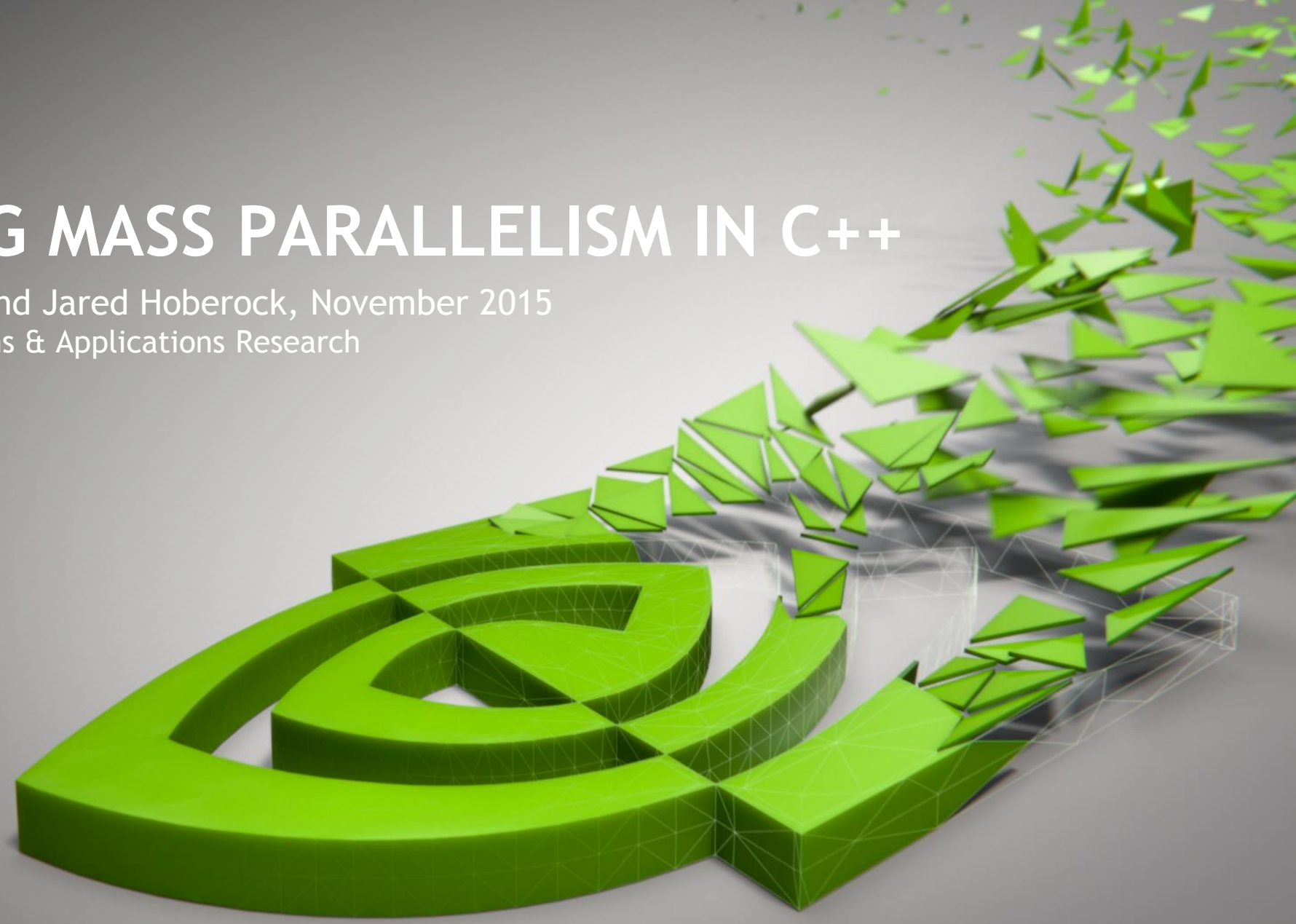
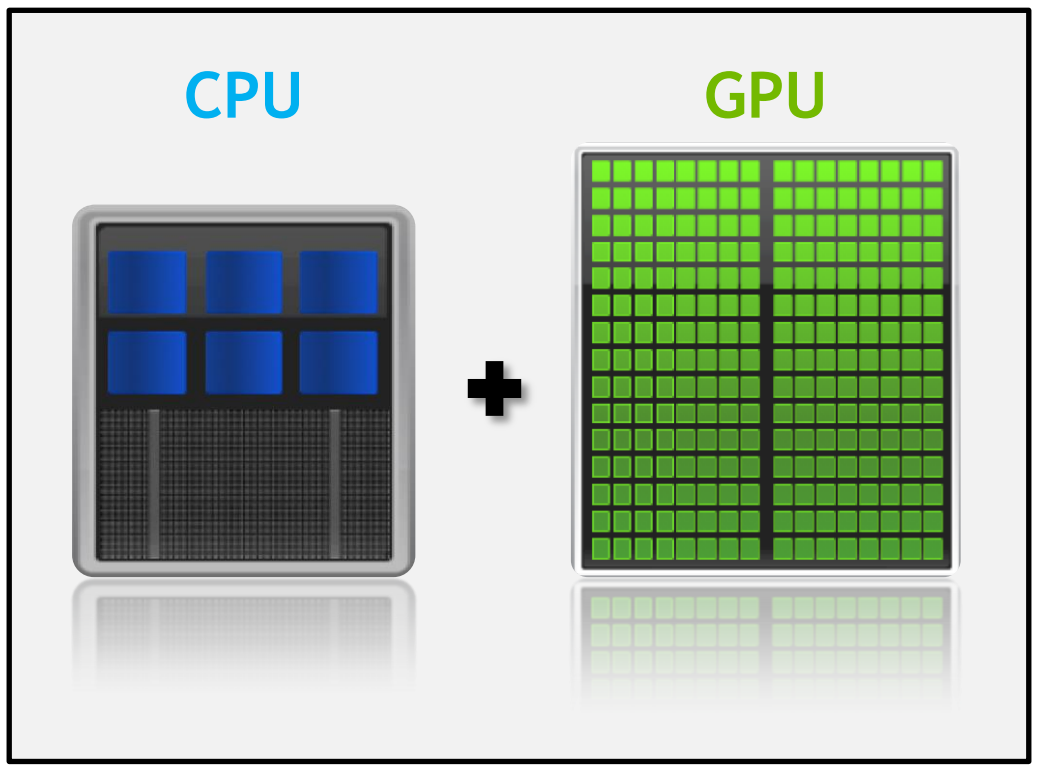
EXPOSING MASS PARALLELISM IN C++

