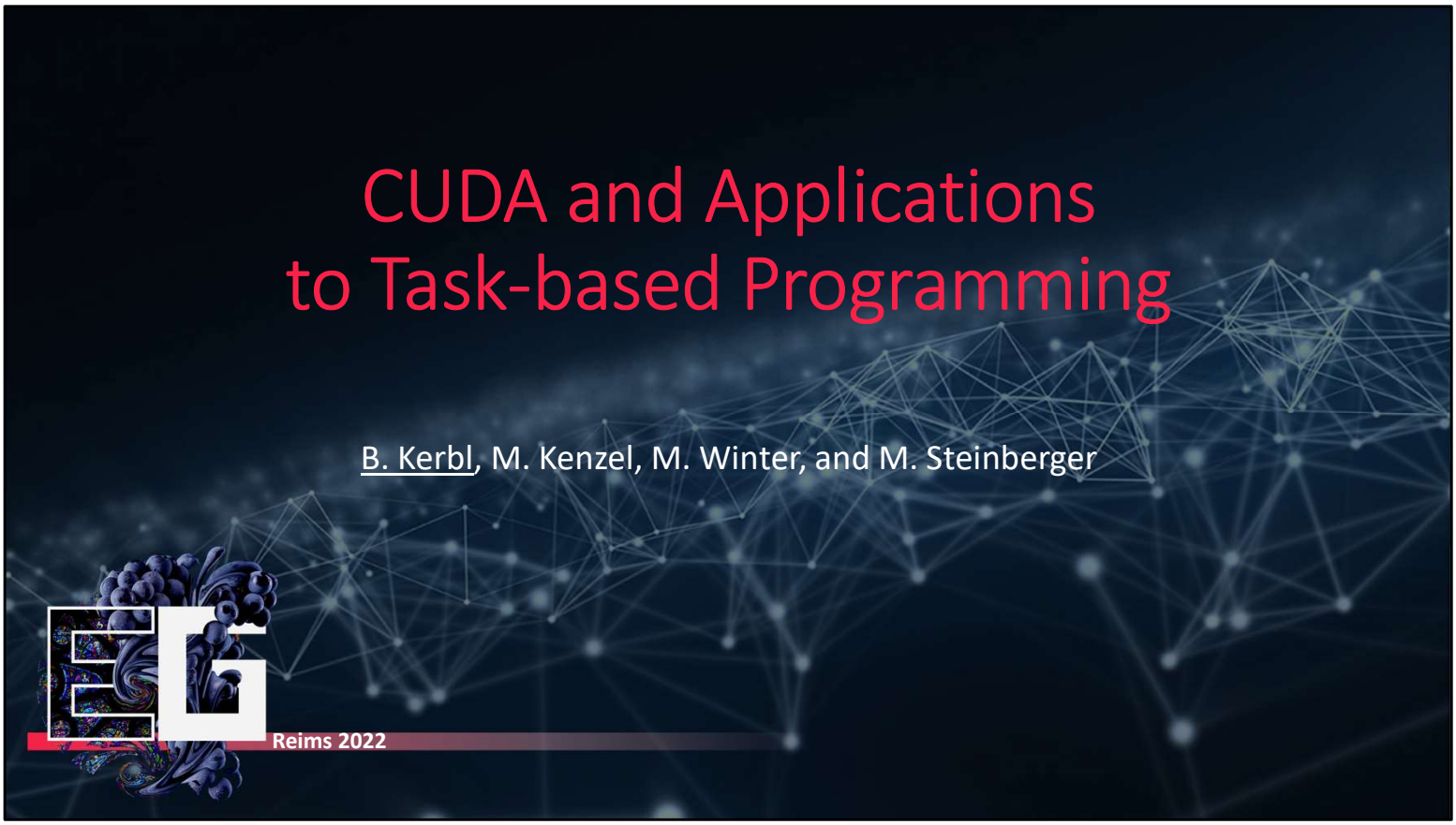


These are the course notes for the third part of the tutorial on “CUDA and Applications to Task-based Programming”, as presented at the Eurographics conference 2022. In this part, we treat advanced mechanisms of CUDA that were not covered by earlier parts, novel features of recent toolkits and architectures, as well as overall trends and caveats for future developments.

Tutorial Resources

Reims 2022

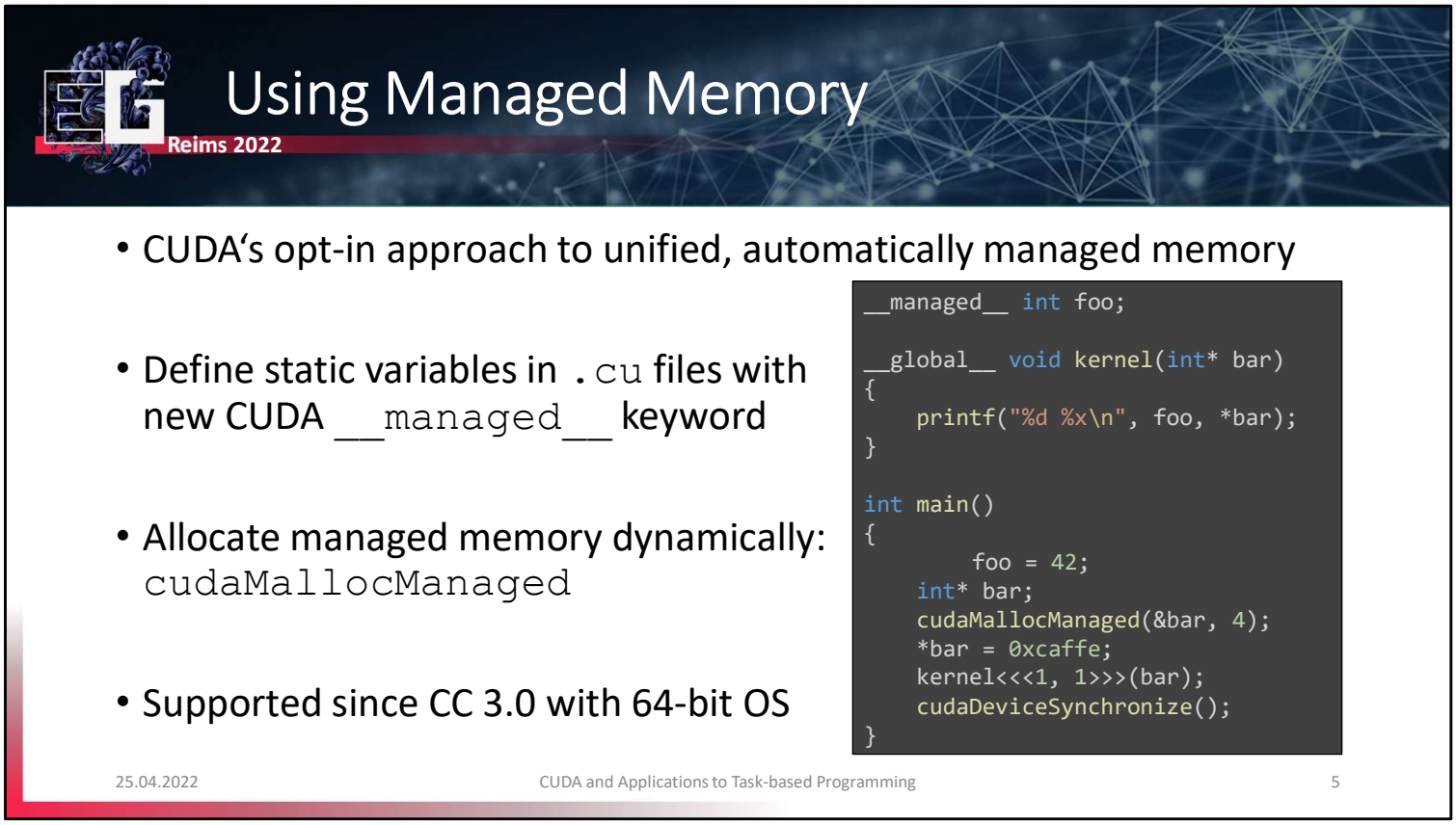
- We have prepared course notes and code samples for this tutorial
  - Course notes include presented slides with additional notes
  - Code samples include buildable versions of the code in the tutorial
    - If you want to try them as you go along, feel free to fetch and build them now
    - Important: CMake 3.20 required!
- May also be extended in the future to stay up to date ☺
- Find them here: https://cuda-tutorial.github.io

The first topic that we want to consider in this portion of the tutorial is CUDA‘s opt-in approach for unified memory between host and device, managed memory.

Controlling Global Memory

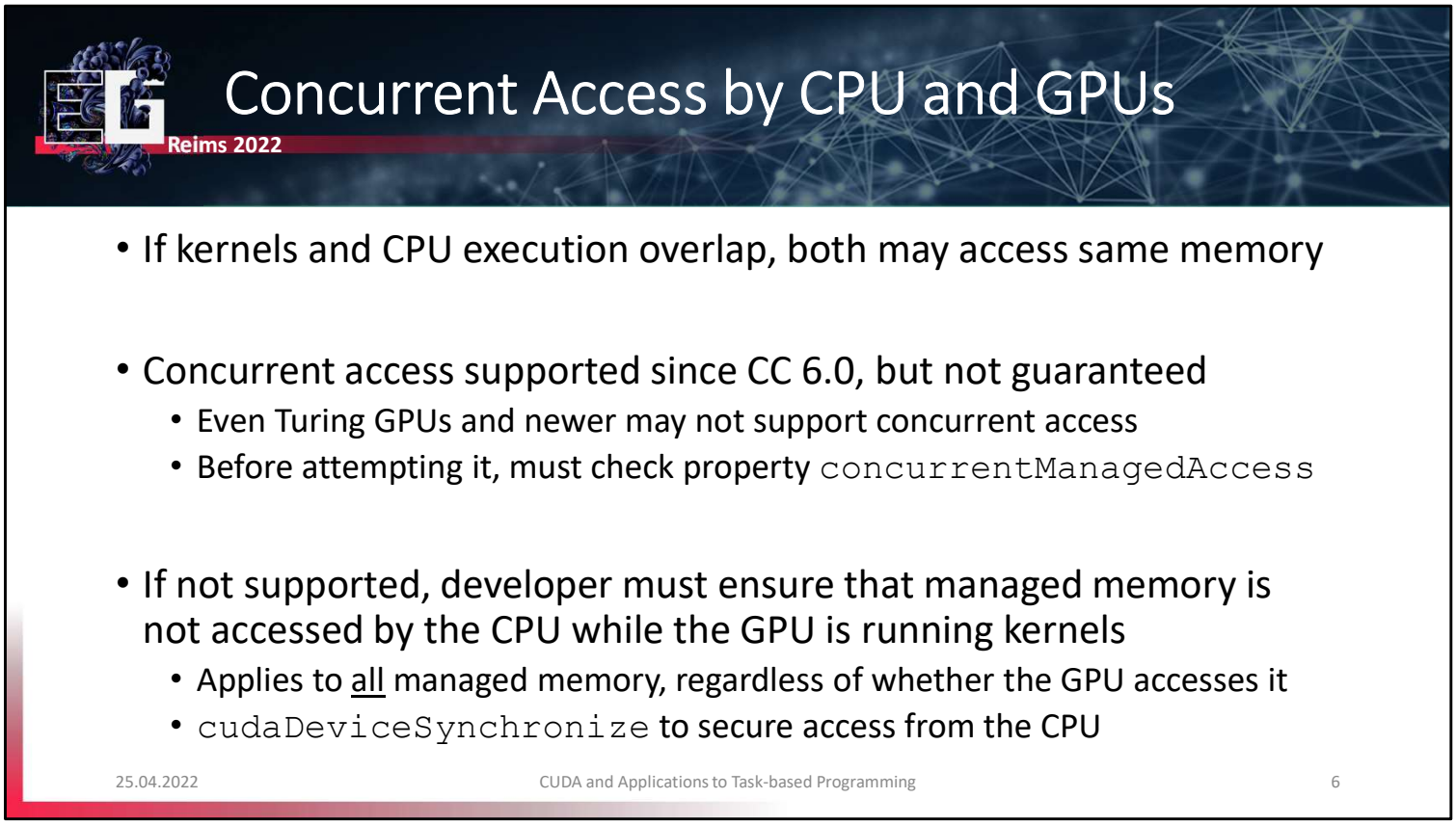
- Static allocation with `__device__` variables declared in source file
- Dynamic management of memory on host
  - Allocate with `cudaMalloc`
  - Eventually, free with `cudaFree`
- To move data between CPU and GPU, use `cudaMemcpy`, `cudaMemcpyToSymbol`, `cudaMemcpyFromSymbol`, …

```
__device__ int foo = 42;

__global__ void allocKernel()
{
    int* foobar = new int[32];
}

int main()
{
    int* bar;
    cudaMalloc(&bar, 4);
    cudaFree(bar);
}
```

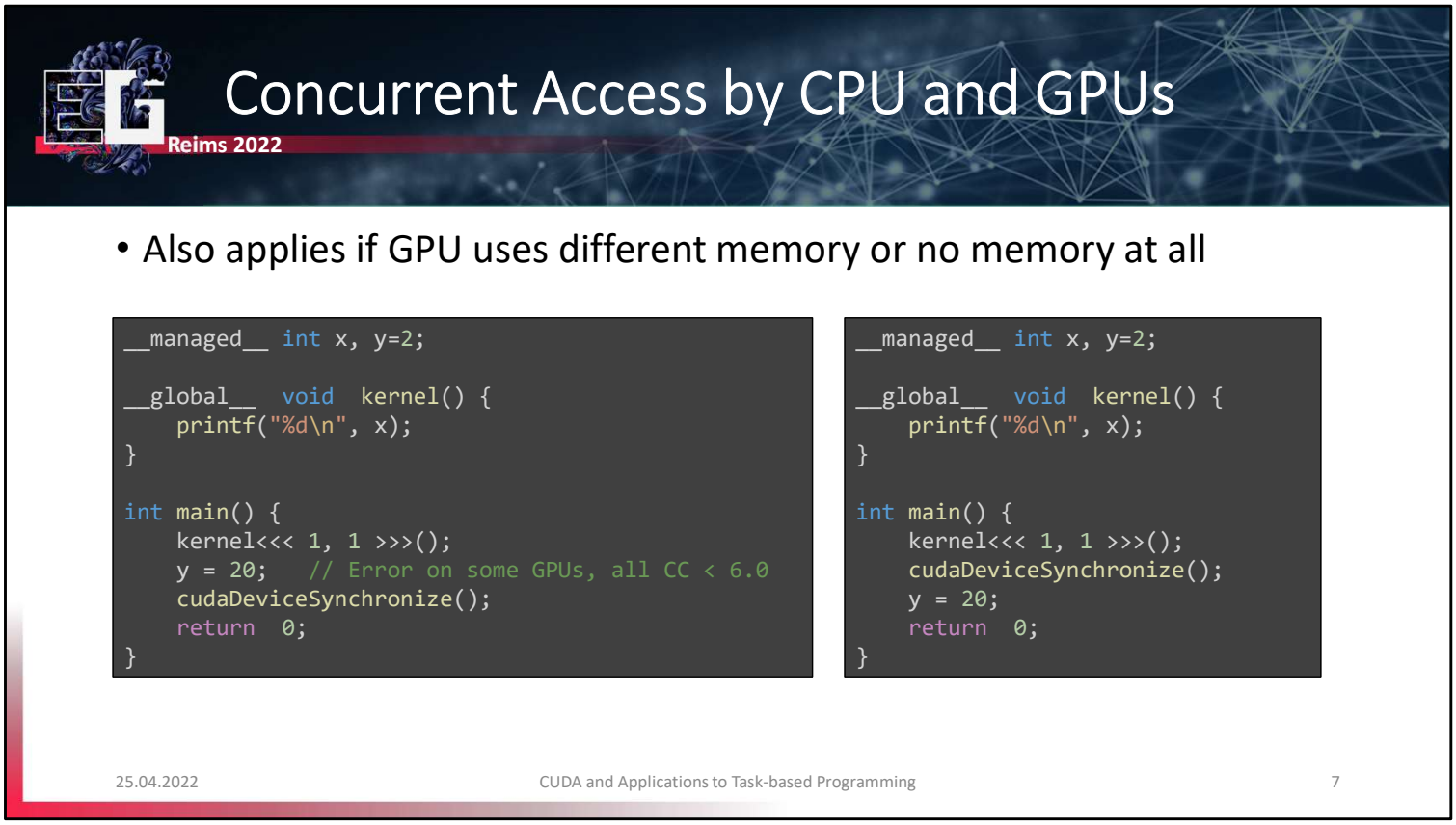


Ever since compute capability 3.0 (Kepler), CUDA has had support for the basic concept of unified memory. The methods for managing it allow for a significant amount of control, even on devices where it is not supported directly by the system allocators. The fundamental additions to the CUDA architecture that managed memory provides are the __managed__ keyword for defining variables in memory, as well as the cudaMallocManaged method to allocate storage on the host side. The managed memory will automatically be migrated to the location where it is accessed, without explicit commands to trigger the transfer. This solution decouples the handle to a memory range from its actual physical storage, which is transient and may change multiple times during execution.

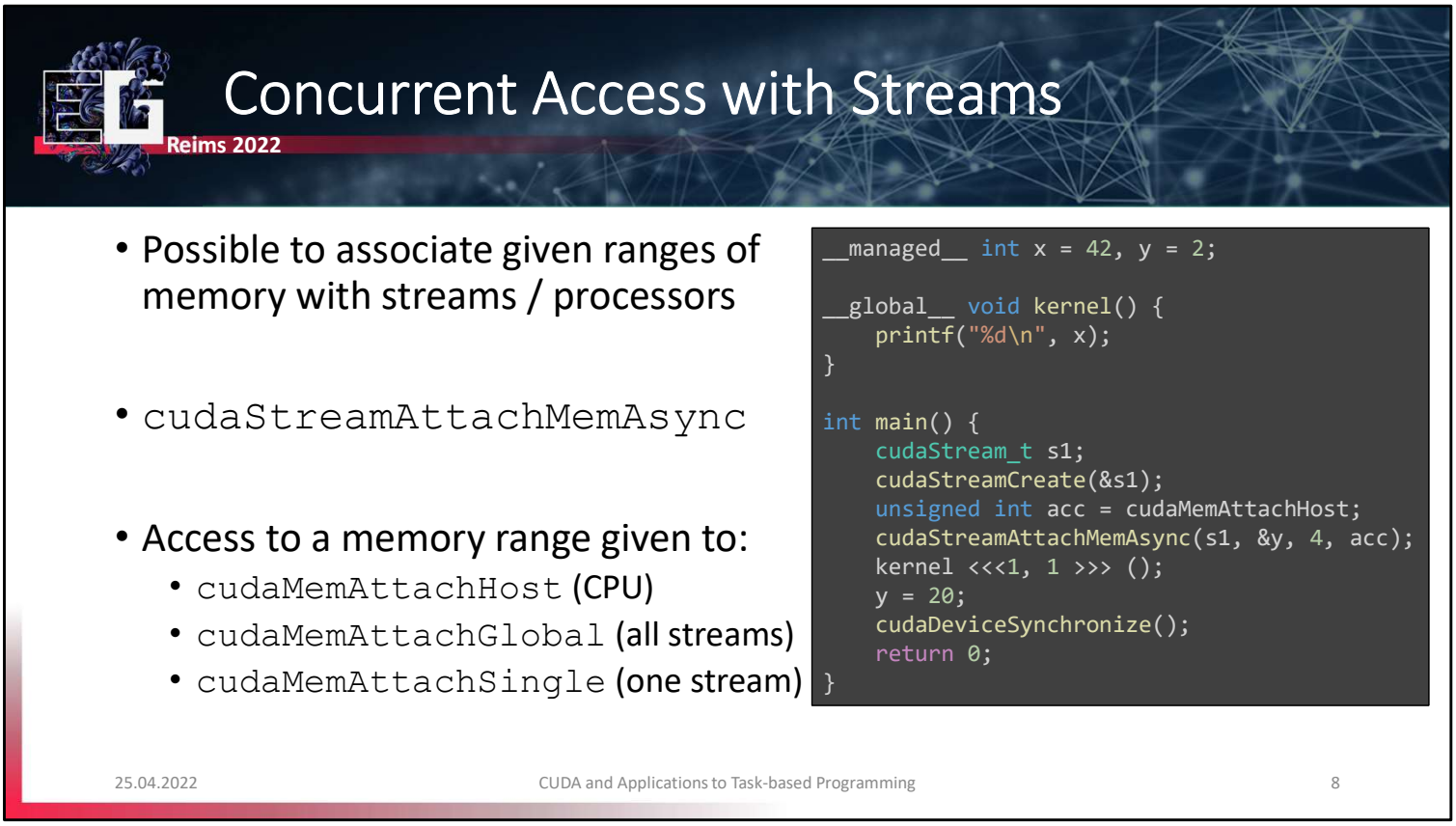
Initially, there was a noticeable performance penalty associated with the use of unified memory, but recently, managed memory has experienced a significant boost, making it much more practical than it used to be in addition to simplifying the code base, so we will quickly revisit it here.



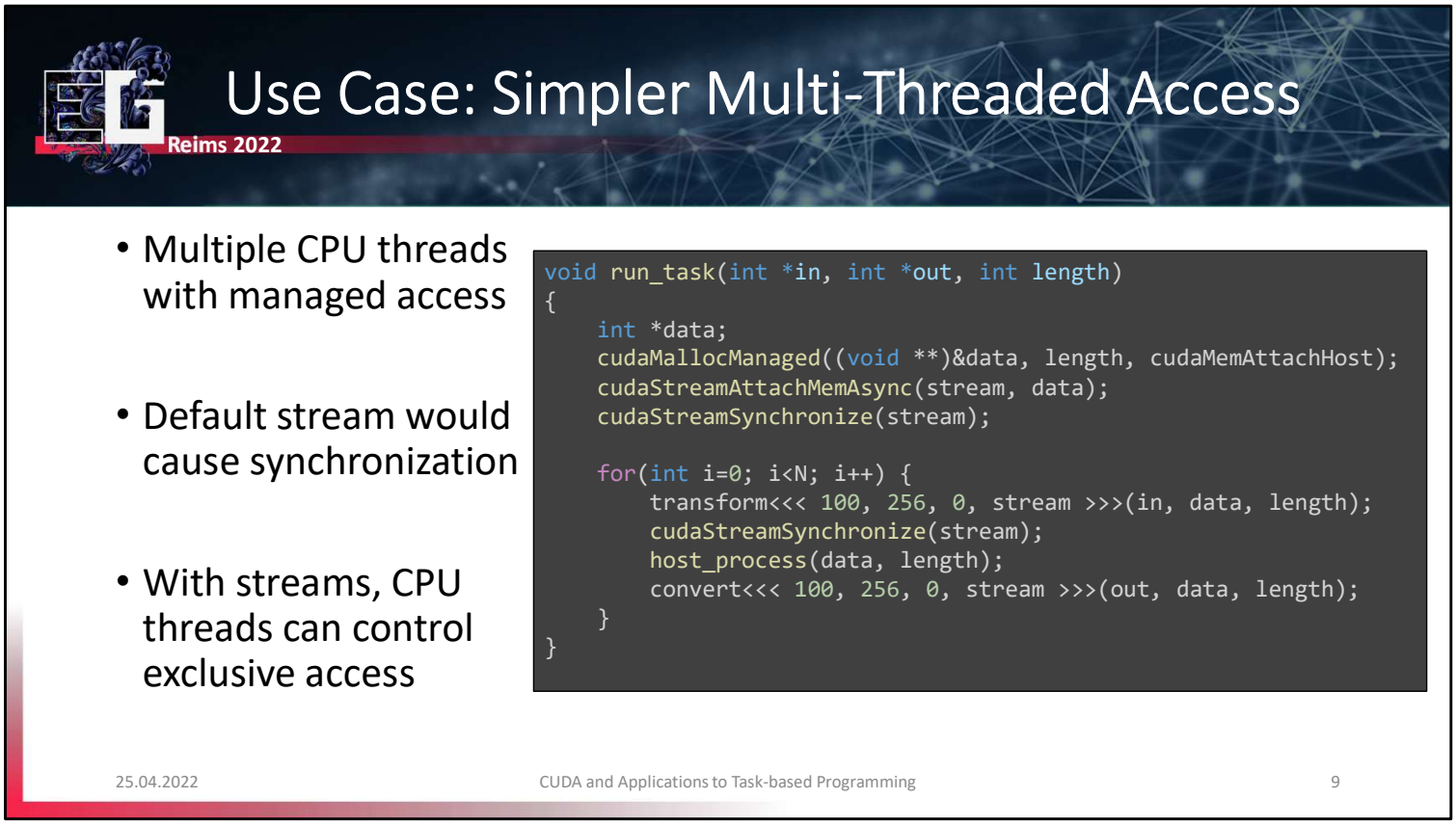
With unified or managed memory, both the CPU and GPU may try to access the same variables at the same time, since kernel launches and CPU-side execution are asynchronous. While it is now possible on some systems to have concurrent accesses, older cards with compute capability lower than 6.0 and even moderately modern ones may not support it. In this case, the CPU must ensure that its access to managed memory does not overlap with kernel execution. This can for instance be achieved with synchronization primitives.



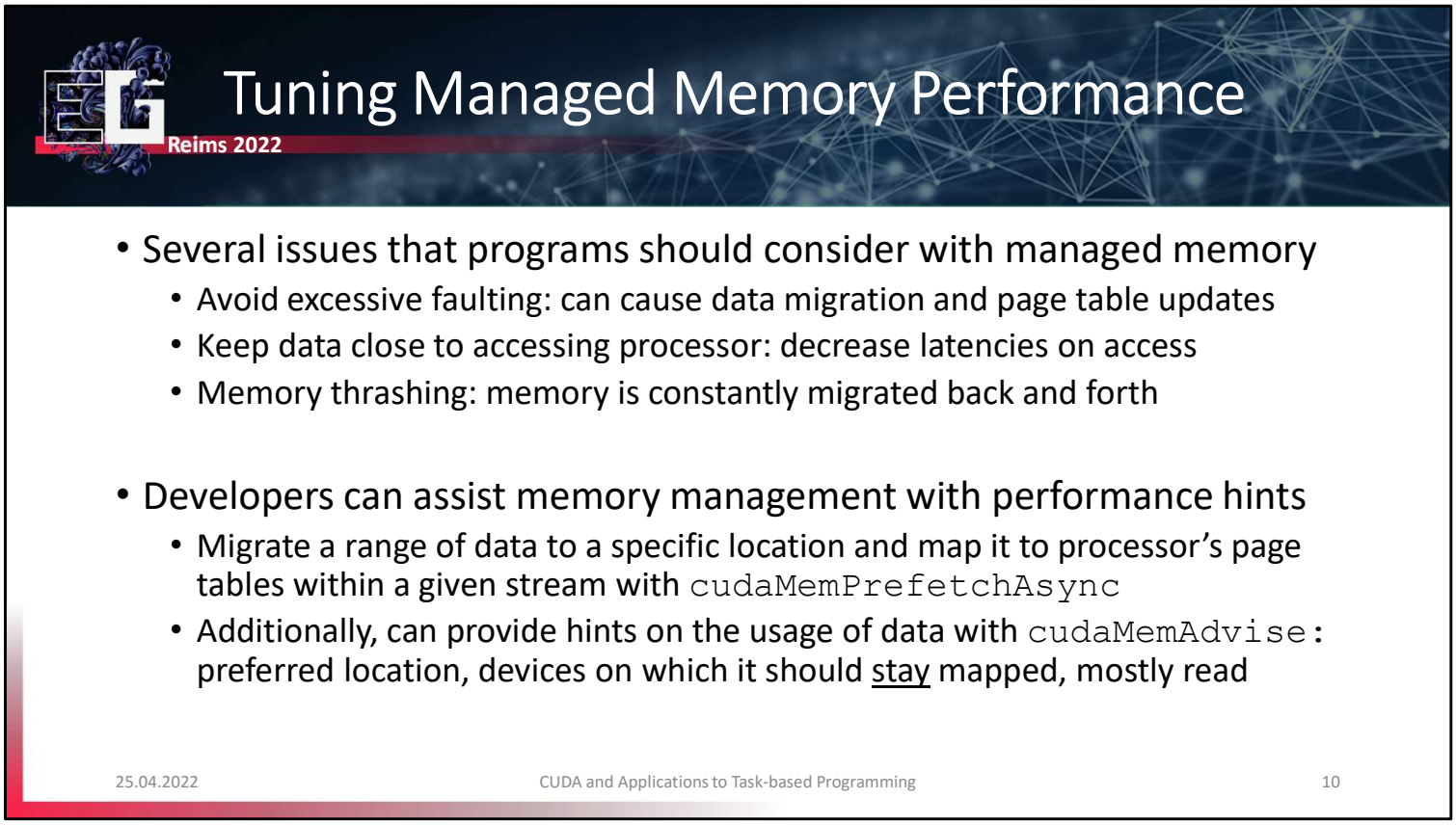
In this example, we see on the left a code segment that is problematic on cards without concurrent access support. On the right is an alternative implementation that makes sure to separate access from CPU and GPU temporally. This version is safe to execute on older hardware as well.



