FUTURE OF ISO AND CUDA C++

The NVIDIA ISO C++ Delegation

Bryce Adelstein Lelbach **`@blelbach`**
Chief ISO C++ Library Designer, US Programming Language Standards Chair

Olivier Giroux **`@__simt__`**
ISO C++ Concurrency and Parallelism Chair

Michał Dominiak **`@Guriwesu`**
Polish C++ Standards Chair, Extended Floating Point Author

Jared Hoberock
Parallelism TS Project Editor, Executors Author

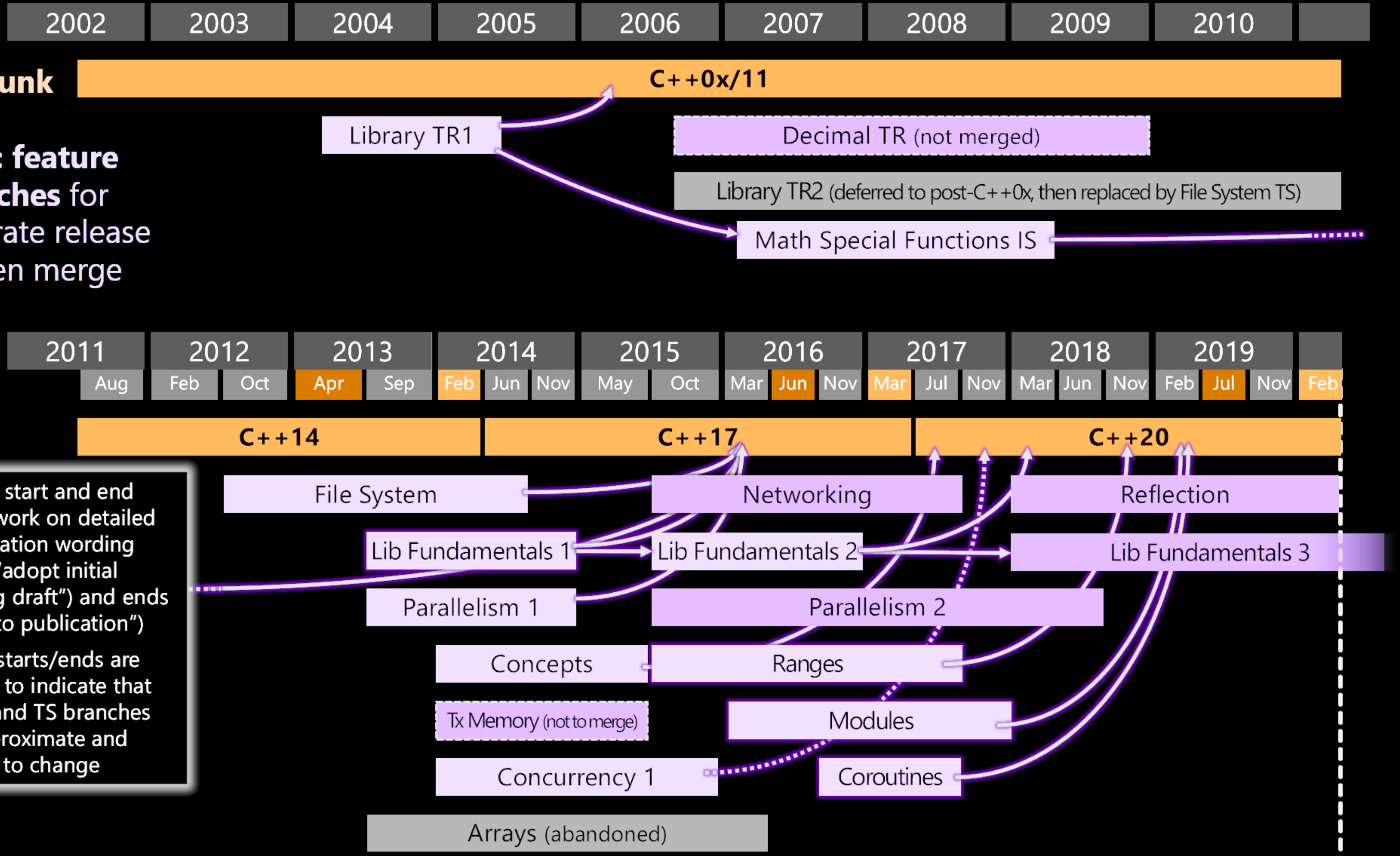
David Olsen
Scalable Synchronization Library Author, Extended Floating Point Author

Timothy Costa
Product Manager, HPC Software

Graham Lopez
Product Manager, HPC Compilers

IS: trunk

TSes: feature branches for separate release & then merge



TS bars start and end where work on detailed specification wording starts ("adopt initial working draft") and ends ("send to publication")

Future starts/ends are shaded to indicate that dates, and TS branches are approximate and subject to change

Source: https://isocpp.org/std/status

C++20

The Biggest Release in a Decade

- Modules
- Coroutines
- Concepts
- Ranges
- Scalable Synchronization

C++23

Asynchrony and Parallelism

- Standard Library Modules
- Coroutine Support Library
- Executors
- Networking
- `mdspan/mdarray`

C++23 Executors

Simplifying Work Creation

```
void compute(int resource, ...) {
  switch(resource) {
    case GPU:
      kernel<<<...>>>(...);
    case MULTI_GPU:
      cudaSetDevice(0);
      kernel<<<...>>>(...);
      cudaSetDevice(1);
      kernel<<<...>>>(...);
    case COOP_GPU:
      cudaLaunchCooperativeKernel(...);
    case GRAPH:
      cudaGraphLaunch(...);
  }
}
```

VS

```
void compute(executor auto ex, ...) {
  execute(ex, ...);
}
```

C++23 Executors

```
static_thread_pool pool(16);
executor auto ex = pool.executor();

execute(ex, []{ cout << "Hello world from the thread pool!"; });

sender auto begin    = schedule(ex);
sender auto hi_again = then(begin, []{ cout << "Hi again! Have an int."; return 13; });
sender auto work     = then(hi_again, [](int arg) { return arg + 42; });

receiver auto print_result = as_receiver([](int arg) { cout << "Received.\n"; });

submit(work, print_result);
```

Linear Algebra & C++23 mdspan/mdarray

```
auto x = ...; // An `mdspan<double, dynamic_extent>`.
auto y = ...; // An `mdspan<double, dynamic_extent>`.

auto A = ...; // An `mdspan<double, dynamic_extent, dynamic_extent>`.

// y = 3.0 * A * x;
matrix_vector_product(par, scaled_view(3.0, A), x, y);
// y = 3.0 * A * x + 2.0 * y;
matrix_vector_product(par, scaled_view(3.0, A), x,
                           scaled_view(2.0, y), y);

// y = transpose(A) * x;
matrix_vector_product(par, transpose_view(A), x, y);
```

C++23 Extended Floating Point Types

```
std::float16_t     // IEEE-754-2008 binary16.

std::float32_t     // IEEE-754-2008 binary32.

std::float64_t     // IEEE-754-2008 binary64.

std::float128_t    // IEEE-754-2008 binary128.

std::bfloat16_t    // binary32 with 16 bits truncated.
```

Why does NVIDIA care about ISO C++?

What does NVIDIA hope to accomplish in ISO C++?

What is the relationship between ISO C++ and CUDA C++?

Modern NVIDIA GPUs

implement the C++ execution model.

We spent transistors to get there.

WHY C++?

WHAT MAKES C++ PORTABLE?

WHAT MAKES C++ PORTABLE?

	Relevant in the 80s/90s
Non-8-bit `char`	✓
Noncommittal `sizeof`	✓
Non-2’s comp. `int`	✓
Non-IEEE `float`	✓
Segmented memory	✓
Non-endian pointers	✓

WHAT MAKES C++ PORTABLE?

	Relevant in the 80s/90s	Relevant Today
Non-8-bit `char`	✓	
Noncommittal `sizeof`	✓	
Non-2's comp. `int`	✓	
Non-IEEE `float`	✓	
Segmented memory	✓	✓
Non-endian pointers	✓	

WHY IS THIS NOT HELPFUL?

FALSE CHOICES

Most options are **dictated**.

New CPU? Match AARCH64.

New GPU? Match the host.

GPUs match *all the hosts*.

BAD CHOICES

Most alternatives are **bad**.

