GPU-Lab

The Concepts of HPC
The Formalization of Hierarchical Parallelism

Máté Ferenc Nagy-Egri

HAS Wigner RCP – GPU-Lab administrator

HAS Wigner RCP – Young researcher, VIRGO group

Eötvös Loránd University of Sciences – PhD student

RoLCG – Grid, Cloud & High Performance Computing in Science

2015 Cluj-Napoca, Romania

GPU-Lab

Provide infrastructure to any academic institute interested in trying massive parallelism

Give seminars to encourage taking advantage of GPGPU

University course at the Eötvös Loránd University of Sciences

Annual conference for promoting GPGPU and exchange expertise

Generic and generative numeric algorithm research

gpu.wigner.mta.hu

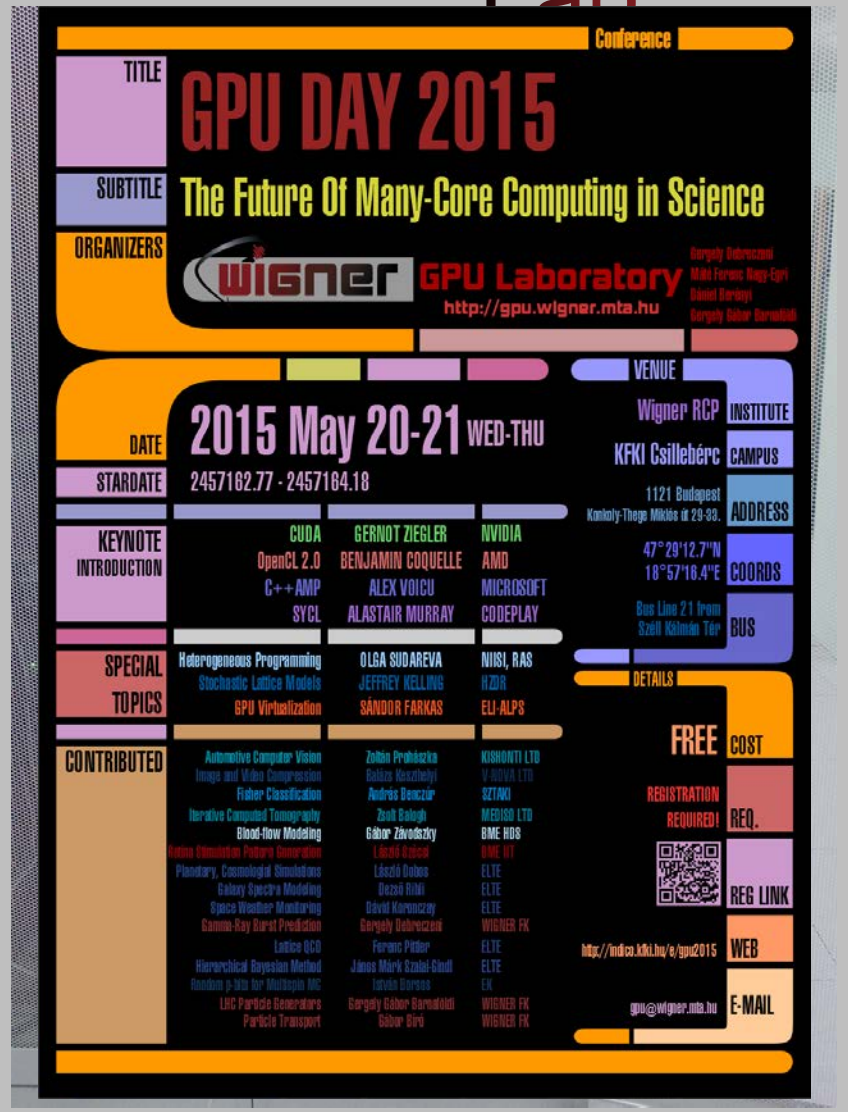
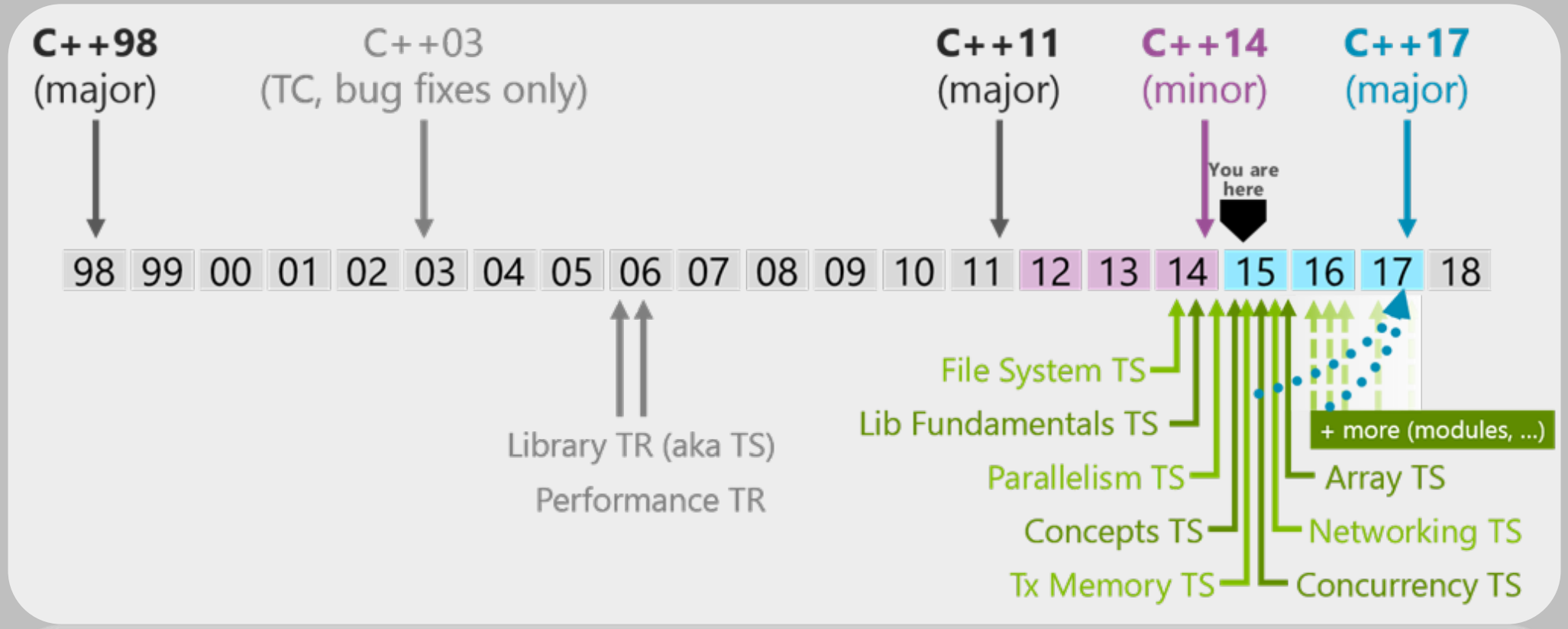
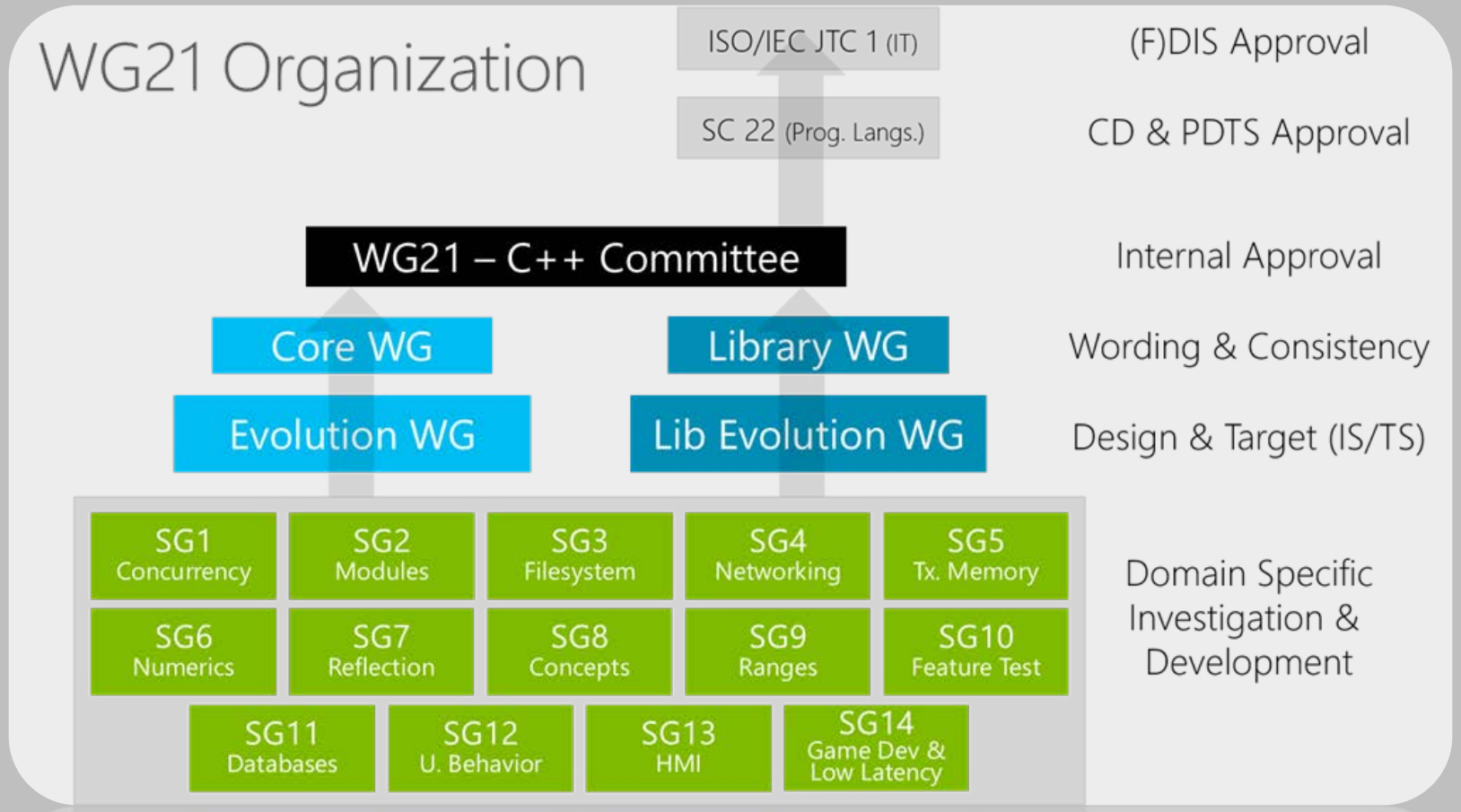