Alternatively, it is also possible to attach particular managed memory ranges to streams. This way, the access to particular managed memory ranges can be exclusively associated with a particular stream. Furthermore, the access to the range can be restricted to, e.g., signify that until further notice, managed memory may only be accessed by the host.

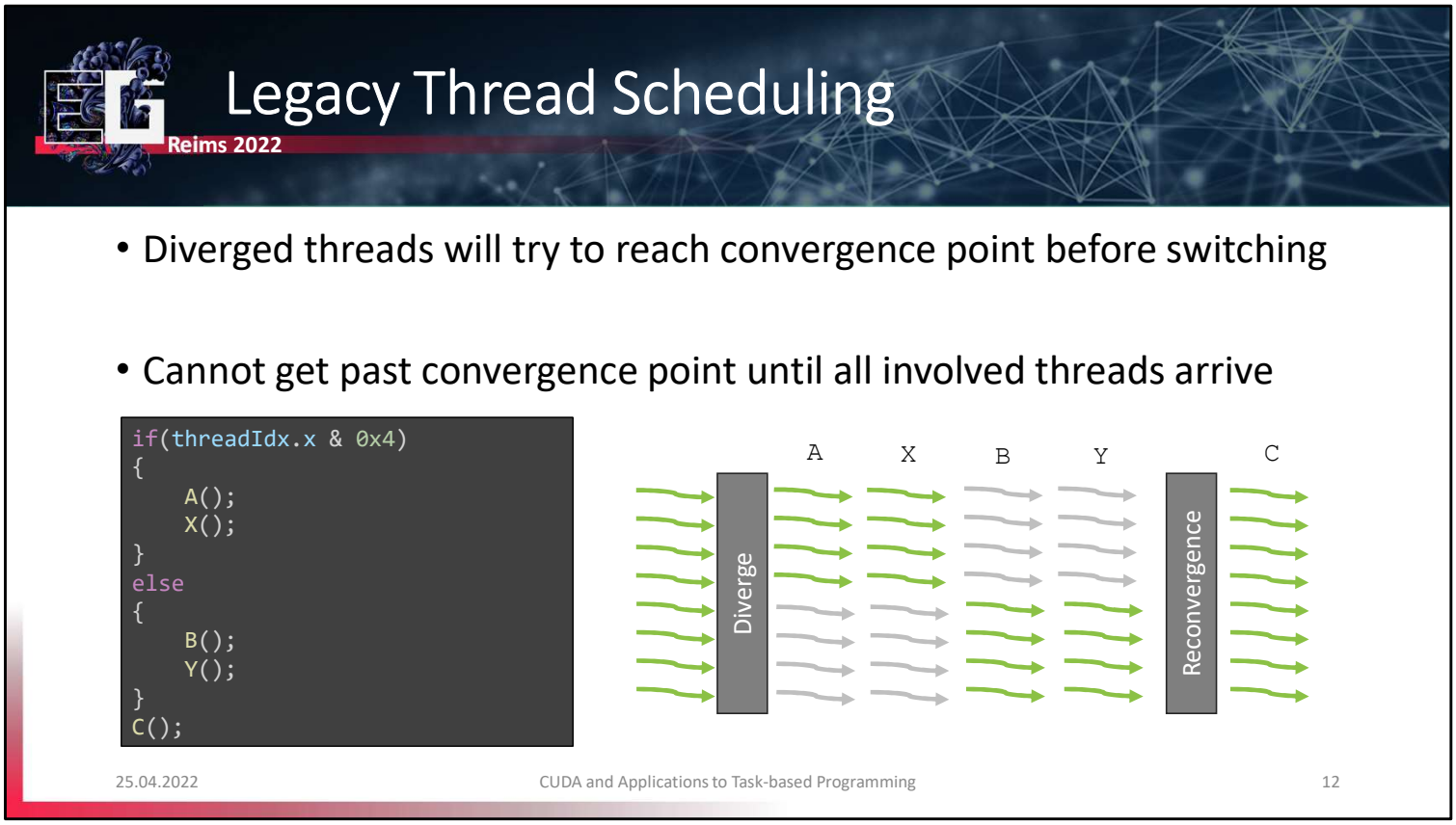


A common use case for the assignment of managed memory to streams is the processing of separate tasks in individual CPU threads. With every thread creating and associating a separate stream to the memory it intends to use, they are free to use managed memory concurrently without the need for synchronization across multiple threads. An exemplary setup that achieves this is given in the code segment above.

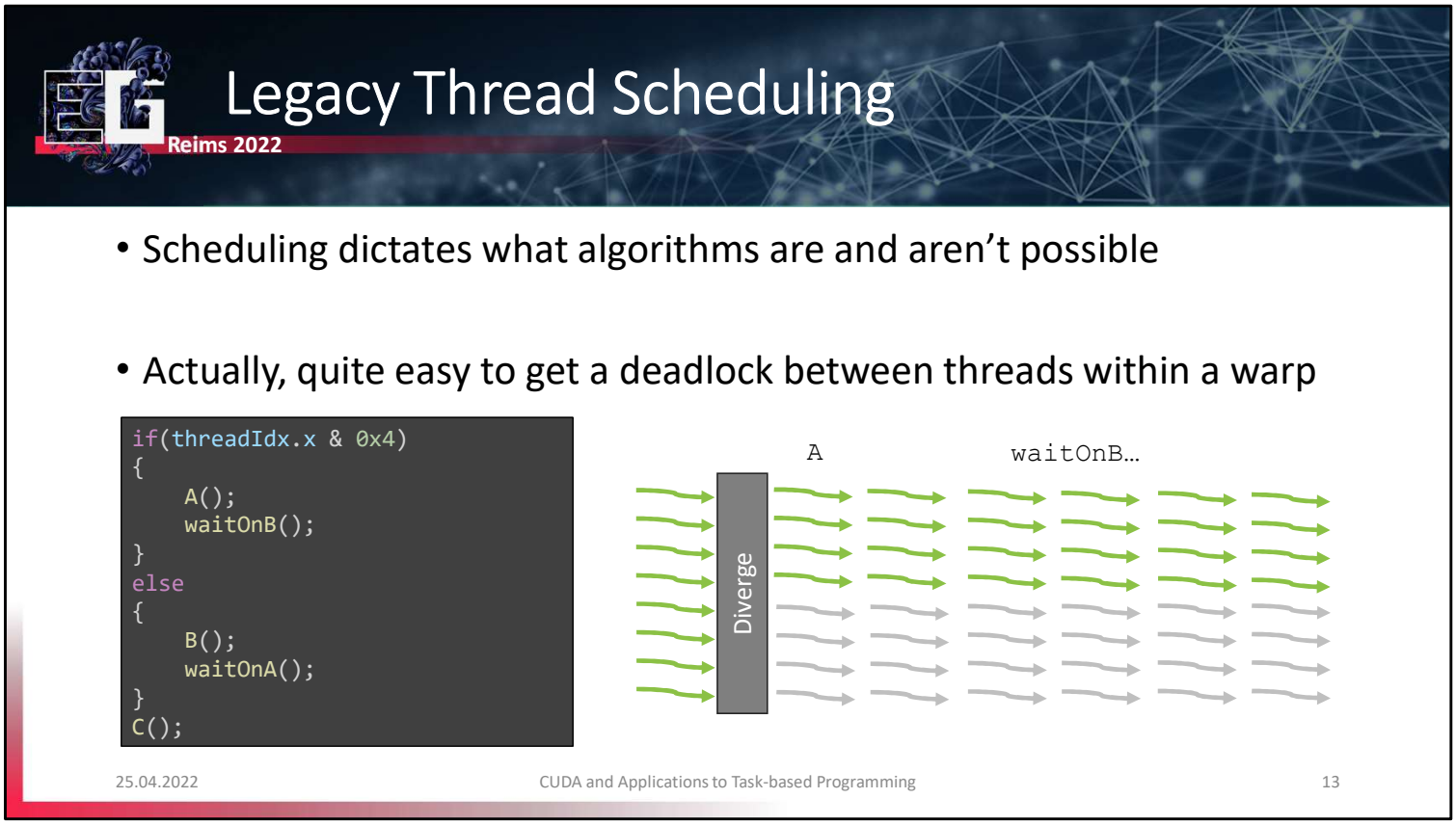


Important performance guidelines for managed memory is the avoidance of excessive faulting, since this negatively impacts performance. Furthermore, it should be ensured that data is always close to the processor that accesses it. Lastly, when memory is often migrated between host and device, this can quickly lead to thrashing, which is detrimental to performance as well. Managed memory has recently been made significantly more effective, insofar as the migration of data can now occur with a fine-granular page faulting algorithm, which somewhat alleviates these problems. However, developers can additionally provide hints that make memory management easier at runtime. In order to do so, they can „prefetch“ memory to a certain location ahead of it being used. Furthermore, developers can define general advice on the utilization of memory to indicate the preferred location of physical storage, the devices where it should remain mapped, and whether or not the access is governed by reading rather than writing.

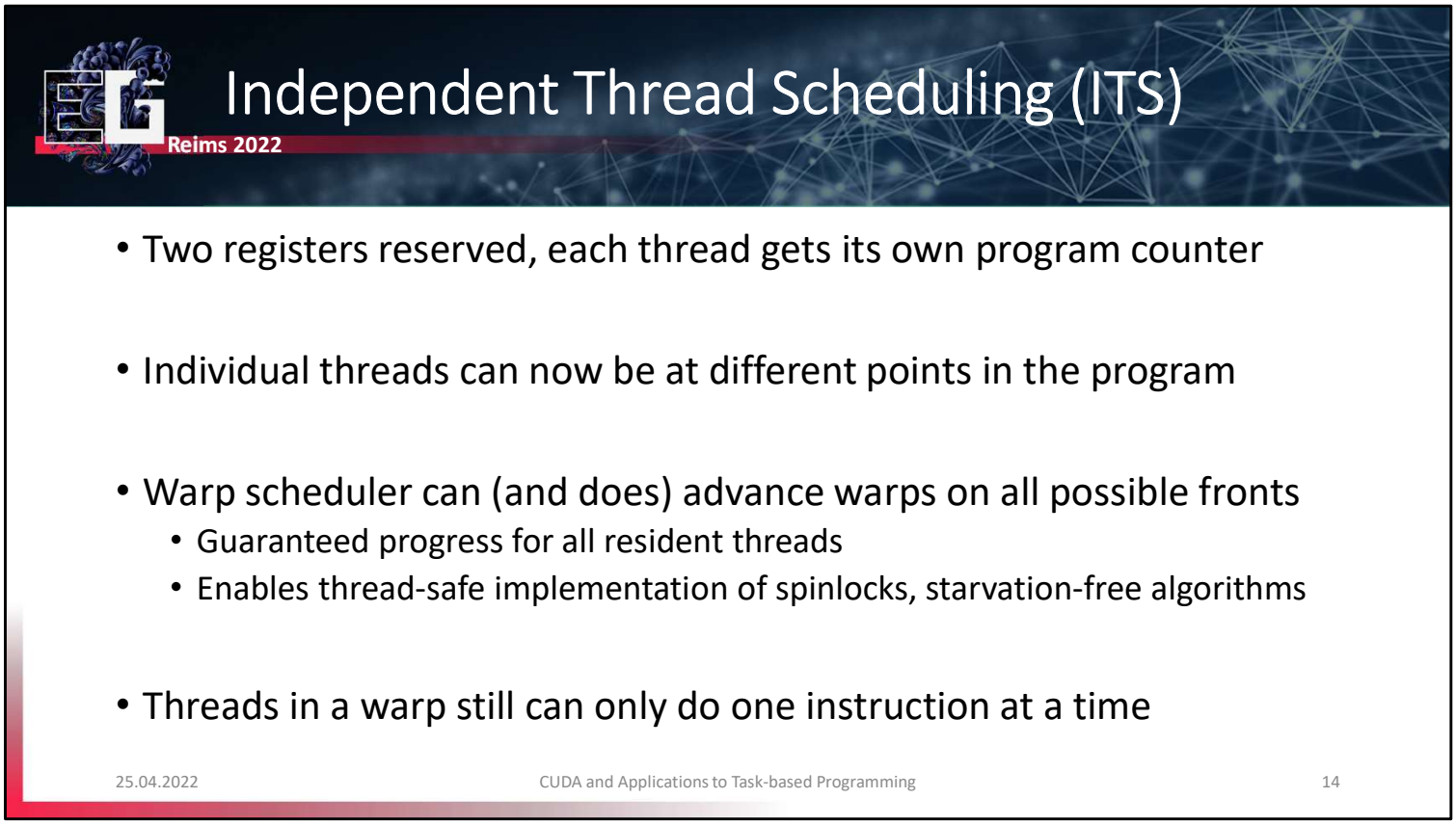
Next up, we will take another look at some of the details of Independent Thread Scheduling, which was introduced with the Volta architecture.



Let's quickly recap legacy scheduling. Consider for instance the branch given based on the thread ID. The lower four threads will enter one branch, the remaining threads will enter the other. However, once a branch has been chosen, it must be completed before the other branch can begin because the warp only maintains a single program counter for all threads. It can, for instance, not switch to execute B directly after A, because that would imply that half of the threads are at one point in the program, while the others are at another instruction, hence both branches would need to maintain separate program counters.



This has several implications that programmers must respect when they program for individual threads. For instance, consider the case where half of the threads in a warp are waiting on the other half. This is illustrated in this code sample. Because with the legacy thread scheduling model, threads cannot execute a different branch until the first chosen branch is complete, this program will hang since either A or B will never be executed, but each branch is waiting on an event that occurs in the other.



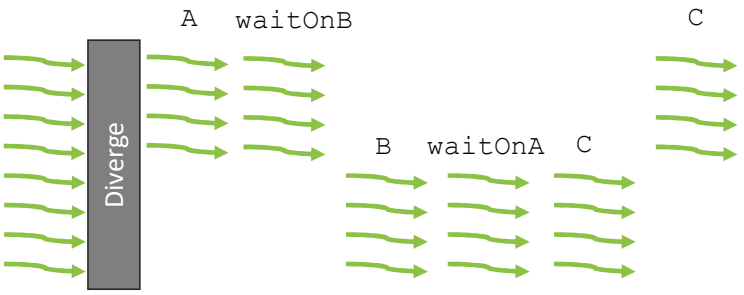
With independent thread scheduling, situations like this are no longer an issue. Each thread is given its own, individual program counter, meaning that theoretically, each thread can store its own unique instruction that it wants to perform next. The execution of threads still happens in warps, this has not changed. It is not possible for threads in a warp to perform different instructions in the same cycle. However, a warp may now be scheduled to progress at any of the different program counters that the threads within it are currently holding. Furthermore, ITS provides a "progress guarantee": eventually, over a number of cycles, all individual program counters that the threads in a warp maintain will be visited. This means that if, for instance, the execution has diverged and two branches, both are guaranteed to be executed sooner or later.

Independent Thread Scheduling (ITS)

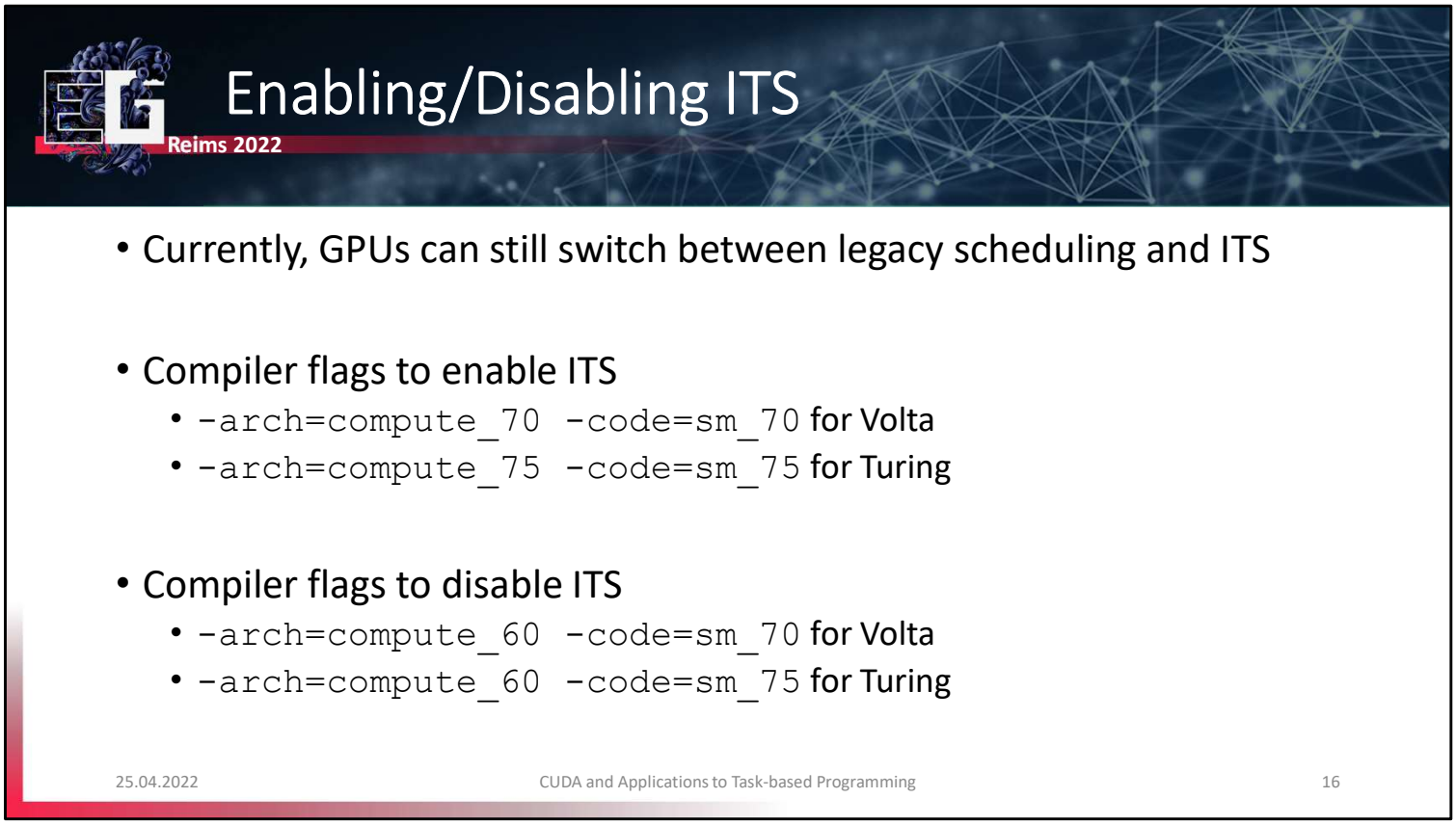
Reims 2022

- Guaranteed progress, one branch can wait on another branch
- Diverged threads may not reconverge, should be explicitly requested!

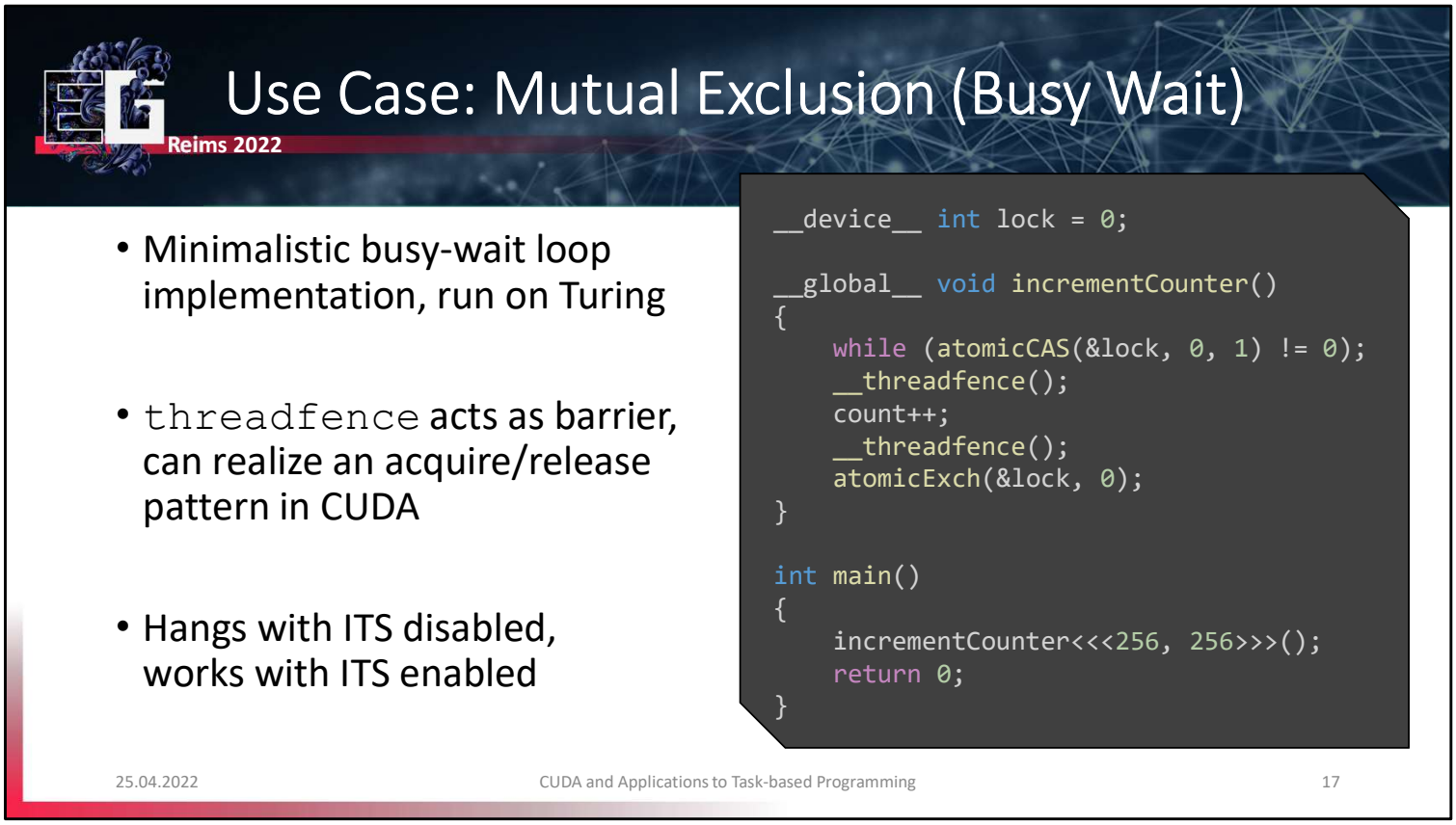
```
if(threadIdx.x & 0x4)
{
    A();
    waitOnB();
}
else
{
    B();
    waitOnA();
}
C();
```



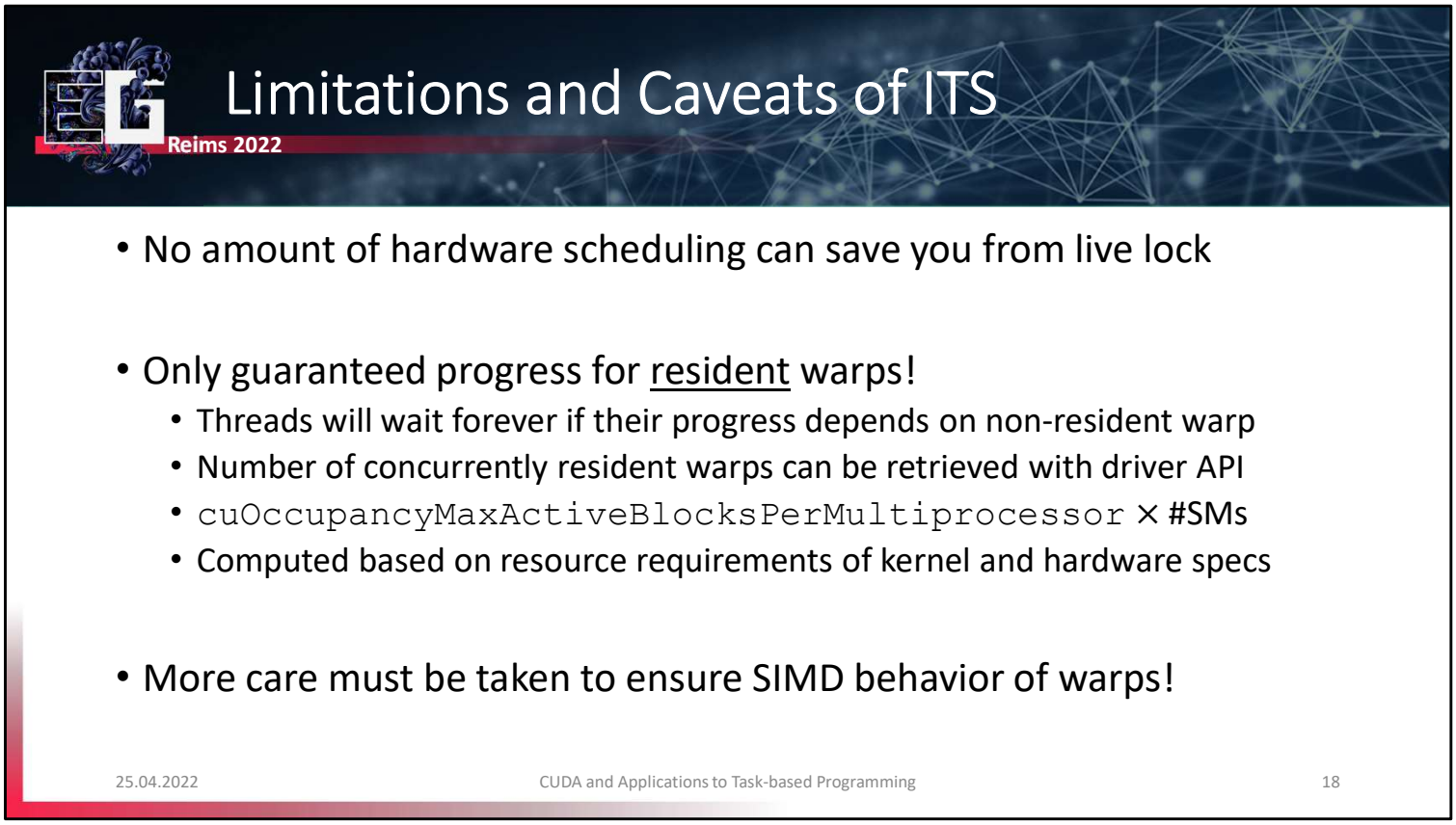
With ITS enabled, the previous setup no longer poses a problem. A branch may be chosen as before and start waiting on the other branch. Due to the progress guarantee of ITS, sooner or later, the other branch will be scheduled and its threads will proceed, which is possible because every thread has a program counter to maintain its own unique position in the program code. A side effect of the new design, however, is that program code can no longer make any assumptions about threads moving in lockstep since they are free to stay diverged until the program finishes. The GPU will try to make threads reconverge at opportune times, but if it is desired that threads are guaranteed to perform given instructions in groups of 32, e.g., to exploit SIMD behavior, this must now be explicitly requested with a synchronization command.