Michael Garland and Jared Hoberock, November 2015
Programming Systems & Applications Research



ACCELERATED COMPUTING

Highly Parallel Program



High Performance Execution

RESEARCH QUESTION

How might programmers write highly parallel programs in a mainstream language like C++?

SUPPOSE PARALLELISM IS AVAILABLE

Example: Independent loop iterations represent latent parallelism

```
void saxpy(int n, float a, float *x, float *y)
{
  // Sequential code with latent parallelism
  for(int i=0; i<n; ++i)
  {
      y[i] = a*x[i] + y[i];
  }
}
```

EXPRESSING PARALLELISM IN C++11

Individually launching threads leads to inefficiency

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

ANNOTATION WITH DIRECTIVES

Requires stepping outside the language

```
void saxpy(int n, float a, float *x, float *y)
{
  #pragma omp parallel for
  for(int i=0; i<n; ++i)
  {
      y[i] = a*x[i] + y[i];
  }
}
```

MASS PARALLELISM WITHIN C++

Library implementation of parallel constructs

```
void saxpy(int n, float a, float *x, float *y)
{
  auto I = interval(0, n);

  for_each(par, I.begin(), I.end(), [=](int i)
  {
      y[i] = a*x[i] + y[i];
  }
}
```

STANDARD TEMPLATE LIBRARY

Higher-level library built around algorithms

```
for_each(begin, end, function);
```

Operator	Data	Function
A named pattern of computation and communication.	One or more collections to operate on.	Caller-provided function object injected in pattern.

PARALLEL STL

Algorithms + Execution Policies

```
for_each(par, begin, end, function);
```

↑

Execution Policy

Specify *how* operation may execute.

PARALLEL ALGORITHMS

Parallelizable algorithms in STL

```
for_each
transform
copy_if
sort
set_intersection
```

etc.

New additions for parallelism

```
reduce
exclusive_scan
inclusive_scan
transform_reduce
transform_inclusive_scan
transform_exclusive_scan
```

EXECUTION POLICIES

Specify how algorithms may execute

POLICY NAME	MEANING
`seq`	Sequential execution alone is permitted.
`par`	Parallel execution is permitted.
`par_vec`	Vectorized parallel execution is permitted.

parallel :: provided function objects can be executed in any order on one or more threads.

vectorized :: provided function objects can be also be interleaved when on one thread.

EXECUTION POLICIES

Provide opportunities for customization

Standard Policies

POLICY NAME	MEANING
`seq`	Sequential execution alone is permitted.
`par`	Parallel execution is permitted.
`par_vec`	Vectorized parallel execution is permitted.

Custom Policies (non-standard)

`cuda::par`	Always choose a CUDA-capable GPU.
`on_some_fpga`	Permit execution on FPGA where possible.
`vendors_policy`	Custom semantics not listed above.

STANDARDIZING PARALLEL STL

Technical Specification for C++ Extensions for Parallelism

Published as ISO/IEC TS 19570:2015, July 2015.

