Wigner GPU Lab

Introduction to GPU Programming

The transition from multi-core to many-core computing

Lectures on Modern Scientific Programming
Wigner RCP
23-25 November 2015

D. Berényi – M. F. Nagy-Egri

IN PARALLEL COMPUTING, AN EMBARRASSINGLY PARALLEL WORKLOAD, OR EMBARRASSINGLY PARALLEL PROBLEM, IS ONE FOR WHICH LITTLE OR NO EFFORT IS REQUIRED TO SEPARATE THE PROBLEM INTO A NUMBER OF PARALLEL TASKS.

Unknown source

Because in computer graphics, the elements of a scene can be transformed onto the screen (mostly) independently, computer graphics is often referred to as an embarrassingly parallel problem. The series of transformations is (mostly) identical for all elements, therefor it is also a data parallel problem.

What is the Fundamental difference?

Wigner GPU Lab

CPU

- General purpose hardware
- Latency optimized core (LOC)
- Generic/task parallel processing unit
- Fixed instruction set architecture (ISA)

GPU

- Dedicated hardware
- Throughput optimized core (TOC)
- Data parallel processing unit
- Virtual instruction set architecture (VISA)

intel

NVIDIA

AMD

How can these be achieved?

CPU

- Tune for instructions per cycle (IPC)
- Large cache sizes so data is always near the registers
- Complicated instruction arbitration (peek ahead, branch prediction, etc.)
- Monolithic instruction set

GPU

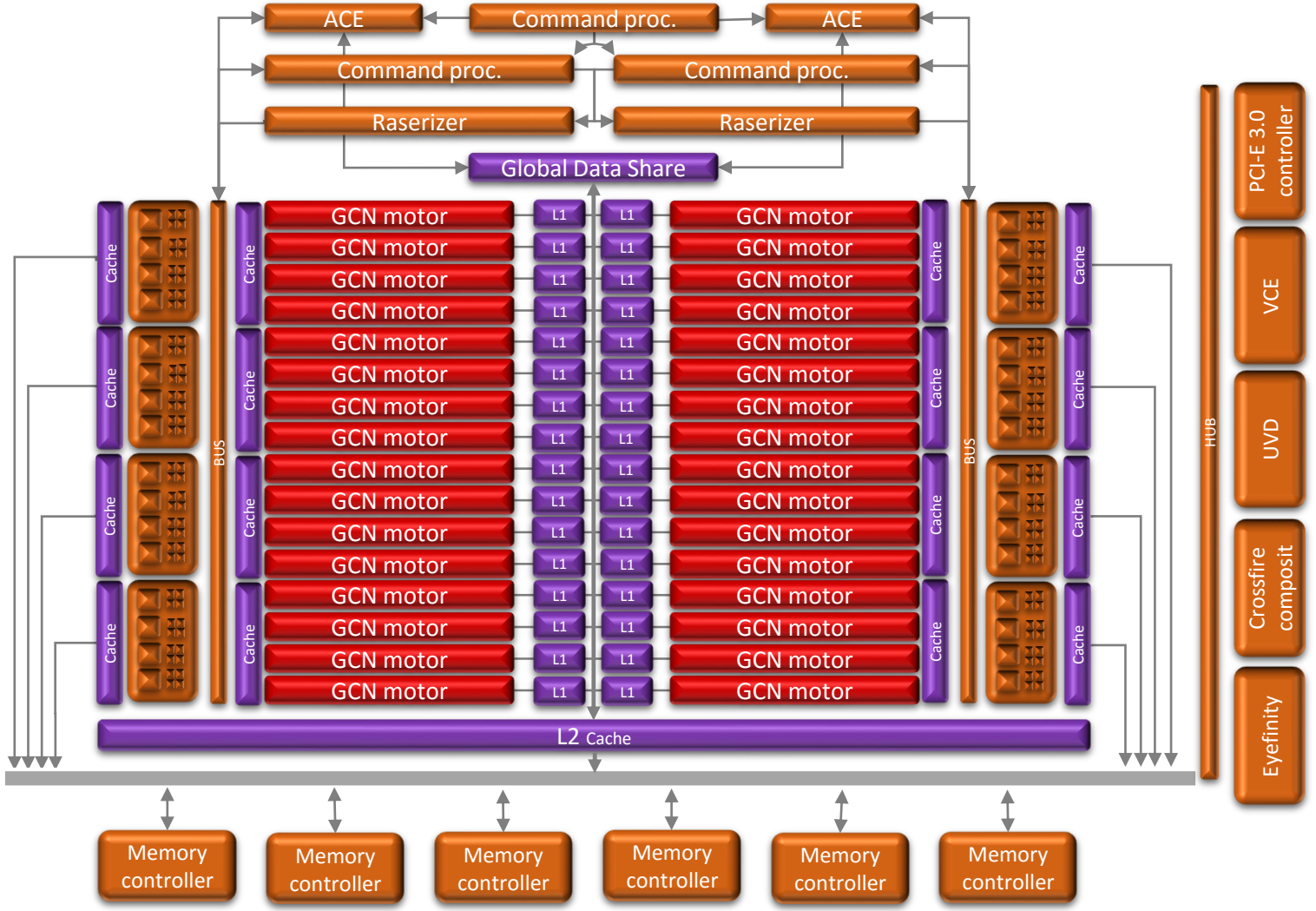
- Tune for reduced transistor and energy footprint
- Small cache sizes with latency hiding
- Simple instruction arbitration (lock-step execution, reduced branching…)
- Reduced instruction set

Just how massively parallel is it?

- Take a fairly modern consumer GPU: AMD Radeon R9 Fury X
  - Core clock: 1050 MHz
  - GCN engine count: 64
  - Stream processing units: 4096
  - Global Memory bandwidth: 512 GB/s
  - Local Memory Bandwidth: 7578 GB/s
- What does that mean to a programmer?
  - Minimum in-flight threads: 4*4096=16384
  - Peak SPFP: 8,6 GFLOPS
  - Peak DPFP: 2,15 GFLOPS