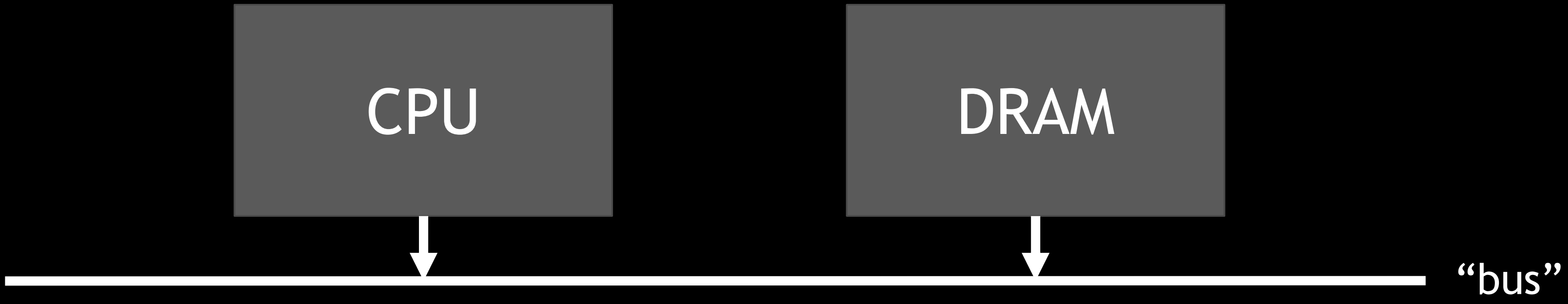
Negligible area savings.

Negligible power savings.

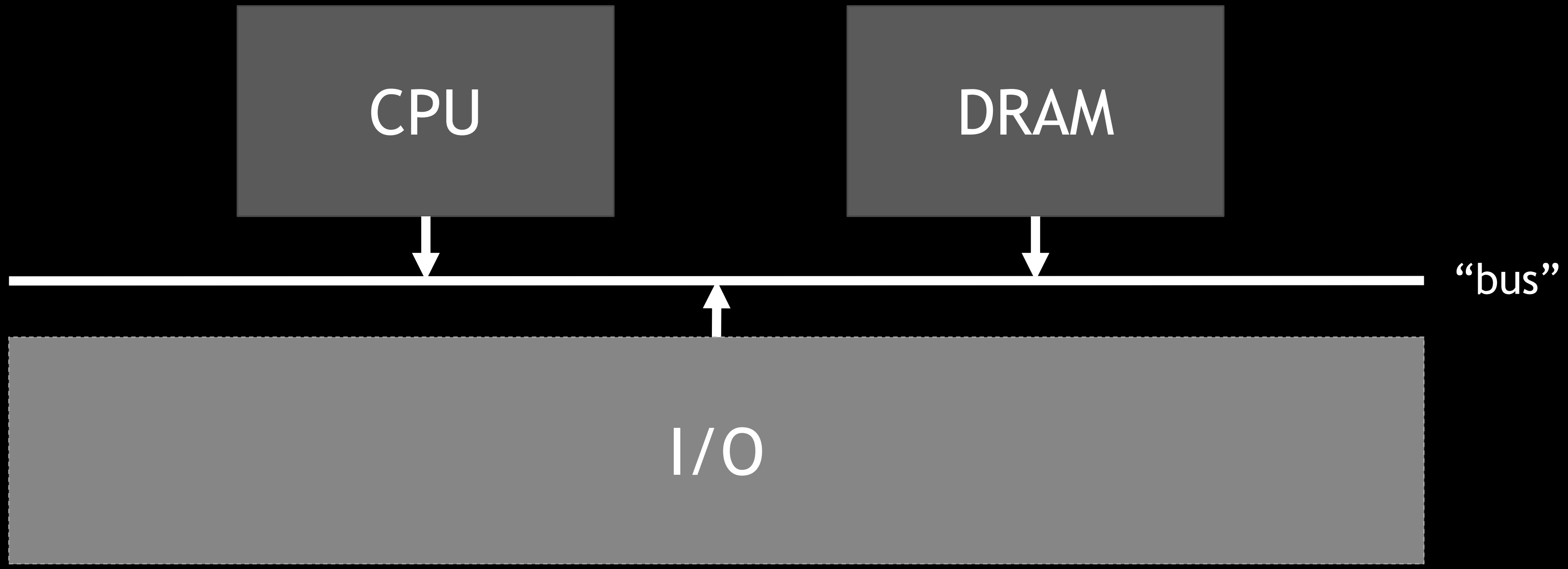
Programmer surprise.