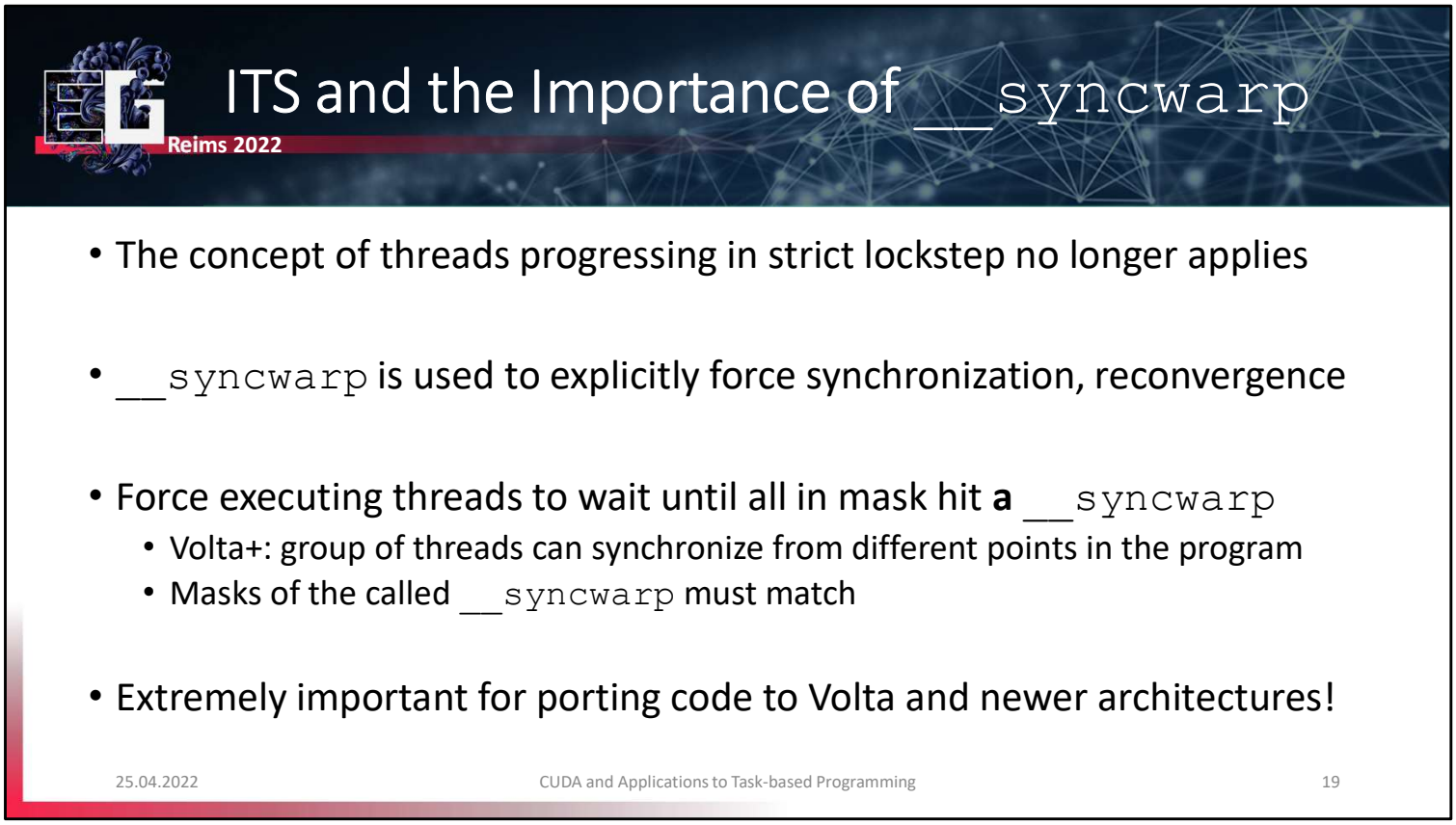
The switches to disable or enable ITS are listed here. Currently, GPU models still support both modes, so it is possible to run the previous example on newer GPUs with ITS enabled/disabled to see the results. It is not yet certain if legacy scheduling will eventually be abandoned in favor of ITS, however, other GPU compute APIs, like OpenGL‘s compute shader, appear to default to legacy scheduling for compatibility reasons.



A simple test to demonstrate the new capabilities of ITS is given by this minimal example, in which we control a critical section that has exclusive excess to a counter. __threadfence can be understood as a general barrier, and therefore can model access patterns like release and acquire. Here, we combine it with atomic operations on a global variable to secure the counter variable. Every thread will attempt to acquire the lock, change the counter and release the lock again. In a warp, only one thread can succeed at any time. If after succeeding the other branch is executed, with legacy scheduling, the routine can never finish. Running without ITS support, this example will therefore likely cause a hang. With ITS enabled, it is safe to execute and eventually terminates.

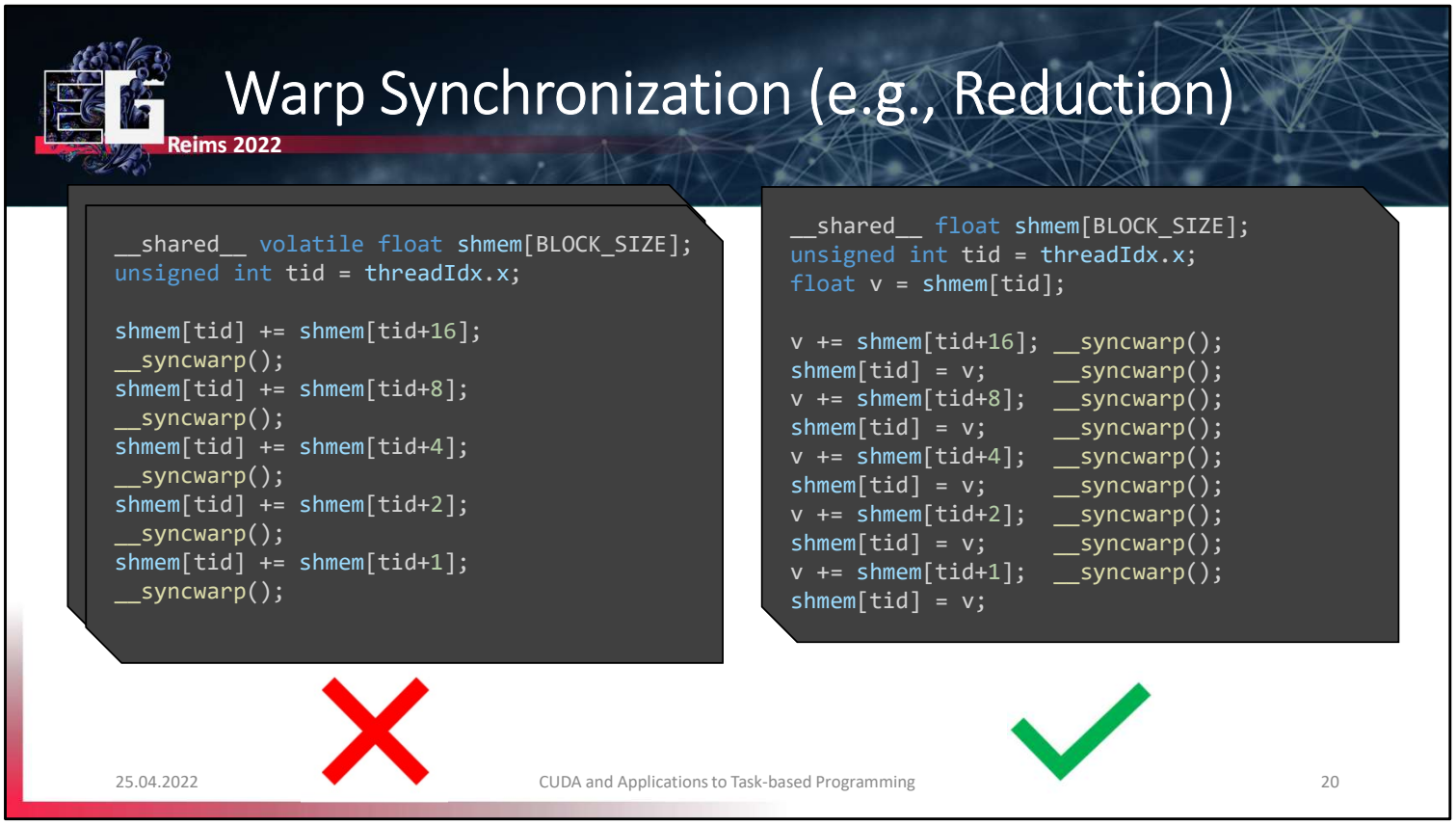


There are of course a few limitations to ITS. First of all, ITS cannot absolve developers of improper parallel coding. While it can in fact take care of deadlocks, it is still very much required of developers to be aware of the scheduling model of GPUs to make sure they can avoid live locks as well. Second, ITS can only provide a progress guarantee for threads and warps that are resident at any point in time. That is, in case of a large launched grid, if the progress of threads depends on a thread that was not launched until all SMs were filled up, the system cannot progress and will hang, since resident warps are not switched out until they complete execution. Lastly, ITS, due to the fact that it is not guaranteed to reconverge, may break several assumptions regarding warp level programming. In order to ensure a fully or partially reconverged warp, developers must make proper use of __syncwarp and can no longer assume lockstep progress at warp level, which is a hard habit to break.

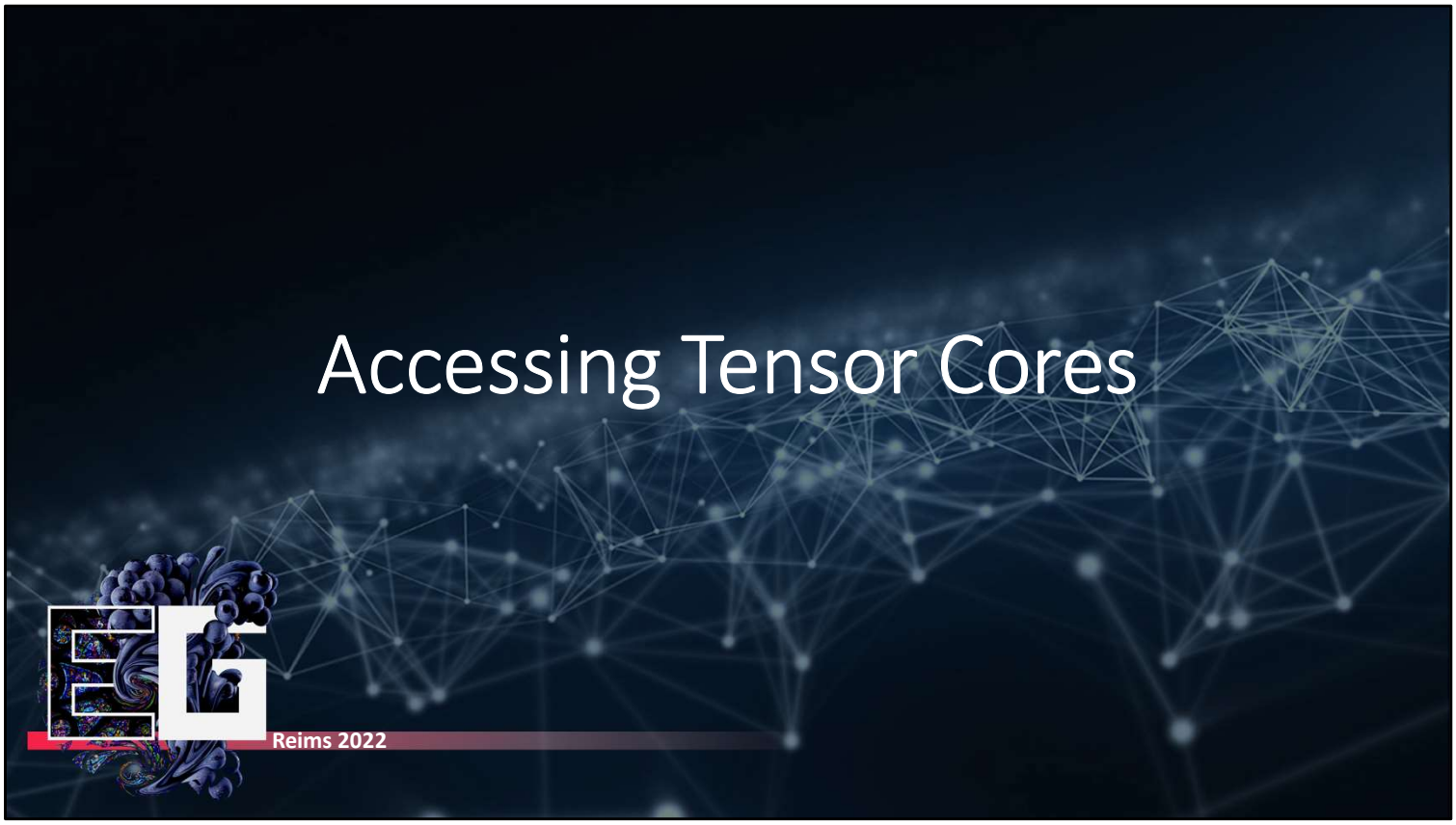


__syncwarp may, at first glance, seem like a smaller version of __syncthreads, however, it has a number of interesting peculiarities that make it more versatile. Most importantly, __syncwarp is parameterized by a mask that indicates the threads that should participate in synchronization, in contrast to __syncthreads, which must always include all non-exited threads in the block.

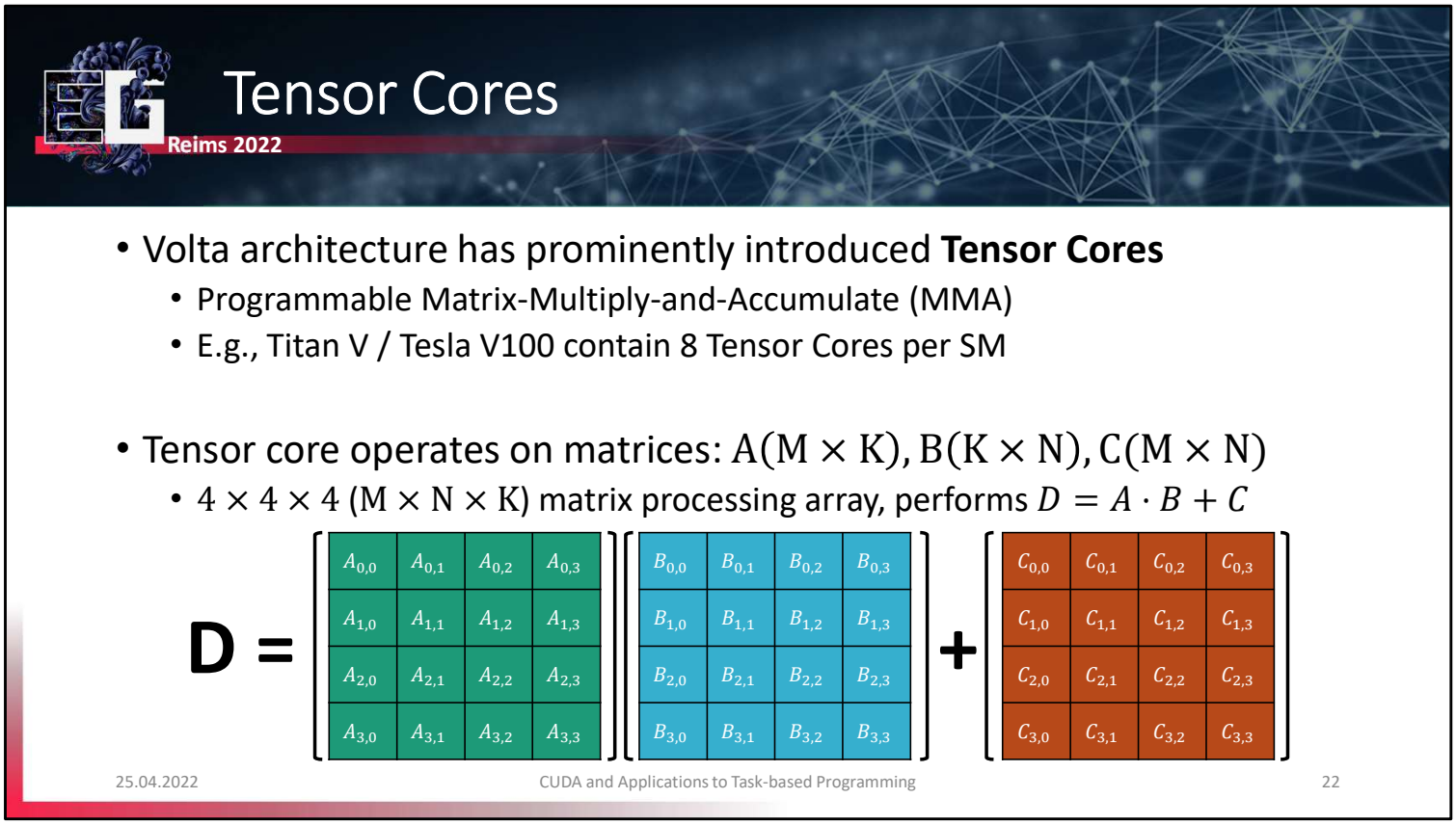
__syncwarp may be executed from different points in the program, enabling for instance a warp to synchronize across two different branches, as long as the masks match. If optimizations at warp-level are made by developers, in order to write correct code, they will need to make generous use of __syncwarp in many common patterns.



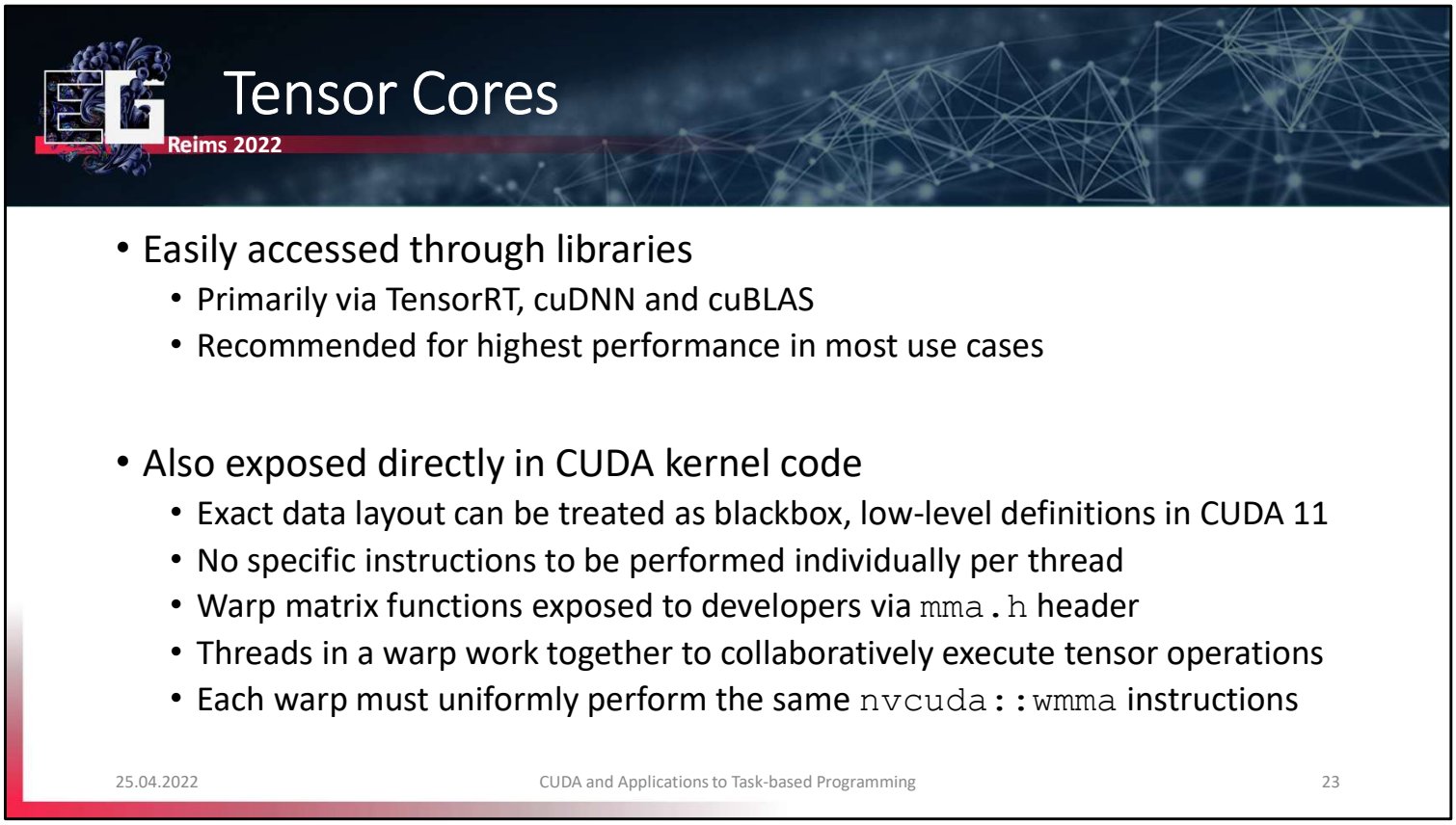
Consider the example on the left, which outlines the last stages of a parallel reduction. Naturally, if we know that ITS is active, we cannot assume lockstep progress and must secure every update of the shared variables with a __syncwarp operation. However, the initial response of many developers is not sufficient. In this case, the access in each step is not secured by an if clause to restrict the participating threads. Hence, the threads with a higher ID might overwrite their results before they are read by lower-ID threads. In order to make these updates secure, either additional if clauses would have to be introduced that exclude higher thread IDs, or a more generous use of __syncwarp is required.



A highly popular topic of GPUs today is the introduction of tensor cores and their crucial role in many machine learning algorithms. For those of you who wondered what exactly it is that tensor cores do, we will now take a short look under the hood and describe what makes them tick.



With the arrival of the Volta architecture, NVIDIA GPUs have added a new function unit to the streaming multiprocessors, that is, the tensor core. The number and capability of tensor cores is rising quickly, and they are one of the most popular features currently. A tensor core and its abilities are easily defined: each tensor core can perform a particular fused matrix operation based on 3 inputs: a 4×4 matrix A, a 4×4 matrix B, and a third 4×4 matrix for accumulation, let's call it C. The result that a single tensor core can compute is $A \times B + C$, which on its own does not seem too helpful. However, the strength of tensor cores originates from its collaboration with other cores to process larger constructs.



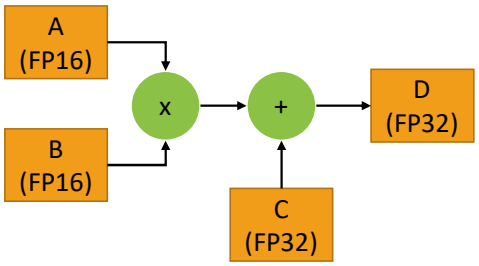
This collaboration can be achieved in one or two ways. The first is by using one of the readily-available libraries that make use of these capabilities in highly-optimized kernels, such as TensorRT, cuDNN or cuBLAS. For general purpose applications, it is recommended to use these solutions for higher performance.

However, the access to tensor cores is also exposed in CUDA directly via a separate header for matrix multiplication and accumulation of small matrices, which are usually only a part of the total input. These matrix tiles, or „fragments“, can be larger than 4×4 if threads in a warp cooperate. The MMA headers define warp-level primitives, that is, tensor cores must be utilized collaboratively by all the threads in a given warp.

Tensor Cores

Reims 2022

- Each core can do 64 floating point fused-multiply-add (FMA) per clock
  - E.g., with 8 tensor cores: 64 * 2 * 8 operations/cycle → 1024 operations/cycle
- Restrictions on format for input fragments, e.g.:
  - A = __half (16bit float), B = __half, C = float →→→→
  - A = __half, B = __half, C = __half
  - A = char, B = char, C = int
  - A = precision::tf32, B = precision::tf32, C = float

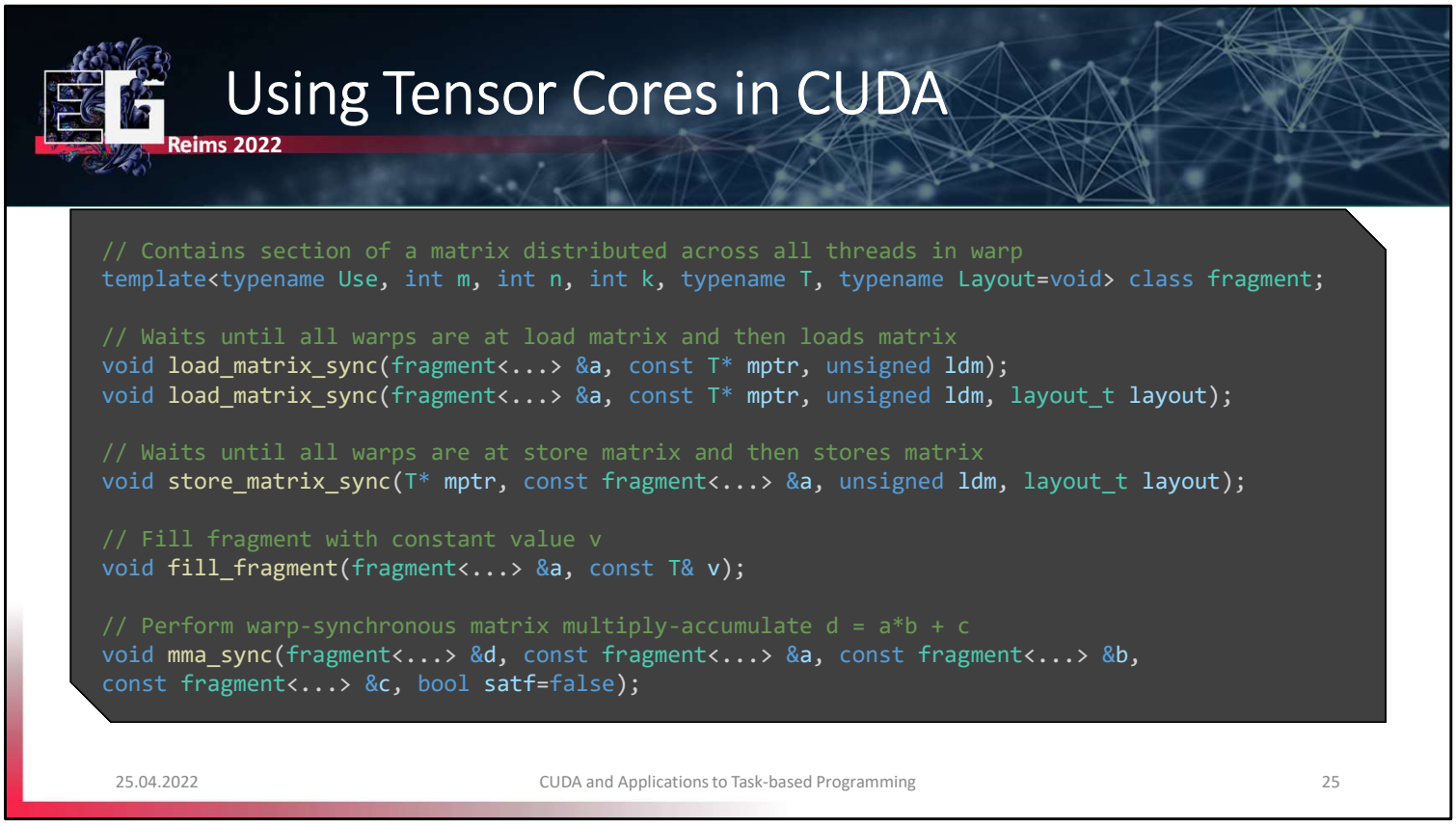


- Warps collaborate to process larger fragments
  - Maximal dimensions governed by data types used
  - E.g., max. 16 × 16 × 16 for A = __half, B = __half, C = float

The performance of these computations is significant since the tensor core is optimized for this very specific operation. A tensor core can achieve 64 fused-multiply-add operations per clocks. With 8 tensor cores per SM, this leads to a vast 1024 operations performed in each cycle.

