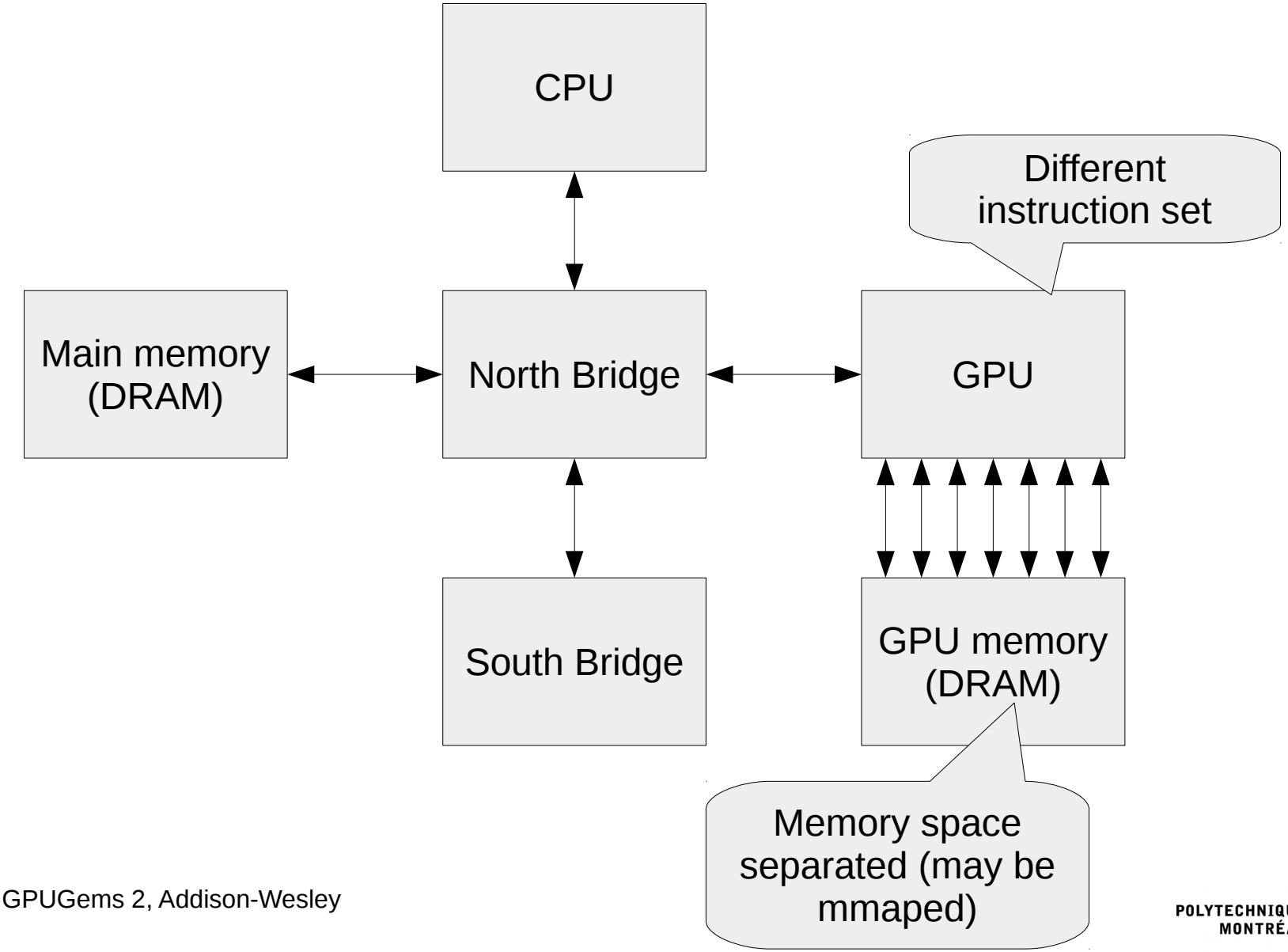
- Principle of operation
- Architecture
- Using OpenCL
- Preview of HCC


