TASK GRAPHS VS. MPI

Stephen Jones, 18th May 2021

WHAT IS A TASK GRAPH?

A task graph is a runtime construct, built dynamically

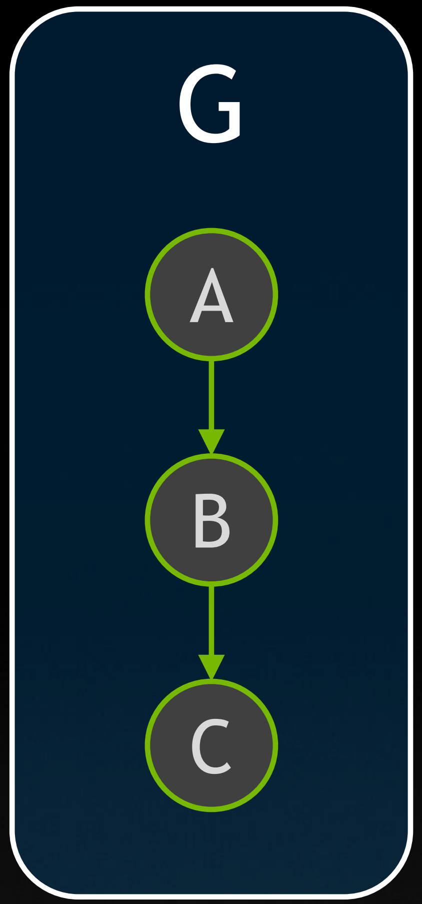
```
void main() {
    a();
    b();
    c();
}
```

Task graph representation of the workflow to the left

PRE-PACKAGED WORKFLOWS ARE EASIER TO COMPOSE

Similar to a library call, but passing runtime execution sequences around

```
Graph *library_call() {
    Graph *g = new Graph();
    g->add_node(a);
    g->add_node(b);
    g->add_node(c);

    return g;
}
```



Task graph encapsulated as an object

PRE-PACKAGED WORKFLOW LENDS ITSELF TO RUN-TIME COMPILATION

```
main() {
    Graph *g = new Graph();
    g->add_node(a);
    g->add_node(b);
    g->add_node(c);

    g->compile();
    g->run();
}
```

COMPILATION LENDS ITSELF TO EFFICIENT RE-LAUNCH

```
main() {
     Graph *g = new Graph();
     g->add_node(a);
     g->add_node(b);
     g->add_node(c);

     g->compile();
     for(i=0; i<N; i++)
          g->run();
}
```