IT’S TOO RISKY IN 2019

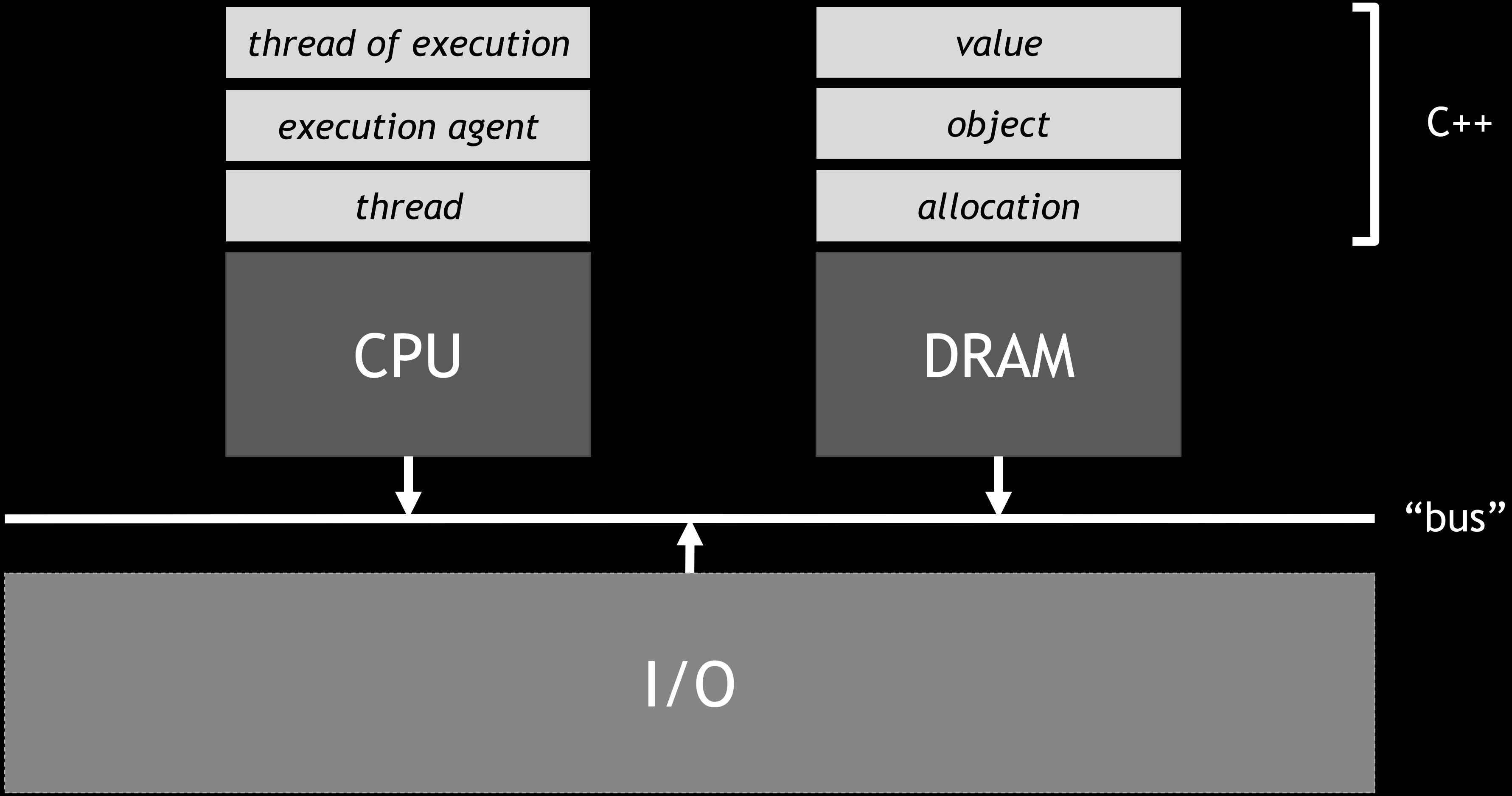
1980’S TOPOLOGY



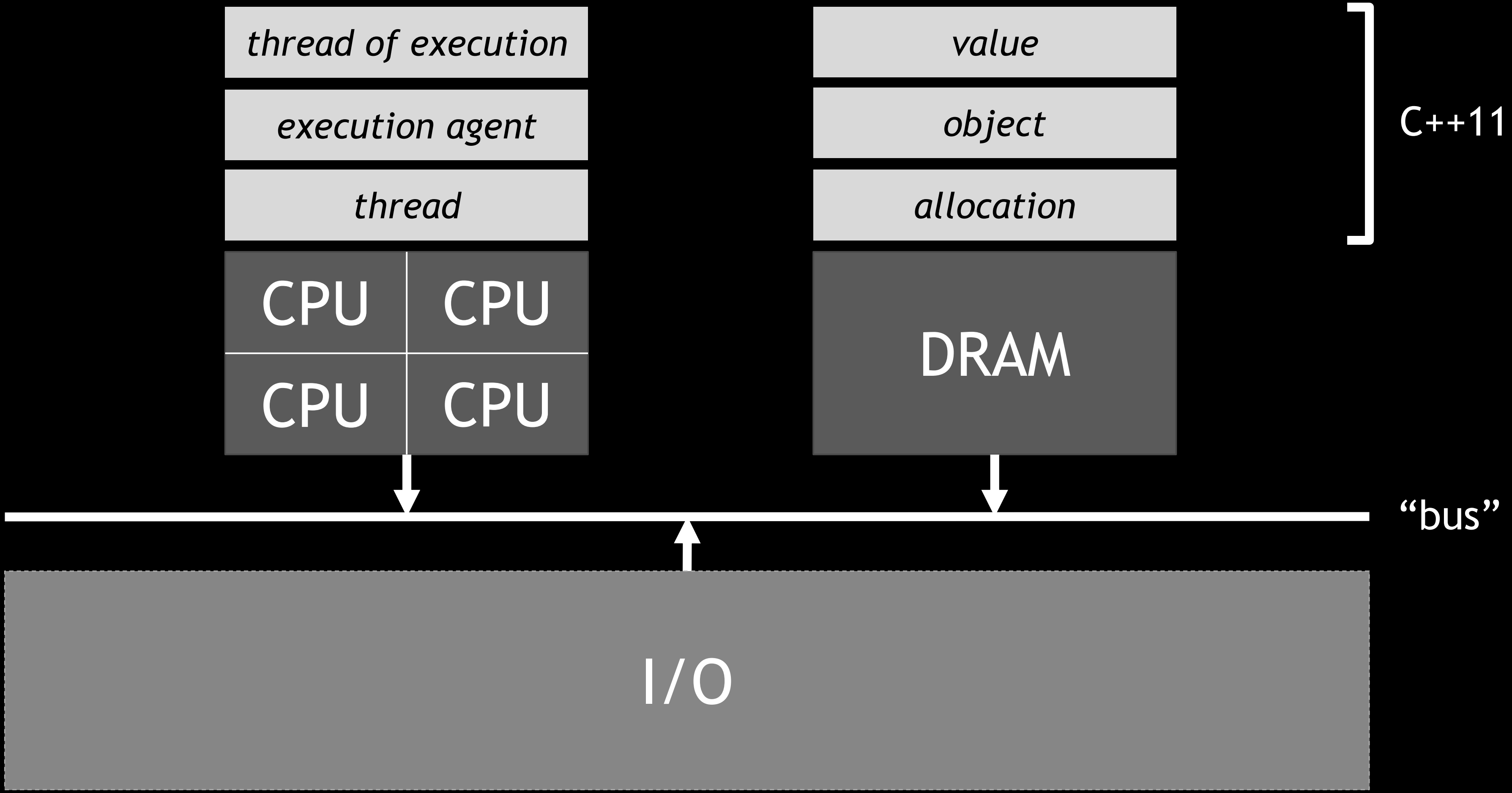
1980'S TOPOLOGY



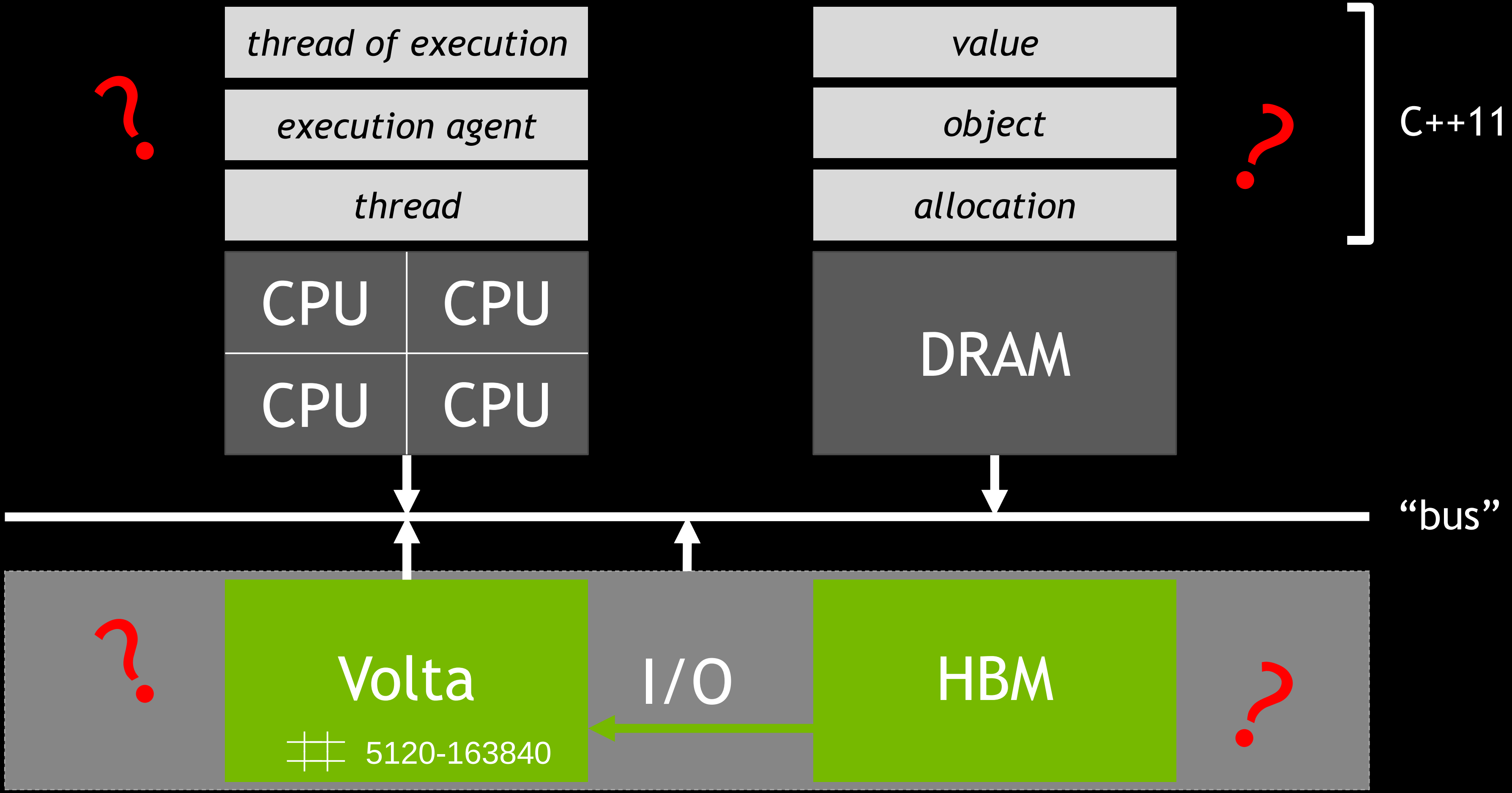
1980'S TOPOLOGY



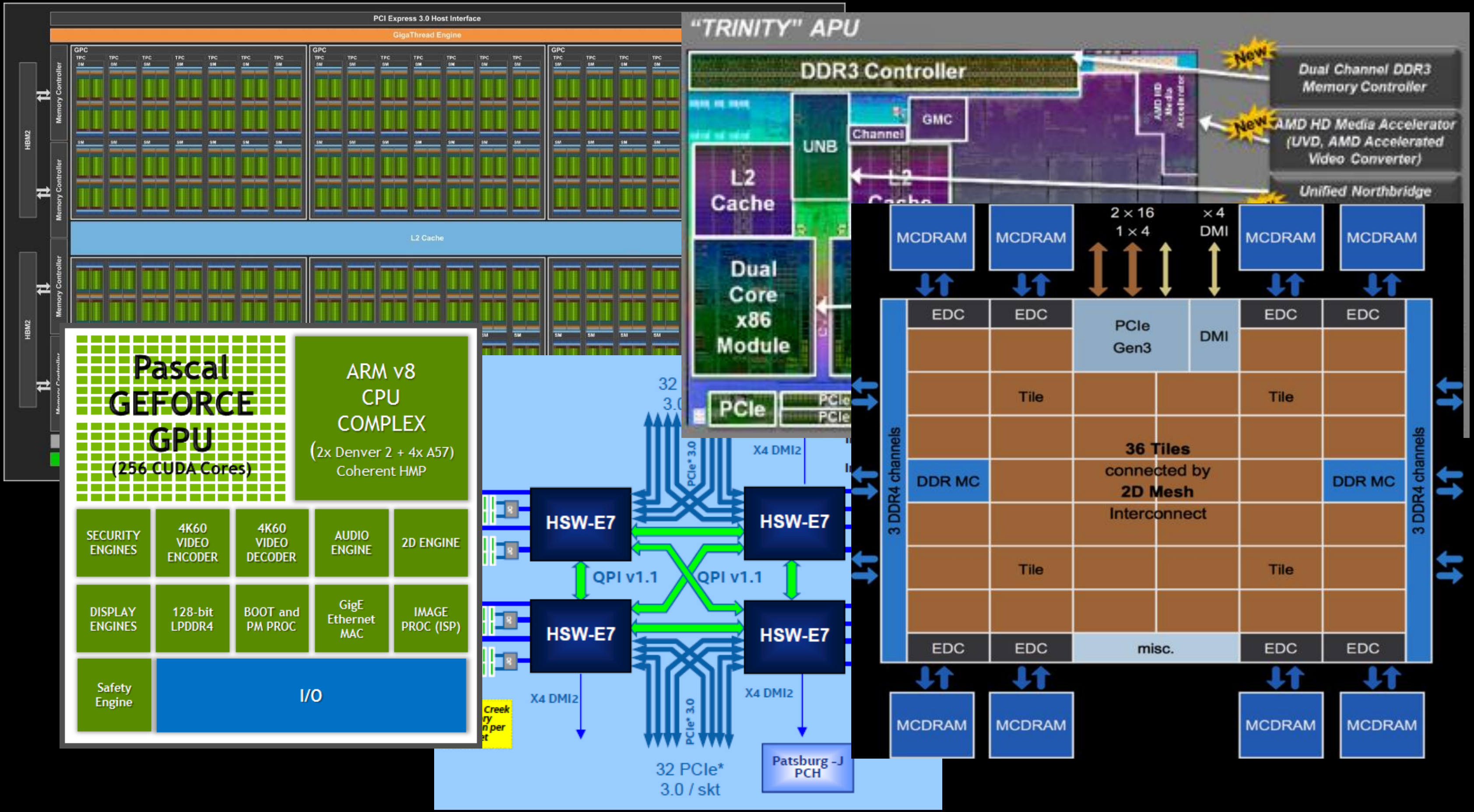
1990-2000'S TOPOLOGY



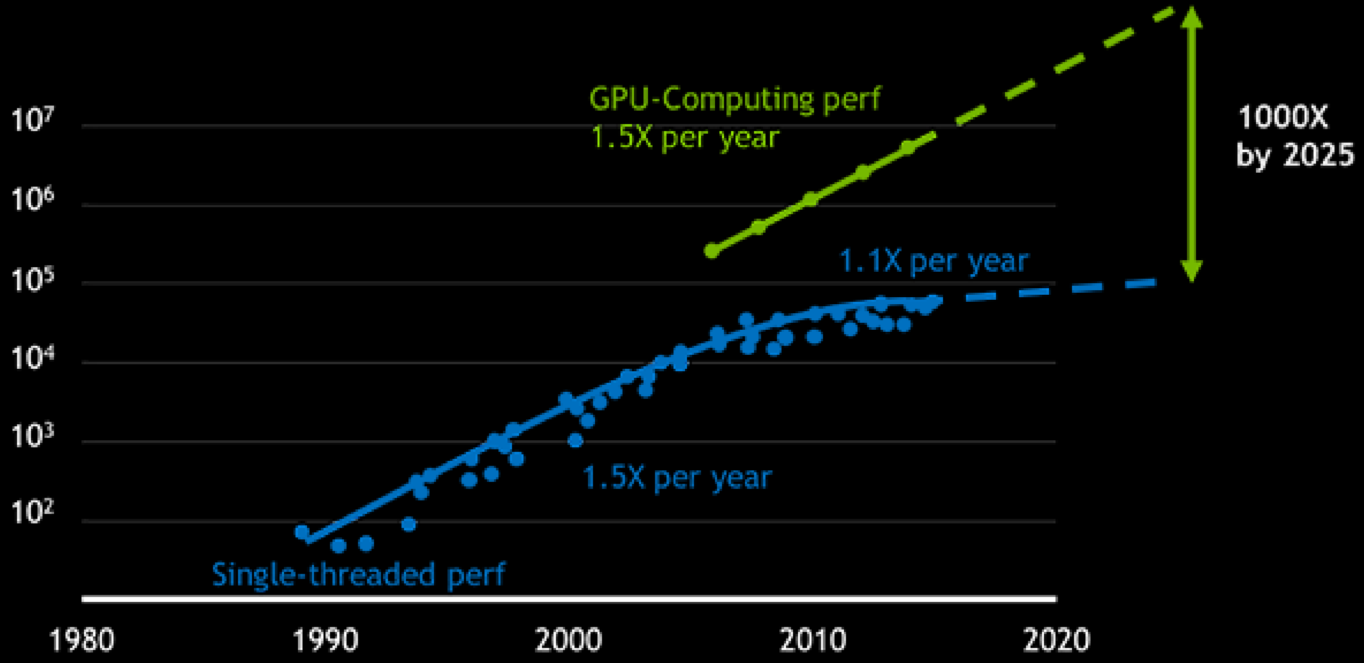
2010'S TOPOLOGY



2010’S TOPOLOGY



WINTER IS COMING



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

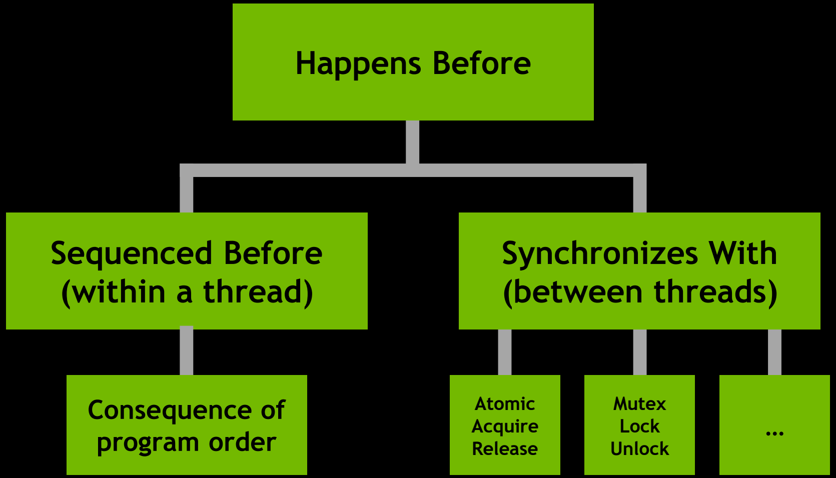
Future silicon performance wins will come from architectural innovation, not transistor density scaling.