GPU task: apply matrix transformations, rendering, apply textures, lightning, reflexion, etc.
Work = scene complexity * resolution * refresh rate

Graphic pipeline



Principle of operation



Source: GPUGems 2, Addison-Wesley

Principle of operation

CPU

Central Processing Unit

```
void saxpy_cpu(float *y,
               float *x,
               float a,
               int n)
{
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

GPGPU

General-Purpose processing on Graphics Processing Units

Intended to be compiled and running on the GPU

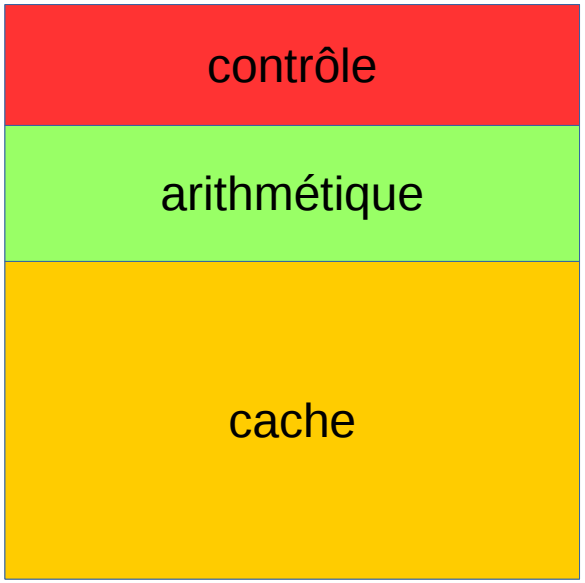
```
__kernel
void saxpy_gpgpu(__global float *y,
                 __global float *x,
                 float a)
{
    int i = get_global_id(0);
    y[i] = a * x[i] + y[i];
}
```

Implicit loop iteration: The function is called for each index