AMD Grphics Core Next architecture 1.0

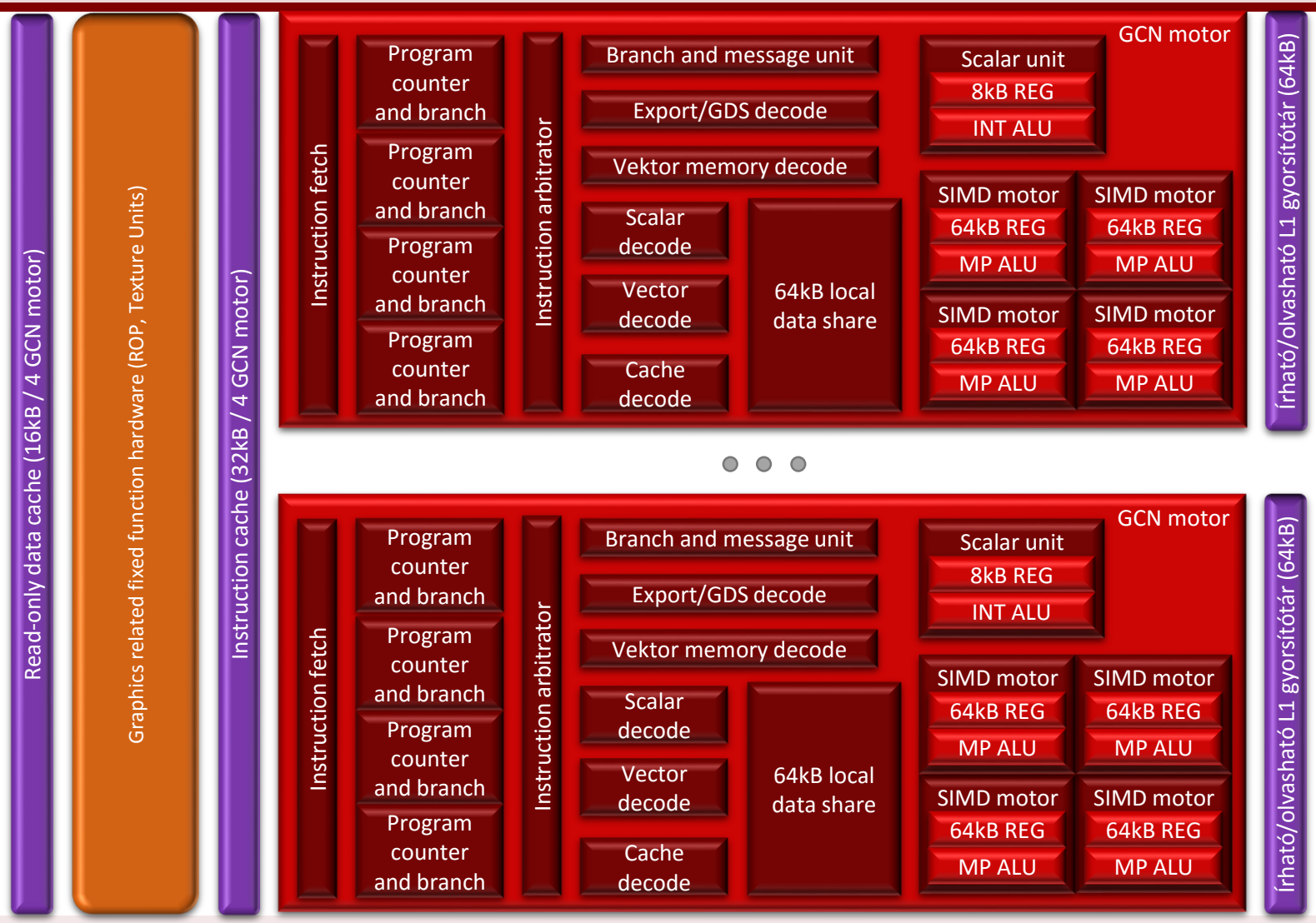
- Highly redundant design
  - Resistant to manufacturing flaws
  - Design once, scale across whole product line
- 3 major parts
  - Setup engine
  - Execution engines
  - Unique fixed function
- Compared to a CPU
  - Hardware thread scheduler (!)
  - Similar cache hierarhcy
  - Multiple memory controllers



AMD Tahiti (Radeon HD7970) schematics

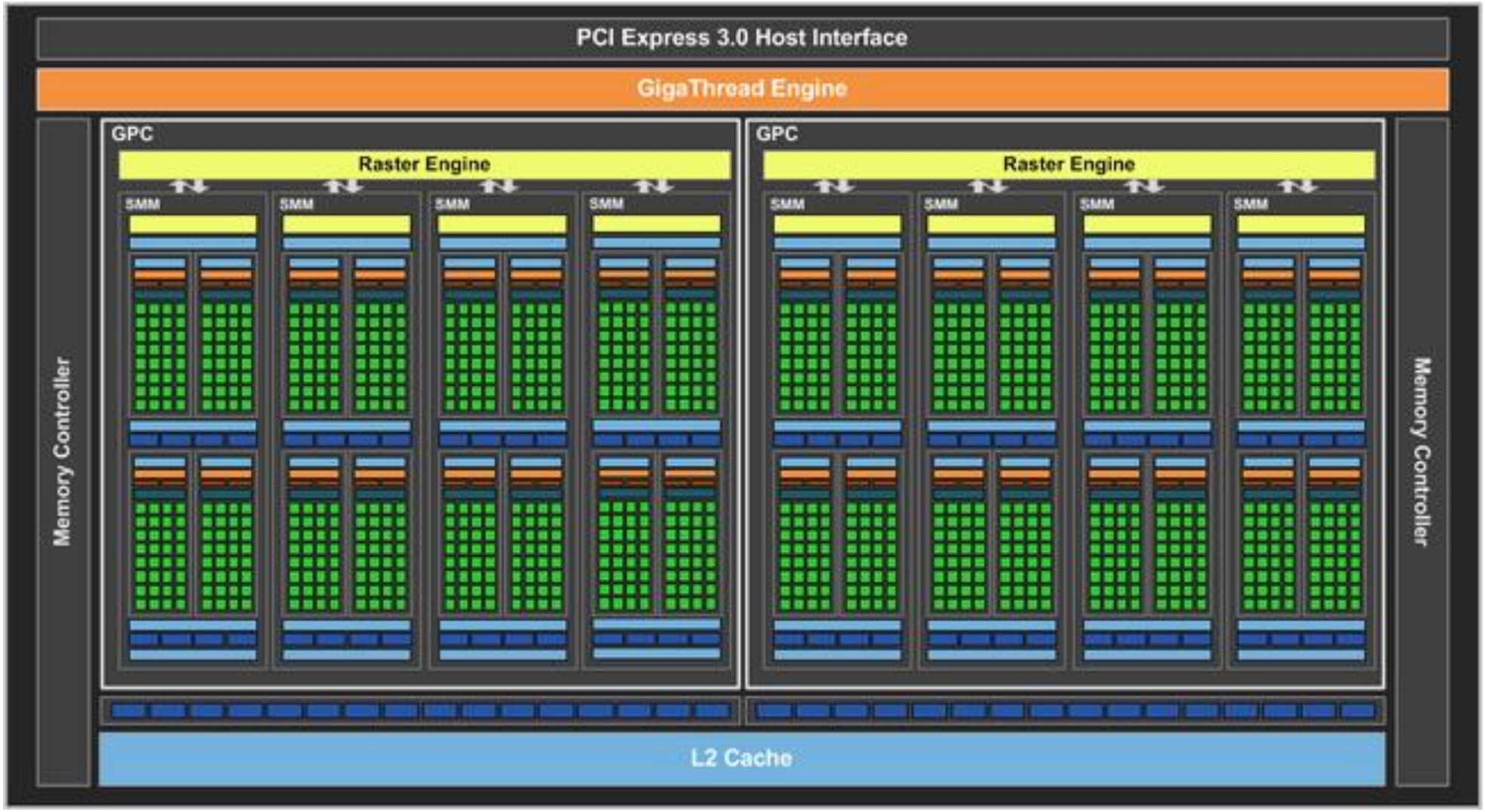
AMD GCN Compute Unit details

- How much a CPU?
  - Overall design is very similar
- How does it differ?
  - Tune for ALU
  - Tune for throughput
  - Extremely small cache sizes
  - Orbital amounts of registers