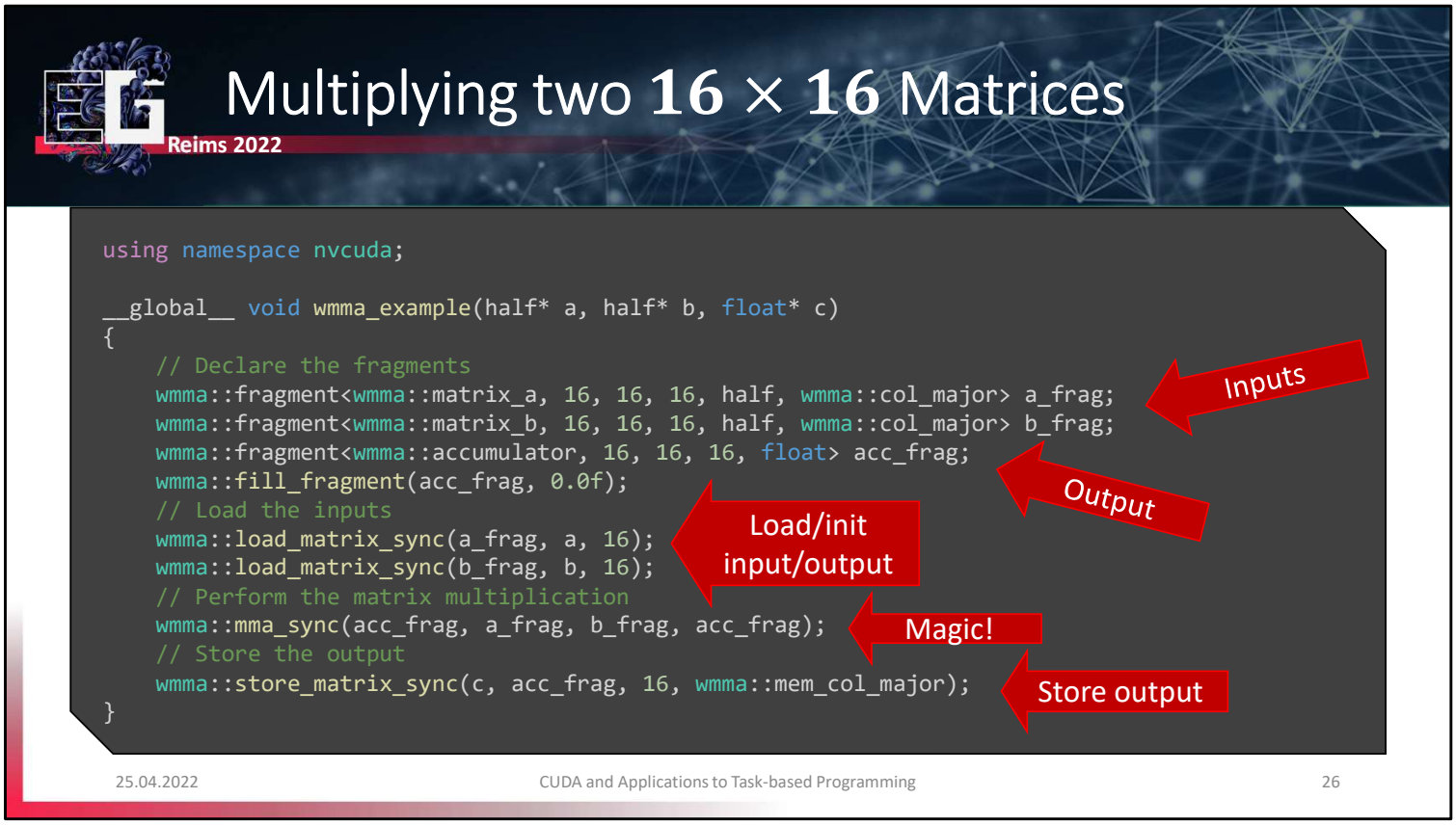
However, restrictions do apply in their utilization. A common assumption is that tensor cores work directly on single-precision floating point values, however, this is only true for the accumulation part of the operation. So far, the input fragments A and B may not be 32-bit wide, but rather 16-bit half-precision or the more adaptive tf32 type, which has a bigger range than half-precision types.

The choice of what data types are used as input directly affects the maximum size of the fragments that can be collaboratively computed. A common configuration, with half-precision for input fragments A and B, enables warps to compute MMA operations on 16×16 fragments. When using, e.g., tf32 for A and B instead, one of the dimensions must be halved.



Here, we list the relevant types and functions that are exposed to warps for performing tensor core operations:

- fragment: Overloaded class, containing a section of a matrix distributed across all threads in a warp. Mapping of matrix elements into fragment internal storage is unspecified (and subject to change). Use can be <matrix_a, matrix_b, accumulator>, M,N,K are shape of matrix.
- load_matrix: waits until all threads in a warp are at load and then loads fragment from memory. ptr must be 256bit aligned, ldm is stride between elements in consecutive rows/columns (multiple of 16 Bytes, i.e. 8 half elements or 4 float elements). All values must be the same for all threads in a warp, must also be called by all threads in a warp, otherwise undefined
- store_matrix: Same ideas as with load
- fill_fragment: Mapping is unknown, v must be the same for all threads
- mma_sync: performs warp-synchronous matrix multiply-accumulate (MMA)



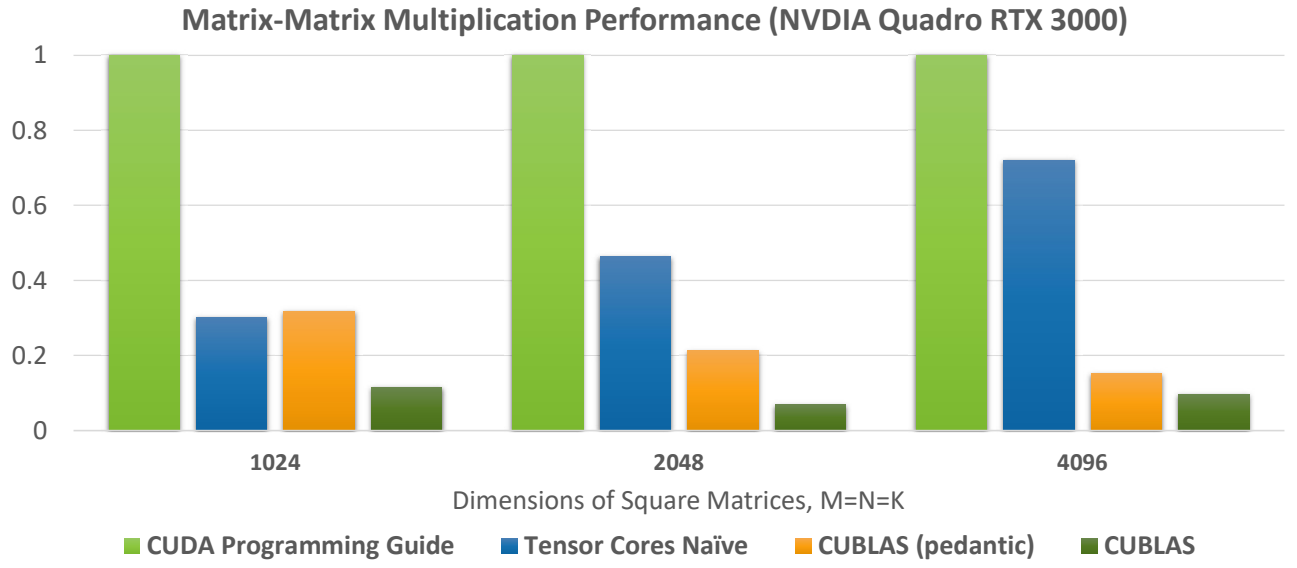
Here, we show a minimal example of using tensor cores with the available functions. First, we define the fragments that a warp can collaboratively work on, in this case, a 16×16 portion of a matrix, with the data format being half-precision floats. The accumulator has a higher precision, it can be single-precision float without reducing the fragment size. After filling the accumulator with all zeros, we collaboratively load in the data to fill the input fragments A and B. Once done, the warp must synchronize and perform the matrix multiplication and accumulation in cooperation. Finally, the result of this computation, stored in the accumulator, is written back to memory.

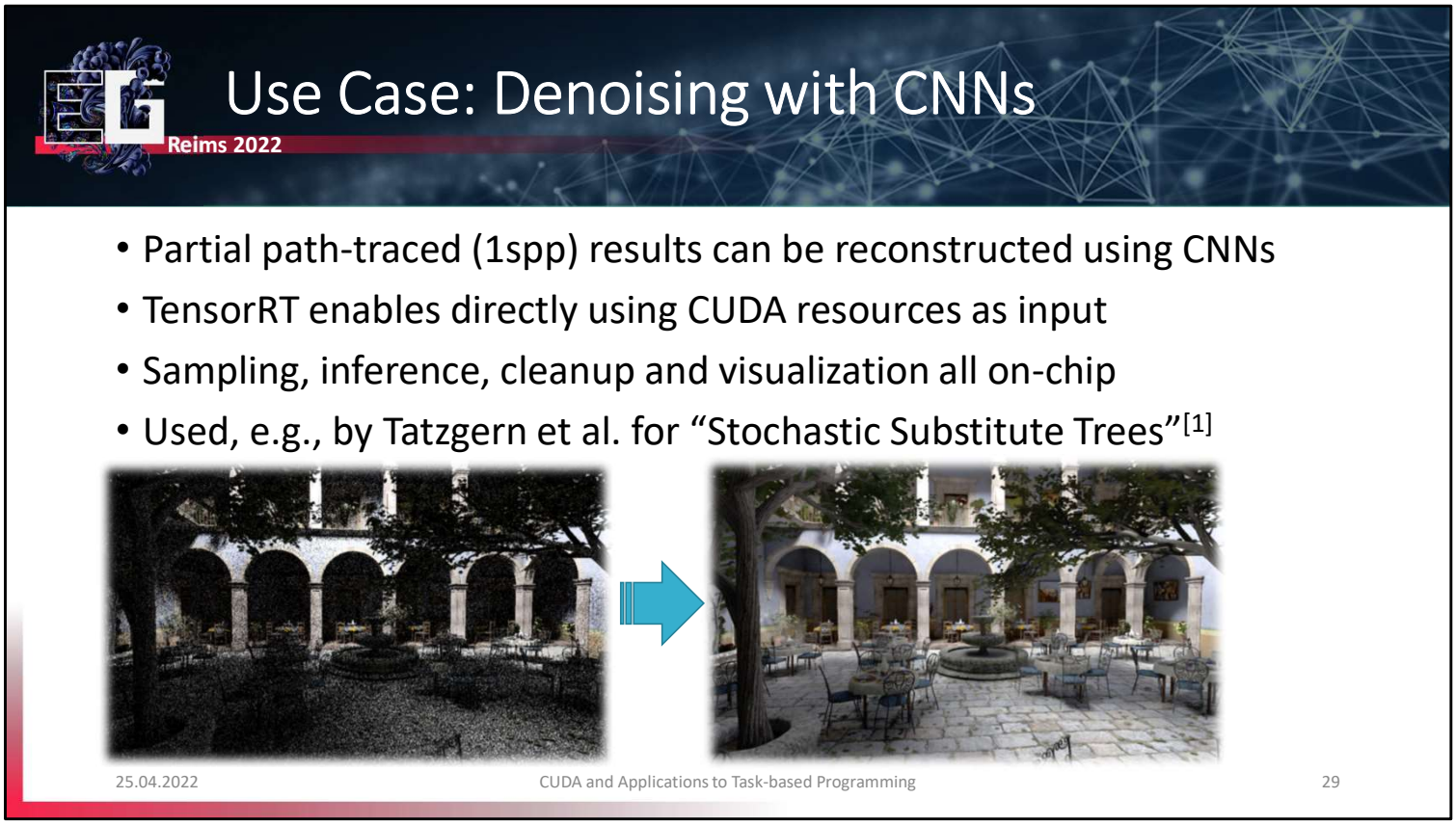
Using CUBLAS

- Tensor cores are now used whenever possible by default
  - Can still be disabled using `CUBLAS_PEDANTIC_MATH`
  - `CUBLAS_TENSOR_OP_MATH` will soon be deprecated
- Copy matrix data from CPU to GPU (`cudaMemcpy` suffices)
- **`cublasGemmEx`**`(cublasHandle_t` **`handle`**`, cublasOperation_t` **`transa`**`, cublasOperation_t` **`transb`**`, const void *`**`alpha`**`, const void *`**`A`**`, cudaDataType` **`Atype`**`, int` **`lda`**`, const void *`**`B`**`, cudaDataType` **`Btype`**`, int` **`ldb`**`, const void *`**`beta`**`, void *`**`C`**`, cudaDataType` **`Ctype`**`, int` **`ldc`**`, cudaDataType` **`computeType`**`, cublasGemmAlgo_t` **`algo`**`)`

Matrix Multiplication Comparison

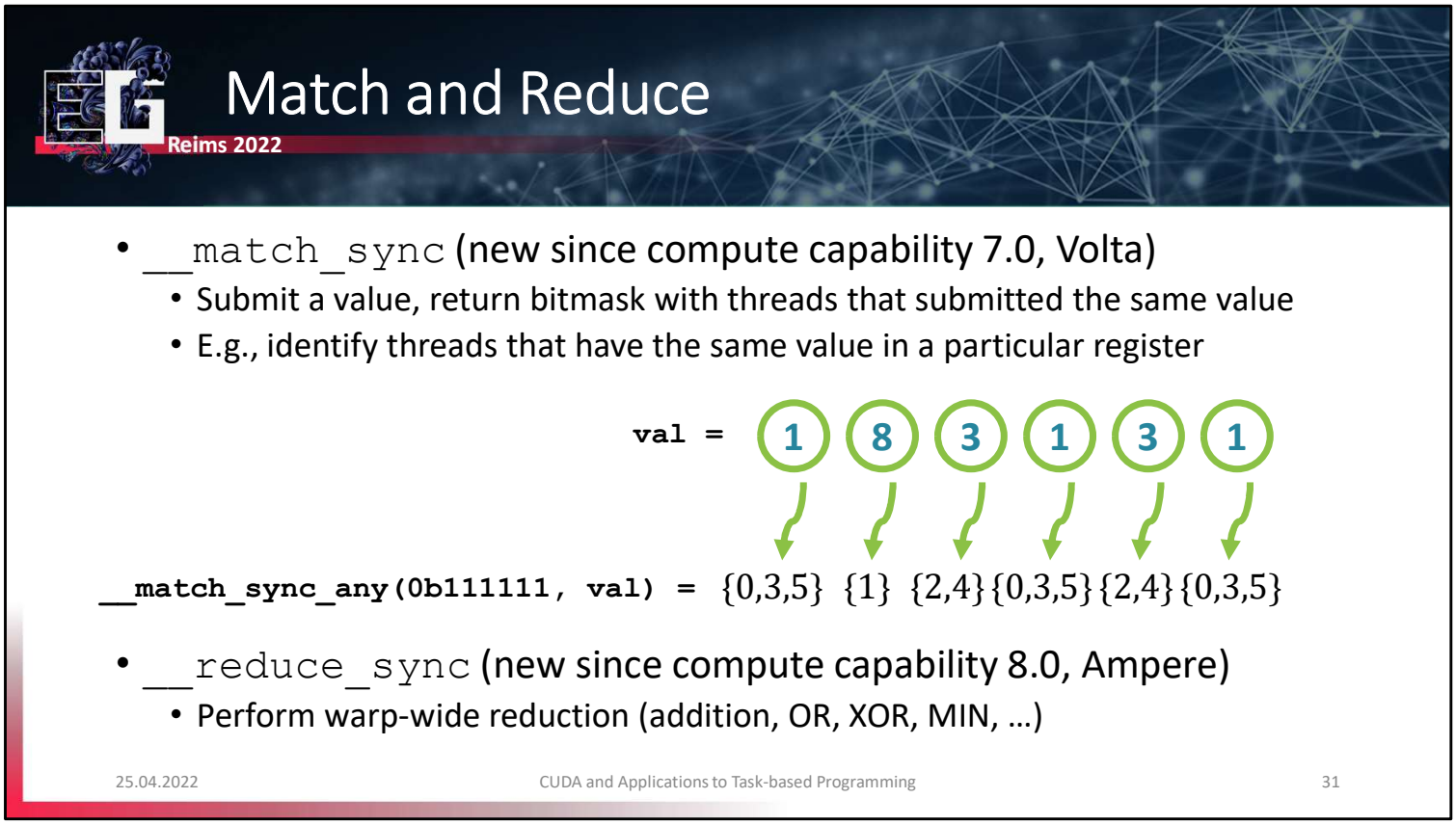
Reims 2022





Although knowing the exact functionality of tensor cores is interesting, a much more practical approach for the most common use cases, like machine learning, is to use the available libraries, like TensorRT. The corresponding solutions support the loading and inference with network layouts in common machine learning formats, such as ONNX, and can compute results with unprecedented performance. For instance, we have used TensorRT to use convolutional networks for the reconstruction of undersampled renderings in previous work, which was published last year at I3D. In the paper, Stochastic Substitute Trees, the sampling, reconstruction, and visualization of an approach inspired by instant radiosity can execute completely on the GPU to give real-time performance in complex lighting scenarios.

Let us now turn to the warp-level primitives that we haven‘t discussed so far. In addition to shuffling and voting, recent architectures have introduced additional primitives that provide interesting use cases for optimization.



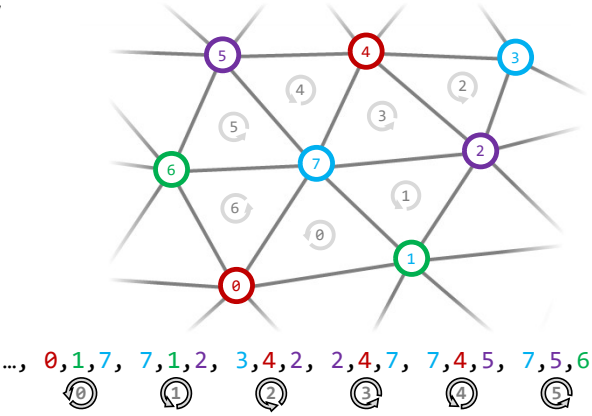
Two new exciting operations can now occur with high efficiency within a warp. One is the __match_sync operation, which has been enabled since Volta. Previously, we had the __ballot operation, which enabled us to find out for which threads in a warp a certain predicated evaluates to true. However, now threads can individually identify the threads whose value in a given register matches their own.

Additionally, it is now possible to reduce results from registers to a single result with a single instruction. This functionality is accelerated in hardware with the Ampere architecture.

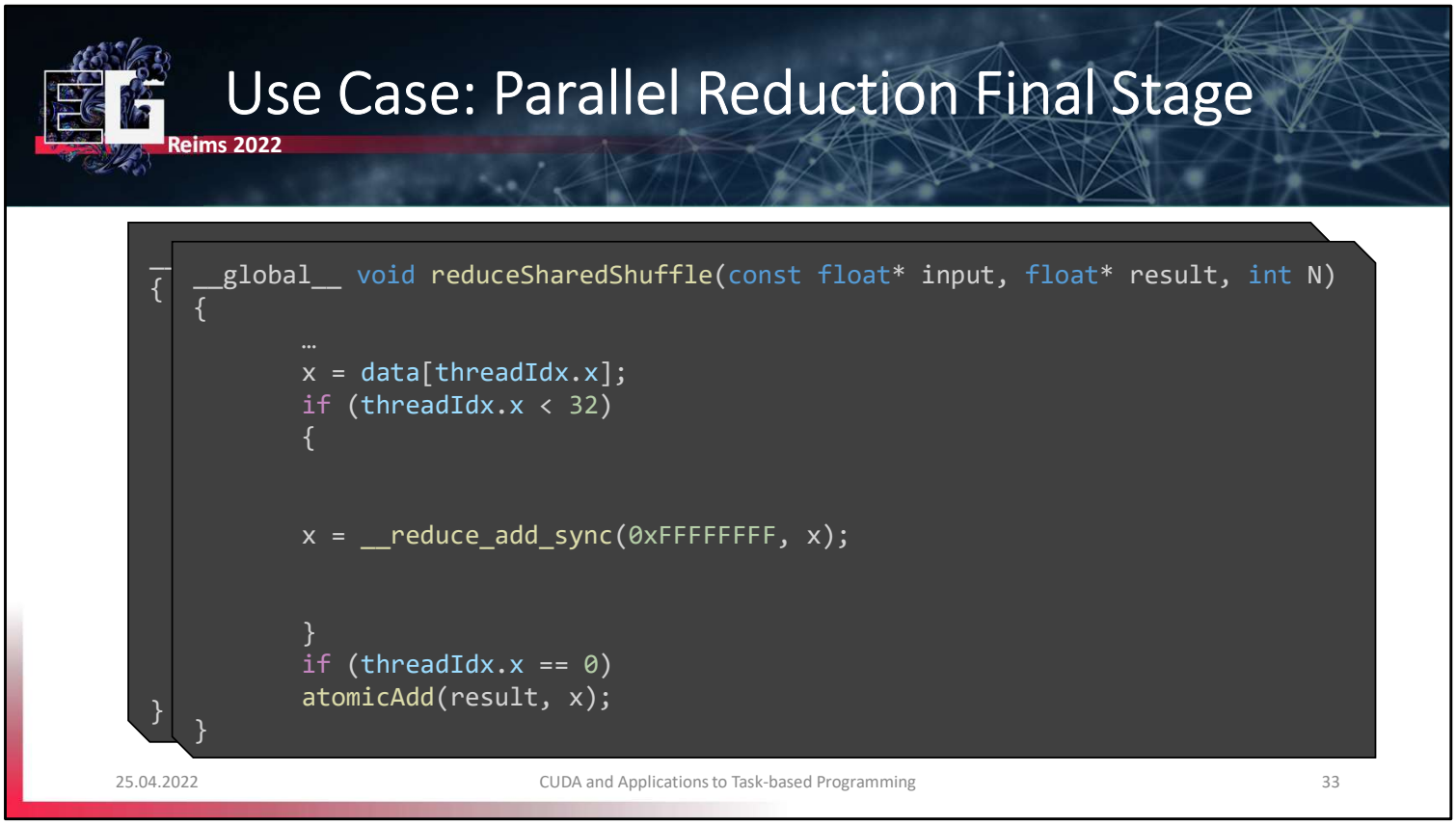
Use Case: Vertex Deduplication

Reims 2022

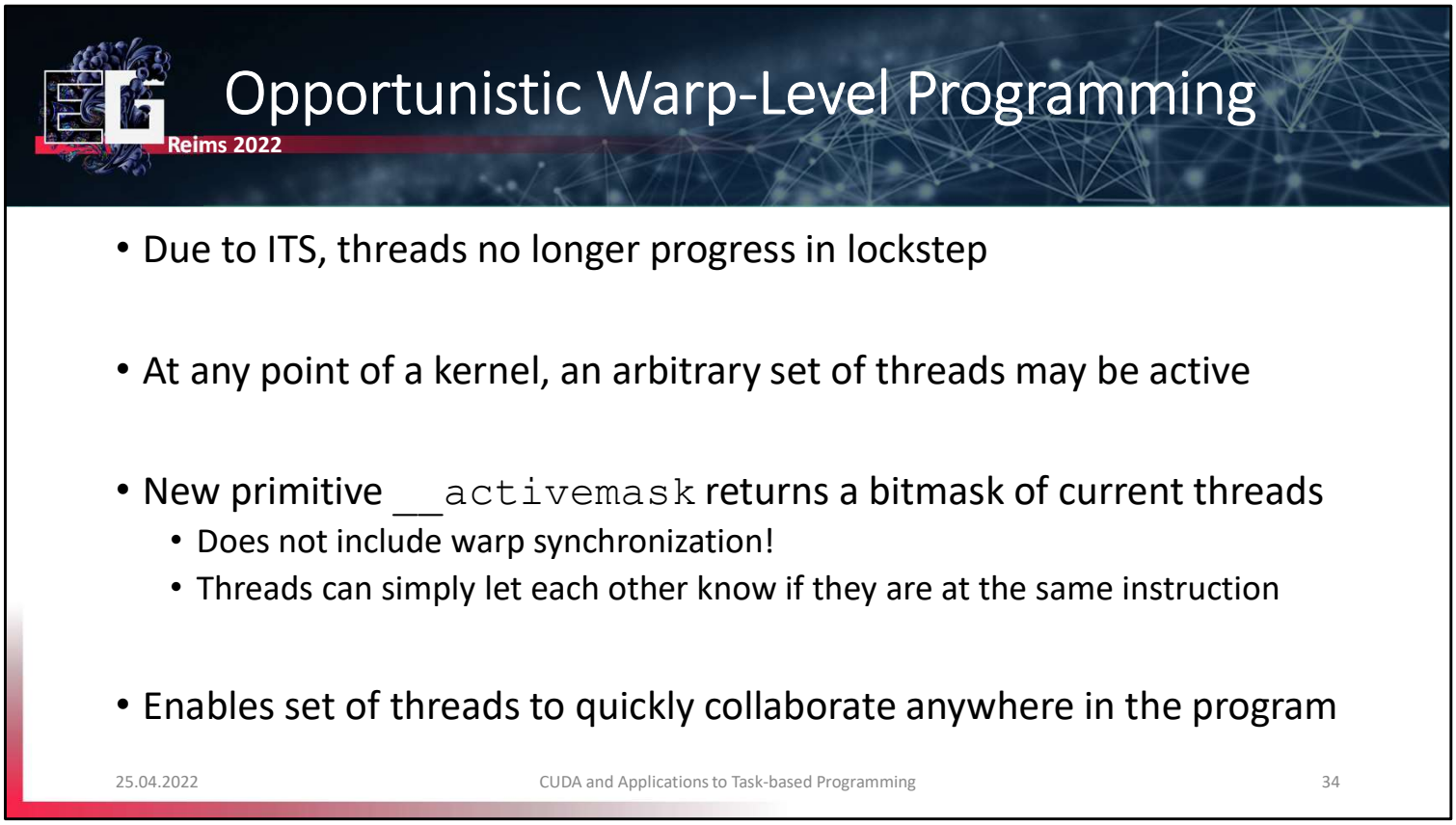
- Use case: identify duplicate vertices in a batch of triangles
  - For rasterization, geometry is usually partitioned into batches
  - Each warp processes a separate triangle batch independently
  - To avoid redundant vertex shading, need to deduplicate indices
  - Can be achieved with shuffles in software (e.g., Kenzel et al.[2])
  - `__match_any_sync` greatly simplifies the deduplication!