NVIDIA

ANALAGOUS TO COMPILERS VS. INTERPRETERS

Especially if the source language is slow, e.g. Python

Runtime compilation-based

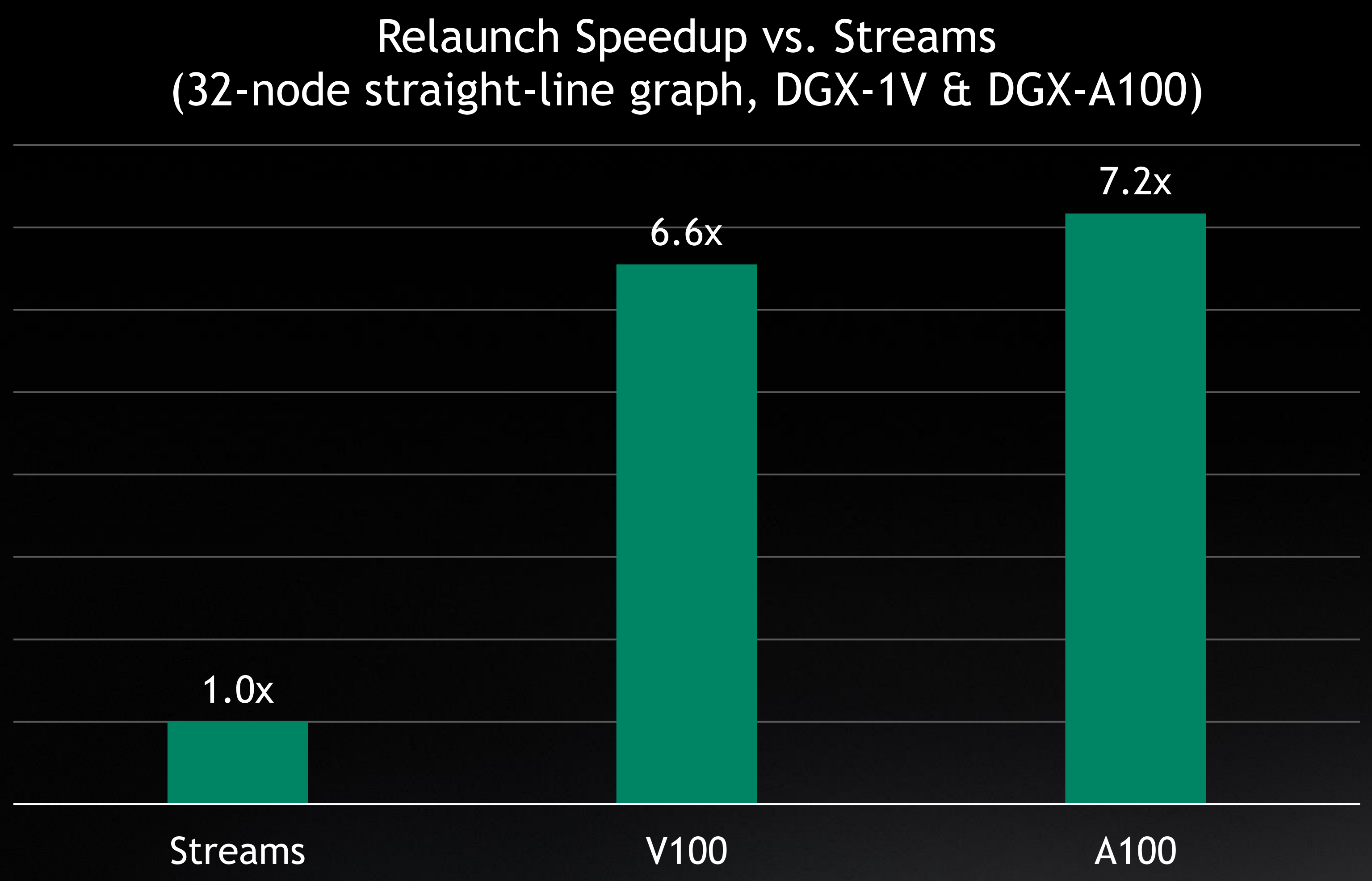
```
main() {
    Graph *g = new Graph();
    g->add_node(a);
    g->add_node(b);
    g->add_node(c);

    g->compile();
    for(i=0; i<N; i++)
        g->run();
}
```