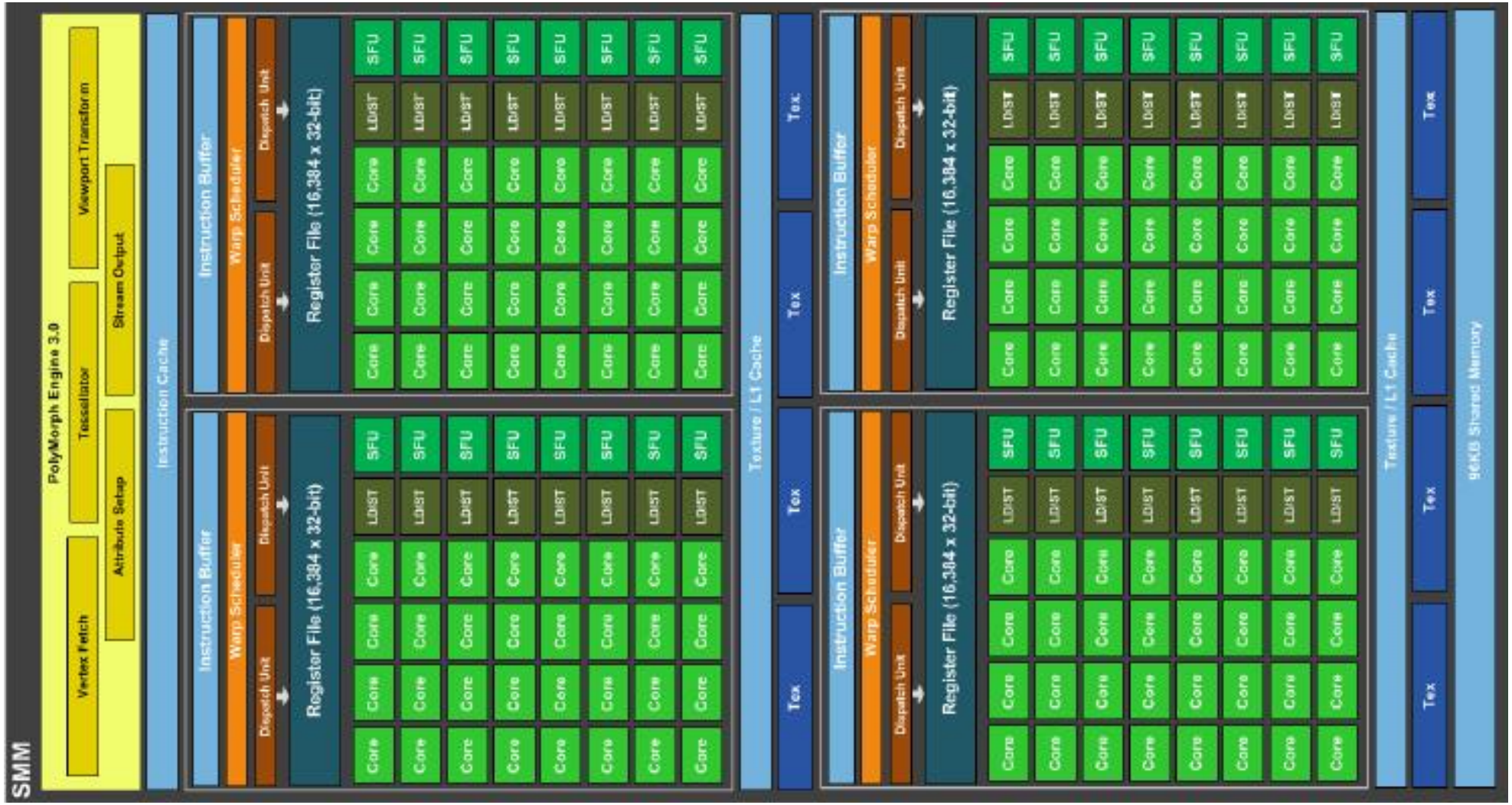


AMD GCN 1.0 Compute Unit schematics

Nvidia Maxwell Architecture



Nvidia SMM Compute Unit details



Timings on a GPU

INT Instruction	Throughput
Shift / Rot	~1
AND / OR / XOR	~1
Compare/test	~1
ADD/SUB	~1
MUL	~4
FMA/MAD	~4
DIV	~40

SFPF Instruction	Throughput
ADD/SUB	~1
MUL	~1
FMA/MAD	~1
INV	~4
LOG	~4
RSQRT	~4
NCOS/NSIN	~10
SIN/COS	~100-200
TAN	~200-300

DFPF Instruction	Throughput
ADD/SUB	~2-24
MUL	~4-48
FMA/MAD	~4-48
INV	~4-48
LOG	~4-48
RSQRT	~4-48
NCOS/NSIN	X
SIN/COS	~200-300
TAN	~300-400

But where's the catch?

- Memory access and branching is critical
  - Most GPU algorithms are memory bandwidth limited
  - The hardware is sensitive to divergence
- Some things are highly vendor/architecture dependent
  - Special hardware can sometimes accelerate operations
  - Overall design impacts behavior

Memory operation	Throughput (NV)	Throughput (AMD)
Register access	~1	~4
Local memory access	~1	~40
Atomic local access	~10-40	~100
Global memory access	~400-1000	~400-1000
Atomic global access	~1000-2000	~200-2000

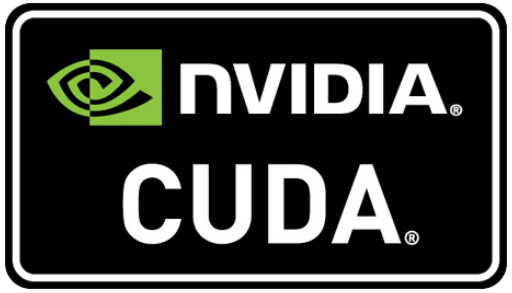
Control flow	Throughput (NV)	Throughput (AMD)
Branching	~20+IF&ELSE	~5+IF&ELSE
Select operator	~20+IF&ELSE	~MIN(IF&ELSE)

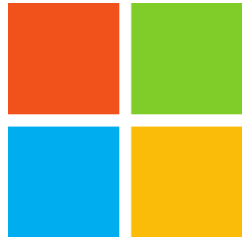
Wigner GPU Lab

Introducing The API Hell

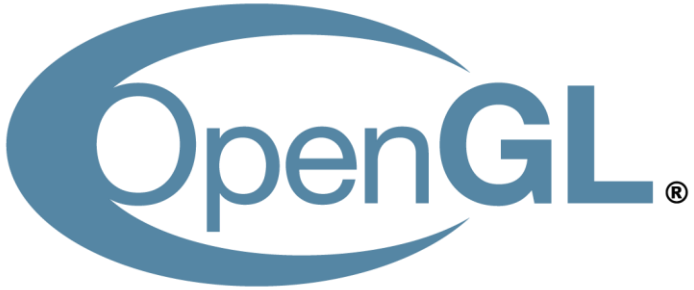
Finding the weapon of choice

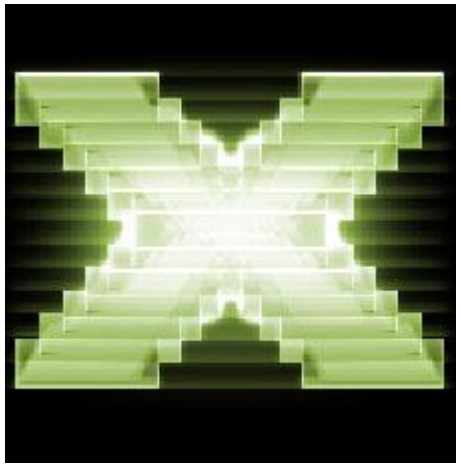
D. Berényi – M. F. Nagy-Egri





C++AMP





OpenACC
Directives for Accelerators

OpenMP

intel

NVIDIA

AMD

Compute Unified Device Architecture

- Invented by Nvidia Corporation to empower developers with GPGPU usage free of the graphics pipeline
- First version released in 2007
- Two levels
  - C driver API
  - C++98 language extension where special operators and decorators appear to enable singe-source GPGPU coding (starting from CUDA 7.0 partial C++11)
- Closed source technology, it is always tuned to achieve maximum performance on Nvidia hardware

Open Computing Language

- An initiative started by Apple Incorporation as an answer to the success of CUDA, howver handling of the standard was handed over to the Khronos Consortion
- Its first release was in 2009
- Seperate host and device side languages and APIs
  - C API for the host-side code
  - C-like language for the device kernels (starting from OpenCL 2.1 static C++14)
- Open technology, anyone can implement it