WHAT MAKES C++ PORTABLE?

What Makes C++ Portable?

The C++ Execution Model: Memory Model + Forward Progress

- Threads evaluate expressions that access and modify flat storage.
- Evaluation within a thread is driven by sequenced before relations.
- Interactions between threads is driven by synchronizes with relations.
- Forward progress promises eventual termination.

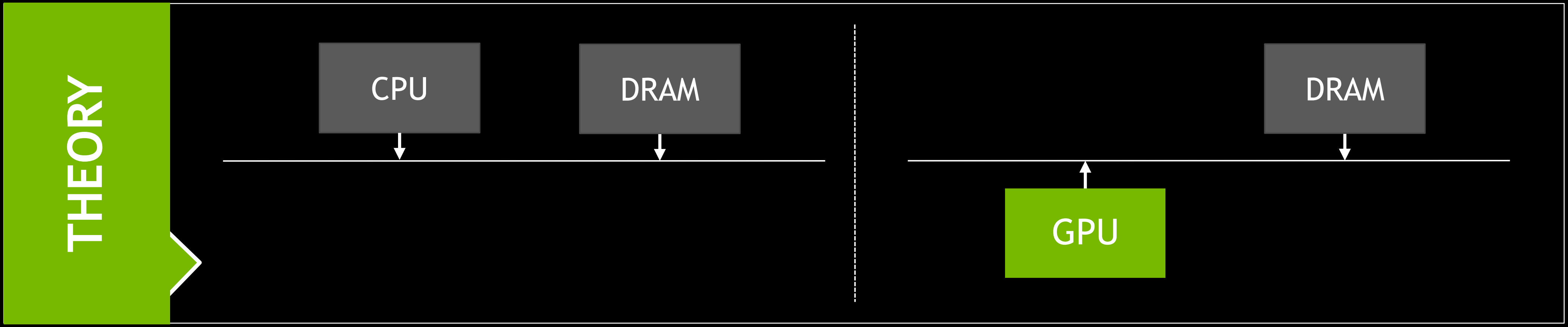

Happens Before
Sequenced Before (within a thread)
Synchronizes With (between threads)
Consequence of program order
Atomic Acquire Release
Mutex Lock Unlock
...