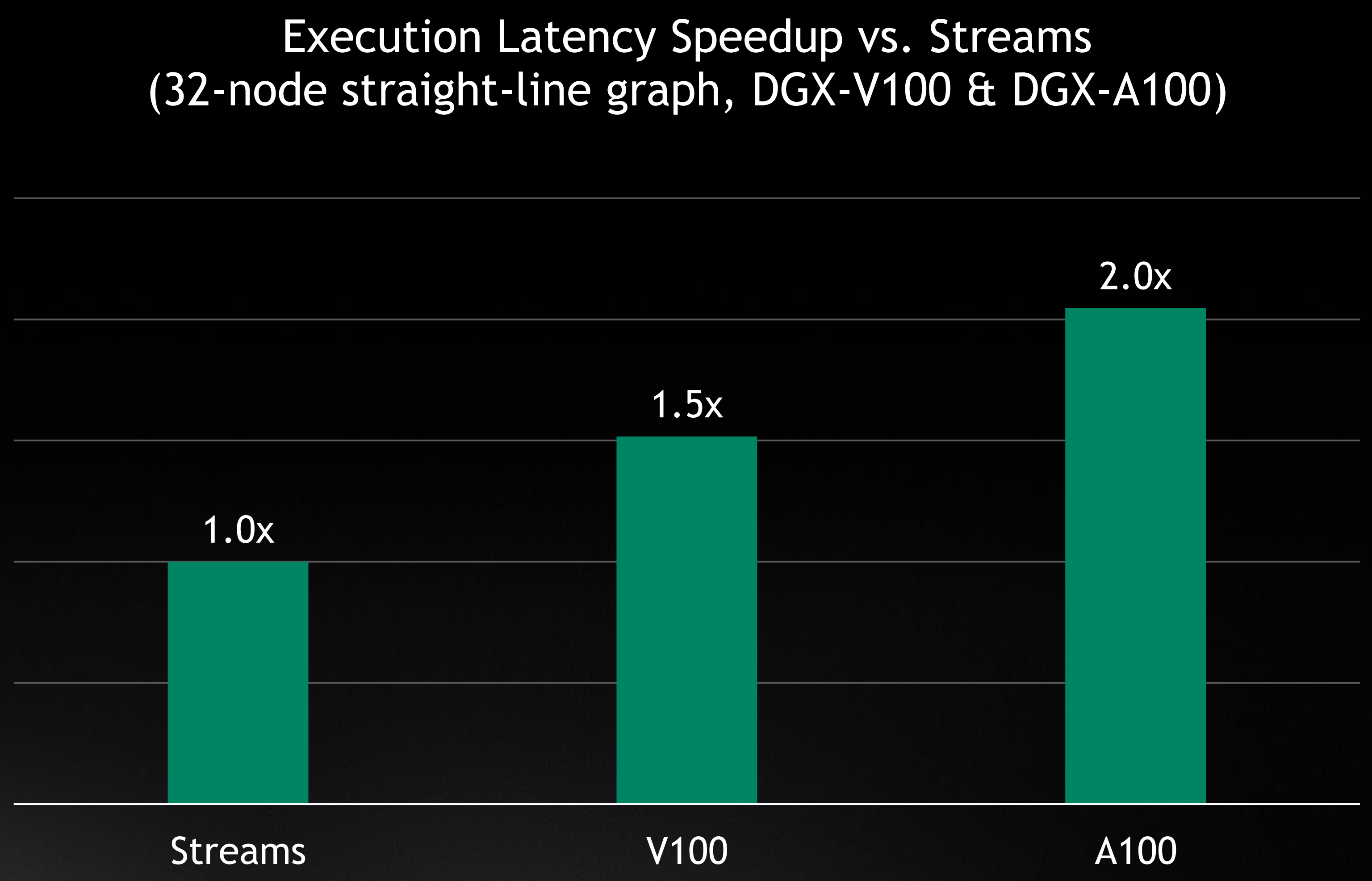
Runtime interpretation-based

```
void main() {
    for(i=0; i<N; i++) {
        a();
        b();
        c();
    }
}
```

7x LAUNCH PERFORMANCE, 2x EXECUTION PERFORMANCE

Comparing latencies, not kernel run-time



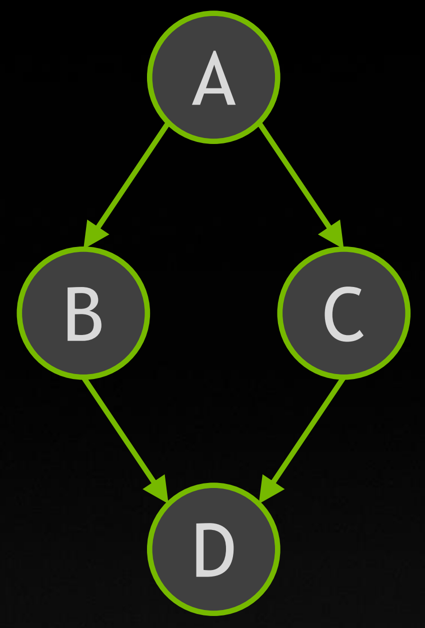


GRAPHS ARE BY DEFINITION ASYNCHRONOUS

A language-agnostic way of defining concurrency

```c
void main() {
    a();
    pthread_create(&thread_b, b);
    pthread_create(&thread_c, c);
    pthread_join(thread_b);
    pthread_join(thread_c);
    d();
}
```