DirectX Compute Shader

Wigner GPU Lab

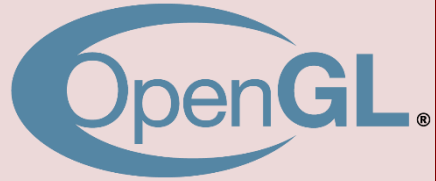
- It is the compute derivate of Microsoft's 3D API, Direct3D
- Debuted in DirectX11 in 2009
- It interfaces with the graphics pipeline in a trivial manner
  - The host side C++ API inherits much of the graphics API
  - Device-side language is HLSL (High-level Shading Language)
- DirectX 12 provides far more explicit access to the hardware

intel

NVIDIA

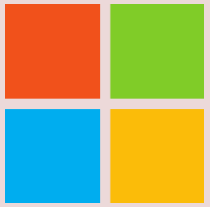
AMD

Open Grahpics Library



- The open version of MicroSoft's DirectX, Khronos standard
- Released in 2012 along with OpenGL 4.3
- It interfaces with the graphics pipeline in a trivial manner
  - The host side C API inherits much of the graphics API
  - Device-side language is GLSL (OpenGL Shading Language)
- Interoperability with OpenCL

C++ Accelerated Massive Parallelism



Wigner GPU Lab

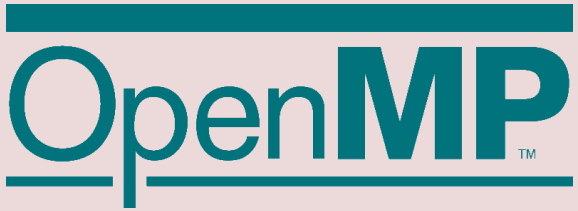
- It is a concept demonstration of standardizing and integrating singe-source GPGPU into C++
- Released in 2012, open to implementation
- C++ language extension that separates host and device-side code using an early version of concepts (C++17 feature)
- Does not specify the Intermediate Language of device-side code

intel

NVIDIA

AMD

Open Multi-Processing



- Maintained by a stand-alone consortion (OpenMP Architecture Review Board). Originally meant for CPU parallel development.
- Starting with version 4.0 from 2013 support offloading parallel work to devices.
- Open standard
- Supports C, C++, Fortran languages through (#pragma) directives

Open Accelerators

OpenACC
Directives for Accelerators

Wigner GPU Lab

- Nvidia initiative for an OpenMP-like directive controlled C, C++, fortran parallel API
- Open standard
- Work in 2013 was started to unify the 2 APIs that might debut in a coming OpenMP standard
- Its newest 2.0 version is already implemented in GCC 5

intel
NVIDIA
AMD

SYCL

- A C++ template library developed along the criticism OpenCL recieved from developers
- It is still in beta (application for academic evaluation is free), public release anticipated Q4 2015
- The host and device side code separate clearly only through library means (no language extensions).
- The means of compilation is not specified, but the intermediate representation is
  - Under the hood, it relies on conforming OpenCL implementations to be present

Open Computing Language

The best for learning GPGPU

D. Berényi – M. F. Nagy-Egri

Why OpenCL?

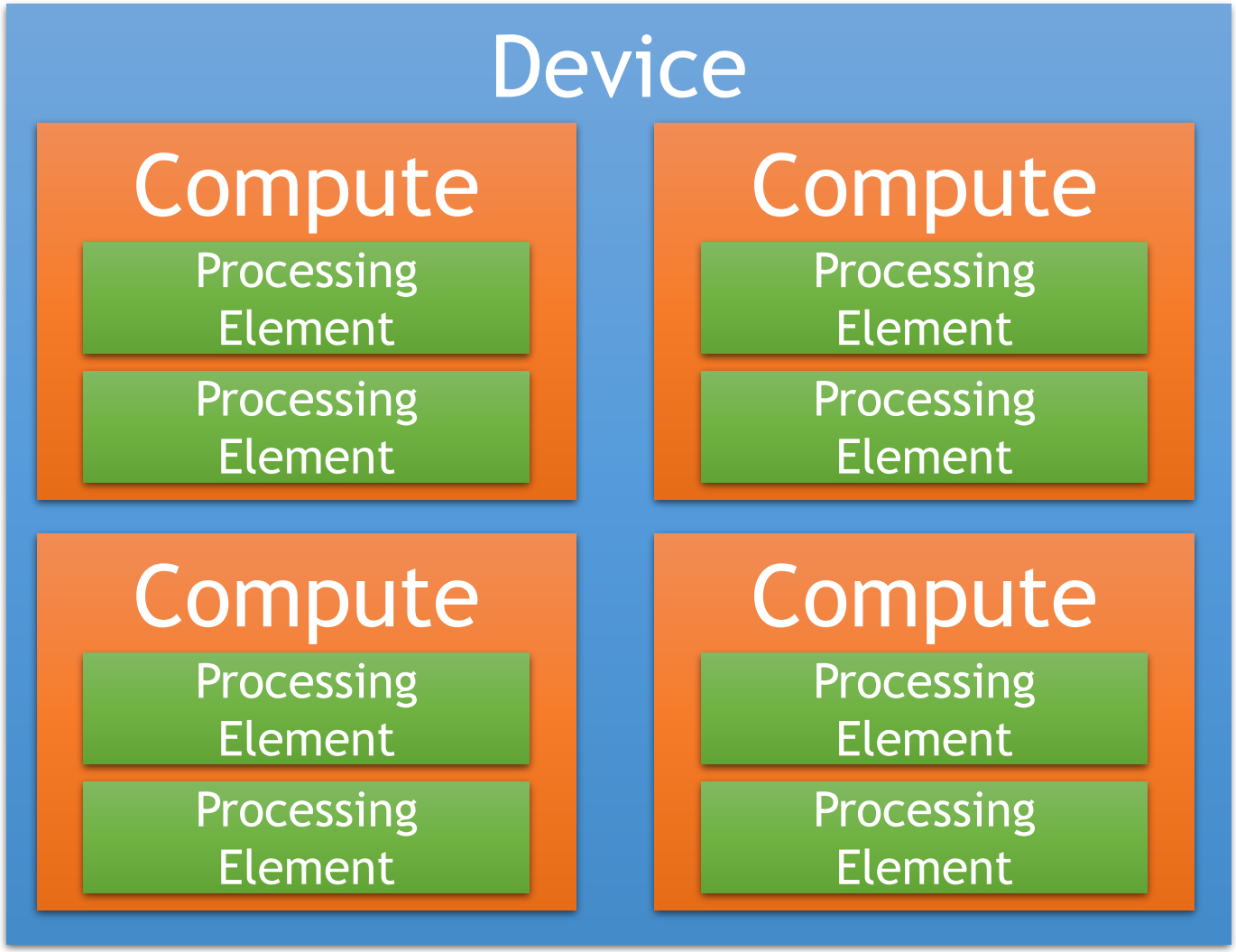
- It abstracts the various levels of parallelism, not the hardware directly
  - LoC versus ToC
  - Task- versus data-parallel
- Hardware elements appear on an abstract level
- No implicit entities
- Maximal control over the hardware
- It promises portable performance

Why not OpenCL?