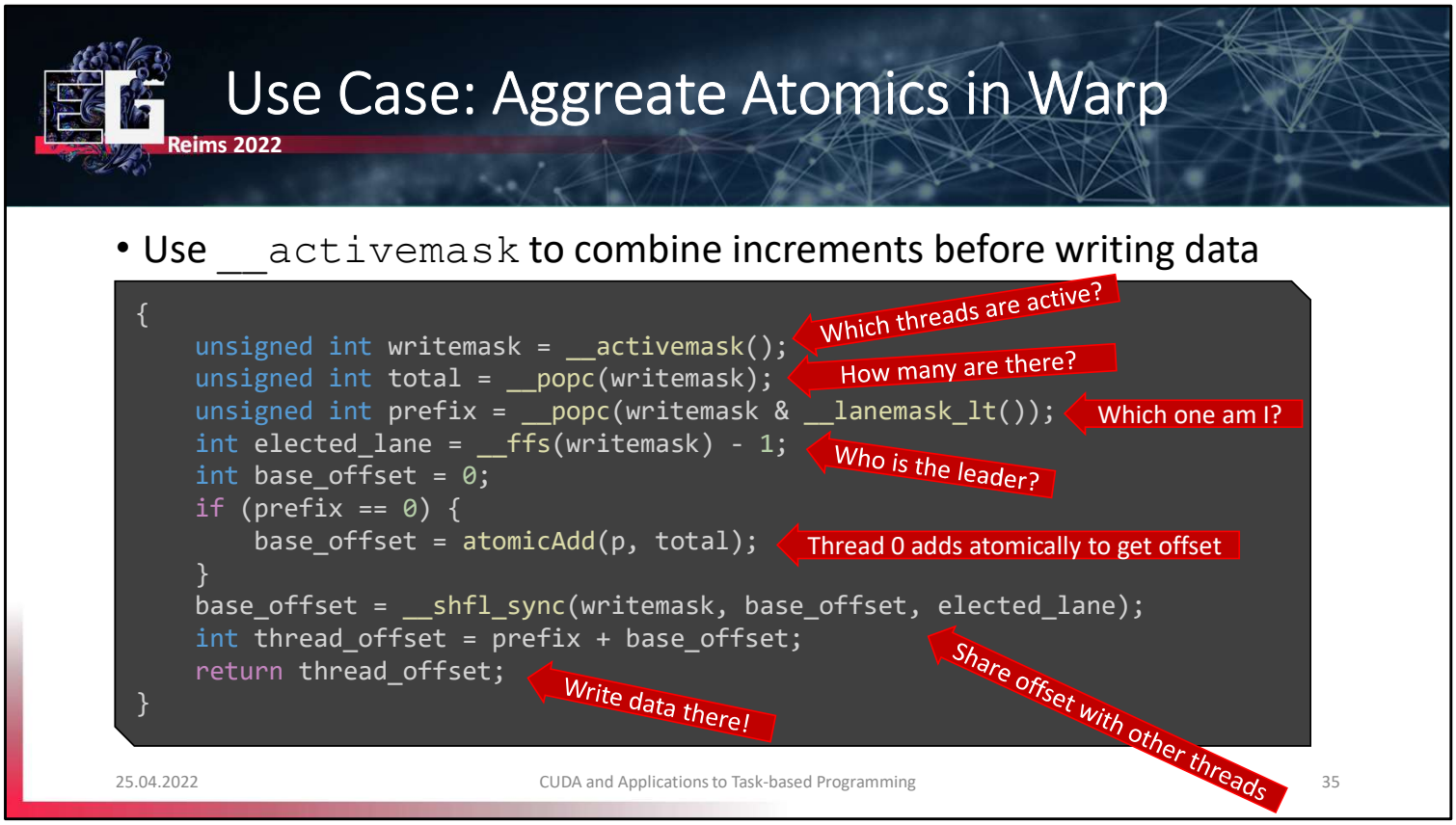
For the first of the two, we can easily find interesting use cases. Consider for instance the task of processing a mesh. For rendering and many other geometry tasks, meshes are split into triangle batches with a given number of indices. When processing must be performed per vertex, e.g., for vertex shading, in order to exploit significant reuse of vertices in a mesh, duplicate vertices can be identified, and each unique vertex can only be shaded once. This was for instance realized in our previous work on enabling vertex reuse on the GPU in software. Previously, we addressed this by shuffling vertex indices and recording duplicates among threads. However, with the Volta architecture, this task maps to a single hardware-accelerated instruction.



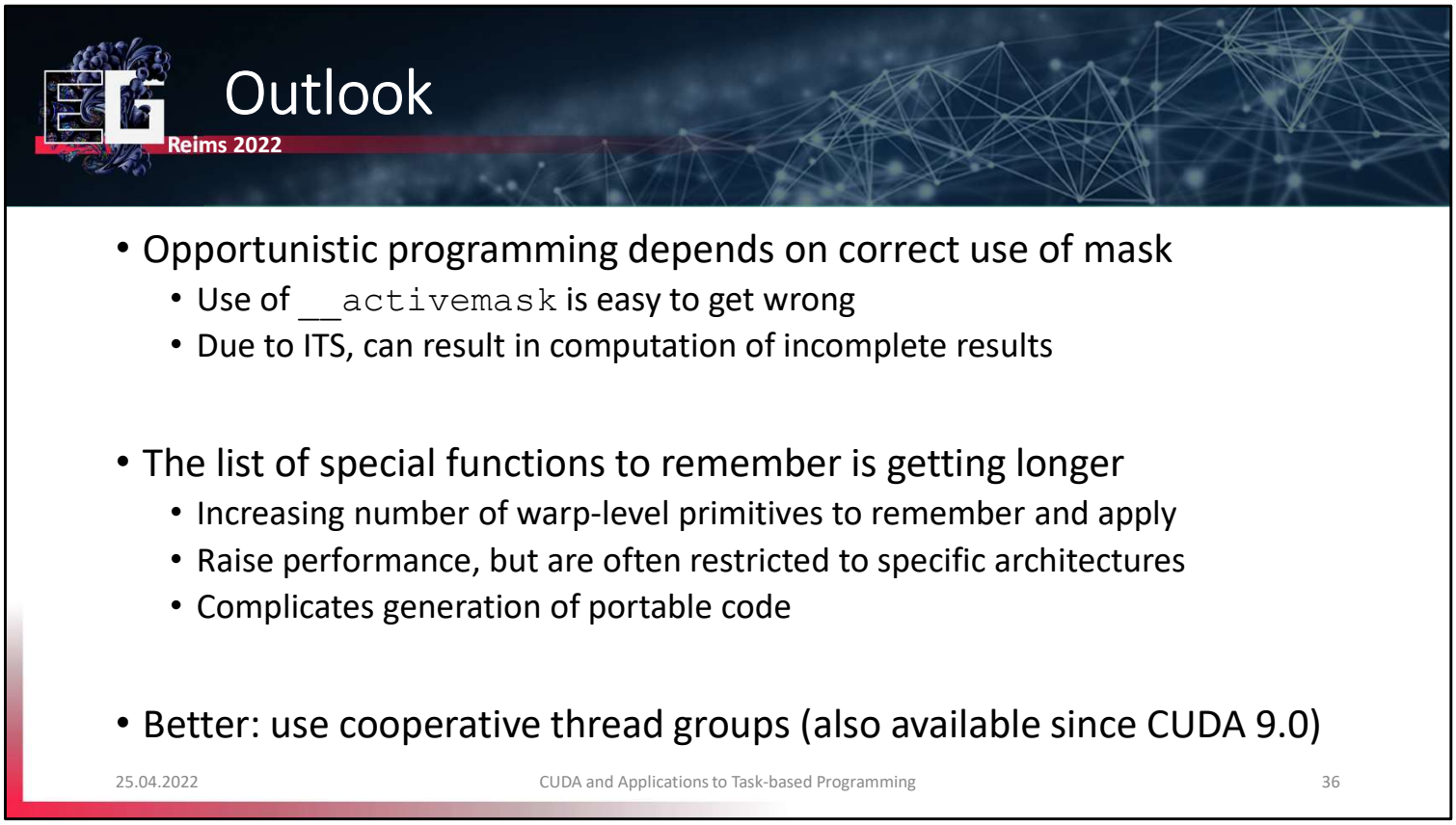
For the latter reduce operation, the application is more straightforward. Consider for instance the implementation of a reduction, where we used shuffling in the later stages to exploit intra-warp communication. The aggregate of different shuffle instructions can now be replaced with a single reduce instruction for the entire warp.



Lastly, another operation is made available that is strongly motivated by the introduction of ITS, and how it affects thread scheduling. With ITS, threads may no longer progress in lockstep, diverge and reconverge somewhat arbitrarily. __activemask is a special warp primitive, since it does not include synchronization and no mask must be provided. This means that it can be called without knowing which threads will be calling it. __activemask returns a set of threads about which it makes no concrete guarantees, other than that these threads are converged at the point where __activemask is called. If the result of this function is used as a mask, other warp-level primitives can use it to opportunistically form groups of threads that are currently converged to optimize particular computations.

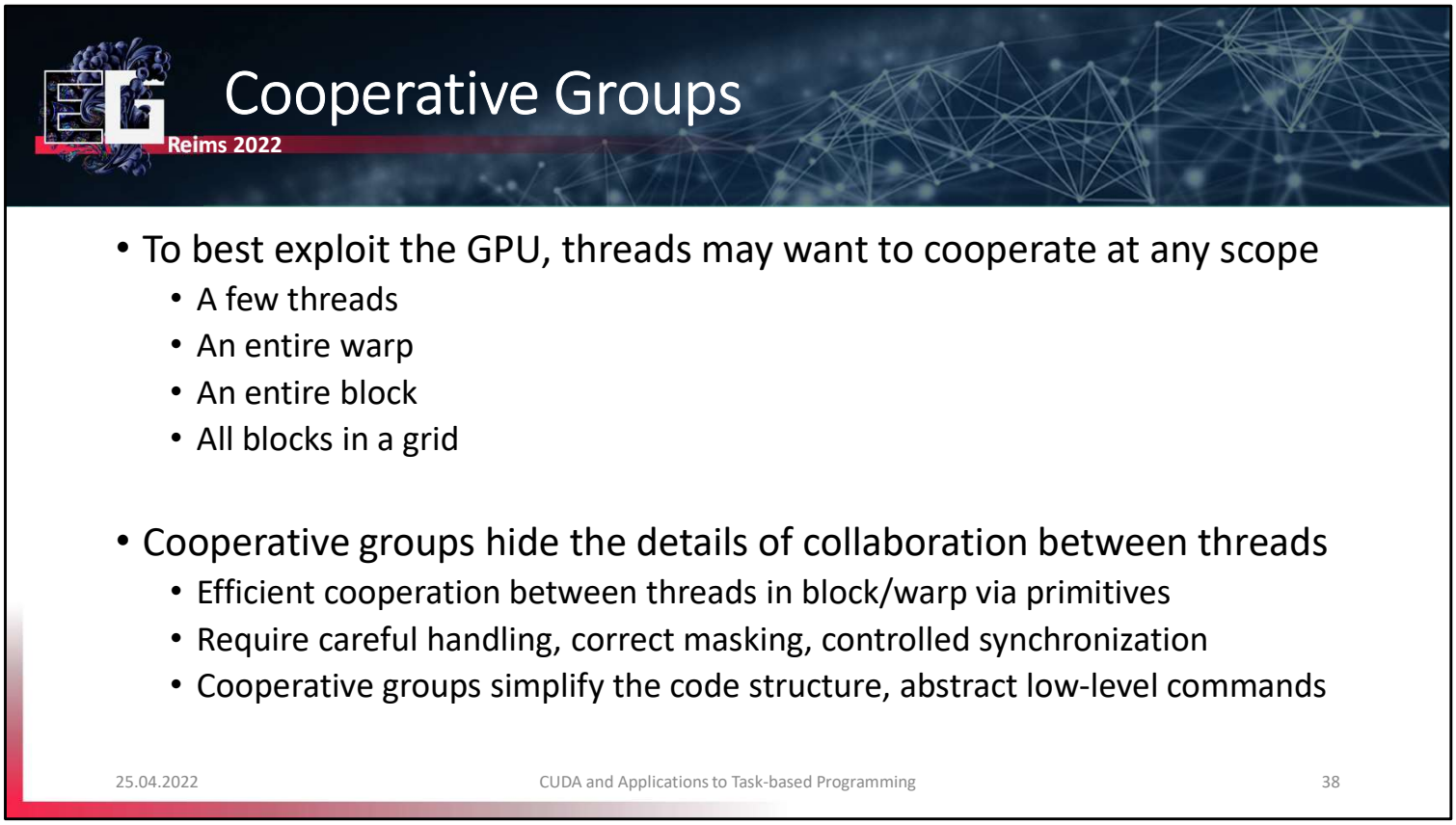


For instance, consider this coding example. While it may be a bit on the intricate side, the goal is actually very simple: At the point where this code is executed, the threads that run it are supposed to write their result to a unique position in a buffer, which they obtain by raising an atomic counter p. To reduce the number of atomic simultaneous operations on the counter p, they opportunistically identify all the threads in the warp that are also currently executing this part of the program, i.e., converged threads. Having identified them, they find the thread in the list with the lowest ID and let it perform a single atomic addition with the size of the converged group. Afterward, every thread in this opportunistic group writes their entry to an appropriate offset in the target buffer.

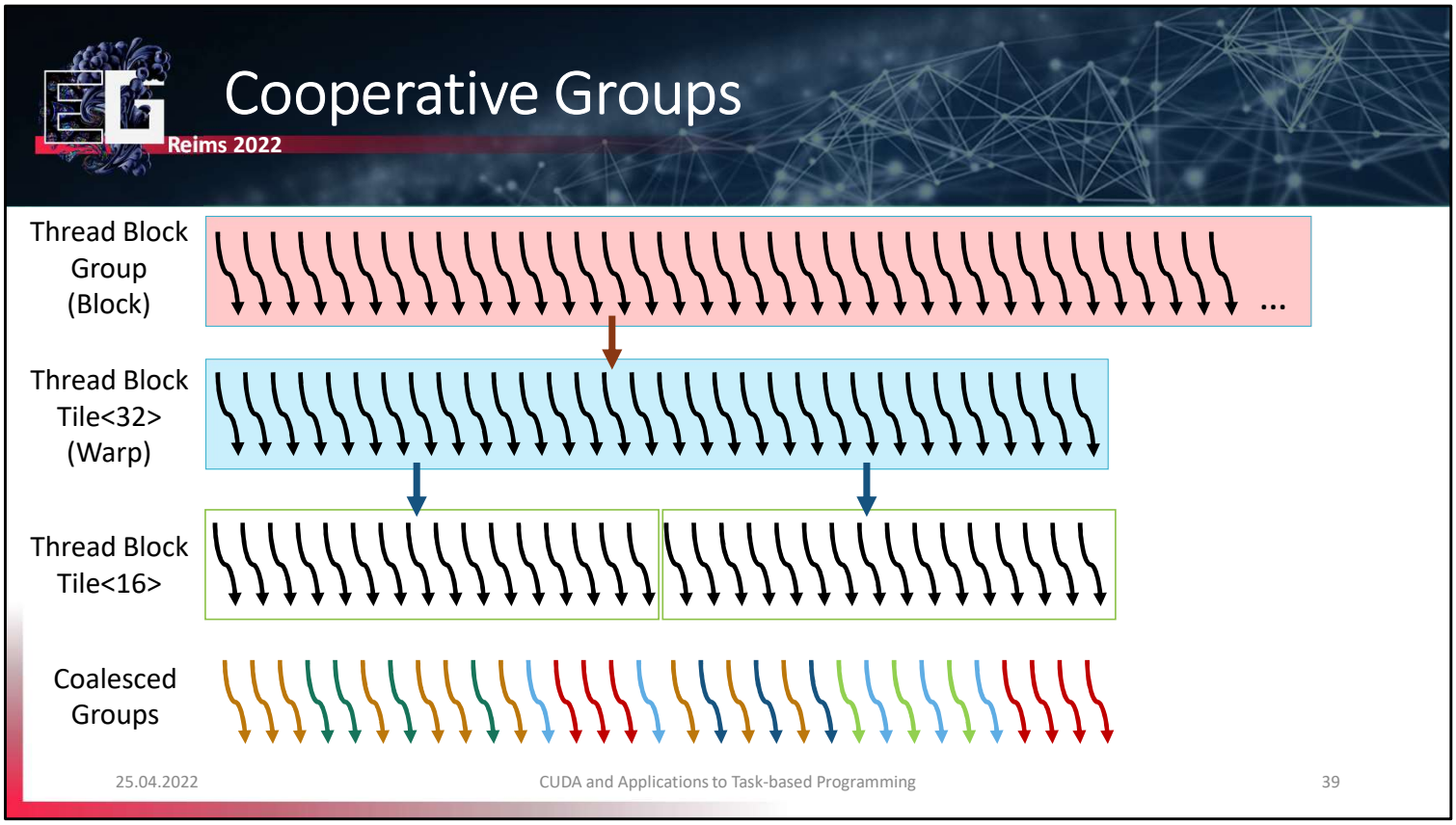


All of these new instructions are helpful, but they also illustrate something else: getting optimal performance out of the GPU is getting more and more intricate. Comparably simple goals, like the one realized in the example we just gave, require a lot of careful design, correct handling and interpreting of bitmasks, and remembering the individual optimizations that can be done in hardware. This may seem discouraging, especially for newcomers to CUDA. However, in addition to exposing these new low-level operations, CUDA also now provides developers with a helpful new library called cooperative groups, which encapsulates these behaviors but abstracts the low-level details for improved usability.

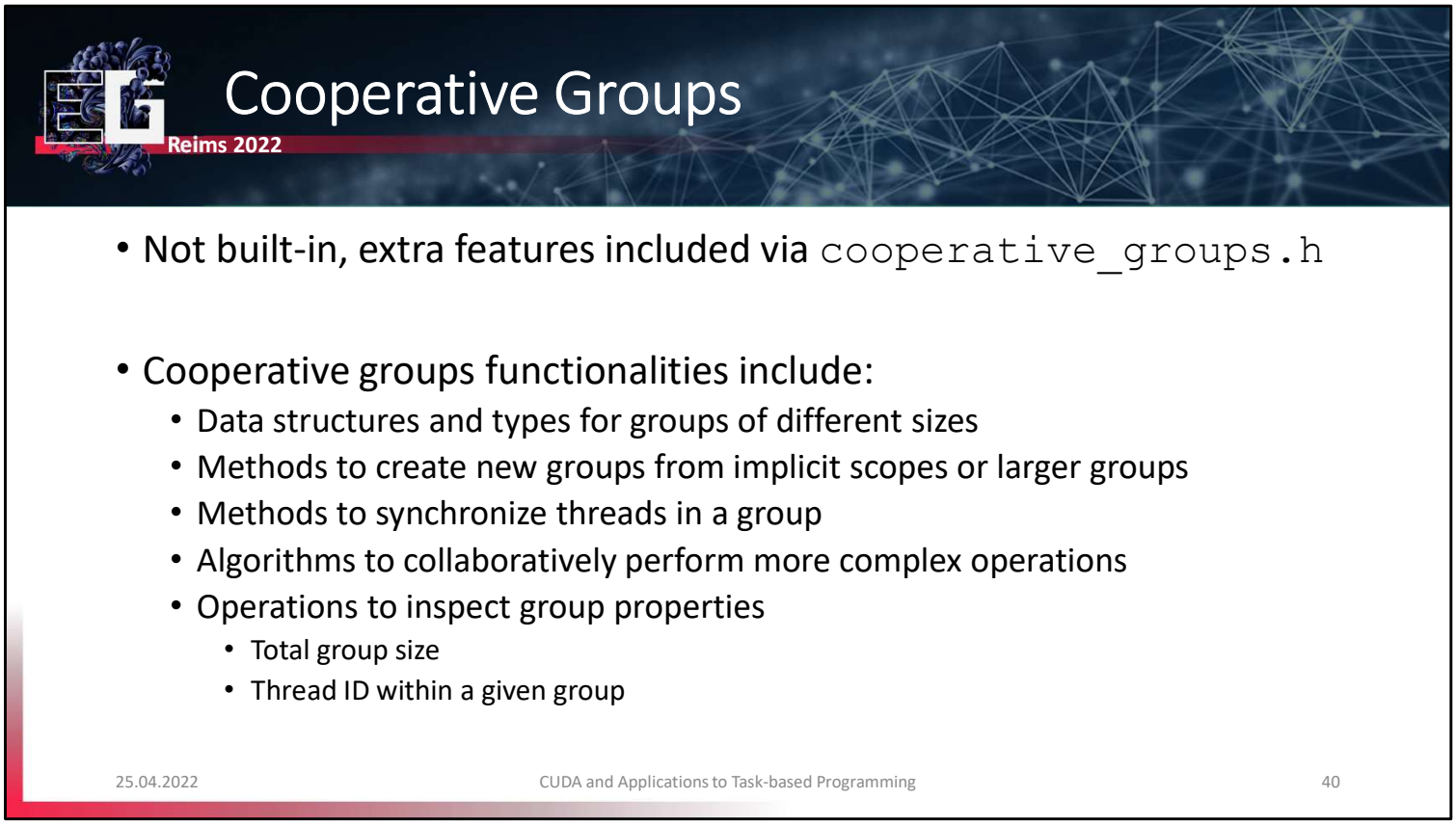
This is exactly the topic that we will be dealing with in the next section of this tutorial.



Cooperative groups can be seen as NVIDIA‘s commitment to the idea that cooperation is key, regardless of whether it happens across multiple blocks, within a block, within a warp, or even just a few threads that happen to execute together. At each of these levels, it is important that developers can exploit the means for cooperation between threads, and that they can exploit it easily. Cooperative groups try to unify the defining properties of thread groups with a common utilization principle that can abstract away many of the intricate, low-level details.

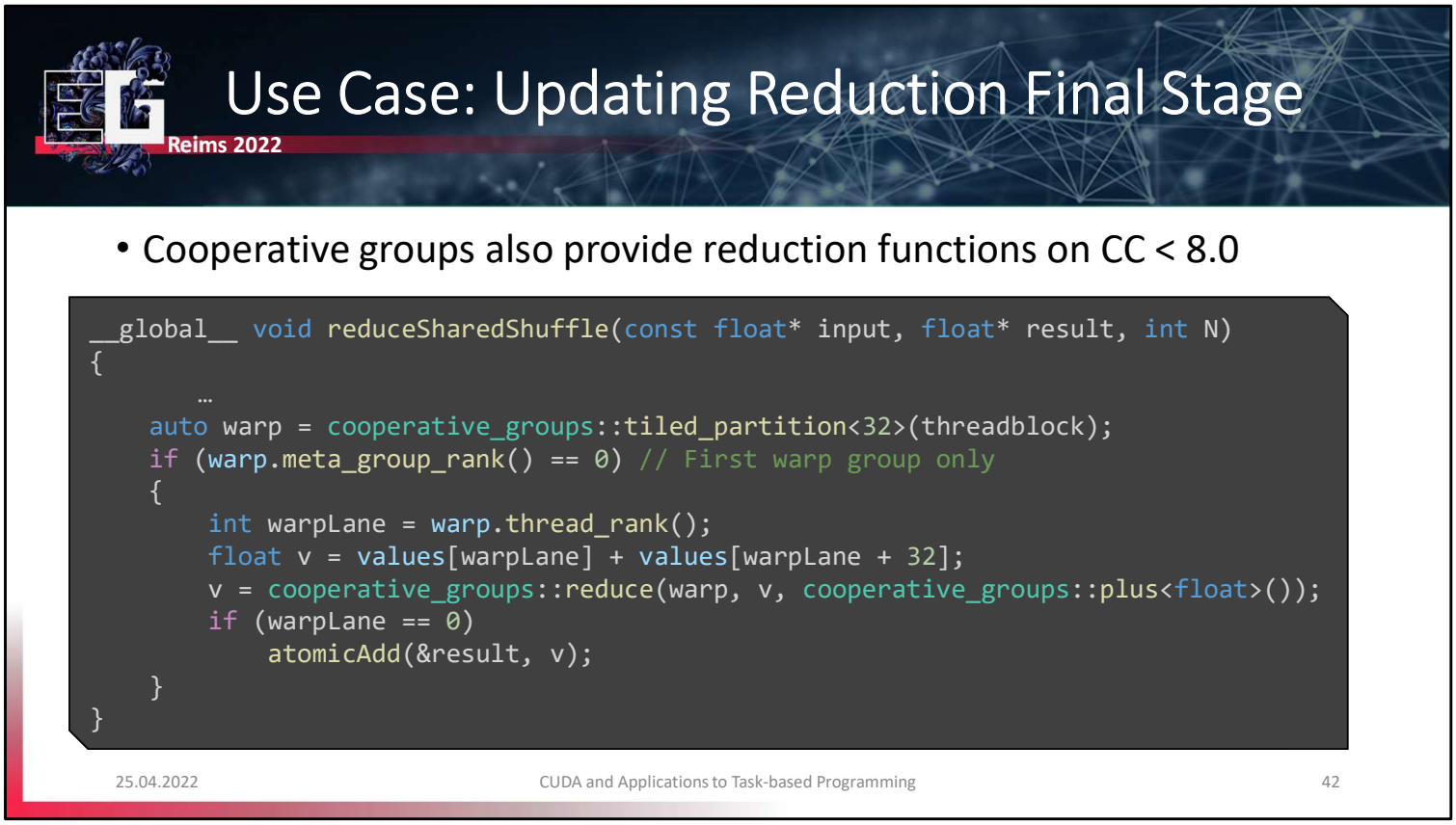


To illustrate this idea, we can visualize different levels of the execution hierarchy and associate each of them with a particular pendant in the cooperative groups model. Conventionally, CUDA uses built-in variables to identify the block that each thread belongs to. With cooperative groups, each thread can retrieve a handle to a group that represents its block, which is of the thread block group type. A thread block group can be further partitioned into thread block tile groups with a given size that must be a power of 2 and no larger than a warp (except for the experimental cooperative groups extensions). Somewhat orthogonal to groups created based on size, but always at most of size 32 is finally the coalesced group, which represents a group of threads that are, at some point in time, converged (compare to our previous example of opportunistic warp-level programming).

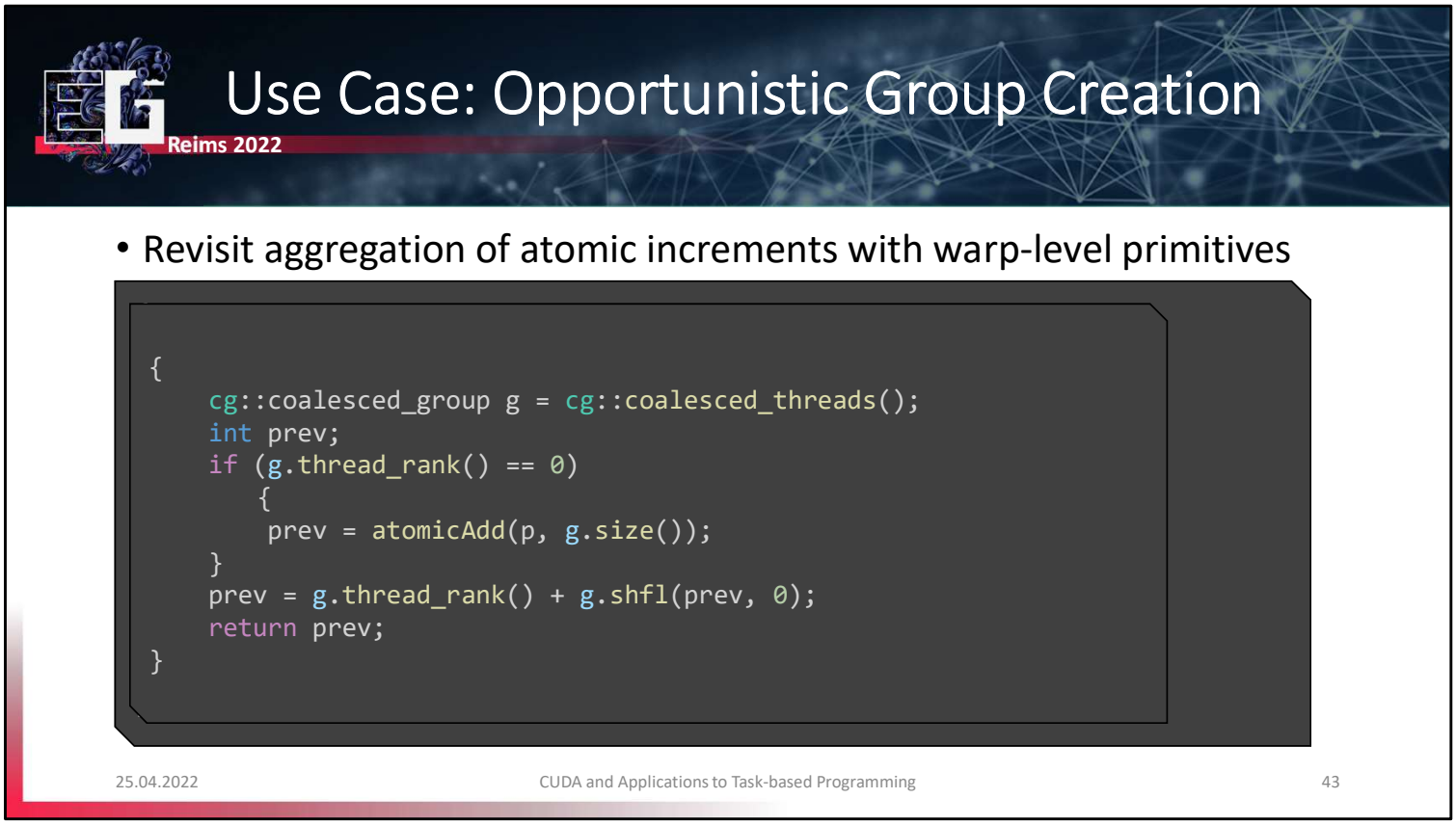


The cooperative groups design is available through an additional header, which includes data structures that describe types for the individual groups of threads, methods to synchronize groups, algorithms that allow them to collaborate toward a specific goal, and functions that developers can use to access generic properties of groups, such as their size.