Modern NVIDIA GPUs

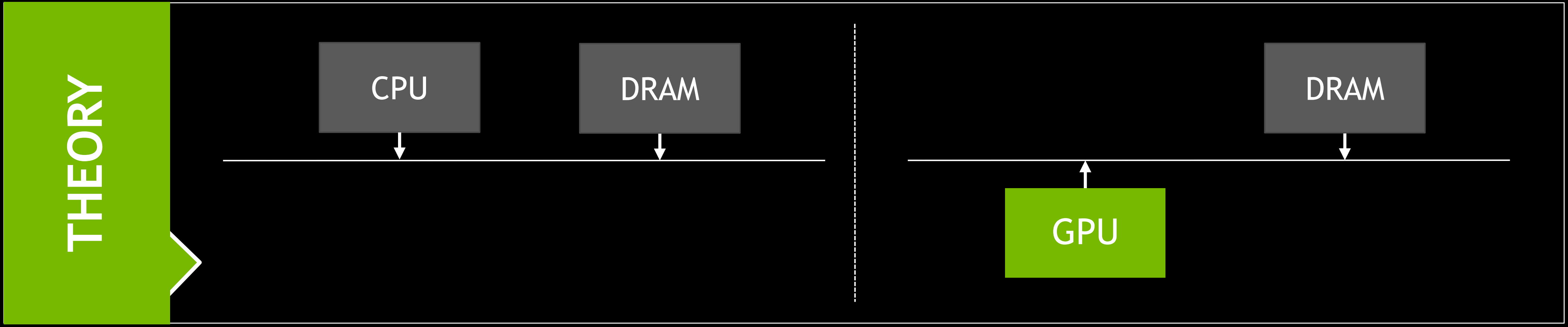
implement the C++ execution model.

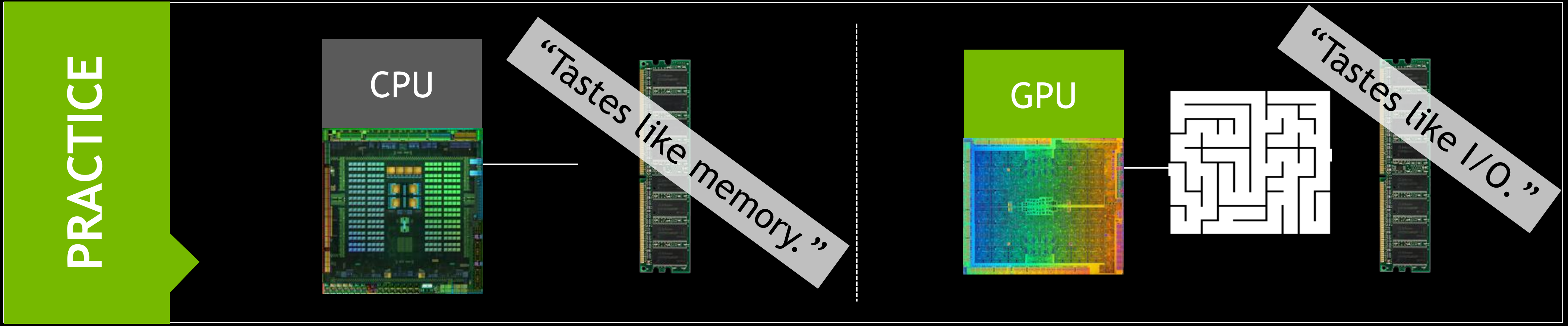
We spent transistors to get there.

COHERENCY

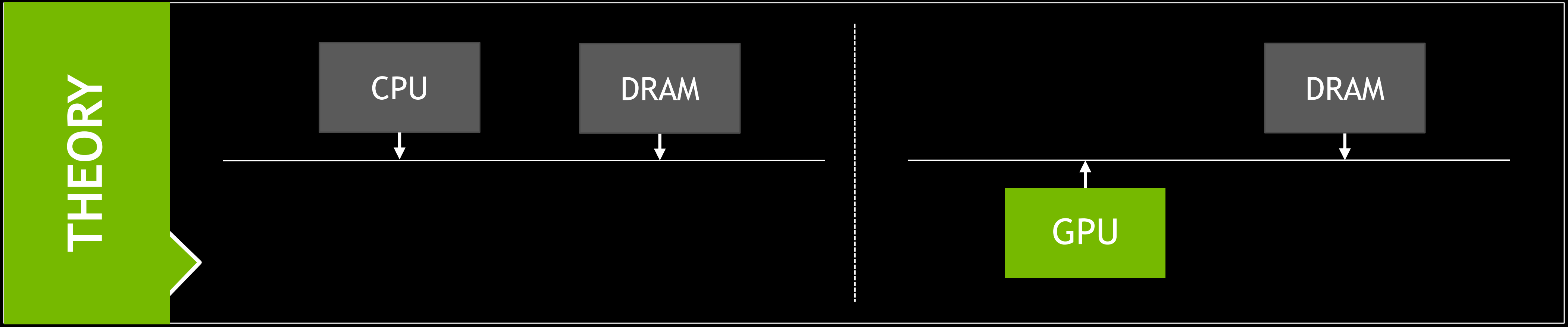


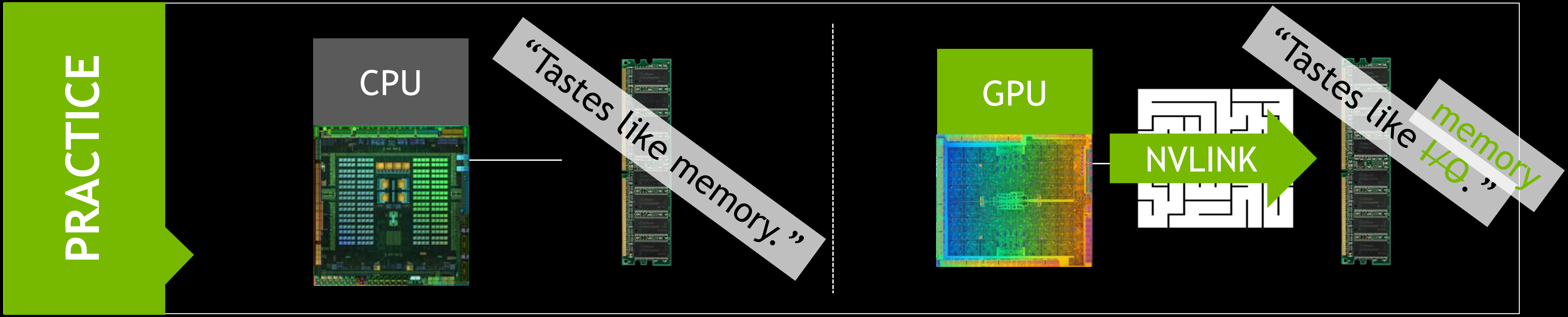
COHERENCY


THEORY
CPU
DRAM
DRAM
GPU


PRACTICE
CPU
“Tastes like memory.”
GPU
“Tastes like I/O.”

COHERENCY

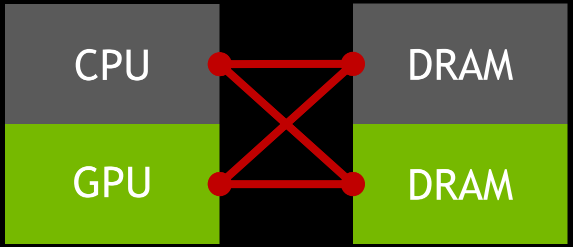

THEORY
CPU
DRAM
DRAM
GPU


PRACTICE
CPU
"Tastes like memory."
GPU
NVLINK
"Tastes like ~~I/O~~ memory."

COHERENCY

GPU ARCH	CUDA	X86	ARM & POWER
Tesla & Fermi	1+	`cudaMalloc` & `cudaMemcpy`	
Kepler & Maxwell	6+	`cudaMallocManaged` (Symmetric Heap)	
Pascal, Volta & Turing	8+	`cudaMallocManaged` (paging)	`cudaMallocManaged` (NVLink)
“Tastes like memory.”	Linux HMM	`malloc` (paging)	`malloc` (NVLink)

CONSISTENCY



:

C++11	load	store	exchange	fence
(not atomic)	✓	✓	-	-
relaxed	✓	✓	✓	-
consume	▼	-	▼	-
acquire	✓	-	✓	✓
release	-	✓	✓	✓
acq_rel	-	-	✓	✓
seq_cst	✓	✓	✓	✓

Completely new hardware memory model in Volta, outline similar to POWER.

Everything but consume is accelerated. Stay tuned about consume.