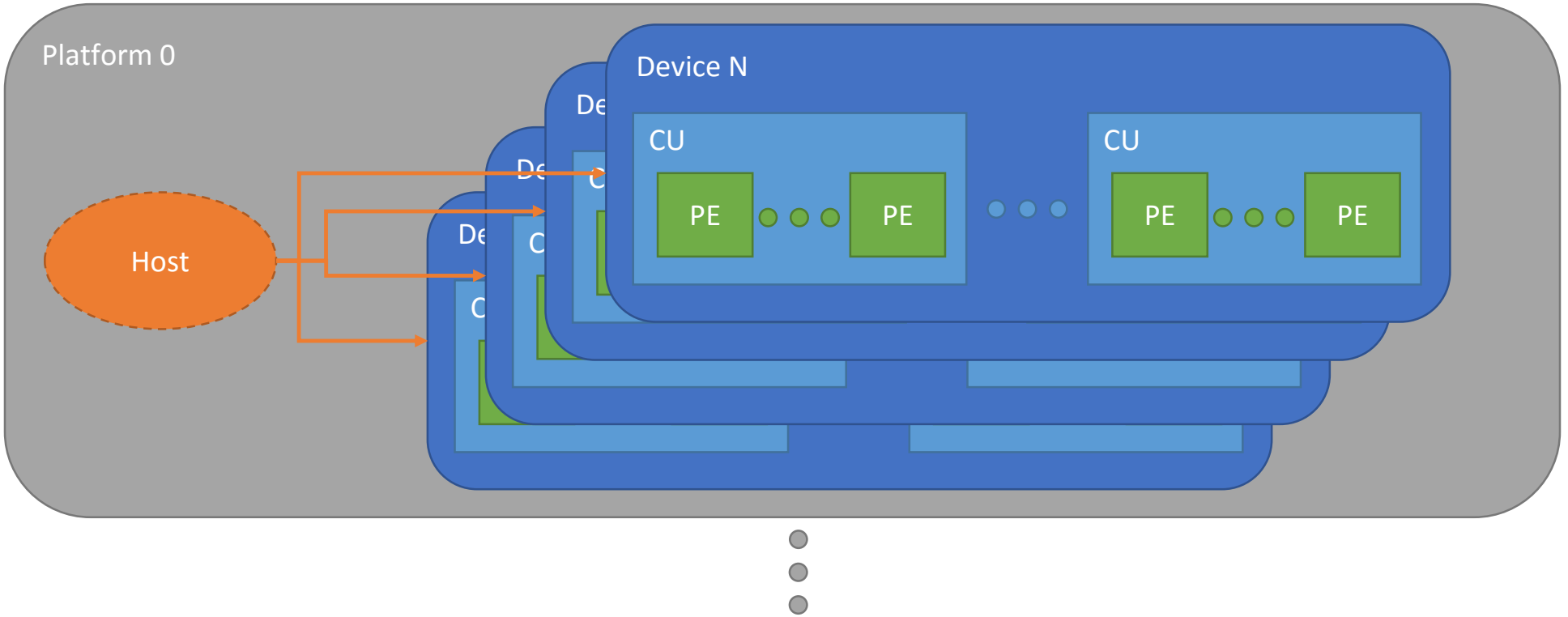
- The API is extremely verbose
- Template metaprogramming inside kernels currently is not available
- Debugging is not easy (it is always hard in GPGPU)
- Generic programming is almost impossible
- It must almost always be wrapped (luckily, cl.hpp exists)

OpenCL Platform Model

- One device consists of several Compute Units
- Each CU has several Processing Elements inside
- Each Compute Unit empowers it's PEs with superpowers
  - Fast memory visible only to the CU's own PEs
  - Synchronization capabilities
  - Collective operations