Here, we see examples for the creation of a thread‘s variable describing a group that represents its thread block, a group that represents its warp, a smaller group representing a 16-wide tile of the block that the thread happens to fall into, and lastly the group of converged threads that this thread is a part of. The threadblock group, like all the others, has the option to synchronize with the other threads in it. Synchronization is now abstracted by the group interface, so instead of calling the specific __syncthreads(), developers may simply call the .sync method. Each group will also provide its members with a unique „rank“ within each respective group, regardless of their higher-level position. E.g., a thread with threadIdx.x == 7 may very well be the thread with rank 0 in a coalesced group, such that ranks always run from 0 to group.size(). Furthermore, tiled partition groups and coalesced groups may exploit fast warp-level primitives as methods of their groups. Note that providing masks is not necessary: the threads that should participate are an implicit property of the group.

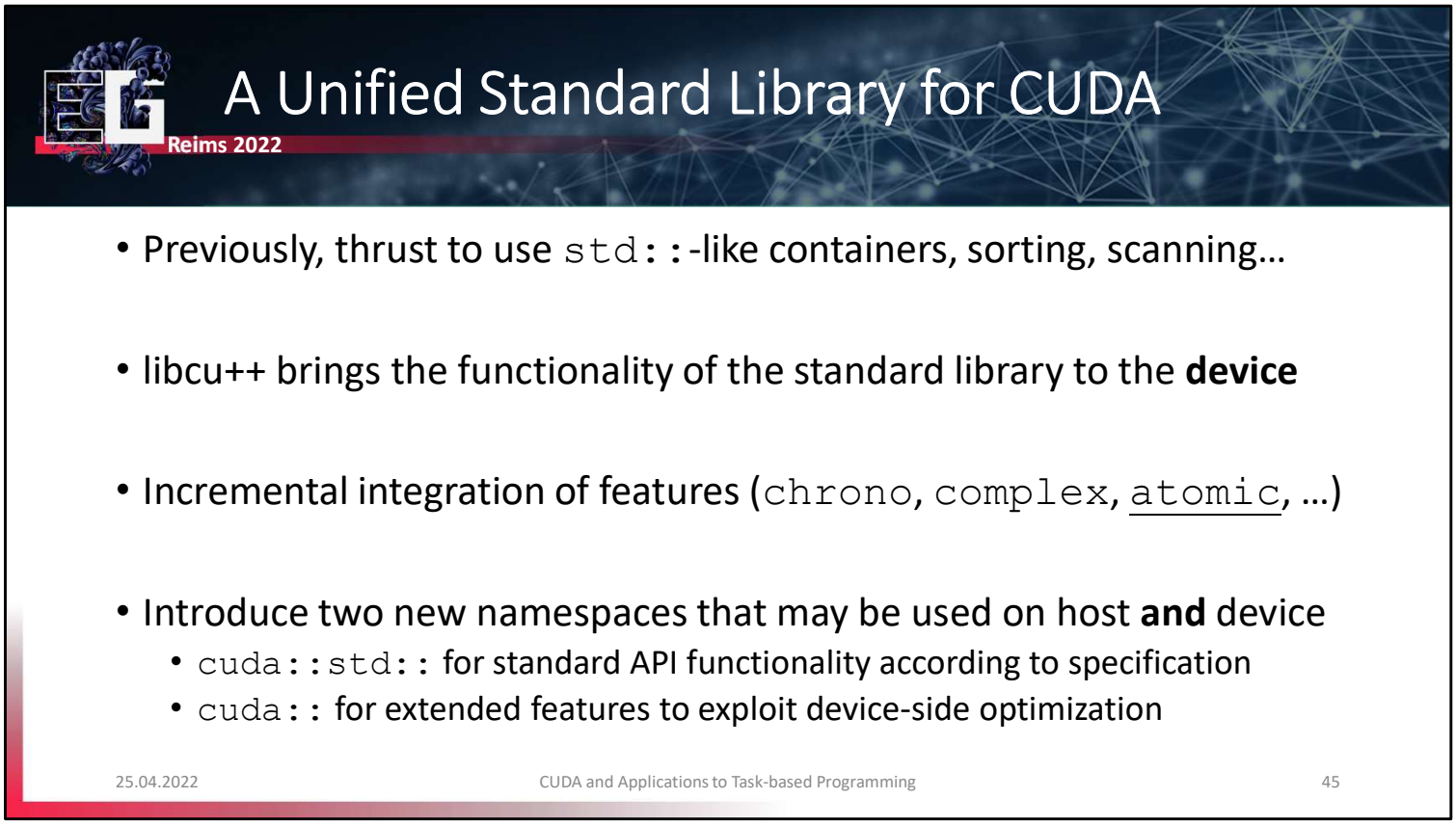


We can use cooperative groups to rewrite the final stage of our reduction with these new mechanics. While in this case, the code does not become shorter, it arguably becomes clearer. Behavior is not explicitly governed based on thread ID. Instead, a block is first partitioned into warps, and only a single warp chooses to participate in the final stages of the reduction. Second, the warp then proceeds to call the more general reduce method, which now may be called even on architectures that do not support the __reduce intrinsic. E.g., on Turing cards or earlier, the reduce method will default to shuffle operations. The inclusion of high-performance primitives where possible and efficient software fallbacks elsewhere is an important step toward additional relief for developers who can now quickly write code that performs well on multiple architectures without introducing special control flow paths.

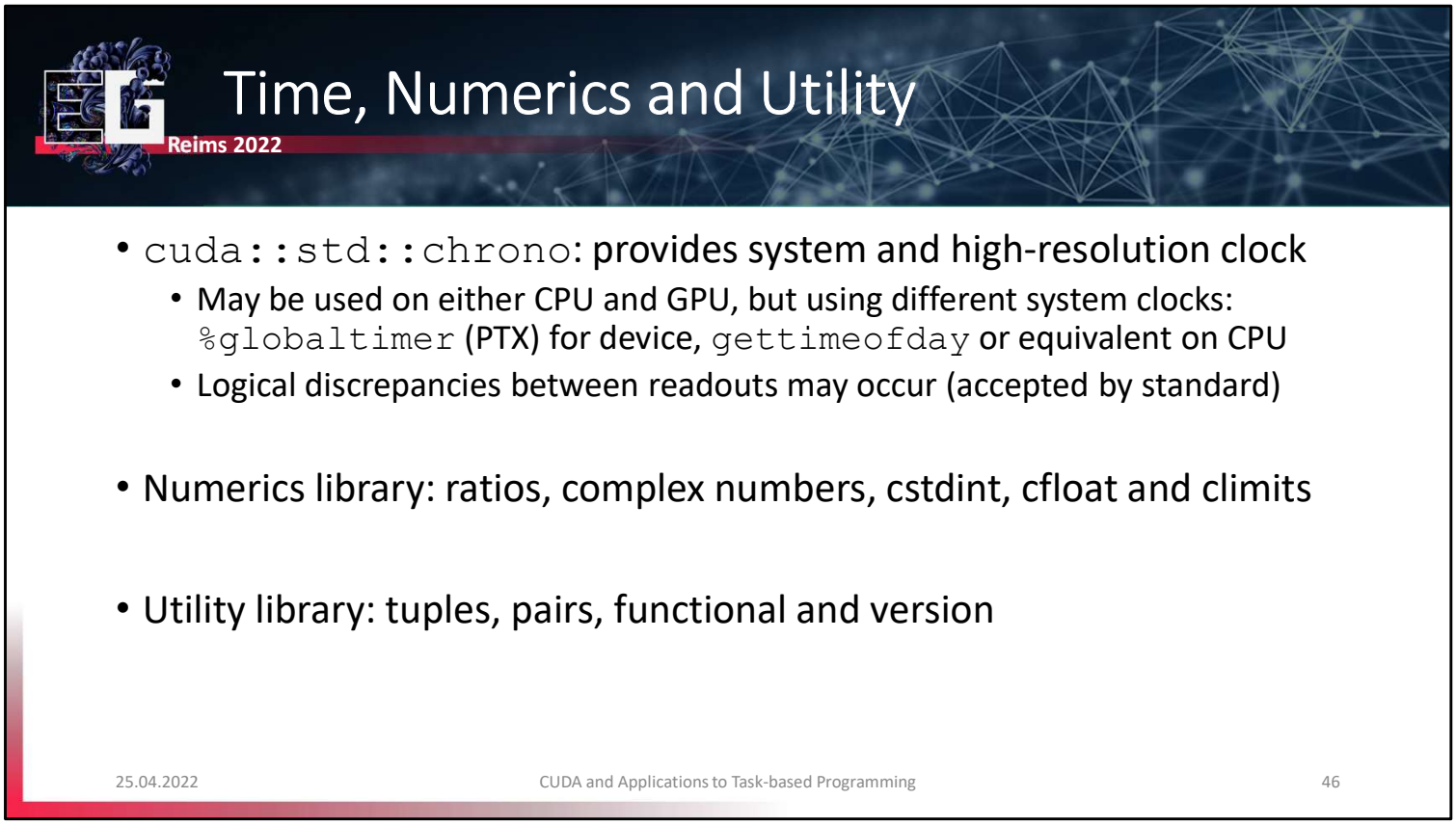


Finally, we can revisit the solution we previously explored for opportunistic warp-level programming. The intrinsics and manipulations we used before enabled us to recreate the behavior that cooperative groups is built upon: the focus on collaborative threads. With the creation of a coalesced group, identifying leader threads, group size or shuffling results among coalesced threads becomes trivial. Internally, of course, the same manipulations are still taking place, but are now hidden from the developer who can achieve the same efficiency with much cleaner and more comprehensible code.

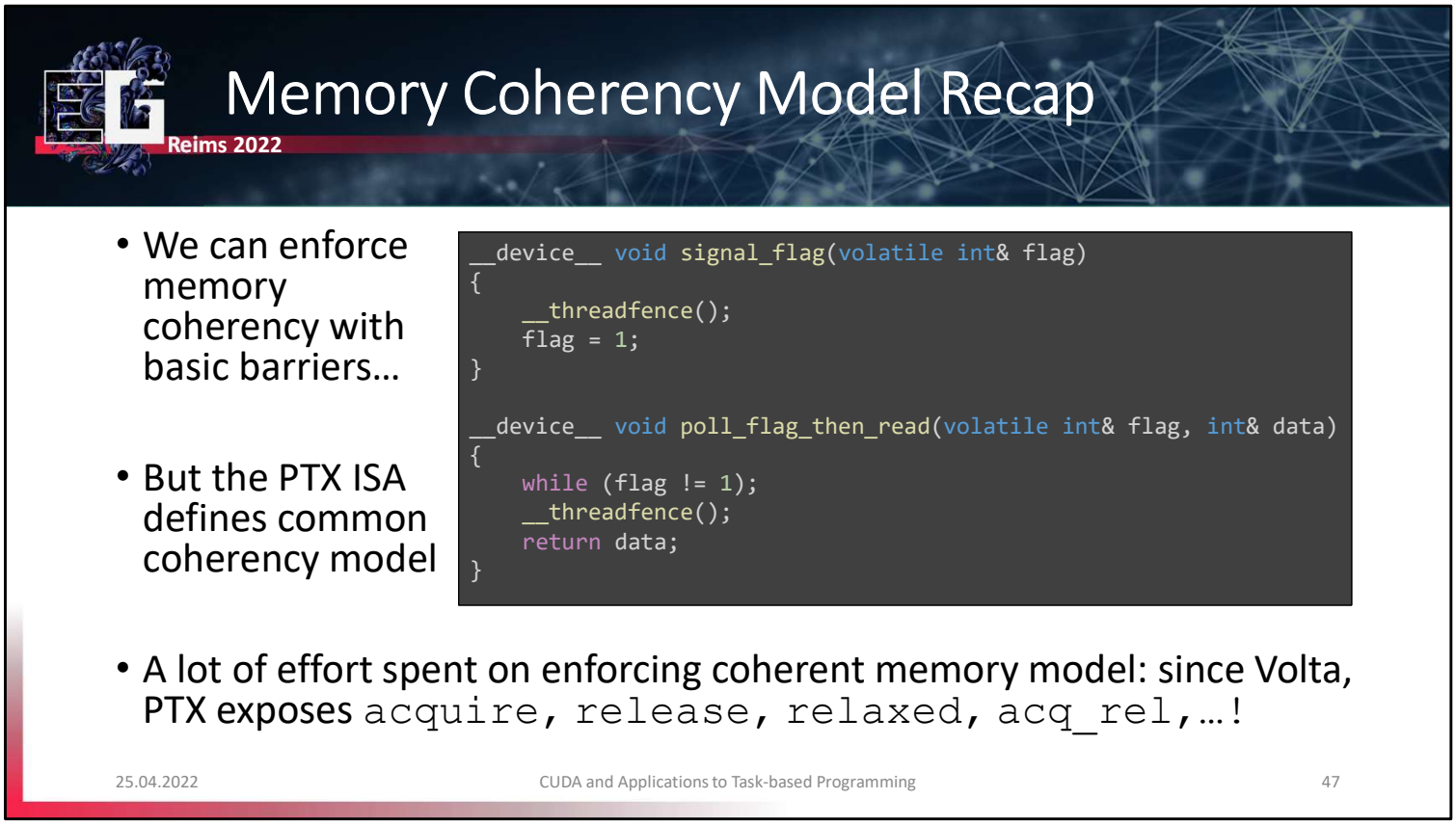
Another exciting new feature that promises to make CUDA much more convenient is the CUDA standard library, libcu++.



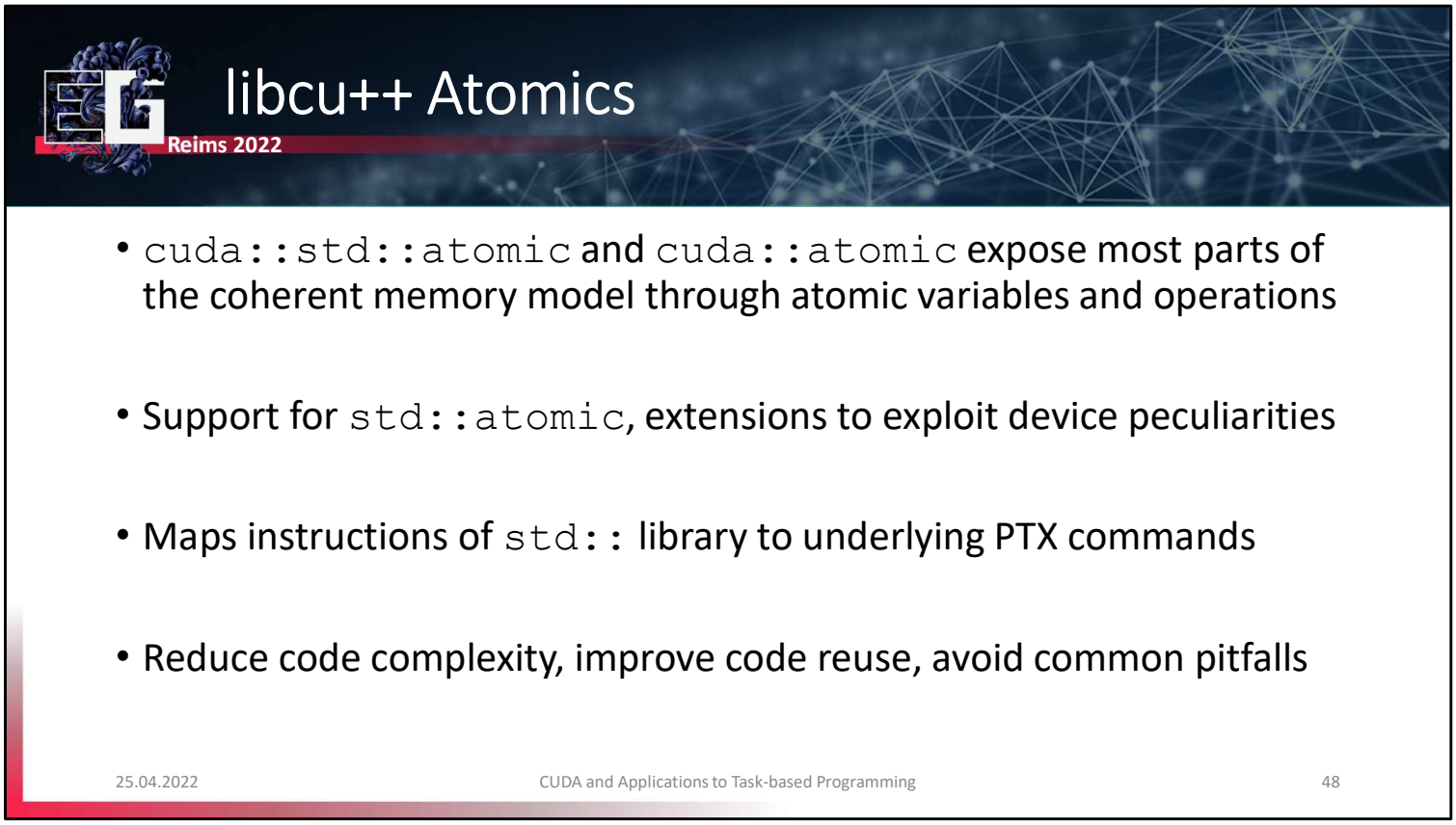
Up until now, to have the comfort of the standard library, CUDA provided thrust, which offers commonly used operations for sorting, scanning, as well as basic containers and interfaces on the host side. However, with libcu++, NVIDIA is bringing the functionality of the standard library, according to specification (and beyond) to the device side. This is an incremental effort. The first parts that have been realized include the chrono library, numeric features such as complex numbers, and atomics. To conform to the specifications, the library provides a namespace cuda::std. However, since the GPU has architectural peculiarities that are not completely captured by specification, it also includes the opt-in name space cuda::, which offers data types and algorithms with additional parameters and settings.



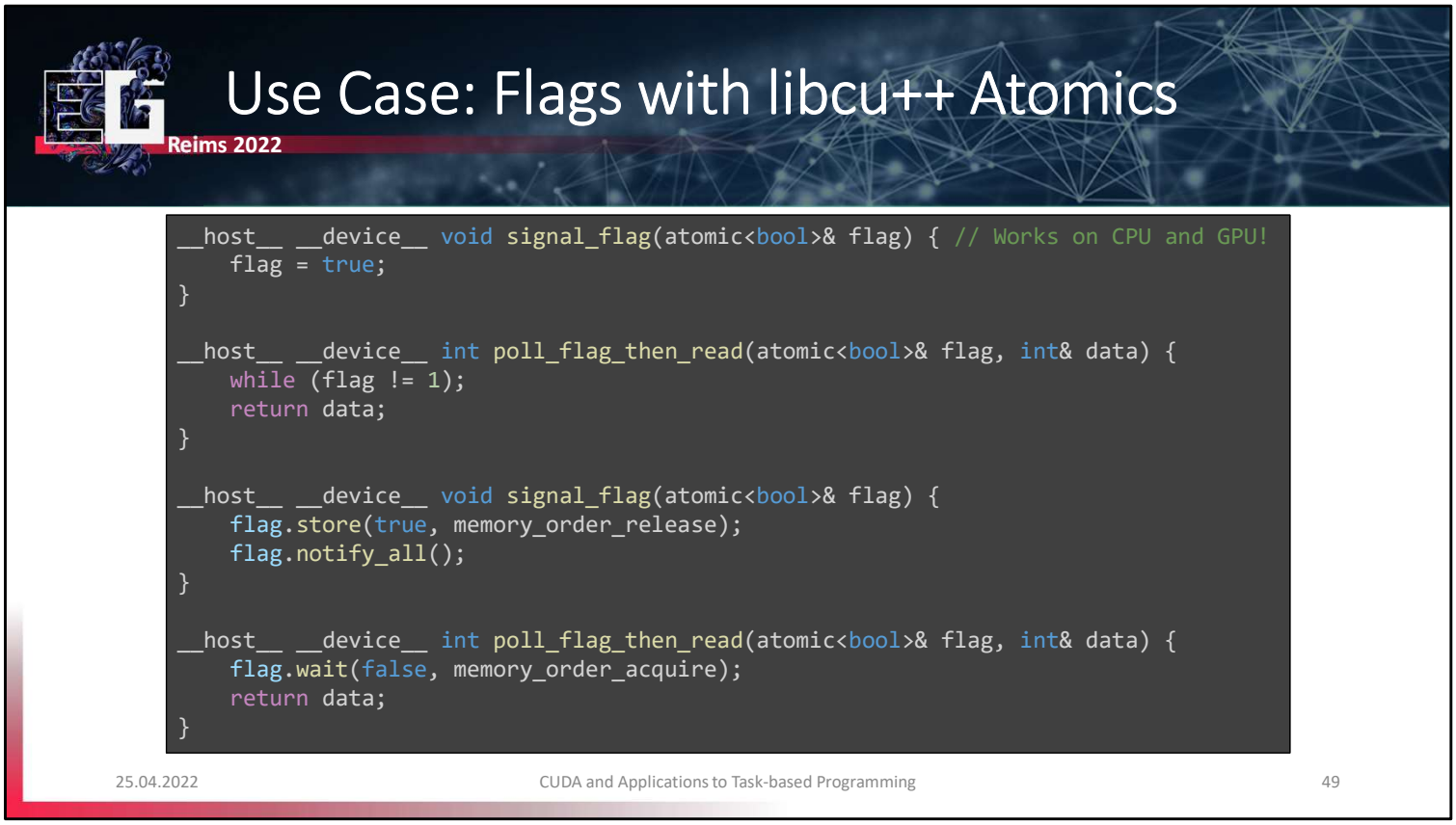
For chrono, the CUDA standard library now offers a system and high-resolution clock that make use of the special built-in clock registers defined by the PTX ISA. In the numerics portion, the library includes support for complex numbers, ratios, as well as limits for built-in types. The utility library currently focuses on the implementation of tuples and pairs. A highly demanded addition is the vector containter which, according to the developers, is already high up on their TODO list.



Before we focus on atomics in the CUDA standard library, let‘s first quickly recap the basic CUDA memory model. Regarding memory ordering, the __threadfence operation is an established, though somewhat crude, mechanism for achieving ordered access, by acting like a general barrier. However, a considerable amount of time has now been spent on actually enforcing a clearly defined memory coherency model on NVIDIA GPUs, that reflects that of common CPUs. This memory coherency model has clear definitions for access with release and acquire semantics, which are much more nuanced than thread fences.



This is where the libcu++ atomics come in. Currently, they are the preferred way to expose this modern memory coherency model to C++ without the need to write explicit PTX instructions. By introducing a memory coherency model that mirrors the CPU, as well as exposing it through a standard library, writing CUDA code now becomes significantly more portable. In addition, the ability to write __device__ __host__ functions that behave the same on both architectures enables a significantly higher code reuse.



This is an example of coding with the libcu++ standard library. Note that we can replicate the same operations as before, which needed to be protected by __threadfence and decorated with a vague, non-portable volatile qualifier, with clear atomic definitions instead. In addition to atomic store, load and arithmetic operations, the new atomics also support waiting and notification of waiting threads.

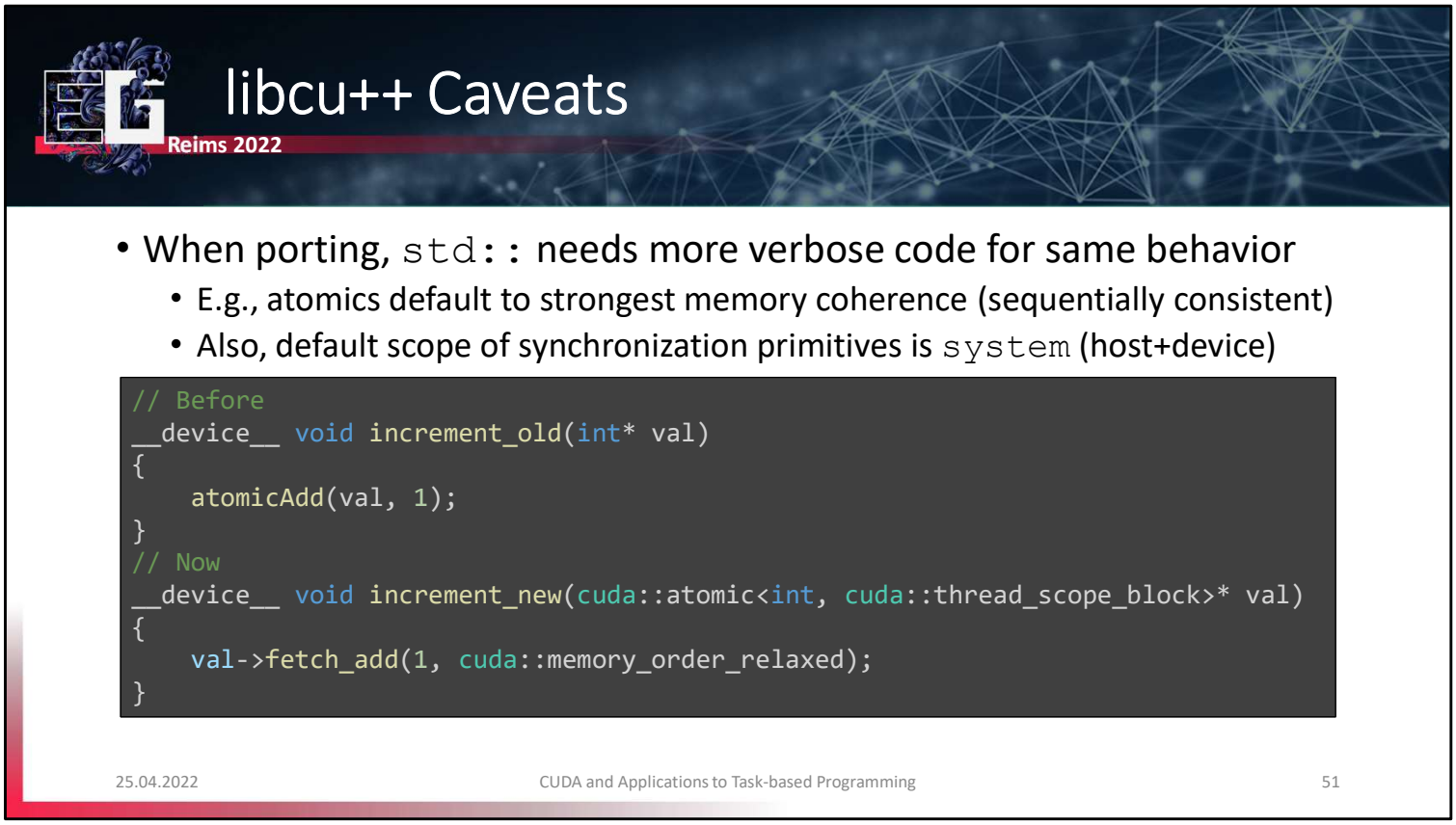
Use Case: Locks and Critical Sections

Reims 2022

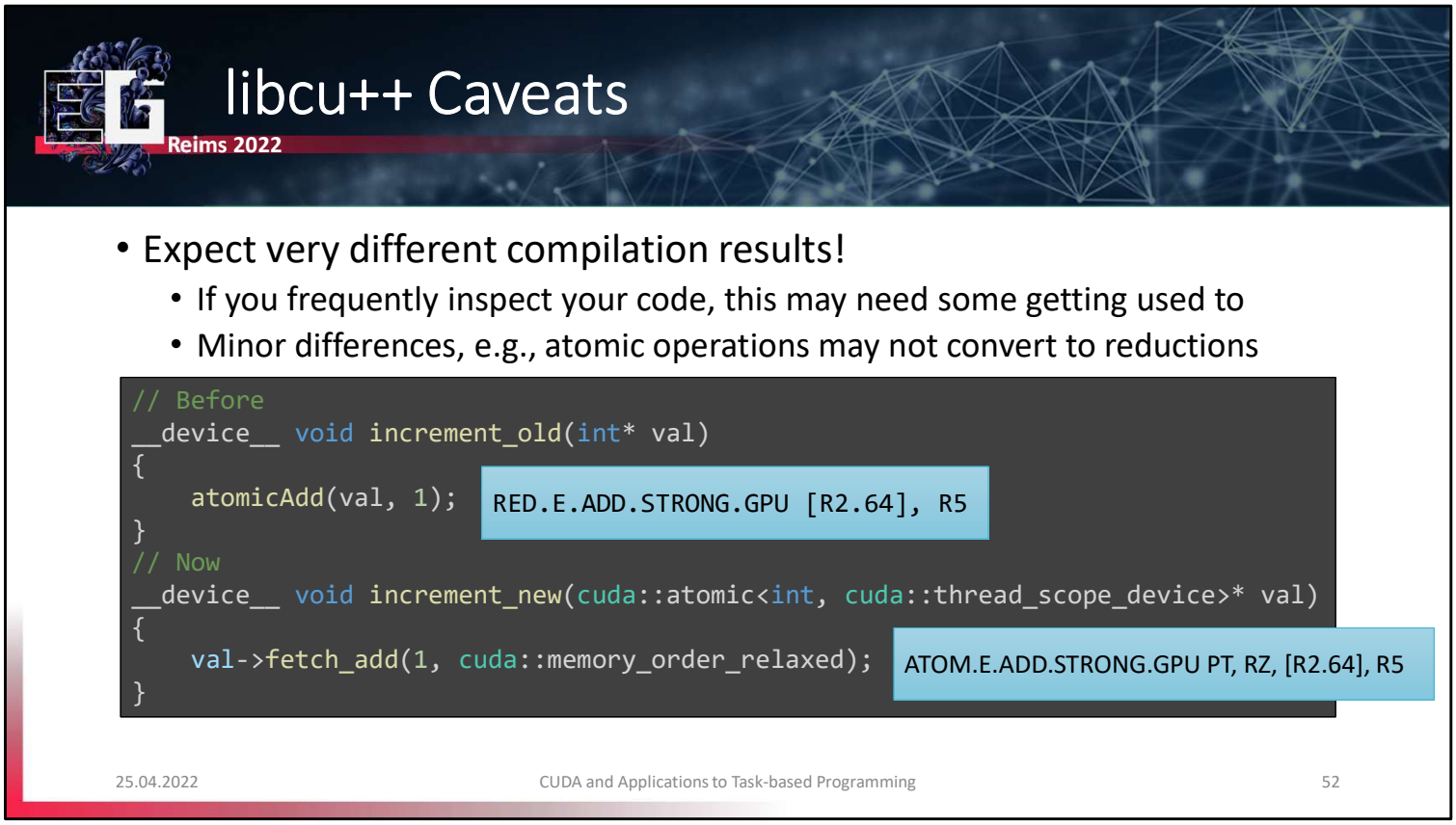
- Synchronization primitives from `std::` already available in libcu++
  - May of course only be used on devices that can utilize them (Volta and later)

```
__global__ void incrementCounter(cuda::std::binary_semaphore* semaphore)
{
    semaphore->acquire();
    count++;
    semaphore->release();
}

int main()
{
    cuda::std::binary_semaphore* sem;
    cudaMallocManaged(&sem, sizeof(cuda::std::binary_semaphore));
    new (sem) cuda::std::binary_semaphore(1);
    incrementCounter<<<256, 256>>>(sem);
        …
```


These new features make it easy to create efficient implementations of common synchronization primitives, however, several of them, like binary semaphores, are already included in libcu++ as well to spare developers the additional effort.