Architecture difference

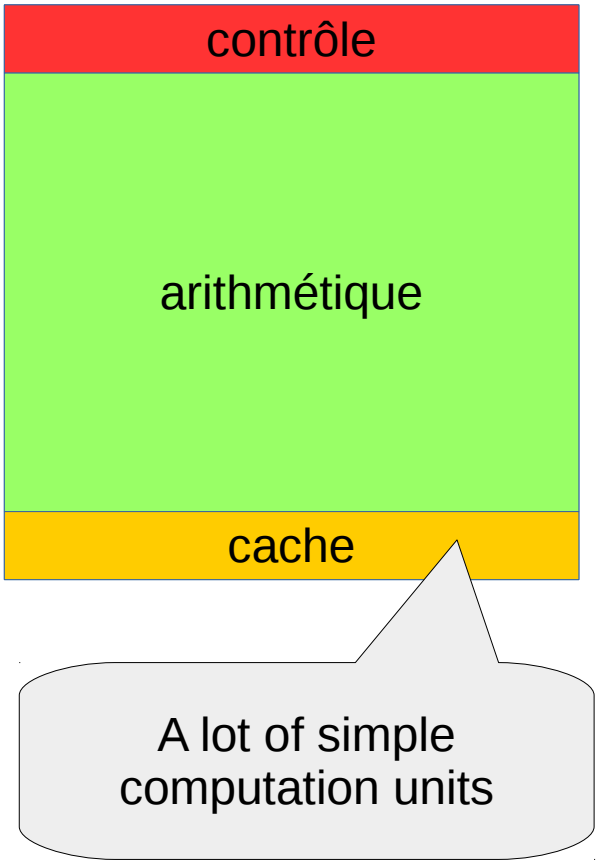
CPU

Central Processing Unit



GPGPU

General-Purpose processing on Graphics Processing Units



Architecture difference

CPU

Central Processing Unit

- Intel i7
- 4-8 coeurs
- 1-2 GFLOPS
- 25 Go/s DRAM

GPGPU

General-Purpose processing on Graphics Processing Units

- NVIDIA GTX 980
- 2048 coeurs
- 5000 GFLOPS
- 224 Go/s DRAM

http://ark.intel.com/fr/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz
http://www.nvidia.fr/object/geforce-gtx-980-fr.html

Execution model

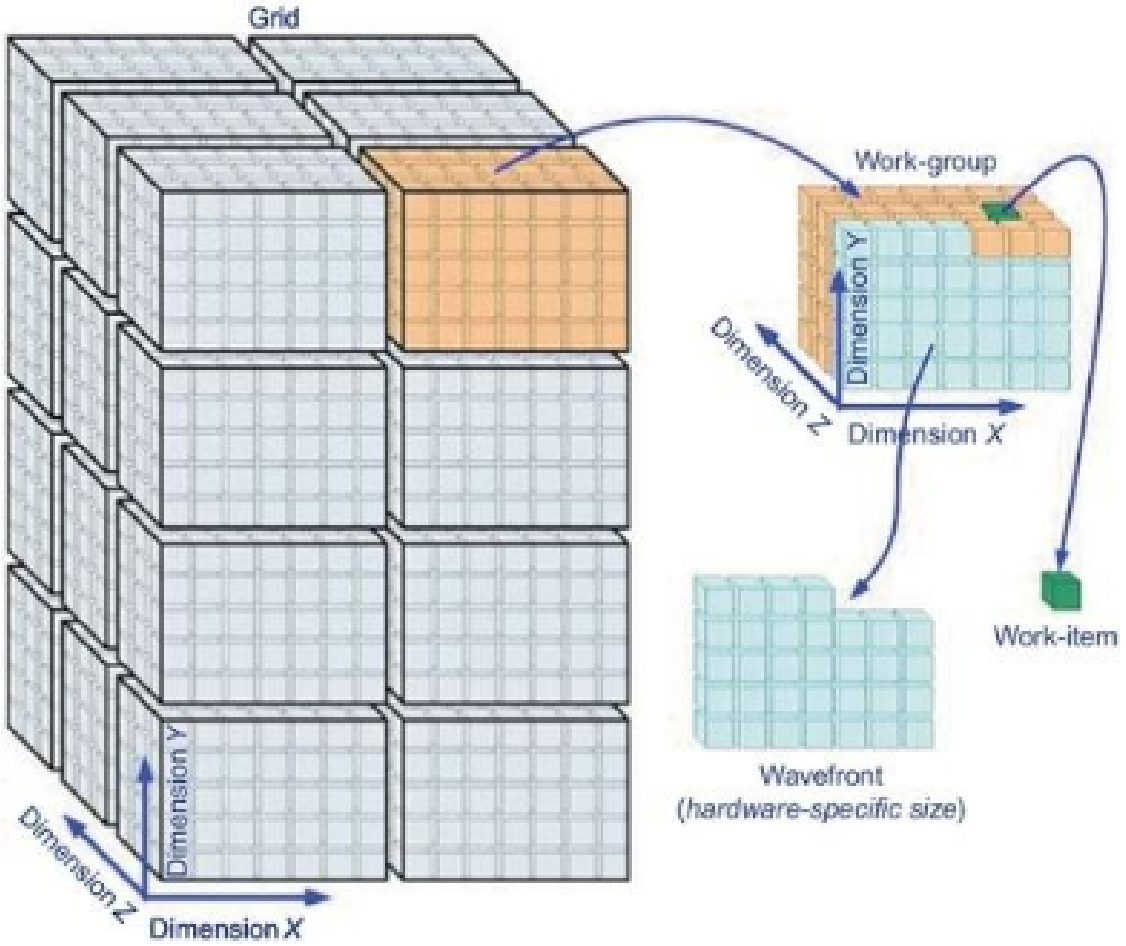


FIGURE 3.2 An HSA grid and its work-groups and work-items.

Source: Heterogeneous Computing with OpenCL 2.0

Memory model

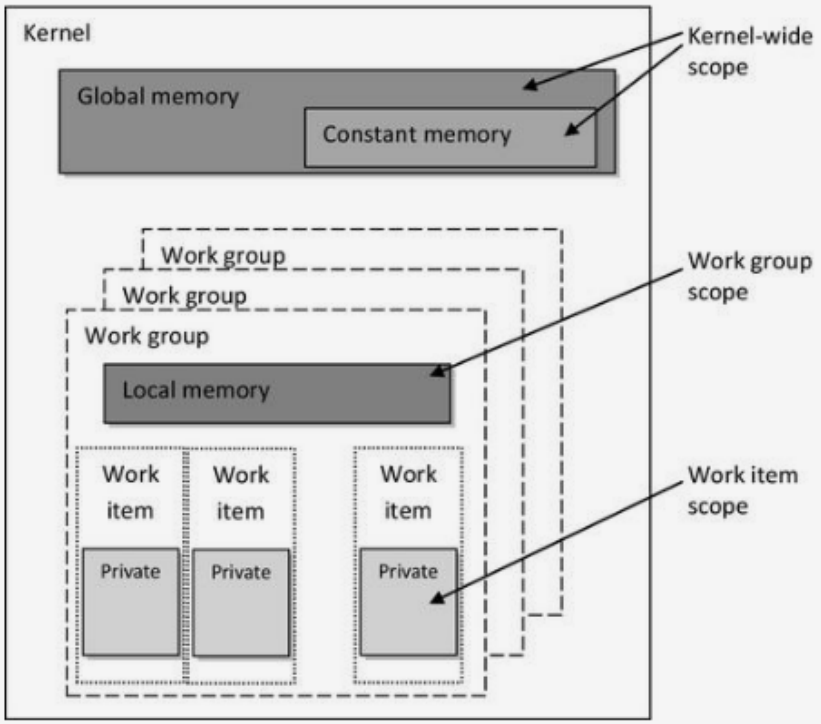


FIGURE 3.6 Memory regions and their scope in the OpenCL memory model.