Posix C



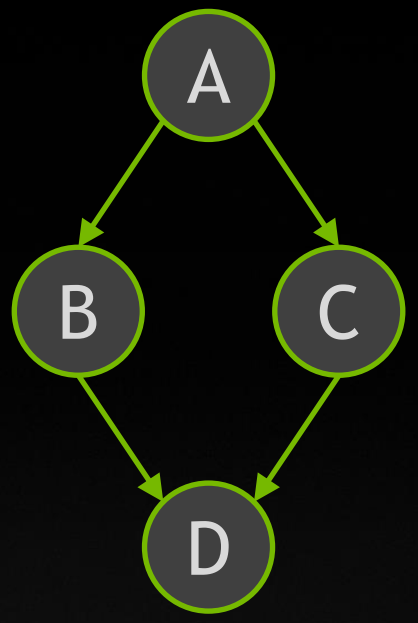

```cuda
void main() {
    a<<< s1 >>>();
    cudaEventRecord(e1, s1);
    b<<< s1 >>>();
    cudaStreamWaitEvent(e1, s2);
    c<<< s2 >>>();
    cudaEventRecord(e2, s2);
    cudaStreamWaitEvent(e2, s1);
    d<<< s1 >>>();
}
```

CUDA

MANY PLATFORM-AGNOSTIC FRAMEWORKS USE TASK GRAPHS

OpenMP, Kokkos, Raja, TensorFlow, etc.

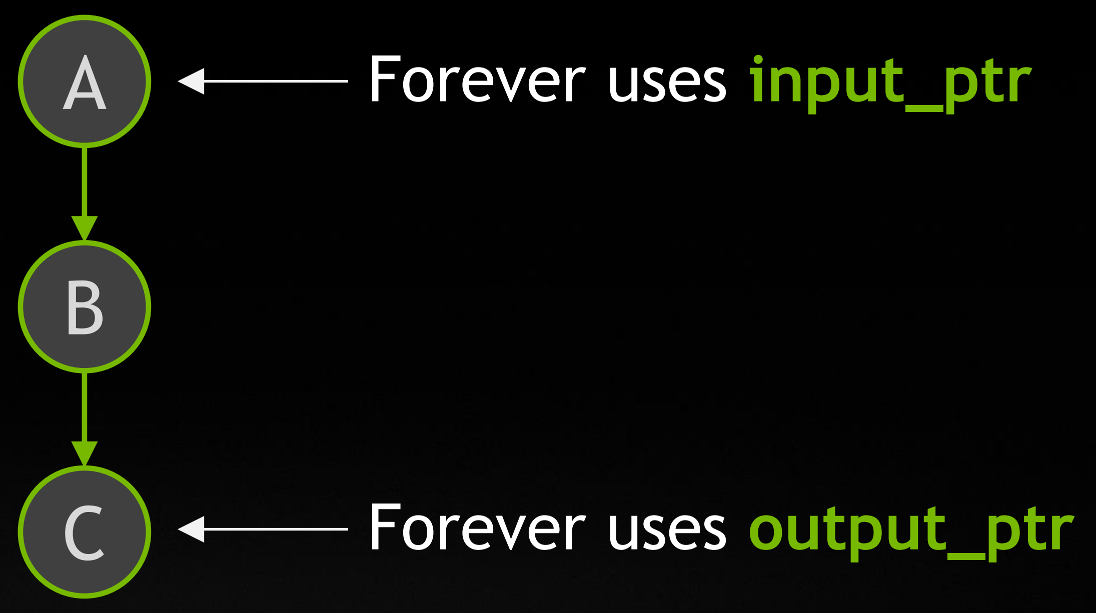
```
Graph *library_function() {
    Graph *g = new Graph();
    g->add_node(a, NULL);
    g->add_node(b, { a } );
    g->add_node(c, { a } );
    g->add_node(d, { b, c } );

    return g;
}
```