While it is definitely on its way to becoming an integral part of CUDA applications, the use of the libcu++ library is not without caveats, especially to long-term users of „conventional“ CUDA. For instance, with the new constructs, achieving the same behavior that developers are used to on the device side can now be much more verbose. E.g., the default behavior of atomic operations conventionally is relaxed, which is not the default in the standard library. Also, care must be taken that, when performance is essential, cuda::std may not be used, since only the primitives in cuda:: offer the ability to define reduced visibility of atomic variables (e.g., shared atomics).



Another, more subtle difference is that while in general, the compiled results of code that uses libcu++ can exploit the memory coherency model better than legacy code, sometimes the result is not what you would expect. For instance, in this case, we perform atomic operations on a variable in both device functions, without using the returned results. The compiler should be able to turn the atomic addition into a simple reduction, however, in the case where the standard library is used, it cannot make this conversion.

libcu++ Caveats

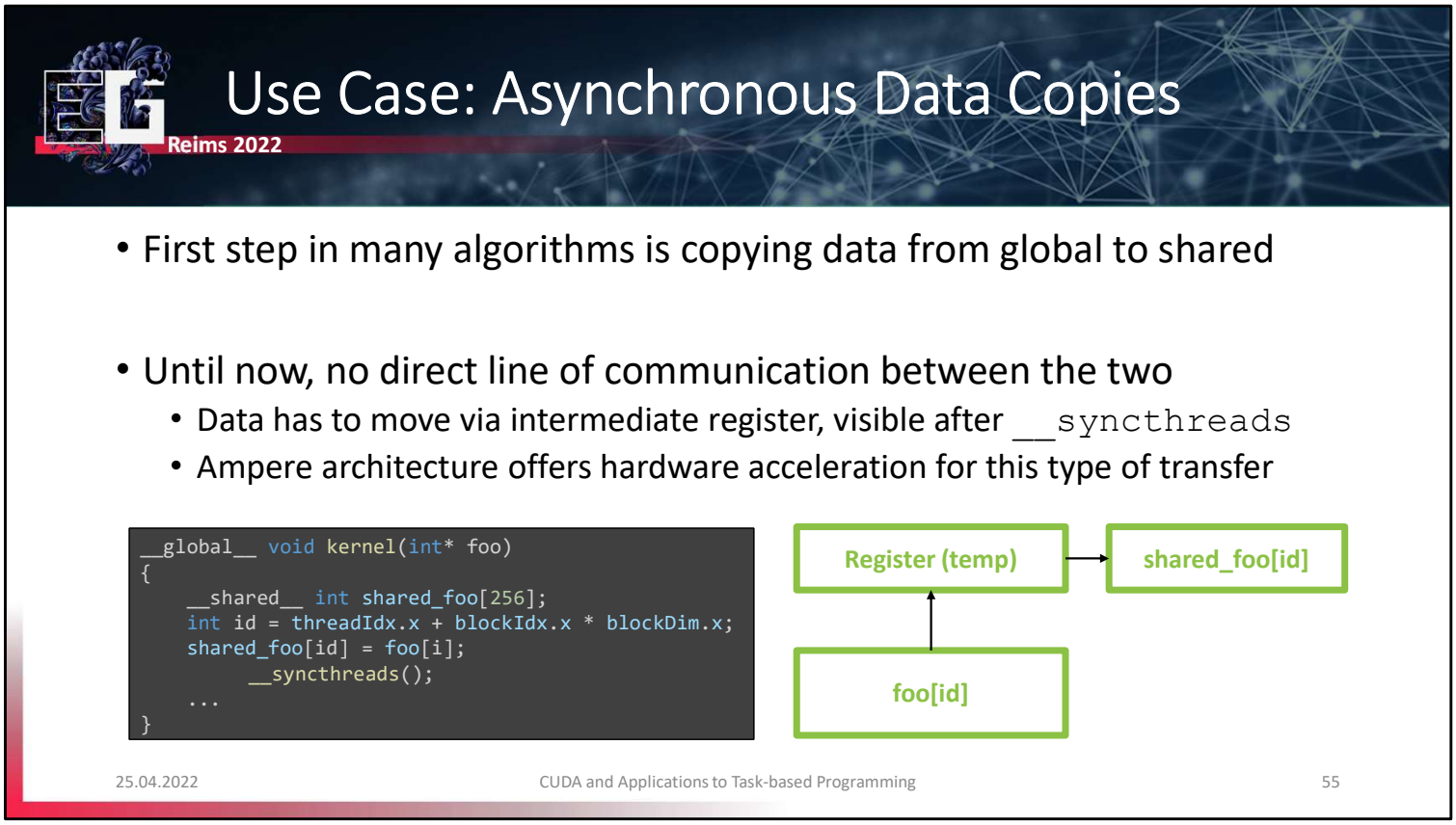
Reims 2022

- Features are still being added incrementally
  - Key motivator: expose memory coherency model through `std::atomic`
  - Aim to improve portability, but majority of `std::` features still missing
  - A full replica of the standard library may not be practical or even feasible
  - Cannot just copy existing CPU code and run it on the GPU
- Support is introduced based on community demand
  - Supporting the full standard library on the GPU is an Herculean effort
  - Coming up: `std::vector`, `std::iostream`, …

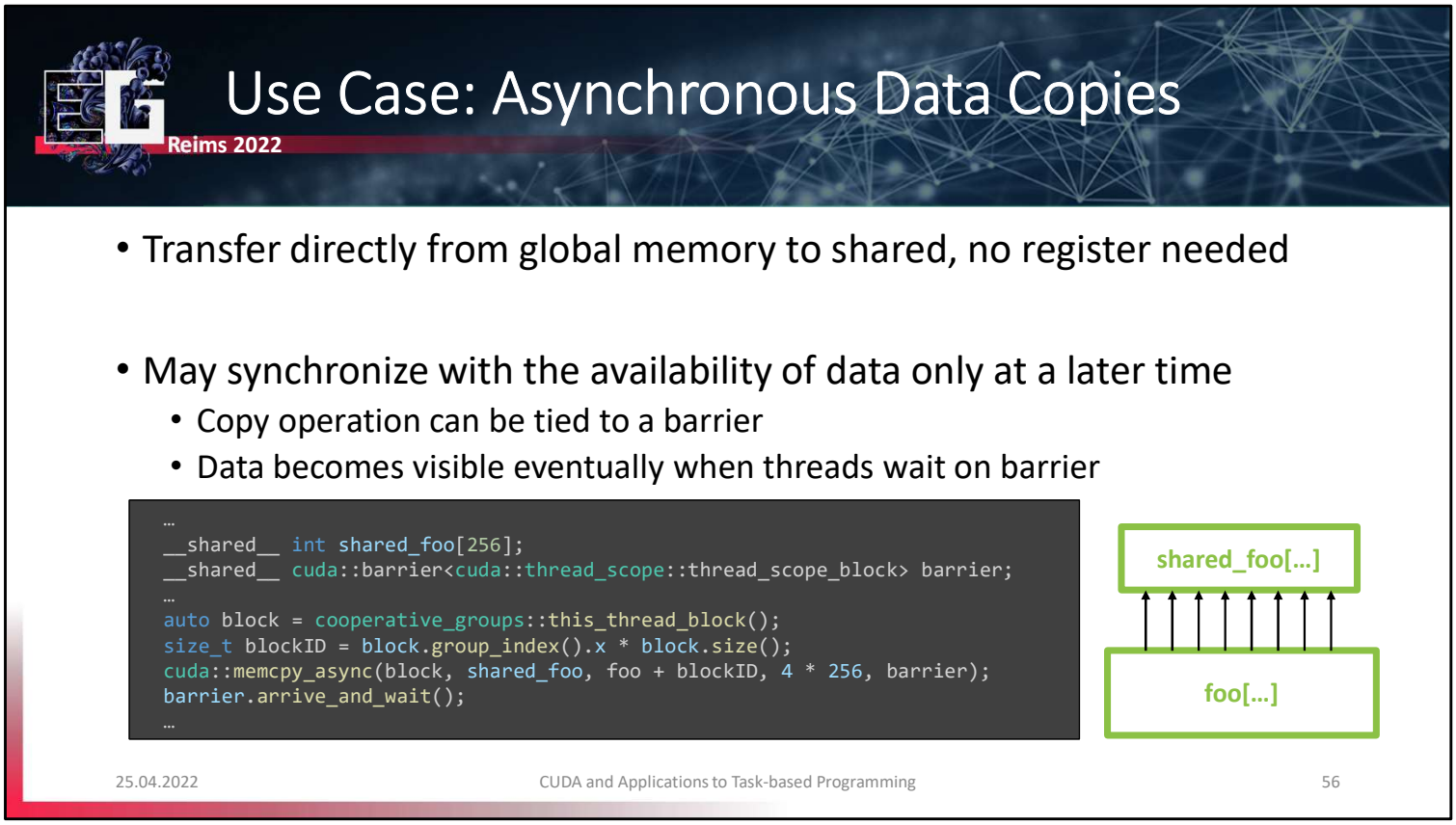
Use Case: Architecture-Agnostic Code

- Basic reduction with global atomics (we saw better solutions already)

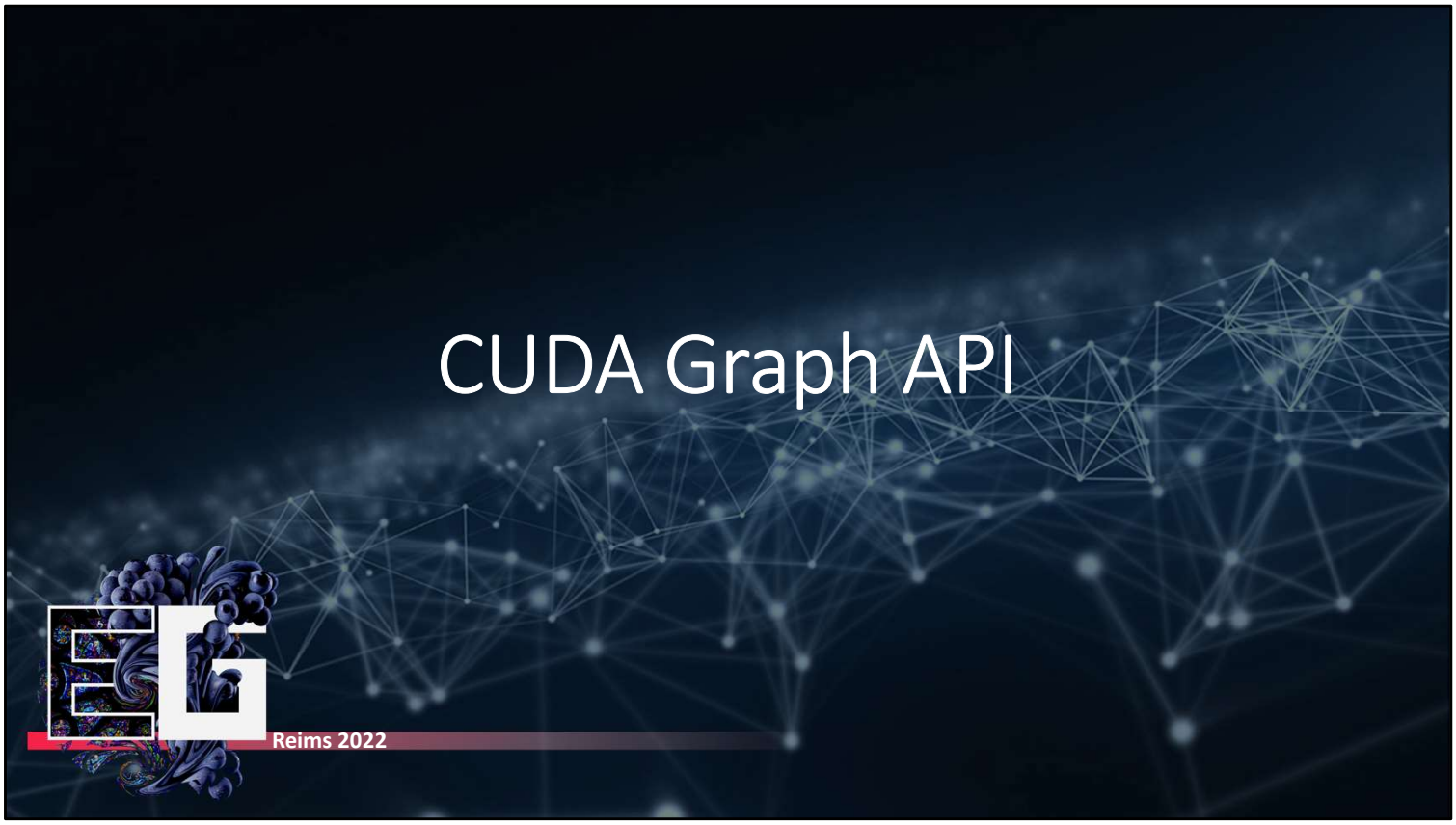
```
__host__ __device__ void reduceAtomic(int tId, int Threads, int N, int* input, cuda::std::atomic<int>* result)
{
    if (tId >= N)
        return;

    // Compute input portion to be handled by each thread
    int perThread = N / Threads;
    int myStart = perThread * tId;
    int myEnd = (tId == Threads - 1) ? N : myStart + perThread;

    // For each value in the assigned portion, atomically add it to accumulated sum.
    for (int i = myStart; i < myEnd; i++)
        result->fetch_add(input[i]);
}
```



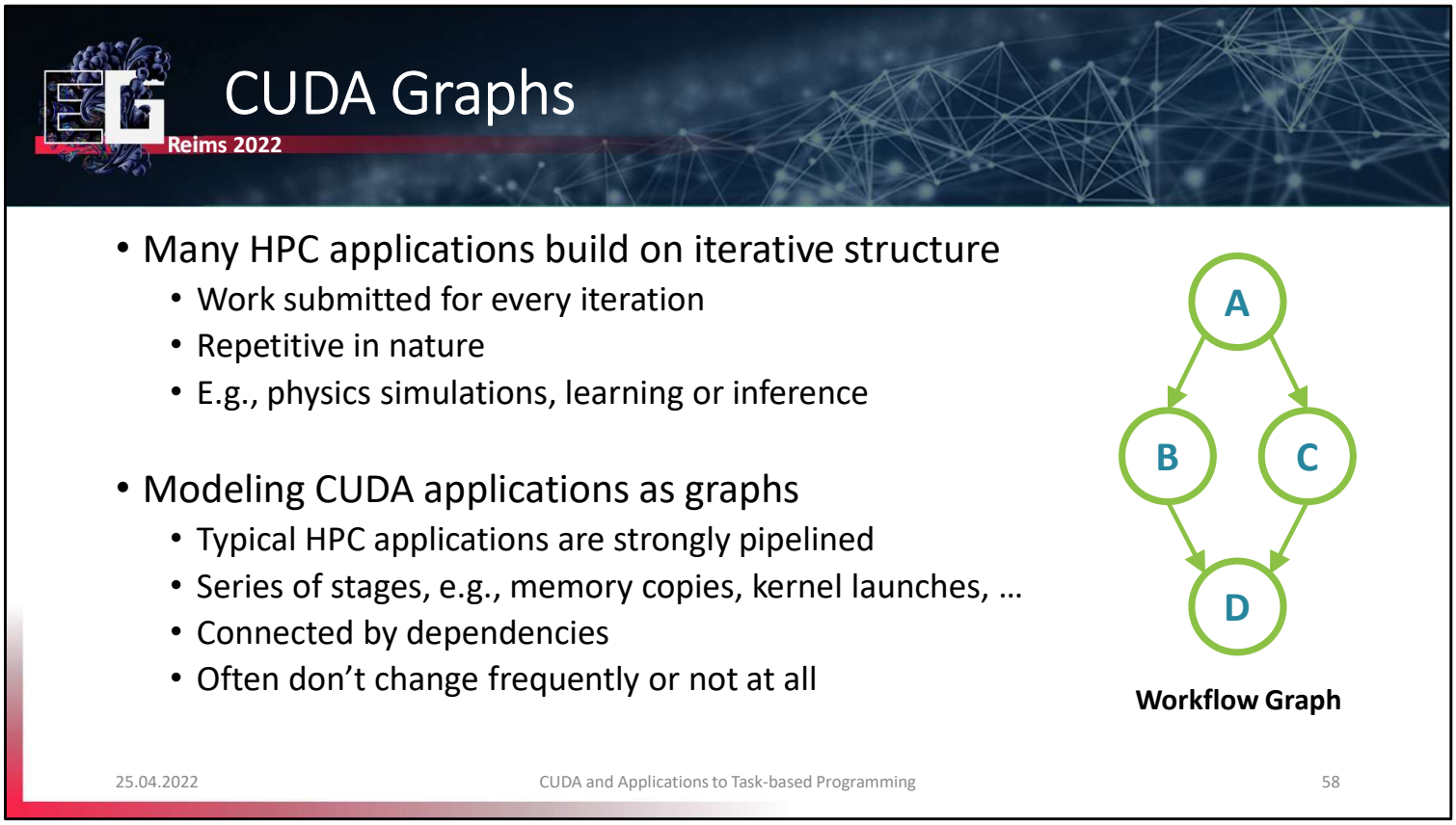
Libcu++ also includes definitions for CUDA barriers, which largely mimic the behavior of std::barrier types. These become important to exploit, e.g., a new feature of the Ampere architecture for efficiently transferring data from global to shared memory. Until now, such transfers, which are very common in most kernels, had to go through an intermediate register before being stored in memory.



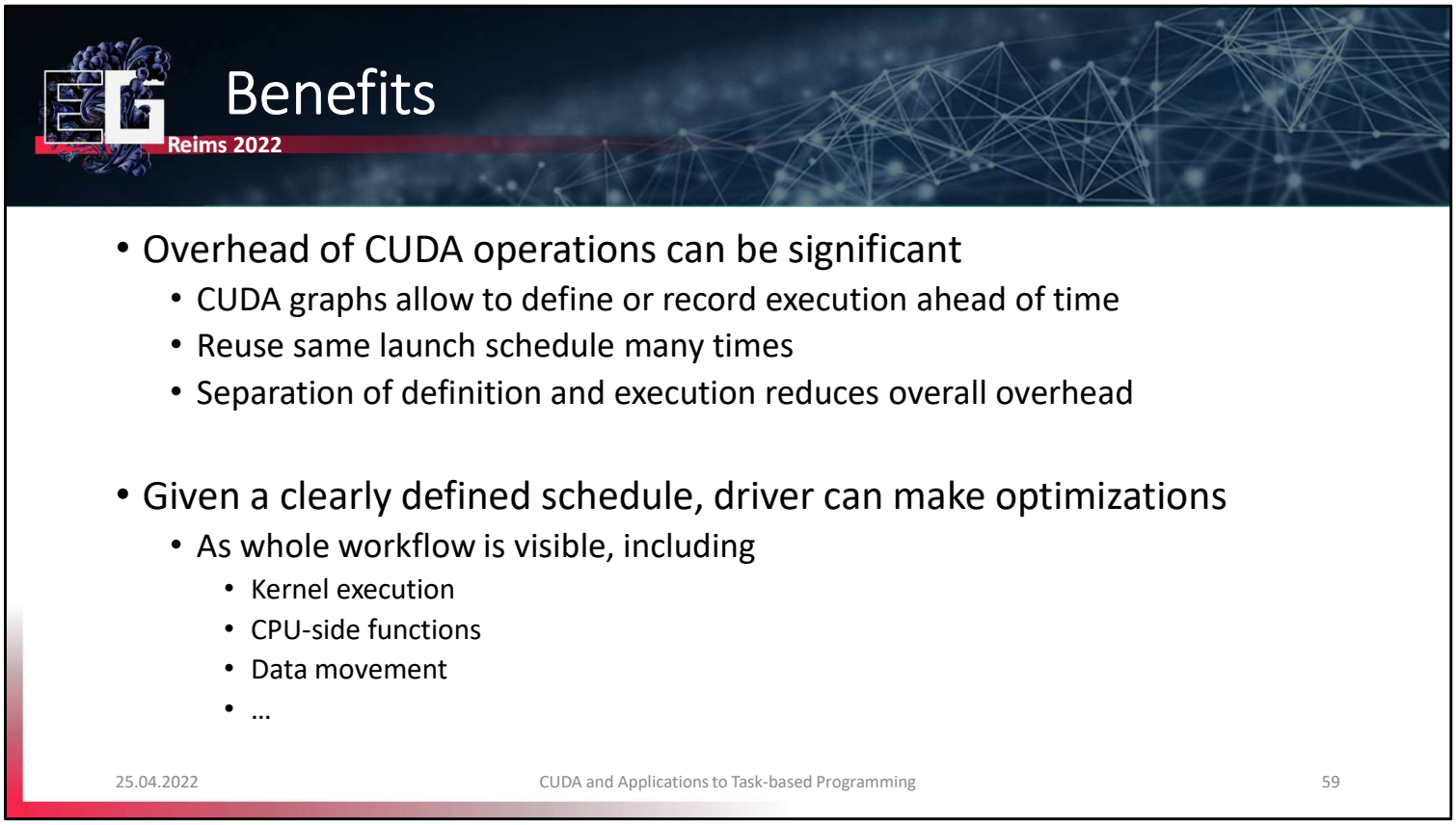
With libcu++, we can use barriers and the new cooperative memcpy_async functionality, which enables us to kick off an asynchronous copy of data from global to shared memory and, at some later point in the program, wait for that transfer to finish before progressing. The true benefit of this new functionality, which enables staging in shared memory, is significant for performance, but its implementation is a bit more involved–we won‘t address it in detail here. However, the interested participant is strongly encouraged to refer to the appendix of the CUDA Programming guide on asynchronous data copies.



In the next section, we will consider the CUDA graph API.



Many applications consist of not one, but a larger number of kernels that are in some way pipelined or processed iteratively. Usually, the nature of the computations that must occur does not change significantly, and a program performs the same steps in the same order for a number of iterations. A good example would for instance be the simulation of game physics, where in each frame, several small, incremental updates are made to achieve adequate precision. These applications can often easily be expressed in the form of a graph, where each step represents a node and edges indicate dependencies. CUDA graphs enable the definition of applications with this graph structure, in order to separate the definition of program flow and execution.

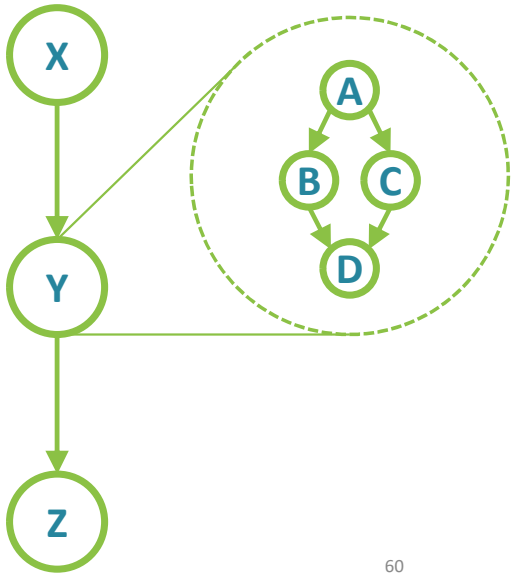


When one places a kernel into a stream, the host driver performs a sequence of operations in preparation for the execution of the kernel. These operations are what are typically called “kernel overhead”. If the driver, however, is aware of the program structure and the operations that will be repeatedly launched, it can make optimizations in preparation for this particular workload. In order to enable the driver to exploit this additional knowledge, developers can construct these graphs either from scratch or existing code.

Node Types

Reims 2022

- Kernel launch
- CPU function call
- Memory copy operation
- Memory setting
- Child graph
  - Option to modularize
  - Attach subgraphs to parent graph
- Empty Node

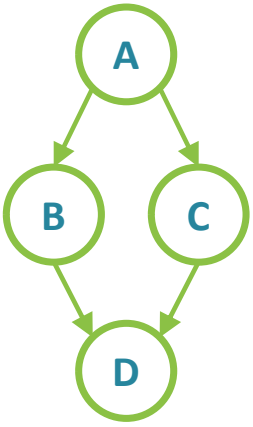


CUDA Graphs support fundamental node types that suffice to build arbitrary applications from their combinations. It is possible to create, attach and parameterize nodes at any point before the graphs are made final.