Draft available online

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf

We've proposed adding this to C++17

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0024r0.html

THRUST LIBRARY

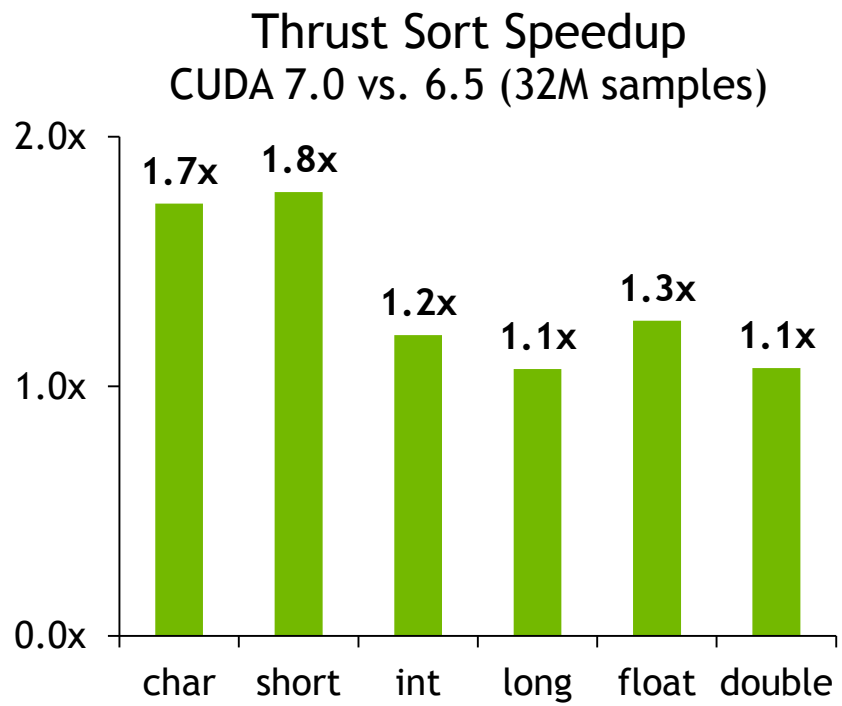
Programming with algorithms and policies today

Bundled with NVIDIA's CUDA Toolkit

Supports execution on GPUs and CPUs

Ongoing performance & feature improvements

Functionality beyond Parallel STL



From *CUDA 7.0 Performance Report*.

Run on K40m, ECC ON, input and output data on device

Performance may vary based on OS and software versions, and motherboard configuration

BEYOND PARALLEL STL

Developing a lower-level framework for parallel execution

```
void saxpy(size_t n, float a, const float* x, const float* y, float* z)
{
  using namespace agency;

  bulk_invoke(par(n), [=](parallel_agent &self)
  {
    int i = self.index();
    z[i] = a * x[i] + y[i];
  });
}
```

BEYOND PARALLEL STL

A lower-level framework with a natural mapping to CUDA

```
void saxpy(size_t n, float a, const float* x, const float* y, float* z)
{
  using namespace agency;

  bulk_invoke(cuda::par(n), [=] __device__ (parallel_agent &self)
  {
    int i = self.index();
    z[i] = a * x[i] + y[i];
  });
}
```

ADDING ASYNCHRONY

Futures track completion

```
void saxpy(size_t n, float a, const float* x, const float* y, float* z)
{
  using namespace agency;

  auto f1 = bulk_async(par(n), [=](parallel_agent &self)
  {
    int i = self.index();
    z[i] = a * x[i] + y[i];
  });

  // Do other work here, asynchronously with kernel above

  f1.wait();
}
```

ADDING ASYNCHRONY

Futures track completion and express dependence

```
void other(size_t n, float a, const float* x, const float* y, float* z)
{
  using namespace agency;

  auto f1 = bulk_async(par(n), [=](parallel_agent &self)
  {
    int i = self.index();
    z[i] = a * x[i] + y[i];
  });

  auto f2 = bulk_then(f1, par(n), other_work);
  auto f3 = bulk_then(f1, par(n), more_work);
  when_all(f2, f3).wait();
}
```

EXECUTORS

Provide control over where execution happens

Placement is, by default, at discretion of the system.

```
for_each(par, I.begin(), I.end(), [](int i) { y[i] += a*x[i]; });
```

In some cases, the programmer might want to control placement.

```
auto place1 = choose_some_place();
auto place2 = choose_another_place();

for_each(par.on(place1), I.begin(), I.end(), ...);
for_each(par.on(place2), I.begin(), I.end(), ...);
```

EXECUTOR FRAMEWORK

Abstract platform details of execution

Create execution agents

Manage data they share

Advertise semantics

Mediate dependencies

```
class sample_executor
{
public:
  using execution_category = ...;

  using shape_type = tuple<size_t,size_t>;

  template<class T>
  using future = ...;

  template<class T>
  future<T> make_ready_future(T&& value);

  template<class Container, class Function, class Future>
  future<Container> then_execute(Function f,
                                 shape_type shape,
                                 Future& dependency);
  ...
};
```

See http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0058r0.pdf for details.

