FUTURE OF ISO AND CUDA C++
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**

- C++0x/11
- Library TR1
- Decimal TR (not merged)
- Library TR2 (deferred to post-C++0x, then replaced by File System TS)
- Math Special Functions IS

**TSes: feature branches for separate release & then merge**

- C++14
  - File System
  - Lib Fundamentals 1
  - Parallelism 1
  - Concepts
  - Concurrency 1
  - Arrays (abandoned)
- C++17
  - Networking
  - Lib Fundamentals 2
  - Parallelism 2
  - Ranges
  - Modules
  - Coroutines
- C++20
  - Reflection
  - Lib Fundamentals 3
  - Concurrency 2

---

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](https://isocpp.org/std/status)

Copyright (C) 2020 NVIDIA
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

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

```cpp
VS
void compute(executor auto ex, ...) {
    execute(ex, ...);
}
```
C++23 Executors

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

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

sender auto begin = schedule(ex);
sender auto hi_again = then(begin, [](std::ostream&) { 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);
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

```cpp
#include <C++>

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.

Copyright (C) 2020 NVIDIA
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?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Relevant in the 80s/90s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-8-bit char</td>
<td>✓</td>
</tr>
<tr>
<td>Noncommittal sizeof</td>
<td>✓</td>
</tr>
<tr>
<td>Non-2’s comp. int</td>
<td>✓</td>
</tr>
<tr>
<td>Non-IEEE float</td>
<td>✓</td>
</tr>
<tr>
<td>Segmented memory</td>
<td>✓</td>
</tr>
<tr>
<td>Non-endian pointers</td>
<td>✓</td>
</tr>
</tbody>
</table>
## WHAT MAKES C++ PORTABLE?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Relevant in the 80s/90s</th>
<th>Relevant Today</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-8-bit <code>char</code></td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Noncommittal <code>sizeof</code></td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Non-2’s comp. <code>int</code></td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Non-IEEE <code>float</code></td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>Segmented memory</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Non-endian pointers</td>
<td>✅</td>
<td></td>
</tr>
</tbody>
</table>

Copyright (C) 2020 NVIDIA
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

CPU

DRAM

“bus”
1980’S TOPOLOGY

CPU

DRAM

“bus”

I/O
1980’S TOPOLOGY

- thread of execution
- execution agent
- thread
- value
- object
- allocation

CPU

DRAM

"bus"

I/O

C++

Copyright (C) 2020 NVIDIA
2010’S TOPOLOGY

- thread of execution
- execution agent
- thread
- CPU
- CPU
- CPU
- CPU
- DRAM
- value
- object
- allocation

"bus"

Volta

I/O

HBM

Copyright (C) 2020 NVIDIA
2010’S TOPOLOGY
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.

```
#include <C++>
```
Modern NVIDIA GPUs implement the C++ execution model.

We spent transistors to get there.
COHERENCY

THEORY

CPU \arrow{down}

DRAM

DRAM

CPU

GPU

PRACTICE

"Tastes like memory."

CPU

"Tastes like I/O."

GPU
COHERENCY

THEORY

CPU → DRAM → GPU → DRAM

PRACTICE

“Tastes like memory.”

GPU

“Tastes like memory.”

Copyright (C) 2020 NVIDIA
## COHERENCY

<table>
<thead>
<tr>
<th>GPU ARCH</th>
<th>CUDA</th>
<th>X86</th>
<th>ARM &amp; POWER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla &amp; Fermi</td>
<td>1+</td>
<td>cudaMalloc &amp; cudaMemcpy</td>
<td></td>
</tr>
<tr>
<td>Kepler &amp; Maxwell</td>
<td>6+</td>
<td>cudaMallocManaged (Symmetric Heap)</td>
<td></td>
</tr>
<tr>
<td>Pascal, Volta &amp; Turing</td>
<td>8+</td>
<td>cudaMallocManaged (paging)</td>
<td>cudaMallocManaged (NVLink)</td>
</tr>
<tr>
<td>“Tastes like memory.”</td>
<td>Linux HMM</td>
<td>malloc (paging)</td>
<td>malloc (NVLink)</td>
</tr>
</tbody>
</table>

Copyright (C) 2020 NVIDIA
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.

<table>
<thead>
<tr>
<th>CPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>CPU</td>
</tr>
</tbody>
</table>

**GPU**
PROGRESS

A chain of evaluations in your code.
A thing that runs your code.
A particularly onerous example of that thing.
Runs things that aren’t onerous.
### PROGRESS

**thread of execution**
- execution agent
- thread

#### thread of execution
- concurrent e.a.
- std:: / main thread

<table>
<thead>
<tr>
<th>CPU</th>
<th>CPU</th>
<th>CPU</th>
</tr>
</thead>
</table>

#### thread of execution
- parallel e.a.
- Volta thread / pool

<table>
<thead>
<tr>
<th>CPU</th>
<th>CPU</th>
<th>CPU</th>
<th>CPU</th>
</tr>
</thead>
</table>

#### thread of execution
- weakly parallel e.a.
- GPU / SIMD lane

<table>
<thead>
<tr>
<th>CPU</th>
<th>CPU</th>
<th>CPU</th>
<th>CPU</th>
</tr>
</thead>
</table>

- Volta
  - 5120-163840

- Other GPU

---

Copyright (C) 2020 NVIDIA
• **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)

Volta
Convergence Optimizer

32 thread warp

32 thread warp with independent scheduling

Copyright (C) 2020 NVIDIA
PASCAL WARP EXECUTION MODEL

No Synchronization Permitted

if (threadIdx.x < 4) {
    A;
    __syncwarp();
    B;
} else {
    X;
    __syncwarp();
    Y;
}
Synchronization may lead to interleaved scheduling!

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Disposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Coherency</td>
<td>Supported since Pascal</td>
</tr>
<tr>
<td>Memory Consistency</td>
<td>Supported since Volta in PTX \prefix{cuda::std::atomic} exposure forthcoming</td>
</tr>
<tr>
<td>Forward Progress Guarantees</td>
<td>Supported since Volta \Clarified in C++17</td>
</tr>
</tbody>
</table>
Modern NVIDIA GPUs implement the C++ execution model.

We spent transistors to get there.
# CUDA C++ IS A SUPERSET OF ISO C++

<table>
<thead>
<tr>
<th>Host processors can use alone</th>
<th>All processors can use isolated</th>
<th>All processors can use together</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>throw</code> <code>catch</code> <code>typeid</code> <code>dynamic_cast</code> <code>thread_local</code> <code>std::</code></td>
<td><code>virtual functions</code> <code>function pointers</code> <code>lambdas</code></td>
<td><code>&lt;rest of ISO C++&gt;</code> <code>cuda::std::†</code></td>
</tr>
</tbody>
</table>

† Coming in a future CUDA release.

COPYRIGHT © 2020 NVIDIA
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 \texttt{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