- Global memory is used to copy to/from host
- Constant memory can be cached on vector processor and reduce memory bandwith
- Local memory: shared between workers of the same group
- Private memory: stack

Source: Heterogeneous Computing with OpenCL 2.0

Mapping memory model

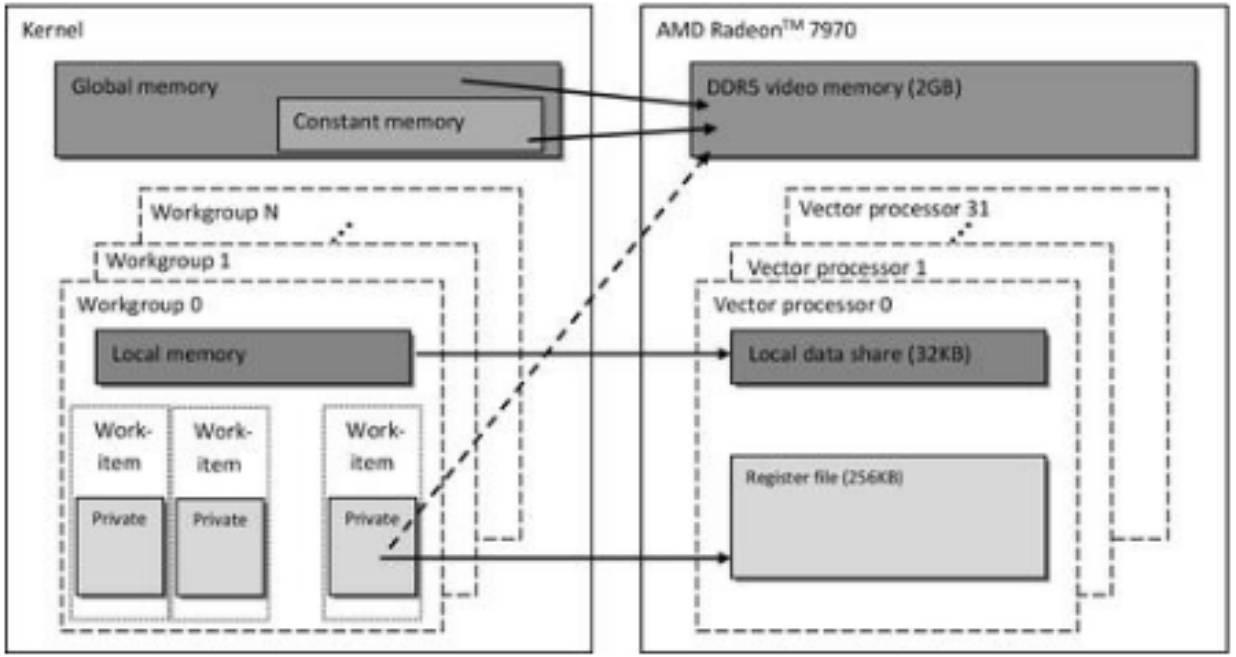


FIGURE 3.7 Mapping the OpenCL memory model to an AMD Radeon HD 7970 GPU.

Source: Heterogeneous Computing with OpenCL 2.0

GPGPU technologies

- CUDA
  - nvcc compiles C++ programs with embedded kernel code, manages kernel launch and device selection
  - NVIDIA specific
- OpenCL
  - Portable (AMD, NVIDIA, Intel), CPU fallback
  - Explicit device, kernel and memory management
- HSA: Heterogeneous System Architecture
  - Portable specification and open source implementation for abstracting heterogeneous devices

CUDA example