TABLE OF CONTENTS





RENAISSANCE

C++11 is a major update to both the core language and the Standard Template Library (STL)

C++14 is a minor update filling in the gaps left by C++11

Provide language features as facilitation for the programmer to express him/herself

Augment the STL to cover more domain specific territories

Zero overhead abstraction for everyone!

Make simple things simpler!

Do not add complexity!

Make the language easier to learn!

Maintain backwards compatibility!

Deprecation to remove anachronistic features

RENAISSANCE

```
auto a = 2.0f;

std::vector<float> vec(10);
for (auto& elem : vec) elem = a;

std::remove_if(vec.begin(), vec.end(),
[](const auto& elem) { return elem < 10; });


auto call std::async(std::launch::async, [](float
x, float y, float z) { return x+y+z; }, a, a, a);

std::cout << typeid(call).name() << std::endl;
// std::future<float(float, float, float)>


auto = [](Class&& class) {return class.func()};
```

New **auto** keyword for automatic type deduction

Range-based for loop
(syntactic sugar + optimization)

Lambda expressions for code clarity
(reduce boilerplate)

Variadic templates for expressiveness

New keywords for metaprogramming
decltype, typeid, sizeof...()

R-value references for performance
And many, many more...

RENAISSANCE

```
static_assert(std::is_base_of<Base,
Derived>::value);

auto start
std::chrono::high_precision_timer::now();

std::complex<double> a{0, 1};


std::string text = "some text";
std::regex parser("text");
if (std::regex_search(text, parser) ) {return 0;}

std::atomic<int> b;
++b;

// We can even launch threads. It's not hard
```

Type-traits STL utilities eliminate compiler hooks

std::chrono utility classes for portable high-percision time measurment

Numeric library including std::complex and std::random

std::regex as a portable, high-speed regular expression parser

std::atomic for built-in and user defined types

std::thread for low-level threading
And many, many more...

NEW PERSPECTIVES

```cpp
#include <string>
#include <memory>

class Solver
{
public:
    virtual void handle(std::string& in) = 0;
};

class Terminator : public Solver
{
public:
    virtual void handle(std::string& in) override { in.erase(); };
};

int main()
{
    std::unique_ptr<Solver> terminator = std::make_unique<Terminator>();
    std::string problems = "pollution, famine, corruption, etc.";

    terminator->handle(problems);

    return problems.empty() ? 0 : -1; // true if the string is empty, false otherwise
}
```

NEW PERSPECTIVES

```
template <typename T> class Charged
{
public: double charge() const { return static_cast<const T&>(*this)->charge(); }
};

class Electron : public Charged<Electron>
{
public: double charge() const { return -1.0; }
};

class UltraStrange
{
public: double hypercharge() { return 1.0; }
};

template <typename T> auto charge(Charged<T>&& in) { return in.charge(); }

int main()
{
    std::cout << charge(Electron()) << std::endl;
    std::cout << charge(UltraStrange()) << std::endl;

    return 0;
}
```

NEW PERSPECTIVES

Concepts aim at providing interfaces that can be implemented statically

Allows static dispatch of interfaces (GPGPU compatible)

Concepts is the compile-time counterpart (and more) to virtual functions

Much more friendly error messages

```
auto concept Charged {
    double charge() const;
};

class Electron {
public:
    double charge() const { return -1.0; }
};

auto charge(Charged&& in) { return in.charge(); }

int main() {

    auto c = charge(Electron());
}
```

C++17

NEW PERSPECTIVES

13 lines of user authored code
Compiler builds ~800.000 lines of code

Compile times shoot for the stars

```cpp
#include <cstdlib>
#include <memory>
#include <fstream>
#include <random>
#include <vector>
#include <algorithm>
#include <cmath>

int main() {

    // Create a smart-pointer to a PRNG
    // Fill STL container with the sqrt of numbers
    // Write to file some distribution
    return EXIT_SUCCESS;
}
```

13 lines of user authored code
Compiler builds 13 lines of code

With modules compile times are comparable to pre-compiled headers

```cpp
import stl.core
import stl.file
import stl.memory
import stl.random
import stl.container
import stl.algorithm
import stl.math

int main() {

    // Create a smart-pointer to a PRNG
    // Fill STL container with the sqrt of numbers
    // Write to file some distribution
    return 0;
}
```

C++17

NEW PERSPECTIVES

std::thread_pool inherits from std::executor, the abstract base class of all schedulers