PRE-COMPILED + ASYNC = PARAMETER LIFETIME ISSUES

Similar to compilation, values are **locked in** at compile time

```
Graph *make_graph(input_ptr, output_ptr) {
    Graph *g = new Graph();
    g->add_node(a, input_ptr);
    g->add_node(b, internal_ptr);
    g->add_node(c, output_ptr);

    g->compile();
    return g;
}
```



ONE OPTION: IN-PLACE PARAMETER UPDATE

“Graph Update” is similar to JIT or incremental compilation – only rebuild the changes

```
Graph *update_graph(g, new_input, new_output) {
    Graph *g2 = new Graph();
    g2->add_node(a, new_input);
    g2->add_node(b, internal_ptr);
    g2->add_node(c, new_output);

    g->update(g2);
    return g;
}
```

WHAT DOES THIS MEAN FOR SOMETHING LIKE MPI?

Asynchronous, pre-compiled execution is problematic

Asynchronous (deferred) execution must

- Record the parameters of the call for later execution
- Manage execution & memory dependencies so that *data_from_c* is visible at the right time
- Set up an efficient callback mechanism to invoke the deferred function

```
A<<< stream >>>();
B<<< stream >>>();
C<<< stream >>>();
MPI_Isend_on_stream( data_from_C, stream );
D<<< stream >>>();
E<<< stream >>>();
cudaStreamSynchronize();
```