Parallel Algorithms

Capture common patterns of parallel computation and communication.

Organized Collections of Parallel Agents

Direct implementation and control of parallel computation (e.g., kernels).

Executor Framework

Lowest-level abstraction of platform-specific details of work creation.

FUTURE WORK

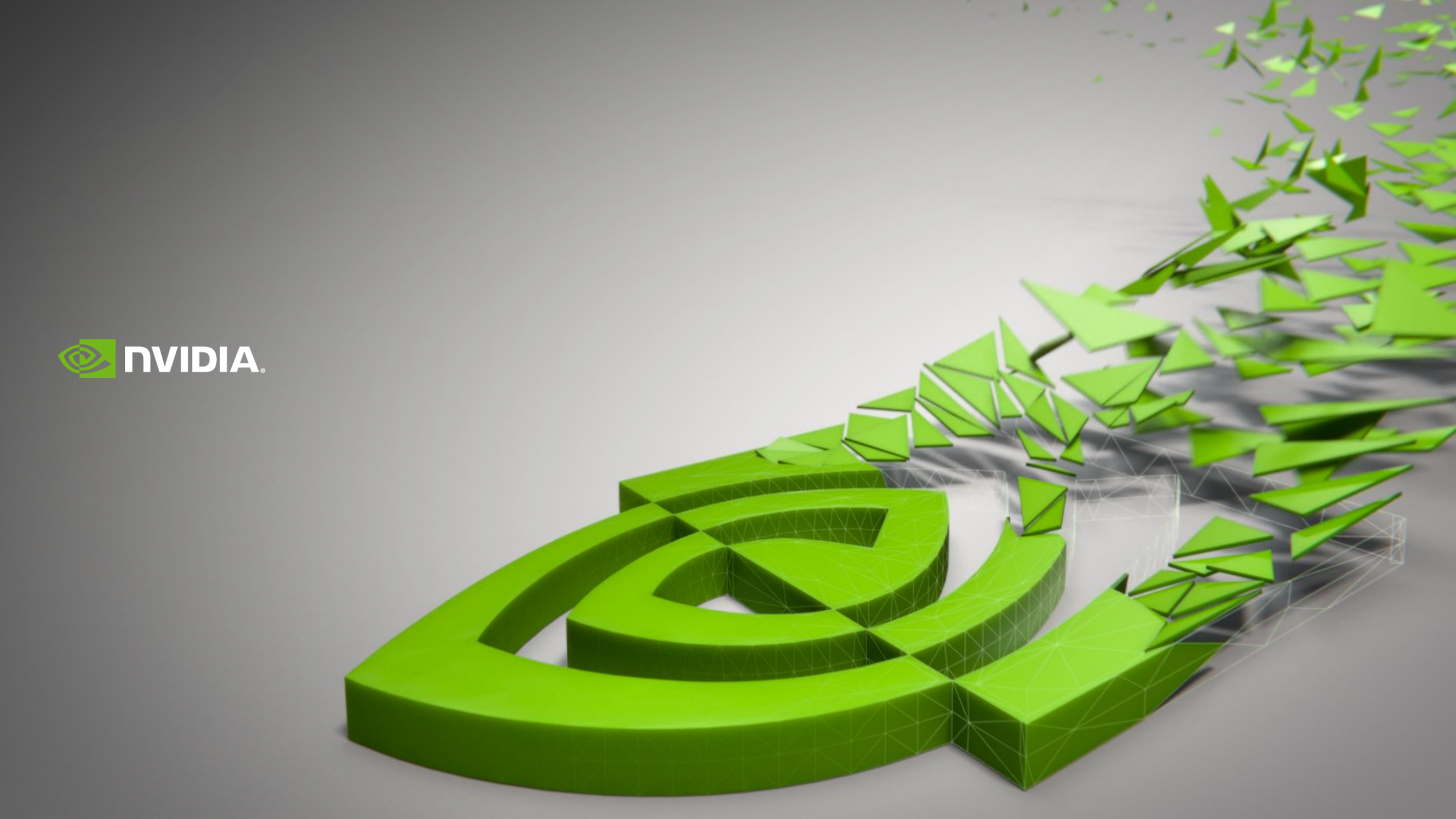
Expanded set of algorithms (cf. Thrust)

Consider range-based interfaces

Support for asynchrony in parallel algorithms

Explore desired parallel control structures

Complete definition of executor framework


NVIDIA