```
/*
 * Summing vectors C = A + B
 *
 * Source: CUDA by example, An Introduction to
 * General-Purpose GPU Programming
 */

#define N 10

__global__ void add(int *a, int *b, int *c)
{
    // built-in kernel variable
    int tid = blockIdx.x;
    if (tid < N) {
        c[tid] = a[tid] + b[tid];
    }
}
```

```
int main()
{
    int a[N], b[N], c[N];

    int *dev_a, *dev_b, *dev_c;
    int buf_size = N * sizeof(int);

    // Allocate memory on the device
    cudaMalloc(&dev_a, buf_size);
    cudaMalloc(&dev_b, buf_size);
    cudaMalloc(&dev_c, buf_size);

    // Init input data
    for (int i = 0; i < N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    // Copy input data to the device
    cudaMemcpy(dev_a, a, buf_size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, buf_size, cudaMemcpyHostToDevice);

    // call the kernel (nvcc compiler specific)
    add<<<N,1>>>(dev_a, dev_b, dev_c);

    // Copy the result to the host
    cudaMemcpy(c, dev_c, buf_size, cudaMemcpyDeviceToHost);

    return 0;
}
```

OpenCL example

```
const char *kernel_text = MULTI_LINE_STRING(
    __kernel void SAXPY (__global float* x,
                         __global float* y,
                         __global float* z,
                         float a) {
        const int i = get_global_id (0);
        z[i] = a * x[i] + y[i];
    }
);
```

```
/*
 * Get available platforms
 */
cl::Platform::get(&m_platforms);

/*
 * Get device info for each platforms
 */
for(const auto platform: m_platforms) {
    std::vector<cl::Device> devices;
    platform.getDevices(CL_DEVICE_TYPE_ALL, &devices);
    m_devices.insert(m_devices.end(), devices.begin(), devices.end());
}
```

OpenCL example

```
/* Create context that will group all related operations on a device */
m_ctx = cl::Context(dev);

/* Load the program (does not compile it yet) */
std::string text(kernel_text);
m_prog = cl::Program(m_ctx, text);

/* Compile the program */
if (m_prog.build({dev}) != CL_SUCCESS) {
    throw std::runtime_error("Failed to compile the OpenCL program");
}

/* Obtain a reference on the main kernel (function annotated with __kernel attribute) */
m_kernel = cl::Kernel(m_prog, "SAXPY");

/* Create the queue for the commands. They are executed in FIFO order. */
m_queue     = cl::CommandQueue(m_ctx, dev);
```