Loosely defined interface

Executors can be used to implement other parallel functions

Many other types of executors could be defined beside the proposed ones

```
std::ifstream input("textures.txt");
std::vector<std::string> file_names(std::istream_iterator(input), std::istream_iterator());

std::thread_pool pool(std::thread::hardware_concurrency() - 1); // pool limited to core_count - 1
std::vector<Texture> textures(file_names.size());

for(std::size_t i = 0; i < file_names.size() ; ++i)
   pool.add([i, textures&](){ textures.at(i) = Texture(file_names.at(i)); });

pool.wait();

// Do something with the textures
```

C++17

NEW PERSPECTIVES

Parallelism is the parallel execution of threads that coexist to perform a given task

The design of inter-communication of threads in a parallel construct is critical to performance

ISO C++ tackles parallelism through high-level constructs (reduce, transform, etc.)

Parallel algorithms may be implemented with either of the two

Implementing composite algorithms is tricky
Composability is not as trivial as it used to be

Concurrency is the concurrent execution of threads performing independent tasks

Concurrent tasks generally do not communicate with each other

ISO C++ tackles concurrency through two methods:

Library feature
std::thread::then()

Language feature
Resumable functions

GUIDELINES SUPPORT LIBRARY

GSL is a minimal library and set of documents

The docs are generic guidelines for writing correct, simple and modern code

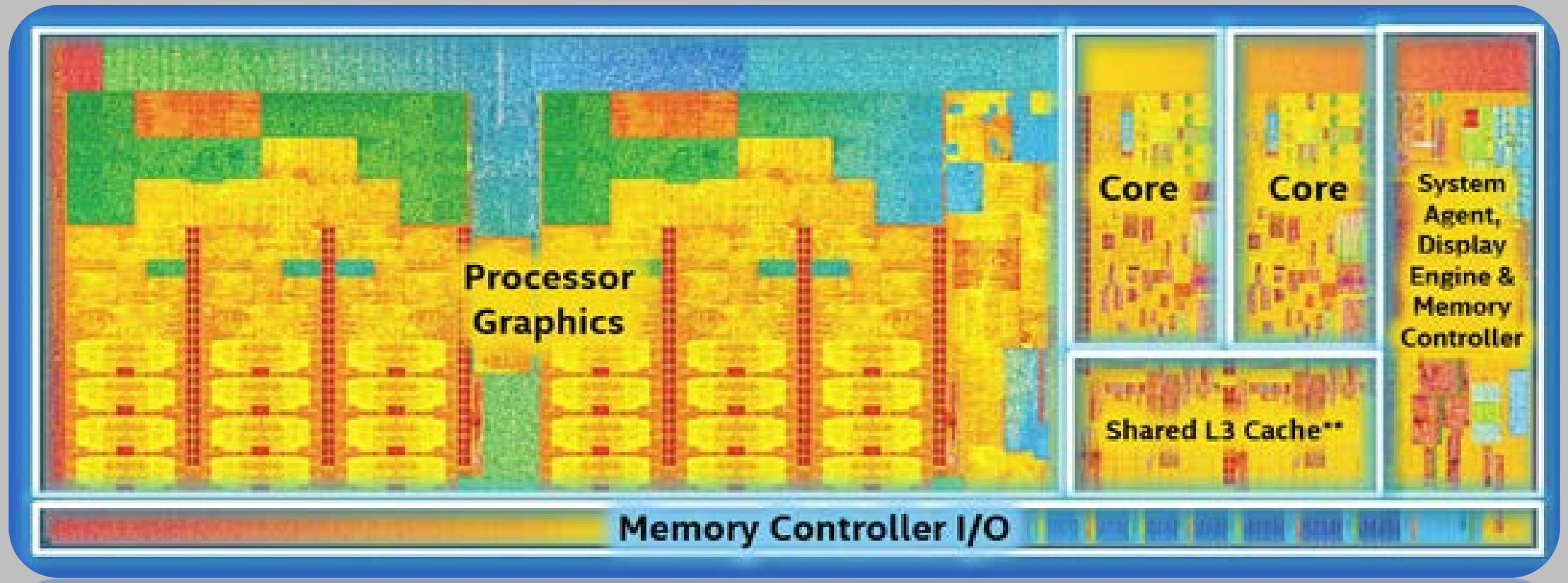
Static analysis tools hook into the library to enhance the debugging experience

Excellent starting point to learning/refreshing knowledge

Domain specific guidelines pending
Input required!