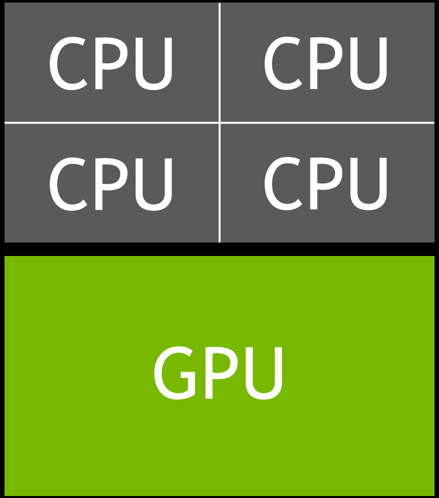
See the PTX 6.0 ISA programming guide, chapter 8.

PROGRESS

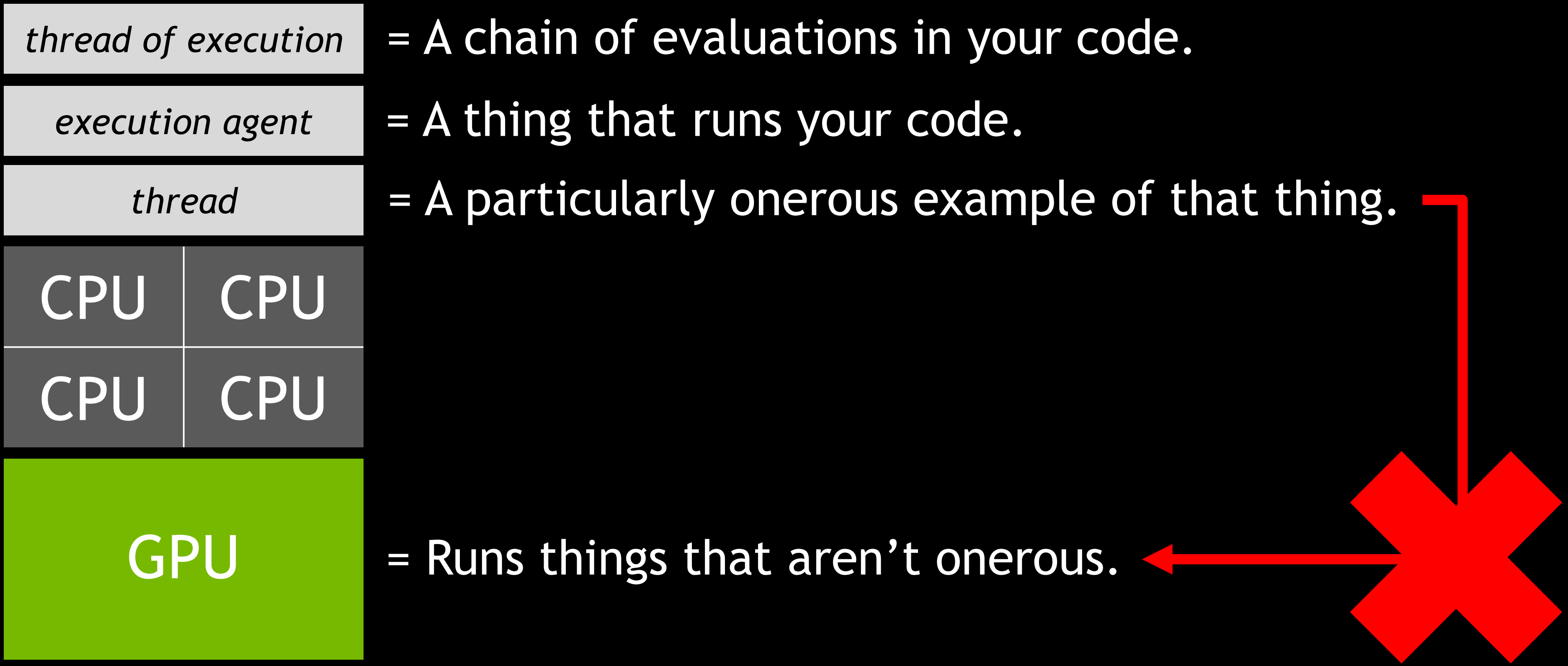
thread of execution = A chain of evaluations in your code.

execution agent = A thing that runs your code.

thread = A particularly onerous example of that thing.



PROGRESS



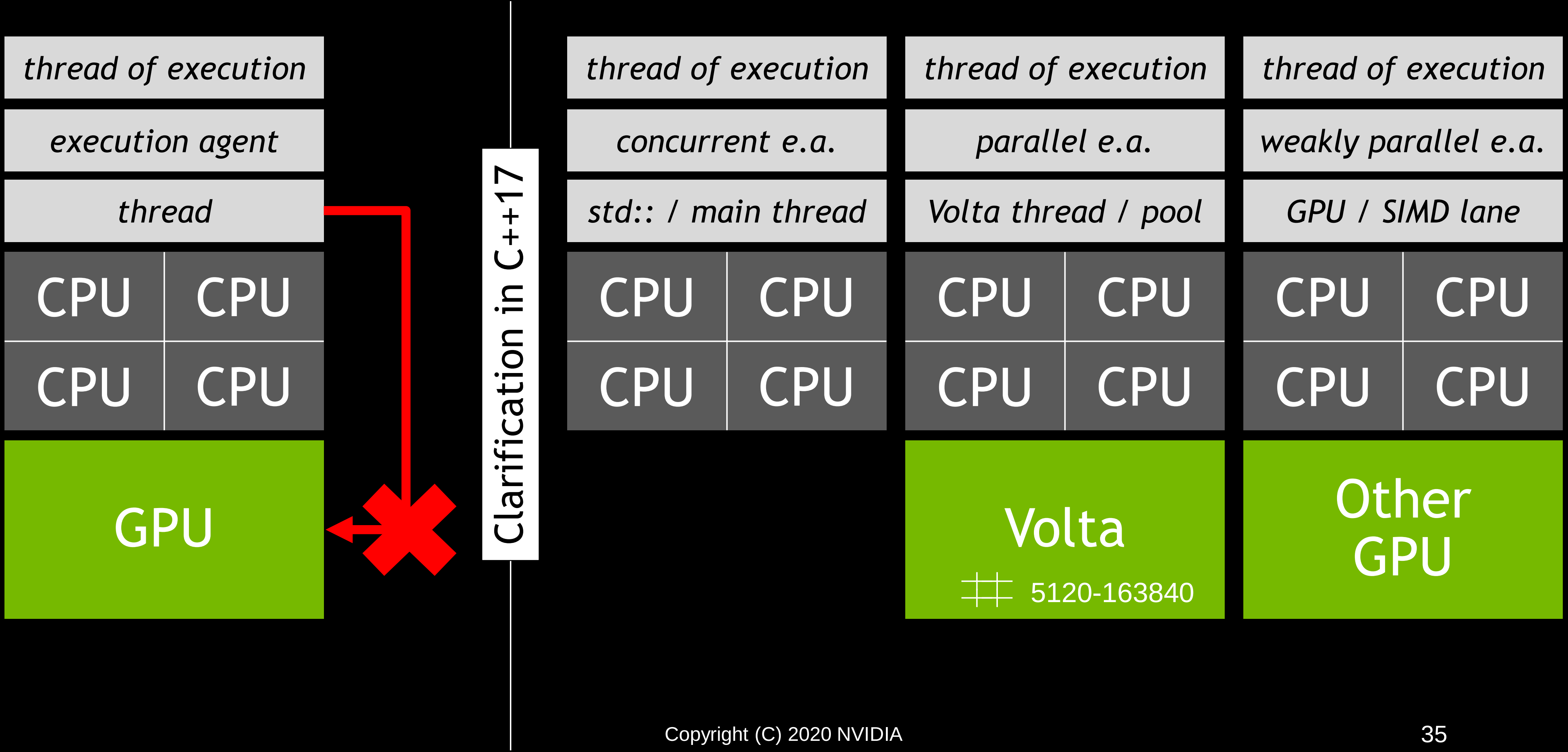
thread of execution = A chain of evaluations in your code.

execution agent = A thing that runs your code.

thread = A particularly onerous example of that thing.

GPU = Runs things that aren’t onerous.