```
/* la taille en octets des tampons, requis pour allouer les tampons sur le périphérique */
size_t size = x.size() * sizeof(float);

/*
 * Create buffers on the device. The operations CL_MEM_READ/CL_MEM_WRITE
 * are from the device point of view. The kernel reads cl_x and cl_y and write to cl_z
 */
cl::Buffer cl_x(m_ctx, CL_MEM_READ_ONLY, size);
cl::Buffer cl_y(m_ctx, CL_MEM_READ_ONLY, size);
cl::Buffer cl_z(m_ctx, CL_MEM_WRITE_ONLY, size);

/*
 * Copy data from the host to the device. The READ/WRITE operations are from the host
 * point of view. For instance, the input vector x is copied to the device in cl_x.
 */
m_queue.enqueueWriteBuffer(cl_x, CL_TRUE, 0, size, x.data());
m_queue.enqueueWriteBuffer(cl_y, CL_TRUE, 0, size, y.data());

/* Set kernel arguments */
m_kernel.setArg(0, cl_x);
m_kernel.setArg(1, cl_y);
m_kernel.setArg(2, cl_z);
m_kernel.setArg(3, a);

/*
 * Call the kernel. cl::NDRange() indicates the size for all dimensions. Here,
 * there is only one dimension and the kernel obtains it by using get_global_id(0)
 */
m_queue.enqueueNDRangeKernel(m_kernel, cl::NullRange, cl::NDRange(x.size()));
m_queue.finish();

/* Fetch result (copy from the device to the host) */
m_queue.enqueueReadBuffer(cl_z, CL_TRUE, 0, size, z.data());
```

Heterogeneous Compute Compiler (HCC)

- Supported languages
  - HC C++ API
  - HIP (CUDA converter)
  - Microsoft C++ AMP
  - C++ Parallel STL
  - OpenMP on the CPU (will support GPU offload in the future)
- Backend
  - Native Graphic Core Next (GCN) Instruction Set Architecture (ISA)
  - HSAIL backend

HC C++ API example

```
const float a = 100.0f;
float x[N];
float y[N];


// make a copy of for the GPU implementation
float y_gpu[N];
std::copy_n(y, N, y_gpu);



// wrap the data buffer around with an array_view
// to let the hcc runtime to manage the data transfer
hc::array_view<float, 1> av_x(N, x);
hc::array_view<float, 1> av_y(N, y_gpu);

// launch a GPU kernel to compute the saxpy in parallel
hc::parallel_for_each(hc::extent<1>(N)
                      , [=](hc::index<1> i) [[hc]] {
  av_y[i] = a * av_x[i] + av_y[i];
});
```

HSAIL example

Heterogeneous **S**ystem **A**rchitecture **I**ntermediate **L**anguage

```
module &m:1:0:$base:$large:$default;

decl prog function &abort()();

prog kernel &__vector_copy_kernel(
    kernarg_u64 %in,
    kernarg_u64 %out)
{
@__vector_copy_kernel_entry:
    // BB#0:                                // %entry
    workitemabsid_u32   $s0, 0;
    cvt_s64_s32 $d0, $s0;
    shl_u64 $d0, $d0, 2;
    ld_kernarg_align(8)_width(all)_u64  $d1, [%out];
    add_u64 $d1, $d1, $d0;
    ld_kernarg_align(8)_width(all)_u64  $d2, [%in];
    add_u64 $d0, $d2, $d0;
    ld_global_u32   $s0, [$d0];
    st_global_u32   $s0, [$d1];
    ret;
};
```

- Cross platform assembly-like
- Compiled to binary intermediate representation (BRIG), embedded into executables
- Finalizer: compile at runtime for the target device

Source: /opt/rocm/hsa/sample/vector_copy_base.hsail

Activity

- Demo of sinoscope
- Copy saxpy-opencl and modify the algorithm to compute an histogram of char values
  - Hint: use atomic_inc()
- Make two benchmarks:
  - Time to launch the kernel with empty data
  - Time to compute large data (1GB)
- Measure the effective speedup compared to computing the histogram on the host