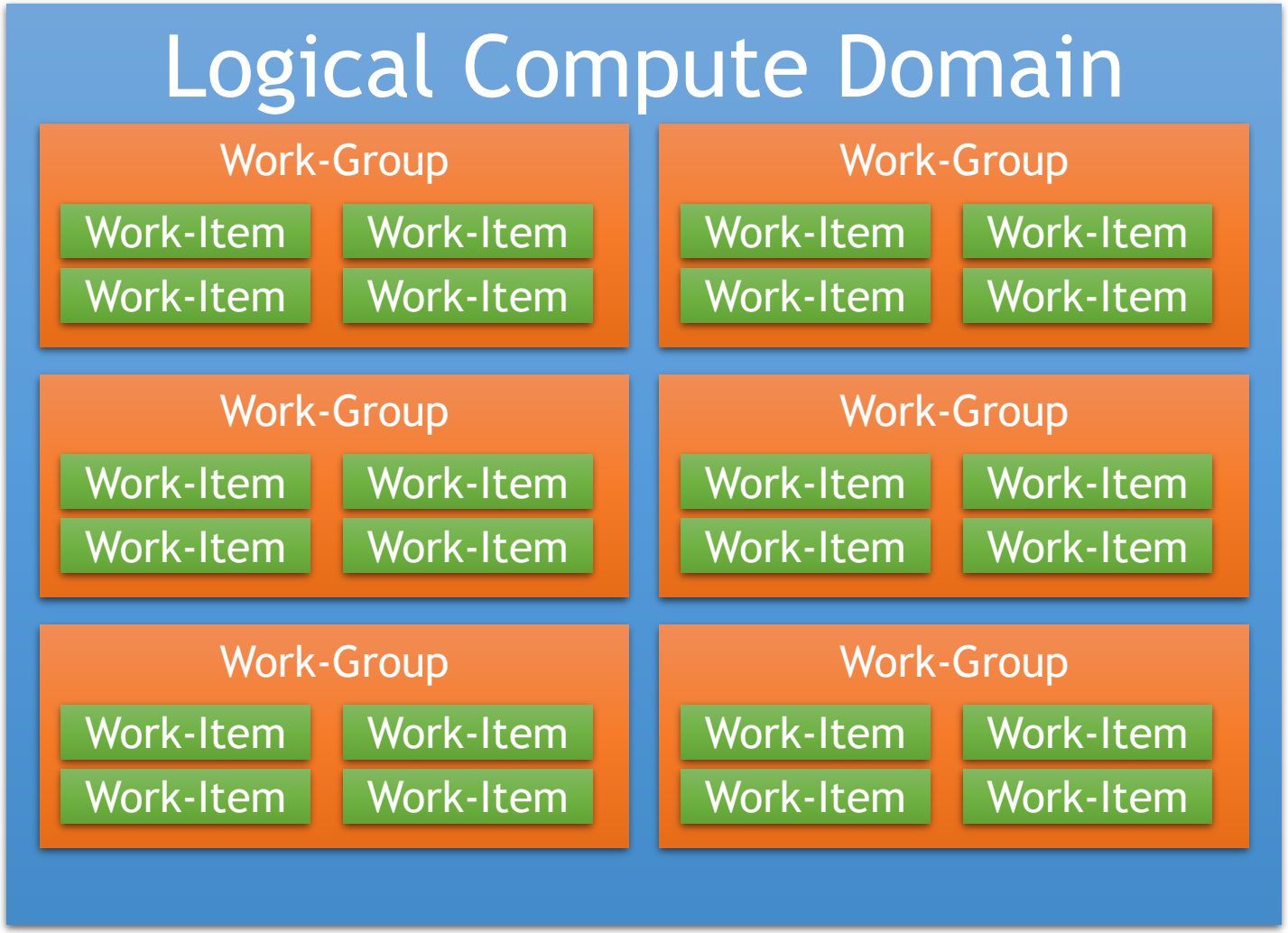
OpenCL Platform Model



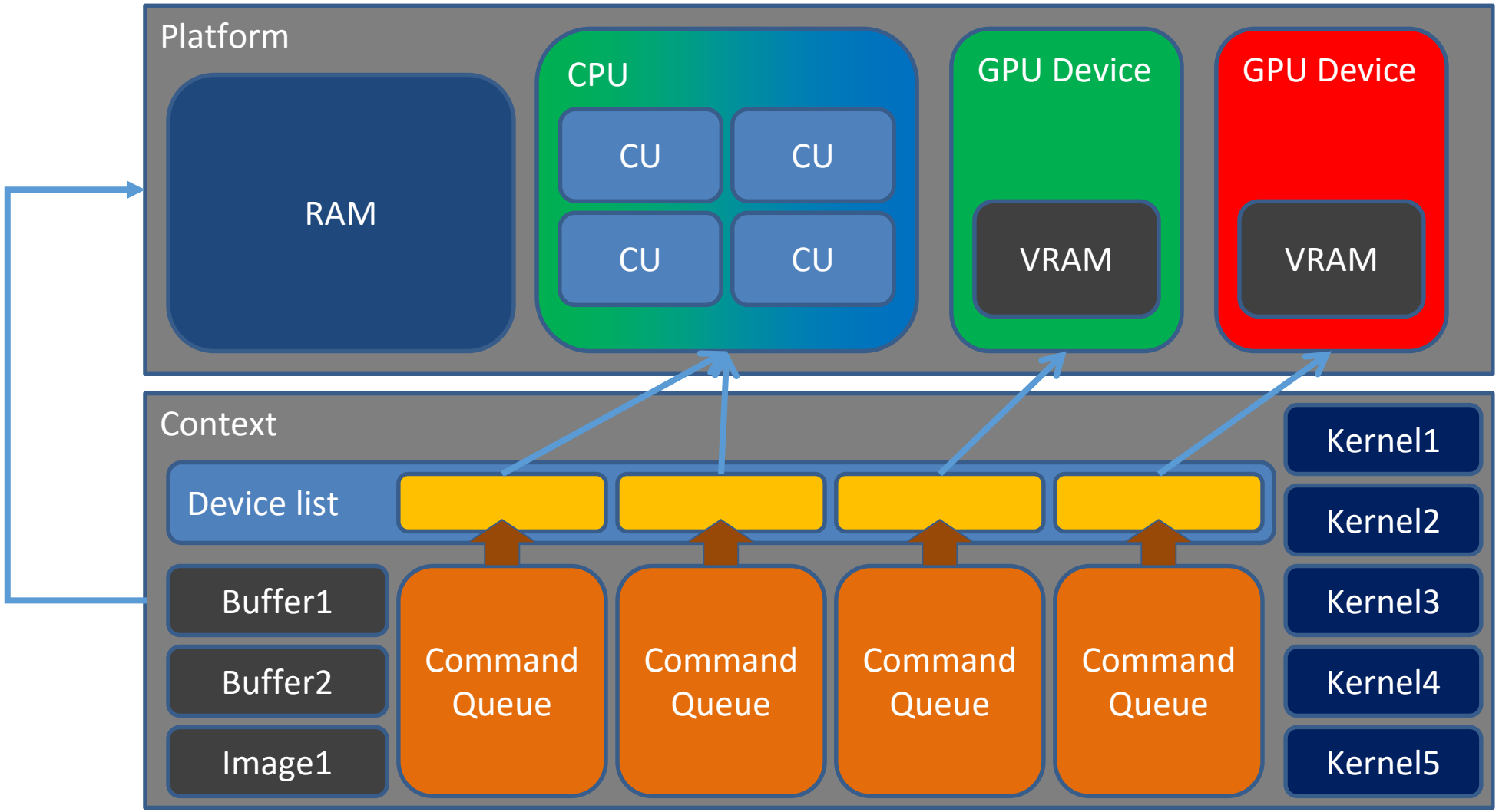
- One implementation of a OpenCL is called a platform (one for each vendor)
- A platform may support multiple devices
- Platforms are loaded at run time
- They can only interact through host memory

OpenCL Threading Model

- We launch threads with a logical indexing ranging from 1 to 3 dimensions
- We create a logical grouping of these threads called work-groups
- Each thread in work-group will be called a work-item
- Each work-group is guaranteed to execute on the same Compute Unit



OpenCL Application Model



OpenCL Application Model

- An application initially queries available platforms
- Chooses one or multiple of them, depending on needs
  - Find fastest GPU
  - Find at least one double-precicion capable device
  - Use absolutely all devices available in a machine
- Selects devices to use
- Load kernel code into memory
- Compile kernels for the selected devices
- Create device memory objects
- Launch kernels

The example: $y = a \cdot x + y$

```
const size_t size = 65536;
const float a = 2.0f;
std::vector<float> x(size);
std::vector<float> y(size);
std::iota(std::begin(x), std::end(x), 1.0f);
std::iota(std::begin(y), std::end(y), 1.0f);

std::transform(begin(x), end(x), begin(y), begin(y),
    [=](const float& b, const float& c) { return a * b + c; });
```

Example: $y = a \cdot x + y : x, y \in \mathbb{V}; a \in \mathbb{R}$

```
const size_t size = 65536;
const float a = 2.0f;
std::vector<float> x(size);
std::vector<float> y(size);
std::iota(std::begin(x), std::end(x), 1.0f);
std::iota(std::begin(y), std::end(y), 1.0f);

std::transform(
    begin(x),
    end(x),
    begin(y),
    begin(y),
    [=](float& b, float& c) { return a * b + c; });
```

OpenCL Sample Application

```
// C standard includes
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
// OpenCL includes
#include <CL/cl.h>

// OpenCL error handling function
void checkErr( cl_int err, const char * name );

// Probe for available OpenCL platforms
void probe_platforms();

// Select platform with most DP capable devices
cl_platform_id select_platform();

// Test device for DP capability
cl_bool is_device_DP_capable( cl_device_id device );
```

OpenCL Sample Application

```
// Count devices with DP capability
unsigned int count_dp_capable_devices(cl_platform_id platform);

// Select devices with DP capability
cl_device_id* select_devices( cl_platform_id platform );

// Create standard context
cl_context create_standard_context( cl_device_id* devices );

// Load kernel file
char* load_program_file( const char* filename );

// Build program file for all devices in a context
cl_program build_program_source( cl_context context, const char* source );

// Obtain kernel from program
cl_kernel obtain_kernel( cl_program program, const char* name );
```

OpenCL Sample Application

```
// OpenCL error handling function
void checkErr( cl_int err, const char * name )
{
    if ( err != CL_SUCCESS )
    {
        printf_s("ERROR: %s (%i)\n", name, err);

        exit( err );
    }
}
```

OpenCL Sample Application

```
// Entry point to our application
int main()
{
    probe_platforms();

    cl_platform_id platform = select_platform();

    cl_device_id* devices = select_devices( platform );

    cl_context context = create_standard_context( devices );
```

OpenCL Sample Application

```
// Probe for available OpenCL platforms
void probe_platforms()
{
    cl_int CL_err = CL_SUCCESS;
    cl_uint numPlatforms = 0;

    CL_err = clGetPlatformIDs( 0, NULL, &numPlatforms );
    checkErr( CL_err, "clGetPlatformIDs(numPlatforms)" );

    if ( numPlatforms > 0 )
    {
        cl_platform_id* platforms = (cl_platform_id*)malloc( numPlatforms *
sizeof( cl_platform_id ) );
        CL_err = clGetPlatformIDs( numPlatforms, platforms, NULL );
        checkErr( CL_err, "clGetPlatformIDs(platforms)" );

        printf_s( "%u platform(s) found:\n", numPlatforms );
```

OpenCL Sample Application

```
        for ( cl_uint i = 0; i < numPlatforms; ++i )
        {
            char pbuf[100];
            CL_err = clGetPlatformInfo( platforms[i], CL_PLATFORM_VENDOR, sizeof(
pbuf ), pbuf, NULL );
            checkErr( CL_err, "clGetPlatformInfo(CL_PLATFORM_VENDOR)" );

            printf_s( "\t%s\n", pbuf );
        }

        free( platforms );
    }
    else
    {
        printf_s( "No OpenCL platform detected.\n" );
        exit( -1 );
    }
}
```

OpenCL Sample Application

```
// Select platform with most DP capable devices
cl_platform_id select_platform()
{
    cl_int CL_err = CL_SUCCESS;
    cl_platform_id result = NULL;
    cl_uint numPlatforms = 0;

    CL_err = clGetPlatformIDs( 0, NULL, &numPlatforms );
    checkErr( CL_err, "clGetPlatformIDs(numPlatforms)" );

    cl_platform_id* platforms = (cl_platform_id*)malloc( numPlatforms * sizeof(
cl_platform_id ) );
    CL_err = clGetPlatformIDs( numPlatforms, platforms, NULL );
    checkErr( CL_err, "clGetPlatformIDs(platforms)" );

    cl_uint max_count = 0;
```