PROGRESS


thread of execution
execution agent
thread
CPU
CPU
CPU
CPU
GPU
Clarification in C++17
thread of execution
concurrent e.a.
std:: / main thread
CPU
CPU
CPU
CPU
thread of execution
parallel e.a.
Volta thread / pool
CPU
CPU
CPU
CPU
Volta
# 5120-163840
thread of execution
weakly parallel e.a.
GPU / SIMD lane
CPU
CPU
CPU
CPU
Other GPU

PROGRESS

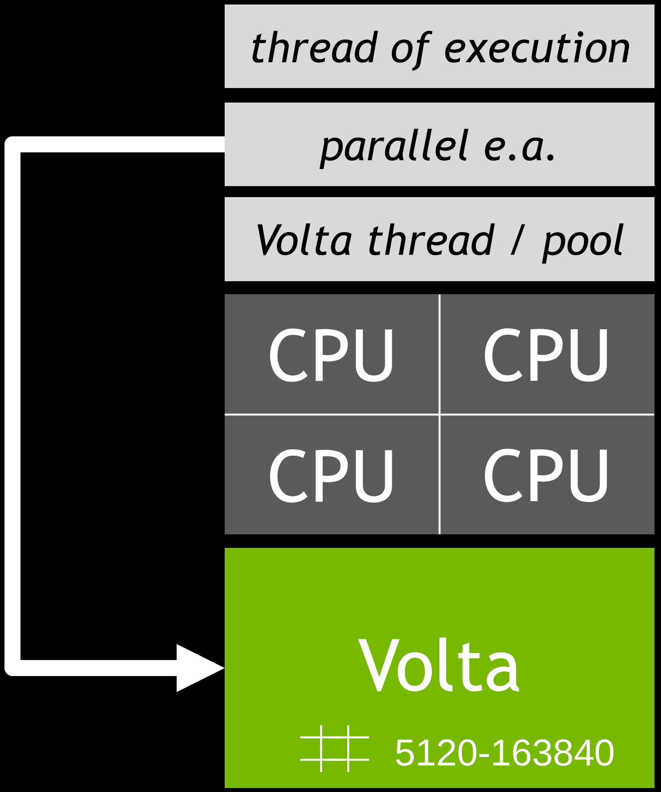
- Concurrent Forward Progress: The thread will make progress, regardless of whether other threads are making progress.
- Parallel Forward Progress: Once the thread has executed its first execution step, the thread will make progress.
- Weakly Parallel Forward Progress: The thread is not guaranteed to make progress.

PROGRESS

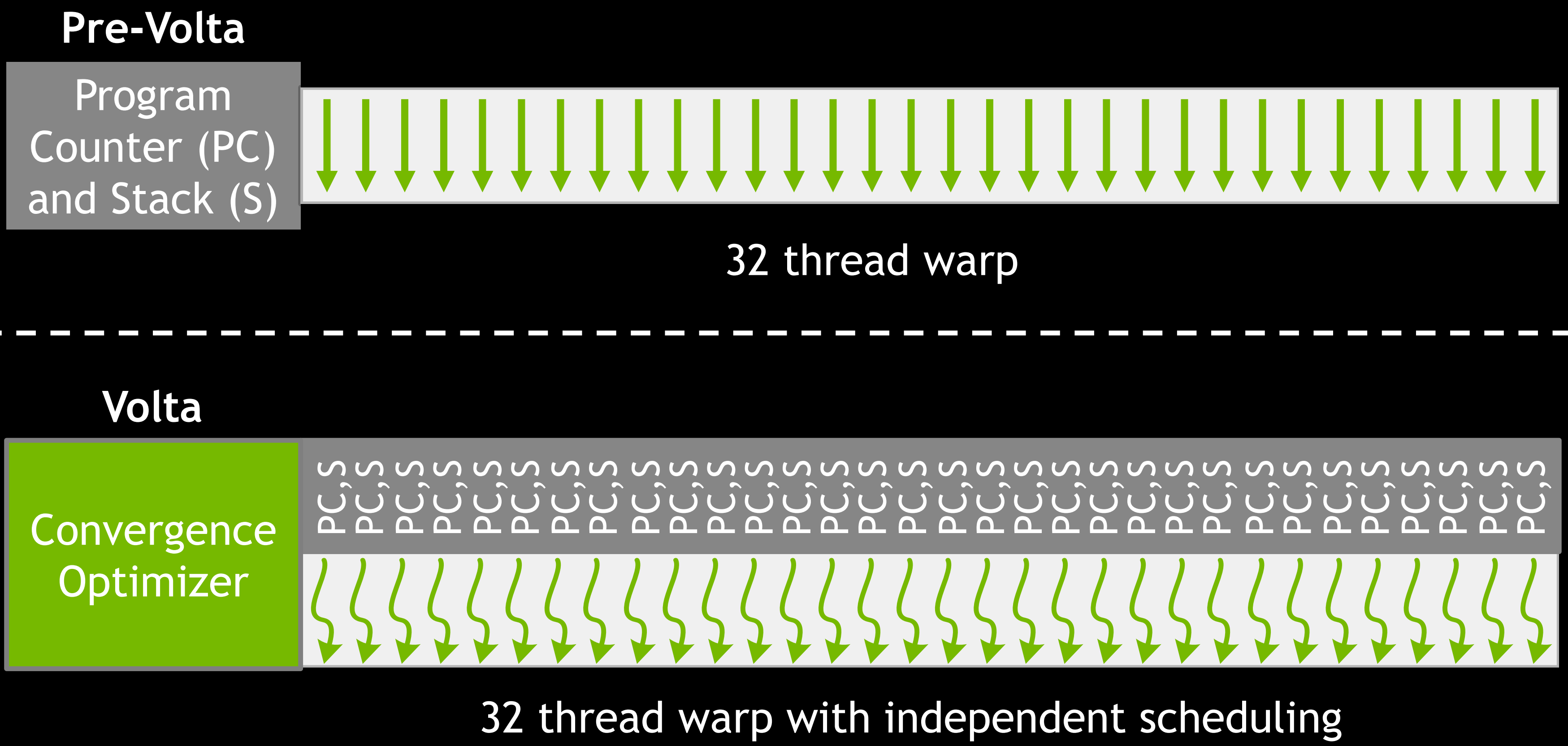
Not "business as usual".

A concerted effort by dedicated engineers.

Volta+ is alone of its kind.

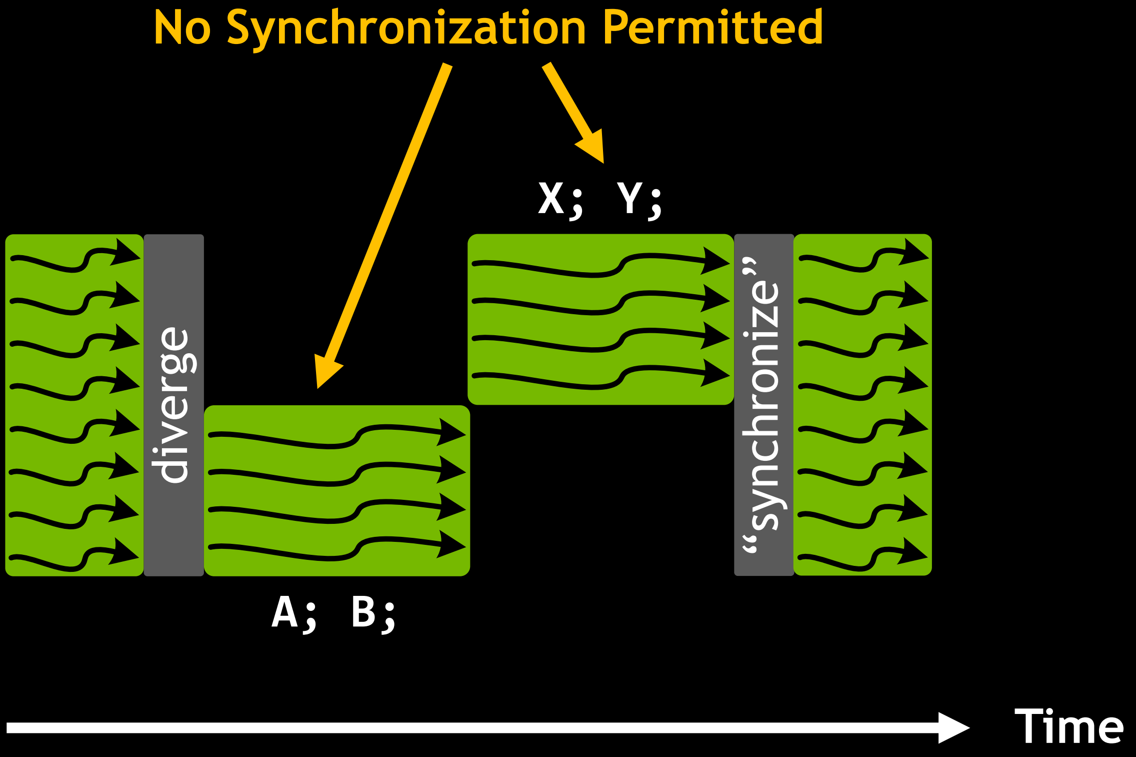


WARP IMPLEMENTATION


Pre-Volta
Program Counter (PC) and Stack (S)
32 thread warp
Volta
Convergence Optimizer
PC,S
32 thread warp with independent scheduling