LAZY vs. SIMPLY ASYNCHRONOUS

Graphs require an extra degree of parameter abstraction

Lazy execution must

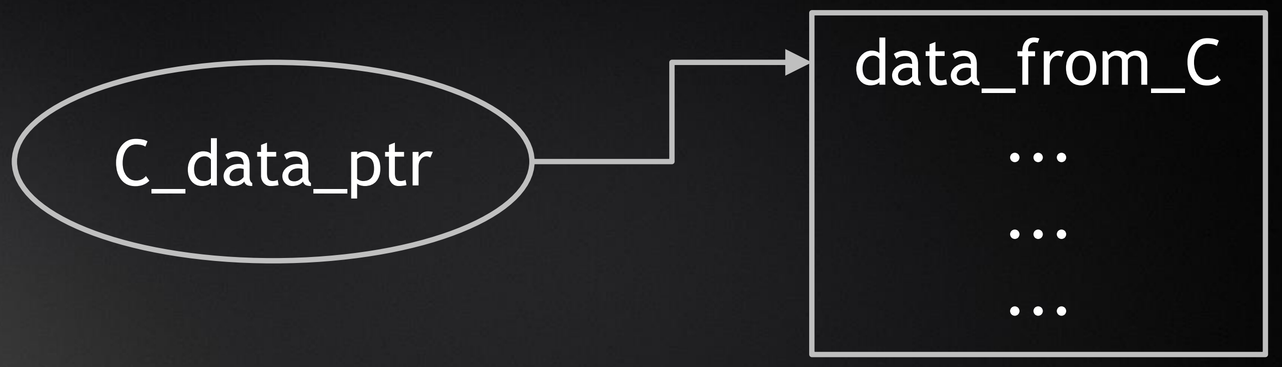
- Do everything async does, PLUS…

To avoid rebuilding graph on each iteration

- Wrap all parameters in **references**
- Callee must dereference parameters

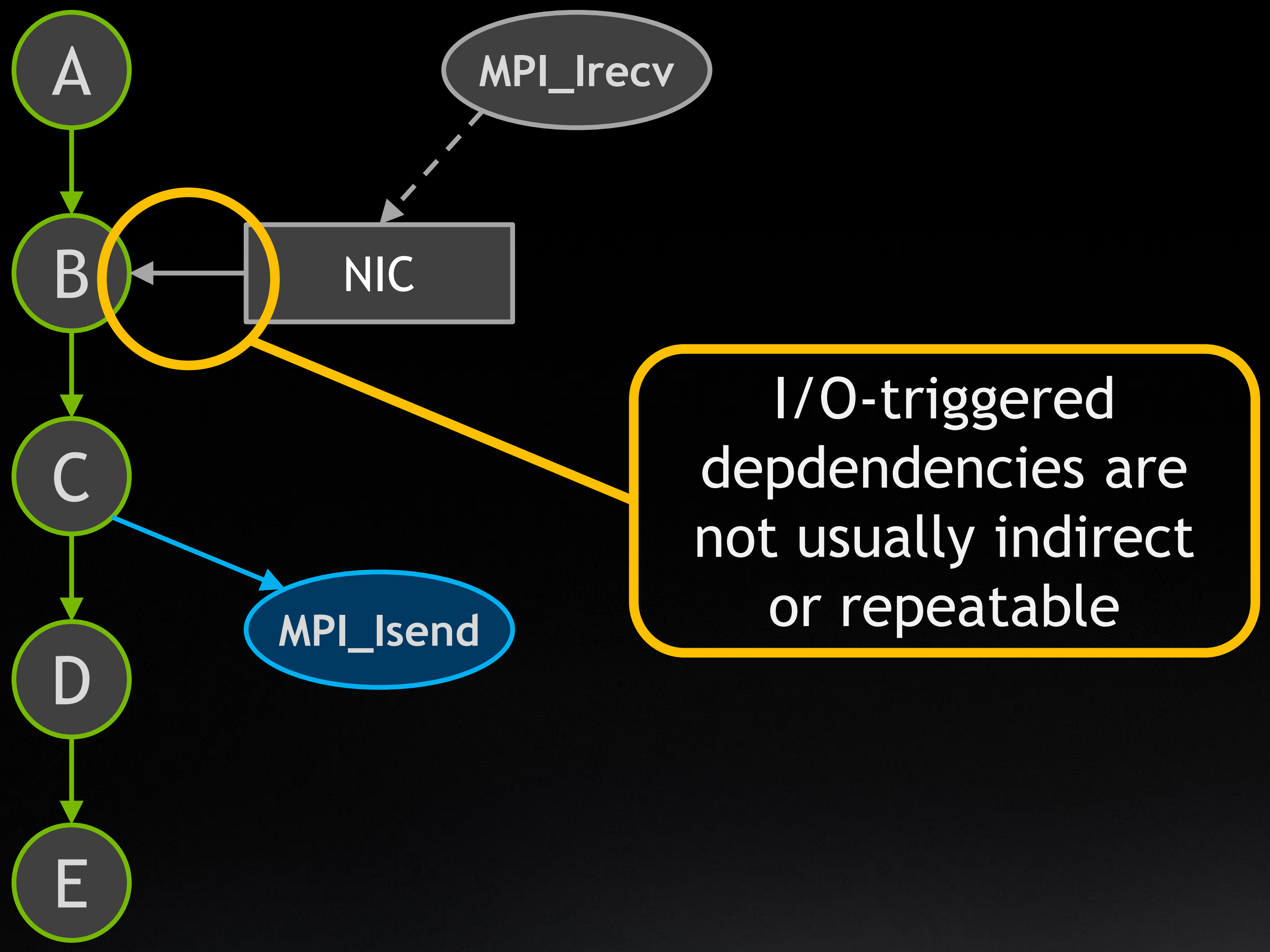
  (simplest example is MPI takes pointer-to-pointer parameters)

```
cudaStreamBeginCapture(stream);
A<<< stream >>>();
B<<< stream >>>();
C<<< stream >>>();
MPI_Isend_on_stream( ptr_to_C_data, stream );
D<<< stream >>>();
E<<< stream >>>();
cudaStreamEndCapture(stream, &graph);
```



PROBLEMS WITH EXTERNAL I/O DEPENDENCIES

May require programmatic solution – To Be Discussed



Lazy execution must

- Do everything async does, PLUS...

To avoid rebuilding graph on each iteration

- Wrap all parameters in **references**
- Callee must dereference parameters

GRAPH-NATIVE OR GRAPH-AGNOSTIC?

I have only problems, not solutions

Graph-Native

Graph describes work; runtime executes it

Runtime can set up all I/O dependencies and dereference memory buffers

BUT: requires some kind of “communication” node type with underlying intelligence

How does runtime communicate with comms library?