OpenCL Sample Application

```
    for ( cl_uint i = 0; i < numPlatforms; ++i )
    {
        cl_uint count = count_dp_capable_devices( platforms[i] );

        if ( count > max_count )
        {
            max_count = count;
            result = platforms[i];
        }
    }
    if ( result == NULL )
    {
        printf_s( "No double precision capable HW detected.\n" );
        exit( -1 );
    }
    free( platforms );
    return result;
}
```

OpenCL Sample Application

```
// Count devices with DP capability
unsigned int count_dp_capable_devices( cl_platform_id platform )
{
    cl_int CL_err = CL_SUCCESS;
    cl_uint result = 0;
    cl_device_id* devices = NULL;
    cl_uint numDevices = 0;

    CL_err = clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, 0, 0, &numDevices );
    checkErr( CL_err, "clGetDeviceIDs(numDevices)" );
    devices = (cl_device_id*)malloc( numDevices * sizeof( cl_device_id ) );
    CL_err = clGetDeviceIDs( platform, CL_DEVICE_TYPE_ALL, numDevices, devices, 0 );
    checkErr( CL_err, "clGetDeviceIDs(devices)" );

    for ( cl_uint i = 0; i < numDevices; ++i )
        if ( is_device_DP_capable( devices[i] ) ) ++result;
    free( devices ); return result;
}
```

OpenCL Sample Application

```
// Test device for DP capability
cl_bool is_device_DP_capable( cl_device_id device )
{
    cl_int CL_err = CL_SUCCESS;
    char pbuf[1024];

    CL_err = clGetDeviceInfo( device, CL_DEVICE_EXTENSIONS, sizeof( pbuf ), pbuf,
NULL );
    checkErr( CL_err, "clGetDeviceInfo(CL_DEVICE_EXTENSIONS)" );

    return strstr( pbuf, "cl_khr_fp64" ) || strstr( pbuf, "cl_amd_fp64" );
}
```

OpenCL Sample Application

```
// Create standard context
cl_context create_standard_context( cl_device_id* devices )
{
    cl_int CL_err = CL_SUCCESS;
    cl_uint count = 0;
    cl_context result = NULL;
    cl_platform_id platform = NULL;

    CL_err = clGetDeviceInfo( devices[0], CL_DEVICE_PLATFORM, sizeof(
cl_platform_id ), &platform, NULL );
    checkErr( CL_err, "clGetDeviceInfo(CL_DEVICE_PLATFORM)" );

    count = count_dp_capable_devices( platform );

    cl_context_properties cps[3] = { CL_CONTEXT_PLATFORM,
(cl_context_properties)platform, 0 };

    result = clCreateContext( cps, count, devices, NULL, NULL, &CL_err );
    checkErr( CL_err, "clCreateContext()" ); return result; }
```

OpenCL By Example

- Who still remembers what we are doing?
  - We are doing a vector equation
- What exactly am I coding?
  - Context is lost in when having to deal with too many details
- We got as far as the 4th line of the main function
  - Main is 50 lines long
  - The entire source code is 399 lines of code
- If only I knew a language a bit more sophisticated than C…

OpenCL By Example

```
// C++ Standard includes
#include <vector>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <ios>

// OpenCL includes
#define __CL_ENABLE_EXCEPTIONS
#include <CL/cl.hpp>
```

OpenCL By Example

```
int main() {
    try {
        // Do all of the OpenCL initialization here. All errors are handled later
    }
    catch ( cl::Error error ) {
        std::cerr << error.what() << "(" << error.err() << ")" << std::endl;

        if ( std::string(error.what()) == "clBuildProgram" )
        {
            if (program.getBuildInfo<CL_PROGRAM_BUILD_STATUS>(devices.at(0)) ==
CL_BUILD_ERROR)
            std::cerr << std::endl << program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(
devices.at(0) ) << std::endl;
        }
        exit( error.err() );
    }
    return 0;
}
```

OpenCL By Example

```
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);

if(platforms.size() != 0)
    std::cout << "Found " << platforms.size() << "platforms.„ << std::endl;
else
{
    std::cout << "No OpenCL platform found.\n";
    std::exit(-1);
}
for(auto& platform : platforms)
    std::cout << platform.getInfo<CL_PLATFORM_VENDOR>() << std::endl;
```

OpenCL By Example

```
// Choose platform with most DP capable devices
auto plat = std::max_element(platforms.cbegin(), platforms.cend(), [](const
cl::Platform& lhs, const cl::Platform& rhs)
{
    auto dp_counter = [](const cl::Platform& platform)
    {
        std::vector<cl::Device> devices;
        platform.getDevices(CL_DEVICE_TYPE_ALL, &devices);

        return std::count_if(devices.cbegin(), devices.cend(), [](const cl::Device&
device)
        {
            return
                (device.getInfo<CL_DEVICE_EXTENSIONS>().find("cl_khr_fp64")) ||
                (device.getInfo<CL_DEVICE_EXTENSIONS>().find("cl_amd_fp64"));
        });
    };
    return dp_counter(lhs) < dp_counter(rhs);
});
```

OpenCL By Example

```
// Obtain DP capable devices
plat->getDevices(CL_DEVICE_TYPE_ALL, &devices);

std::remove_if(devices.begin(), devices.end(), [](const cl::Device& device)
{
    return
        !(device.getInfo<CL_DEVICE_EXTENSIONS>().find("cl_khr_fp64")) ||
         (device.getInfo<CL_DEVICE_EXTENSIONS>().find("cl_amd_fp64"));
});

// Create context
std::vector<cl_context_properties> props{ CL_CONTEXT_PLATFORM,
reinterpret_cast<cl_context_properties>((*plat)()), 0 };
cl::Context context(devices, props.data());
```

OpenCL By Example

```
// Load program source
std::ifstream source_file("C:/Users/nagy-_000/OneDrive/Develop/Egyetem/Active/GPGPU-
speci/OpenCL-C-API/src/OpenCL-C-API.cl");
if (!source_file.is_open())
    throw source_file.exceptions();

std::string source_string( std::istreambuf_iterator<char>(source_file),
(std::istreambuf_iterator<char>()));
```

OpenCL By Example

```
// Create program and kernel
program = cl::Program(context, source_string);

program.build(devices, "-Werror");

cl::Kernel kernel(program, "vecAdd");
auto vecAdd = cl::make_kernel<double, cl::Buffer, cl::Buffer>(kernel);
```

OpenCL By Example

```
//#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void vecAdd(
    double a,
    __global double* x,
    __global double* y)
{
    int gidX = get_global_id(0);

    y[gidX] = a * x[gidX] + y[gidX];
}
```

OpenCL By Example

```
const unsigned int chainlength = 1048576;

cl_double a = 2.0;
std::vector<cl_double> x(chainlength);
std::vector<cl_double> y(chainlength);

cl::Buffer buf_x(context, CL_MEM_READ_WRITE, x.size() * sizeof(cl_double), x.data());
cl::Buffer buf_y(context, CL_MEM_READ_WRITE, y.size() * sizeof(cl_double), y.data());

cl::CommandQueue queue(context, devices.at(0), CL_QUEUE_PROFILING_ENABLE);
```

OpenCL By Example

```
cl::Event kernel_event;

Kernel_event = vecAdd(cl::EnqueueArgs(queue, cl::NDRange(chainlength)), a, buf_x,
buf_y);

kernel_event.wait();

cl_ulong exec_start, exec_end;

std::cout << "Kernel execution took: " <<
kernel_event.getProfilingInfo<CL_PROFILING_COMMAND_END>() -
kernel_event.getProfilingInfo<CL_PROFILING_COMMAND_START>() << " nanosceonds" <<
std::endl;

queue.enqueueReadBuffer(buf_y, CL_TRUE, 0, y.size() * sizeof(cl_double), y.data());
```

OpenCL By Example

- The entire source code is 127 lines of code
- It is „fairly” easy to read
  - Using containers alleviates the user of tedious memory management
  - Using algorithms instead of loops to state your intent
  - Code is so compact, you don’t even need functions at this level of complexity