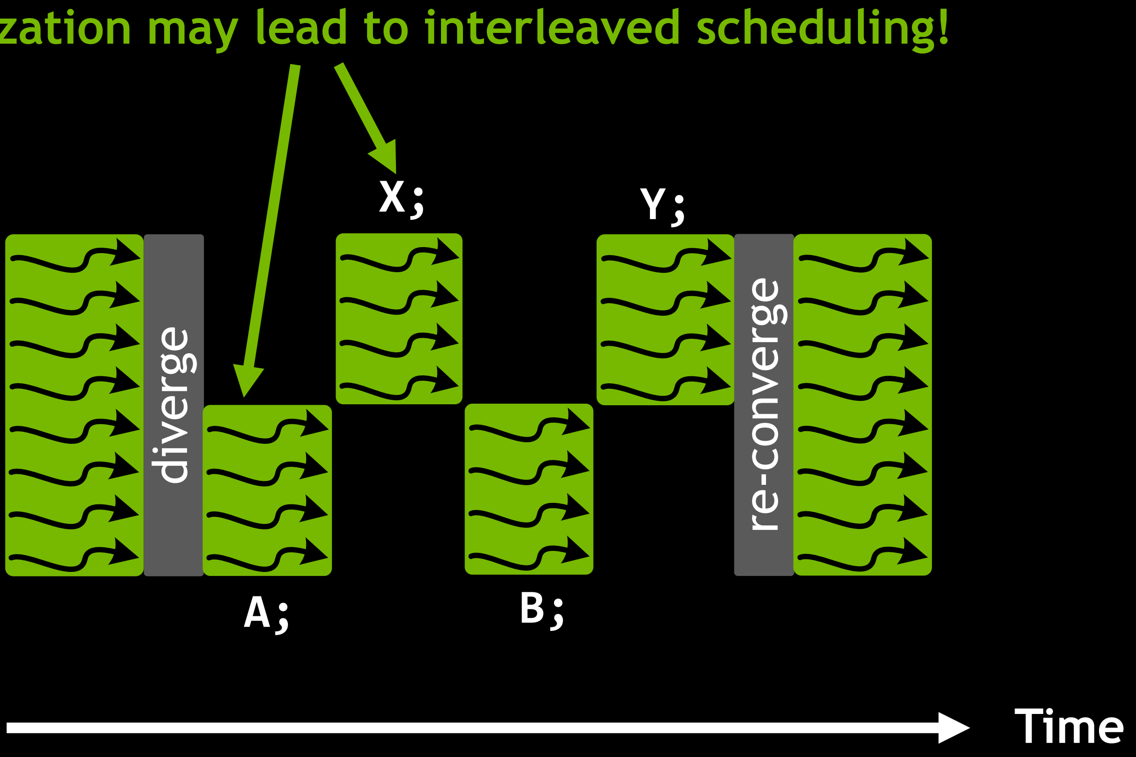
PASCAL WARP EXECUTION MODEL

```
if (threadIdx.x < 4) {
   A;
   ~~__syncwarp();~~
   B;
} else {
   X;
   ~~__syncwarp();~~
   Y;
}
```



VOLTA WARP EXECUTION MODEL

```
if (threadIdx.x < 4) {
   A;
   __syncwarp();
   B;
} else {
   X;
   __syncwarp();
   Y;
}
```



SCORECARD

Problem	Disposition
Memory Coherency	Supported since Pascal
Memory Consistency	Supported since Volta in PTX `cuda::std::atomic<>` exposure forthcoming
Forward Progress Guarantees	Supported since Volta Clarified in C++17

Modern NVIDIA GPUs

implement the C++ execution model.

We spent transistors to get there.

CUDA C++ IS A SUPERSET OF ISO C++

Host processors can use **alone**	All processors can use **isolated**	All processors can use **together**
`throw` `catch` `typeid` `dynamic_cast` `thread_local` `std::`	`virtual` functions, function pointers, lambdas	<rest of ISO C++>, `cuda::std::`†

† Coming in a future CUDA release.

libcu++

The CUDA C++ Standard Library

Opt-in, heterogeneous, incremental C++ standard library for CUDA.

Open source; port of LLVM's libc++; contributing upstream.

Version 1 (CUDA 10.2): `<atomic>` (Pascal+), `<type_traits>`.

Version 2 (CUDA next): `atomic<T>::wait/notify`, `<barrier>`, `<latch>`, `<counting_semaphore>` (all Volta+), `<chrono>`, `<ratio>`, `<functional>` minus function.

Future priorities: `atomic_ref<T>`, `<complex>`, `<tuple>`, `<array>`, `<utility>`, `<cmath>`, string processing, …

libcu++ is the

opt-in,

heterogeneous,

incremental

CUDA C++ Standard Library.

Opt-in

Does not interfere with or replace your host standard library.

```
// ISO C++, __host__ only.
#include <atomic>
std::atomic<int> x;

// CUDA C++, __host__ __device__.
// Strictly conforming to the ISO C++.
#include <cuda/std/atomic>
cuda::std::atomic<int> x;

// CUDA C++, __host__ __device__.
// Conforming extensions to ISO C++.
#include <cuda/atomic>
cuda::atomic<int, cuda::thread_scope_block> x;
```

Heterogeneous

Copyable/Movable objects can migrate between host & device.

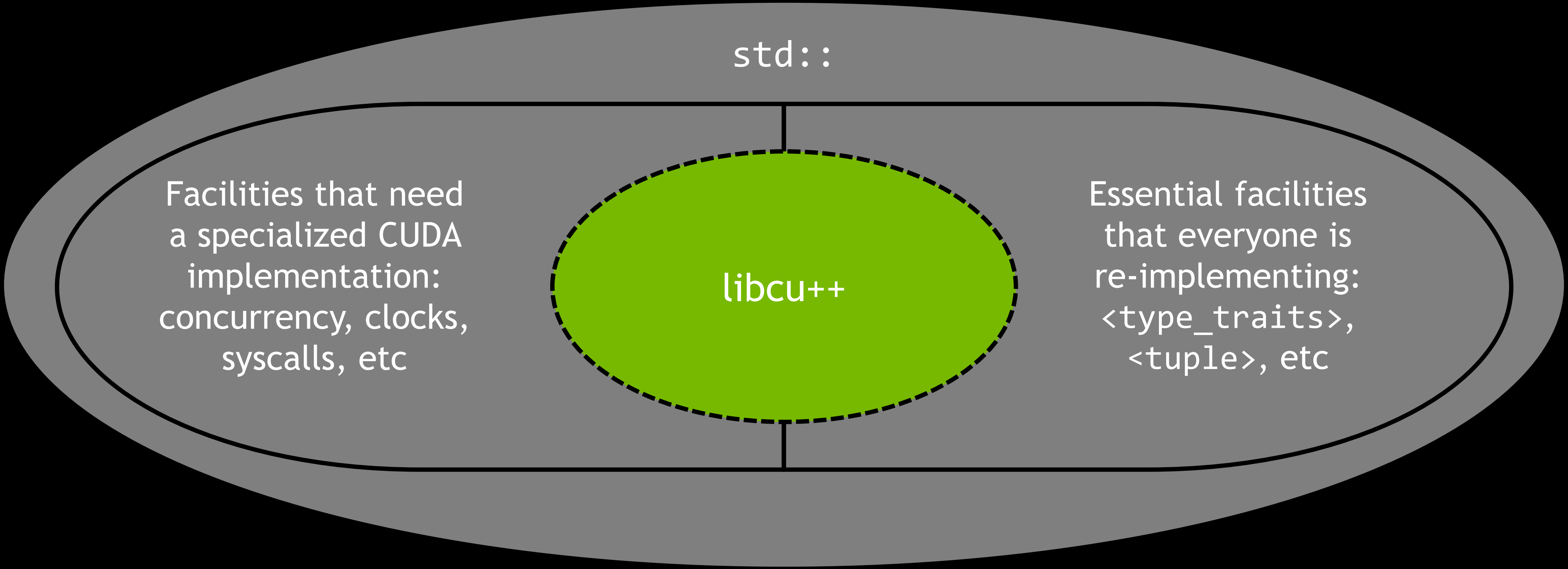
Host & device can call all (member) functions.

Host & device can concurrently use synchronization primitives*.

*: Synchronization primitives must be in managed memory and be declared with `cuda::std::thread_scope_system`.

Incremental

Not a complete standard library today; each release will add more.



std::
Facilities that need a specialized CUDA implementation: concurrency, clocks, syscalls, etc
libcu++
Essential facilities that everyone is re-implementing: <type_traits>, <tuple>, etc

Based on LLVM's libc++

Forked from LLVM's libc++.

License: Apache 2.0 with LLVM Exception.

NVIDIA is already contributing back to the community:

- Freestanding `atomic<T>`: reviews.llvm.org/D56913
- C++20 synchronization library: reviews.llvm.org/D68480