Continously evolving document and library
Available on GitHub!

PARALLELISM AS IT STANDS



PARALLELISM AS IT STANDS

CPU parallel

OpenMP

C++

Graphics

OpenGL

AMD MANTLE API TECHNOLOGY

Vulkan

Propriatary compute

NVIDIA CUDA

ATI STREAM TECHNOLOGY

Open compute

HSA FOUNDATION

C++AMP

OpenACC
Directives for Accelerators

OpenMP

SYCL

PARALLELISM AS IT STANDS

The gfx industry welcomed proprpiatary technologies
Closed/open standards however proved to be better

Standard technologies became monolithic, too complex
Propriatary APIs live their renaisance

Propriatary technologies drive hardware capabilities
Innovative technologies and techniques stem here

Open standards identify the „common denominator"
Wrap new technologies with useful abstractions

C++ is dominant in HPC, both as a back-end and as a front-end

GPGPU as an ISO C++ standard is not possible at this rate of development

Industrial self-defense mechanisms tend to sabotage open standards

The graphics industry has an enormous driving force behind it

Where is the Unreal/Cry/Mono/Unity of compute?

PARALLELISM AS IT STANDS

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

PARALLELISM AS IT STANDS

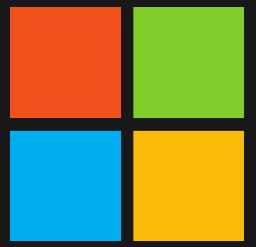
```
const float a = 2.0f;
std::vector<float> x(size);
std::vector<float> y(size);

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

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

PARALLELISM AS IT STANDS

```
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;
});
```

PARALLELISM AS IT STANDS

```
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;
} );
```



PARALLELISM AS IT STANDS

```
const float a = 2.0f; sycl::execution_policy<class Pol> pol;
std::vector<float> x(size);
std::vector<float> y(size);

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

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

PARALLELISM AS IT STANDS

```
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));
```

BOLT

PARALLELISM AS IT STANDS

```
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;
     }
};
);
```

BOLT

FORMALIZING HPC

Putting it all together

GPGPU entities generally map well to STL entities

The STL provides good abstractions, but some vital parts are missing for cluster and GPGPU parallelism

Hierarchy definately needs to be taken into account

cl::CommandQueue is a special kind of std::executor

MPI_Request, cl::Event and gl::Event are std::futures

SVMAlloc, cudaMalloc, concurrency::array are std::allocators

cl::Device, concurrency::accelerator could be execution policies

Essential parts are still missing std::accessor?

```
parallel_for_workgroup(nd_range(range(size), range(groupsize)),
                       lambda<class hierarchical>([=](group_id group)
{
    parallel_for_workitem(group, [=](tile_id tile)
    {
        out_access[tile] = in_access[tile] * 2;
    });
}));
```

FORMALIZING HPC

```
parallel_for_node(nd_range(range(MPI_COMM_WORLD)), [](mpi_rank rank)
{
    parallel_for_device(nd_range(range(devices.size()), [&](device_id dev)
    {
        parallel_for_workgroup(nd_range(range(size), range(groupsize)),
                               lambda<class hierarchical>([=](group_id group)
        {
            parallel_for_workitem(group, [=](tile_id tile)
            {
                out_access[tile] = in_access[tile] * 2;
            });
        }));
    });
});
```

Thank you for your attention!