Going Single-source

- Loading kernel code at run-time?
  - Sounds unneccesary (and is), but allows optimizations impossible otherwise
- Wouldn’t it be better if we could write host-side and device-side code together, and just compile’n’run?
  - People used to CUDA know the feeling
  - People used to OpenCL might not
- Using the same source file and C++ allows different techniques

C++AMP By Example


```
#pragma once

// C++ Standard includes
#include <cstdlib>      // std::exit, EXIT_SUCCESS
#include <vector>       // std::vector
#include <iostream>     // std::cout, std::cerr
#include <algorithm>    // std::remove_if
#include <chrono>       // std::chrono::high_resolution_clock

// C++AMP includes
#include <amp.h>

int main()
{
    ...
}
```

C++AMP By Example


```
auto devices = concurrency::accelerator::get_all();

if (devices.empty())
{
    std::cerr << "No amp-compatible accelerator detected." << std::endl;
    std::exit (EXIT_FAILURE);
}

std::remove_if(devices.begin(), devices.end(),
               [](const concurrency::accelerator& device)
                   { return !device.get_supports_double_precision(); });

if (devices.empty())
{
    std::cerr << "No DP capable accelerator detected." << std::endl;
    std::exit (EXIT_FAILURE);
}
```

C++AMP By Example

```
auto acc_view = devices.at(0).get_default_view();

const unsigned int chainlength = 1048576;

double a = 2.0;
std::vector<double> x(chainlength);
std::vector<double> y(chainlength);

concurrency::array<double> arr_x(concurrency::extent<1>(chainlength), x.cbegin(),
x.cend(), acc_view);
concurrency::array<double> arr_y(concurrency::extent<1>(chainlength), x.cbegin(),
x.cend(), acc_view);
```

C++AMP By Example

```
auto start = std::chrono::high_resolution_clock::now();

concurrency::parallel_for_each(acc_view, arr_x.get_extent(),
    [a,
     view_x = concurrency::array_view<double>(arr_x),
     view_y = concurrency::array_view<double>(arr_y)](const concurrency::index<1> idx)
restrict(amp,cpu)
    {
        view_y[idx] = a * view_x[idx] + view_y[idx];
    });

auto end = std::chrono::high_resolution_clock::now();

std::cout << "Kernel execution took: " <<
    std::chrono::duration_cast<std::chrono::nanoseconds>(end.time_since_epoch() -
start.time_since_epoch()).count() << " nanoseconds" << std::endl;

y = arr_y;
```

Algorithm Libraries

The low hanging fruits?

Low hanging fruits

- Most GPGPU APIs either have the most common algorithms built-in, or 3rd party libraries are developed.
- All of them rely on a GPGPU API to execute
  - They inherit all the limitations of the underlying API (OS, HW, etc.)
- Simple computations can be implemented in seconds.
- Complicated situations are borderline impossible.

Thrust

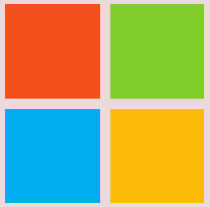
- CUDA template library
- Easy to interface with the STL itself
- Because it relies on CUDA
  - Only usable on Nvidia HW
  - Can only by compiled using nvcc
- Most stable and flexible library

Parallel STL via SYCL

Wigner GPU Lab

- C++17 will feature parallel versions of the STL
- Insert 3 characters, and algorithms use all cores
- The design allows for GPU acceleration in a trivial manner
  - The current SYCL-enabled implementation is only a proof-of-concept
  - It's sole purpose is to demonstrate, that it could be done
  - Feel free to step up and contribute
- Because it builds atop SYCL, it's portable

C++AMP Algorithms Library



- C++AMP template library
- Easy to interface with the STL itself
- Because it relies on AMP
  - Can only be compiled using an AMP-compatible compiler (MSVC, Clang)
  - It is vendor and OS independent
- Still at V1.0 (next version coming soon)

Bolt Template Library

BOLT

Wigner GPU Lab

- C++ template library
- Easy to interface with the STL itself
- It relies on multiple back-ends
  - Intel Thread Building Blocks for CPUs
  - OpenCL for both CPU/GPU
  - It's interface is restricted by both of the above
- open-source, custom types ARE A PAIN

intel

NVIDIA

AMD

Thrust


```
const float a = 2.0f;
thrust::device_array<float> x(size);
thrust::device_array<float> y(size);

thrust::sequence(x.begin(), x.end());
thrust::sequence(y.begin(), y.end());

thrust::transform(x.begin(), x.end(), y.begin(), y.begin(),
__host__ __device__ [=]( const float& x, const float& y )
{
    return a * x + y;
} );
```

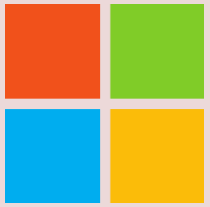
Parallel STL via SYCL

```
const float a = 2.0f; std::sycl::queue q;
std::array<float> x(size);
std::array<float> y(size);

std::iota(begin(x), end(x), 1.0f);
std::iota(begin(y), end(y), 1.0f);

std::transform(sycl::sycl_execution_policy<class VecAdd>(q), x.begin(),
x.end(), y.begin(), y.begin(), [=]( const float& x, const float& y )
{
    return a * x + y;
} );
```

C++AMP Algorithms Library



Wigner GPU Lab

```
const float a = 2.0f;
concurrency::array<float> x(size); array_view<float> x_av(x);
concurrency::array<float> y(size); array_view<float> y_av(y);

amp_stl_algorithms::iota(begin(x_av), end(x_av), 1.0f);
amp_stl_algorithms::iota(begin(y_av), end(y_av), 1.0f);

amp_stl_algorithms::transform(begin(x_av), end(x_av), begin(y_av),
begin(y_av), [=]( const float& x, const float& y ) restrict(amp,cpu)
{
    return a * x + y;
} );
```

intel

NVIDIA

AMD

Bolt Template Library

BOLT

Wigner GPU Lab

```
const float a = 2.0f;
bolt::cl::device_vector<float> x(size);
bolt::cl::device_vector<float> y(size);

std::iota(x.begin(), x.end(), 1.0f);
std::iota(y.begin(), y.end(), 1.0f);

bolt::cl::transform(x.begin(), x.end(), y.begin(), y.begin(),
saxpy_functor(a));
```

intel

NVIDIA

AMD

Bolt Template Library

BOLT

Wigner GPU Lab

```
BOLT_FUNCTOR(saxpy_functor,
struct saxpy_functor
{
    const float a;

    saxpy_functor(float _a) : a(_a) {}

    float operator()(const float& x, const float& y) const
    {
        return a * x + y;
    }
};
);
```

intel

NVIDIA

AMD

But I was told this stuff is hard…

And all I've been showed thus far are kittens and fluffy bunnies

Things they do not teach you in school

- Documentation only speaks about how things generally work
  - What the heck is error code -1001 when the largest documented one is -68?
    - Teach me the ways of the ICD sensei!
  - I have good code, but my computer freezes (to death)
    - WDDM, XServer, Wayland, Mir, pre-emption, QoS, WTF?
  - CUDA always works*
    - Except
      - If you have Intel+Nvidia (99% of machines out there)
      - And want to use you IGP for desktop on linux (99% of people here)
  - Drivers provided as RPM, but cluster is running Ubuntu
    - „Good luck mate! HARRR” – Intel support (they do help though)
  - I was told incremental development is good
    - Have you tried refactoring GPU code? It’s fun, through and through (reboots)!

Summary

- Mastering GPGPU takes time
  - And you’re never finished
  - Handle with care, it is highly addictive
- Using algorithm libraries are good, however
  - They do not teach
  - They rarely achieve anything near to peak performance
- Using domain specific libraries is good, always
  - They are usually faster than you could ever be
  - We beg to differ, but we’re cocky folk