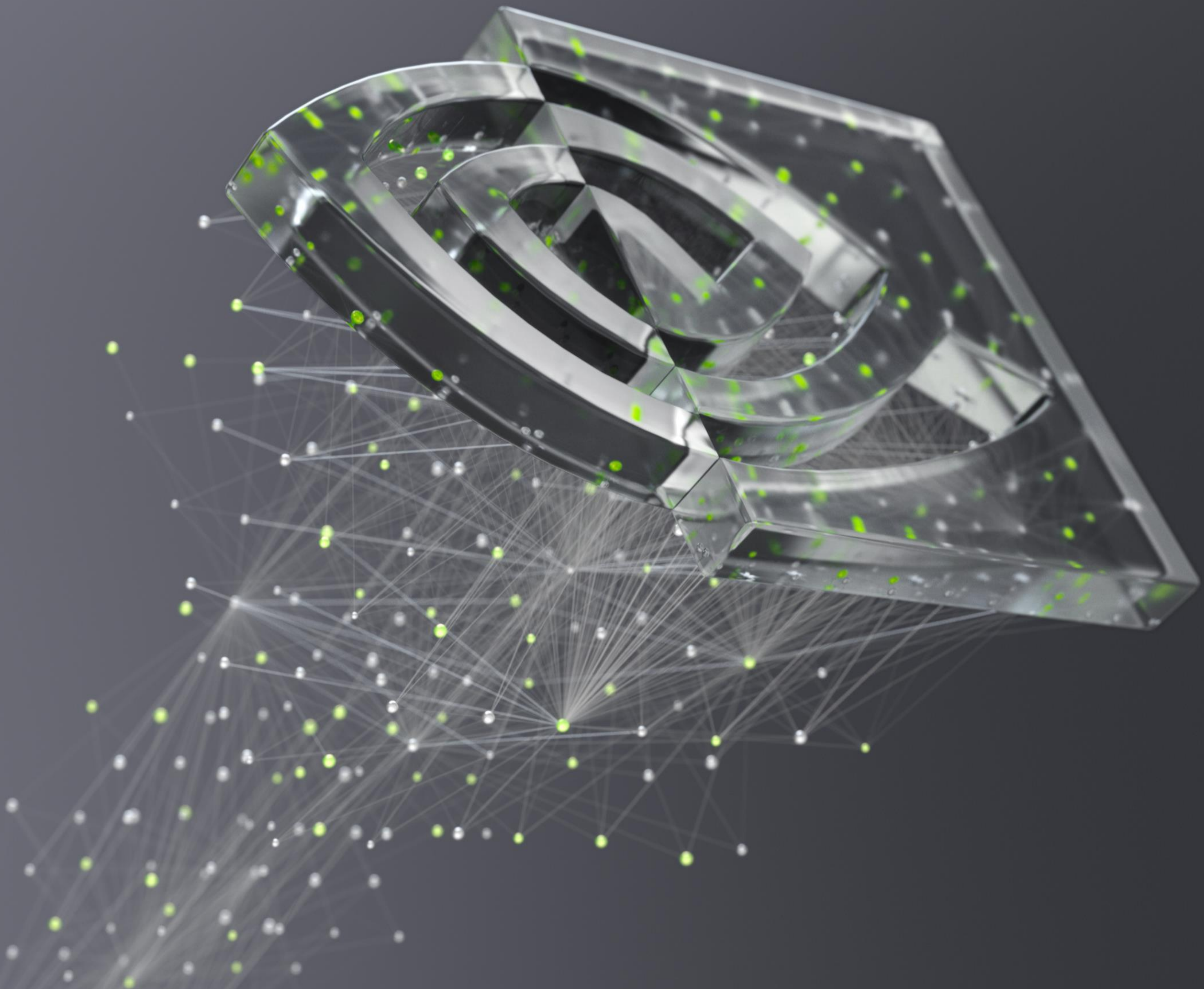
Graph-Agnostic

Comms calls are opaque to the runtime

All responsibility for I/O, memory, sync etc. lies within comms library

May require callbacks from runtime into MPI as part of a graph launch

Significant complexity, and runtime-dependence



NVIDIA