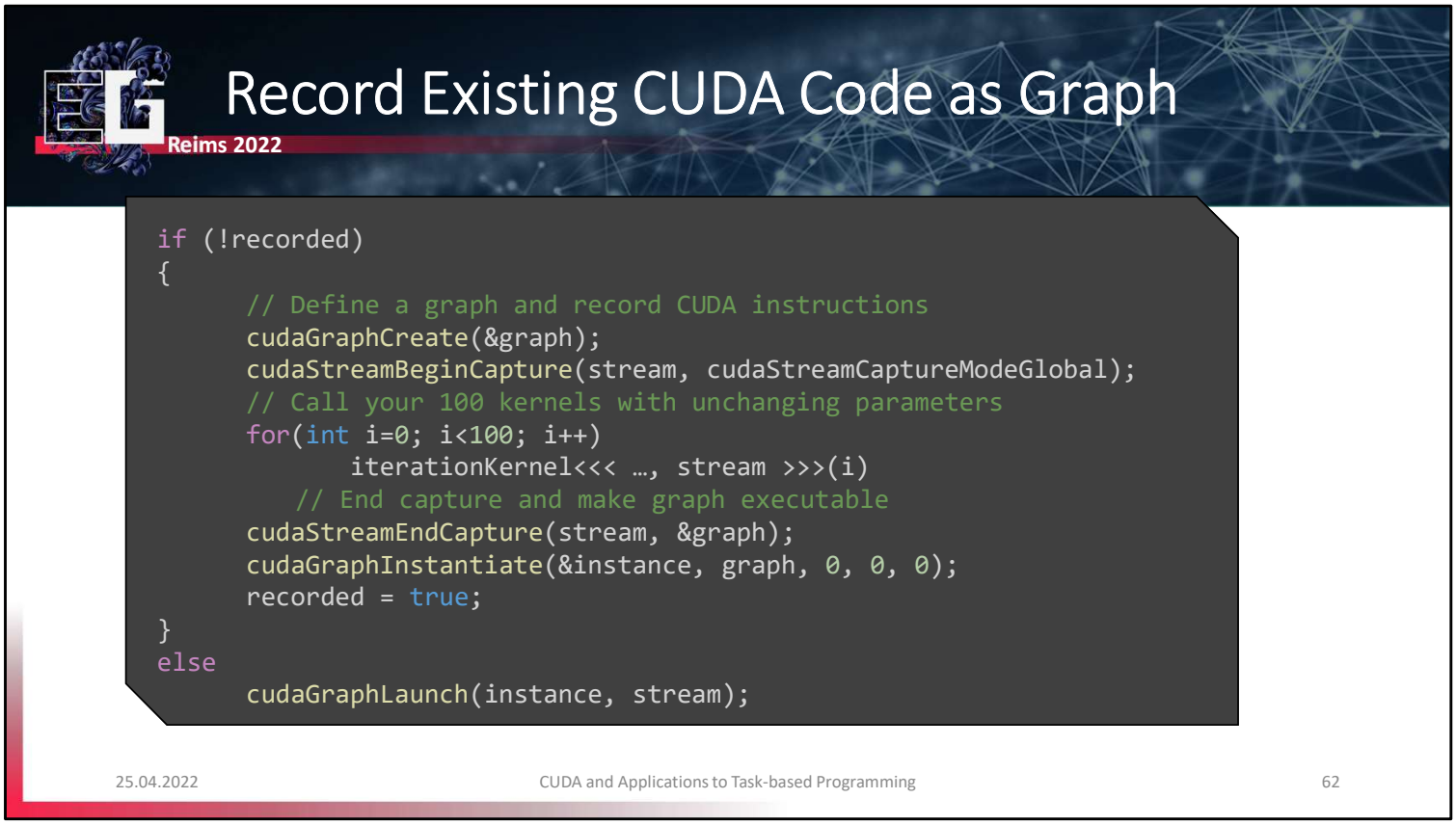
Create CUDA Graph from Scratch

Reims 2022

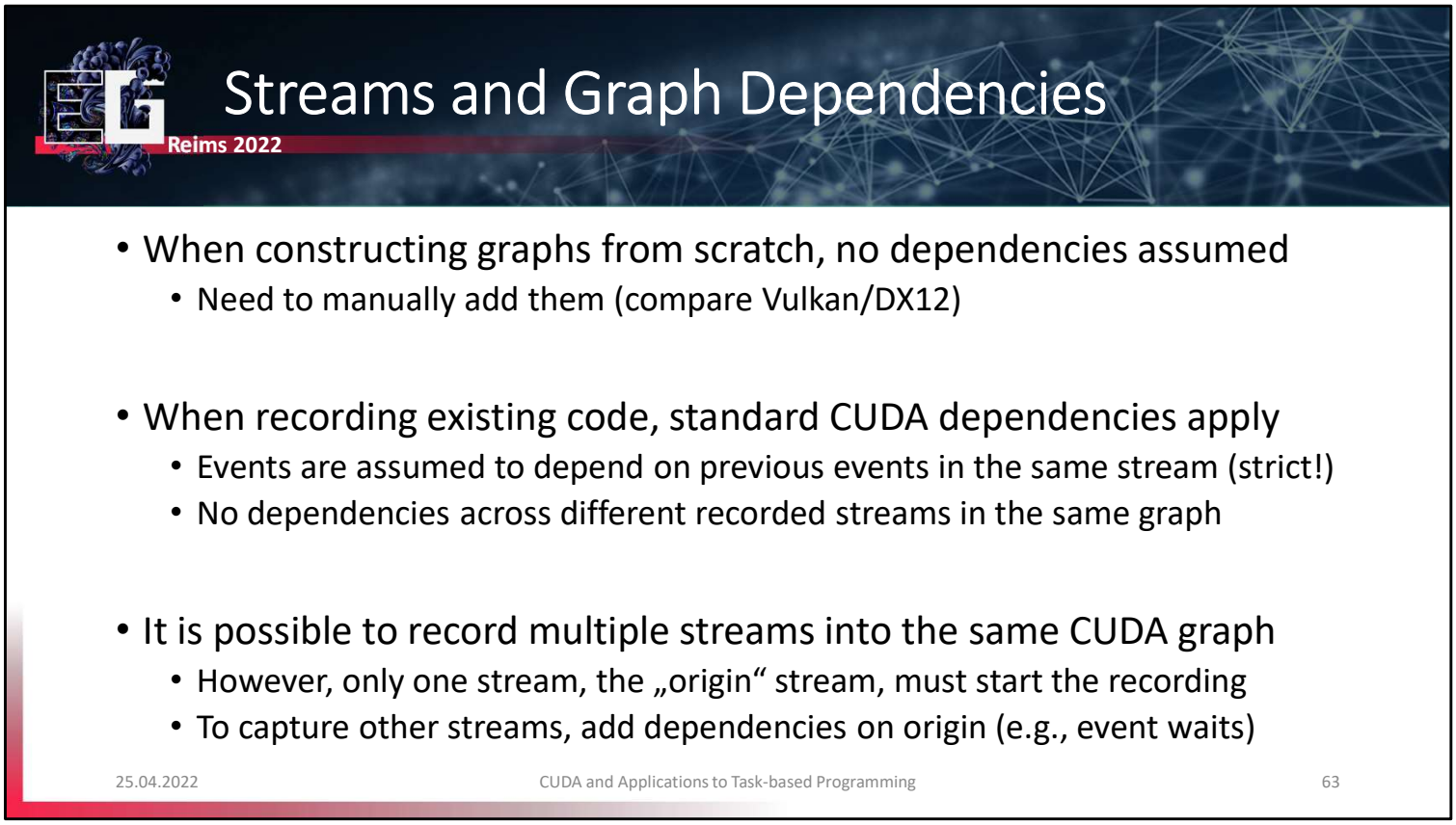
```
cudaGraph_t graph;

// Define graph for work
cudaGraphCreate(&graph);
// Define nodes, dependencies and params
cudaGraphNode_t A, B, C, D;
cudaGraphNode_t d_B[] = {A}, d_C[] = {A}, d_D[] = {B,C};
cudaKernelNodeParams paramsA = {...}, paramsB = ...

cudaGraphAddKernelNode(&A, graph, 0, 0, paramsA);
cudaGraphAddKernelNode(&B, graph, d_B, 1, paramsB);
cudaGraphAddKernelNode(&C, graph, d_C, 1, paramsC);
cudaGraphAddKernelNode(&D, graph, d_D, 2, paramsD);

// Instantiate graph and apply optimizations
cudaGraphInstantiate(&instance, graph);
```



Here we see a minimalistic example for the use of CUDA graphs. First, graphs must be created. After creation, a graph's structure, consisting of individual nodes and their dependencies, is defined. Before execution, a defined graph must be instantiated to enable CUDA to analyze it, validate it, optimize it and eventually yield the final, executable graph. Once instantiated, the executable graph can be reused as often as desired.



However, it is also possible to record code into a CUDA graph instead. This is particularly valuable for the transfer of existing codebases to the graph API. In this example, once at program startup, a collection of commands that are executed in every frame of a simulation are recorded into a graph, which is then instantiated. After the initial recording, the graph is ready for execution and can be executed directly. In the best-case scenario, an existing code segment can be wrapped with the commands for recording and instantiating in order to replicate the behavior of legacy code with the graph API.



In CUDA without graph APIs, we rely on streams in order to define the dependencies between different CUDA operations. By sorting commands into different streams, we indicate that they are not dependent on one another and can be concurrently scheduled. When using the graph API to build graphs from scratch, by default no dependencies are assumed. That is, if multiple kernel execution nodes are added to a graph without the definition of a dependency, they will execute as if they were all launched into separate streams.

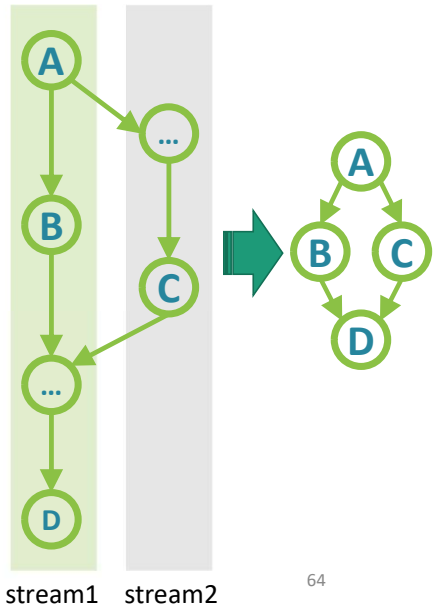
When code is recorded into a graph, the conventional dependency model is assumed. For instance, if a single stream is recorded, all commands that may have potential dependencies on one another are treated as such. If multiple streams are being recorded, the commands in different streams may run concurrently.

Example

Reims 2022

```
// Start by initating stream capture
cudaStreamBeginCapture(stream1, cudaStreamCaptureModeGlobal);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
// Create executable graph instance before launching…
```

Event required to initiate recording another stream

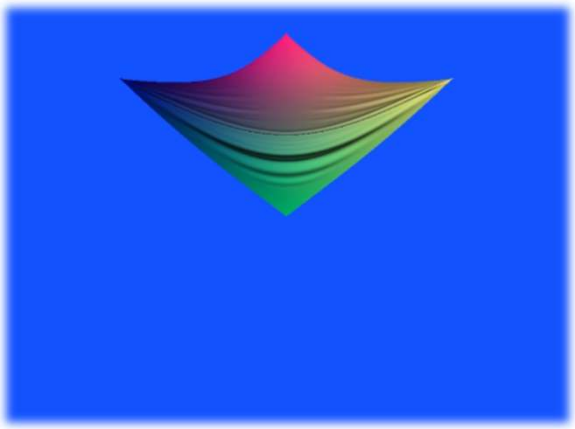


Capturing multiple streams into a graph takes a little extra care. Each captured graph must have an origin stream, and other captures streams must somehow be associated with the origin. Simply starting a capture in one stream before commands are executed in another will not suffice. In order to establish this association, one stream may for instance wait on an empty event from the origin stream. This way, the dependency of one stream on the other is made explicit and captured in the graph as well.

Use Case: Cloth Simulation

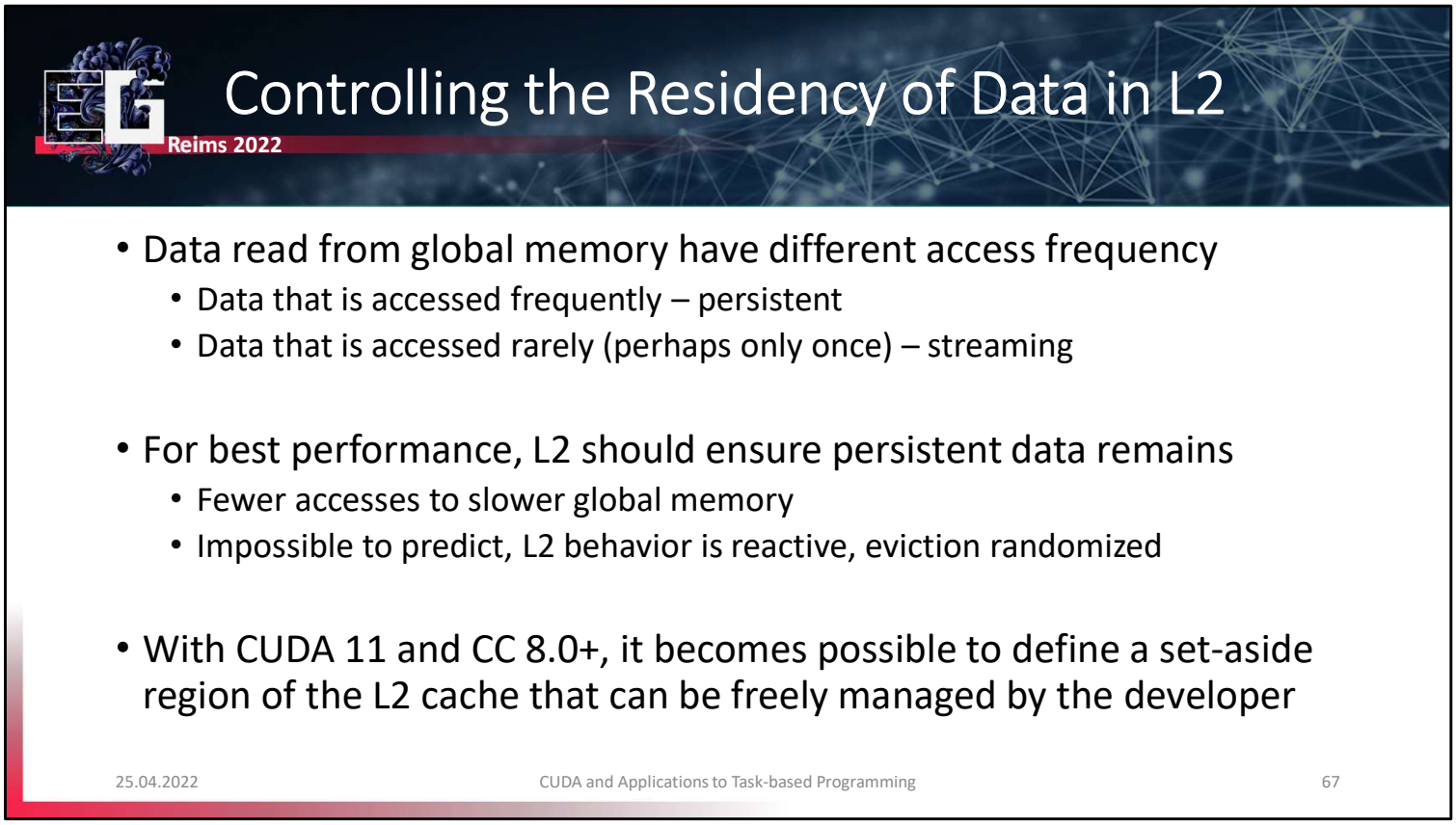
Reims 2022

- Mass-spring cloth model, Verlet integration, 30 iterations per frame…
- Used in GPU programming lecture
- 5ms per frame, initially
- 4.5ms after adding CUDA graphs
  - 5 minutes of effort
  - 10% performance benefit

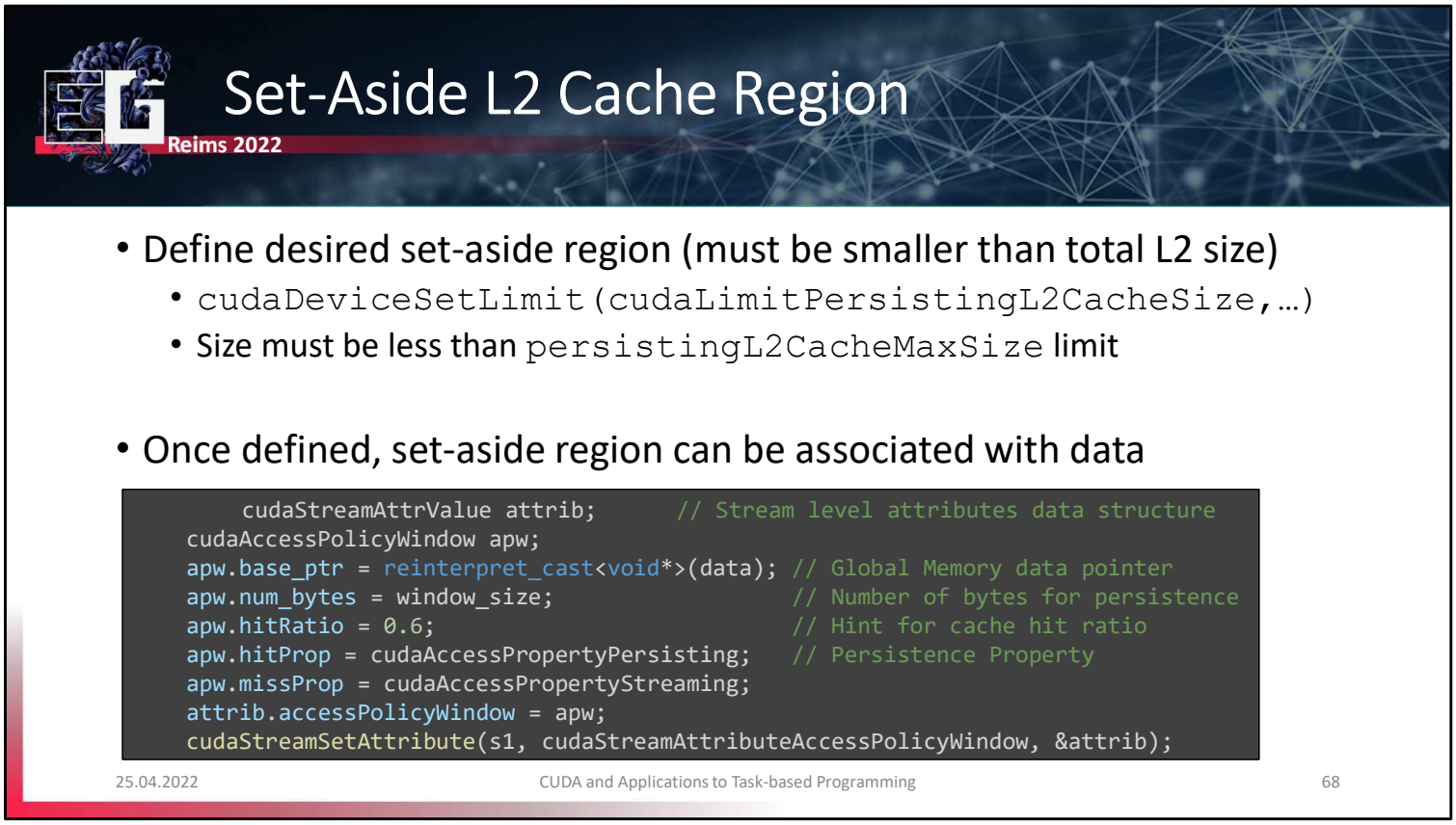


Here we show a use case from our GPU programming lecture. This example implements a simple cloth simulation, where a mass-spring model is solved with Verlet integration. For updating the positions of the individual vertices, a simple update procedure is called many times in each frame with a small time step. Hence, the pipeline is highly repetitive and the kernels extremely simple, which makes the kernel launch overhead more substantial in proportion. By capturing the update routine in a graph and replaying it in each frame, we were able to improve the performance by approximately 10%.

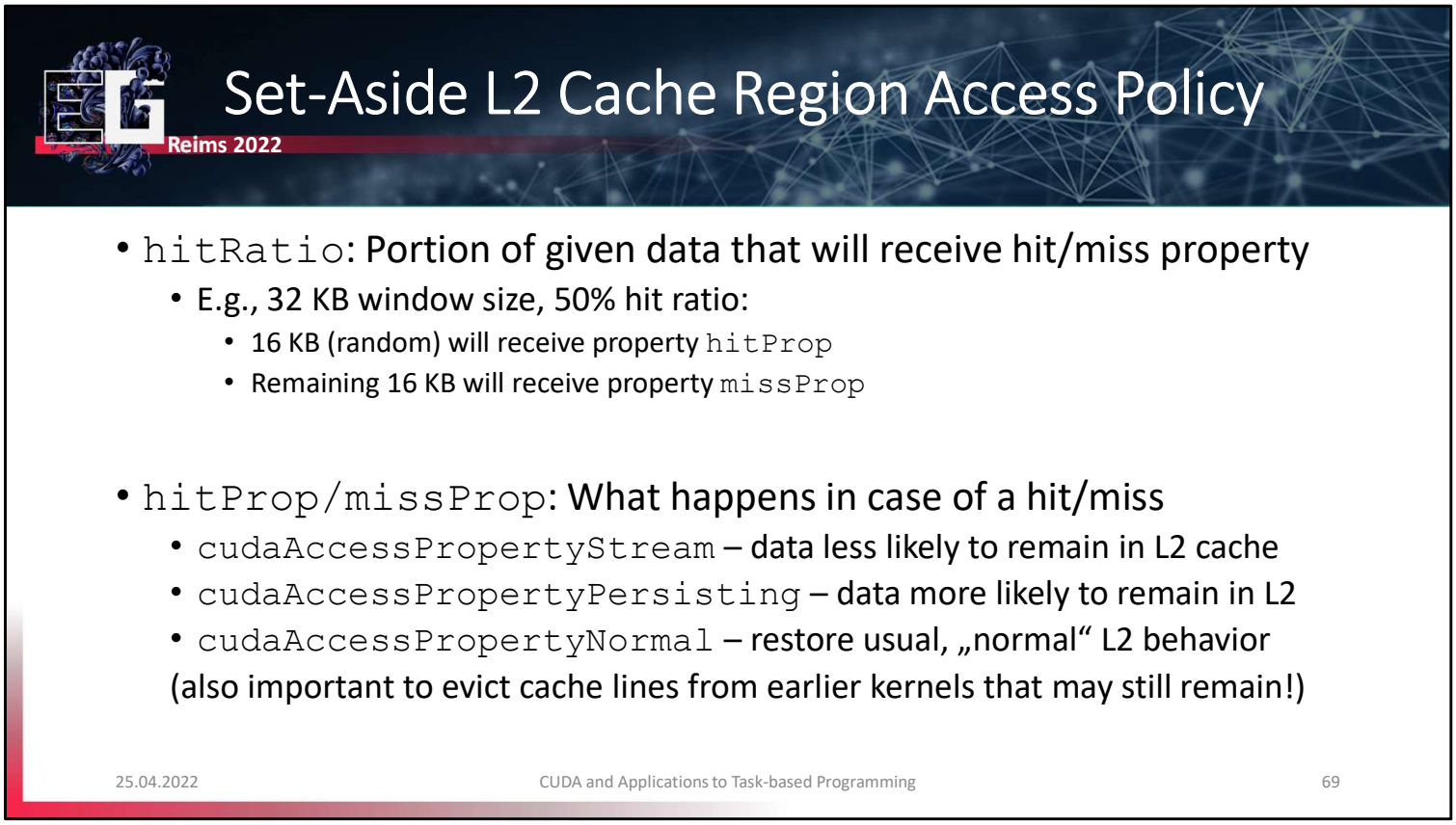
The last recently introduced feature that we want to mention in this tutorial is the set-aside L2 cache.



Not all data is made equal. Some of it used frequently in kernels, other data may be more transient and not used more than once. In the context of the residency in L2 cache, we can distinguish these as persistent data and streaming data. To achieve maximum performance, the L2 cache management should encourage that persistent data remains while streaming data is quickly evicted. However, this behavior is purely reactive, since the cache cannot predict program flow and frequently used information. With CUDA 11 and the Ampere architecture, it is now possible to define set-aside regions of the L2 cache that will be managed according to the definitions by the developer.

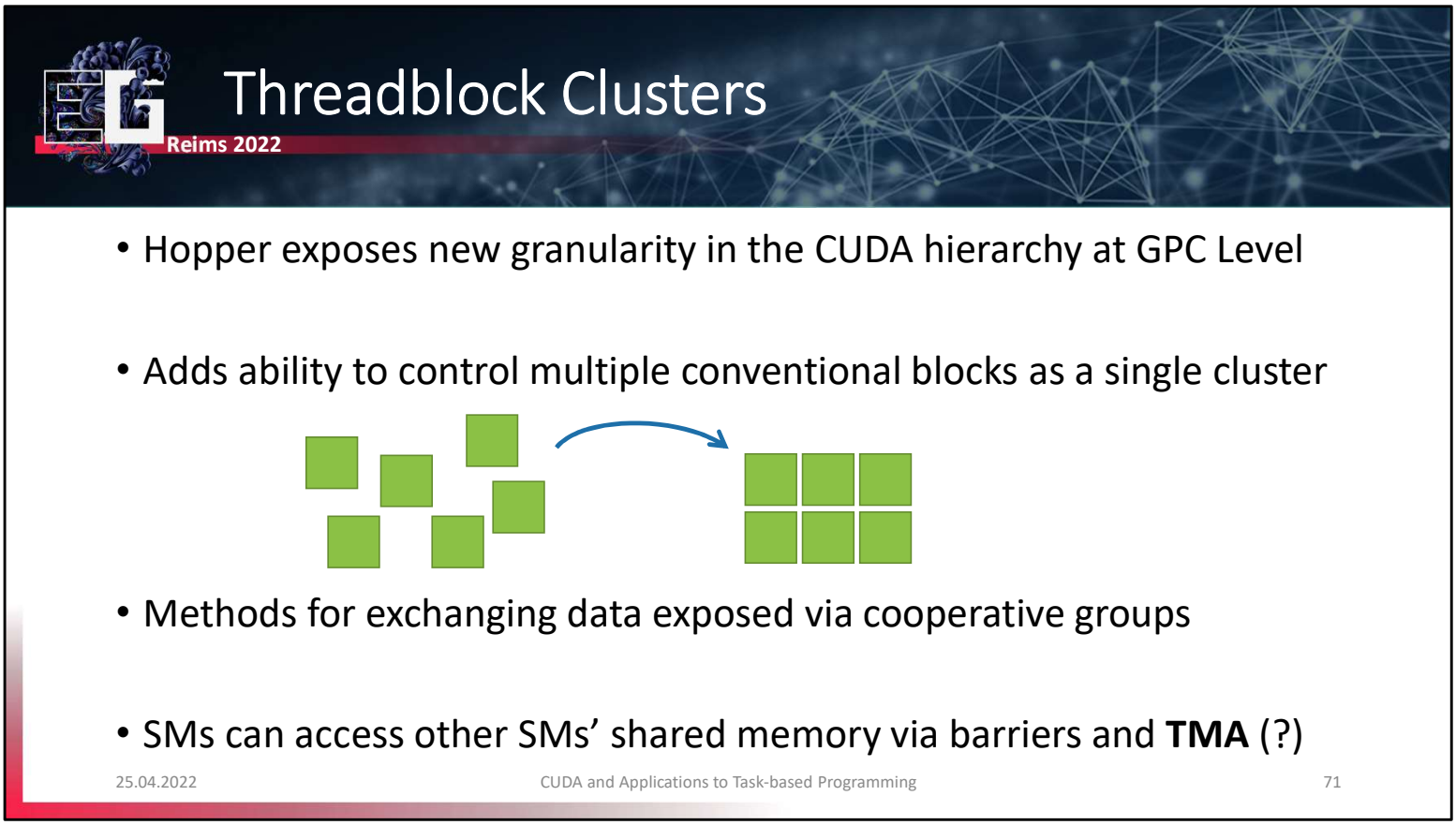


The amount of L2 cache that can be used in this way is defined by a property that can be queried from the active GPU. A memory range can then be associated with a portion of the set-aside L2 cache and configured with various properties that define how it will be maintained.



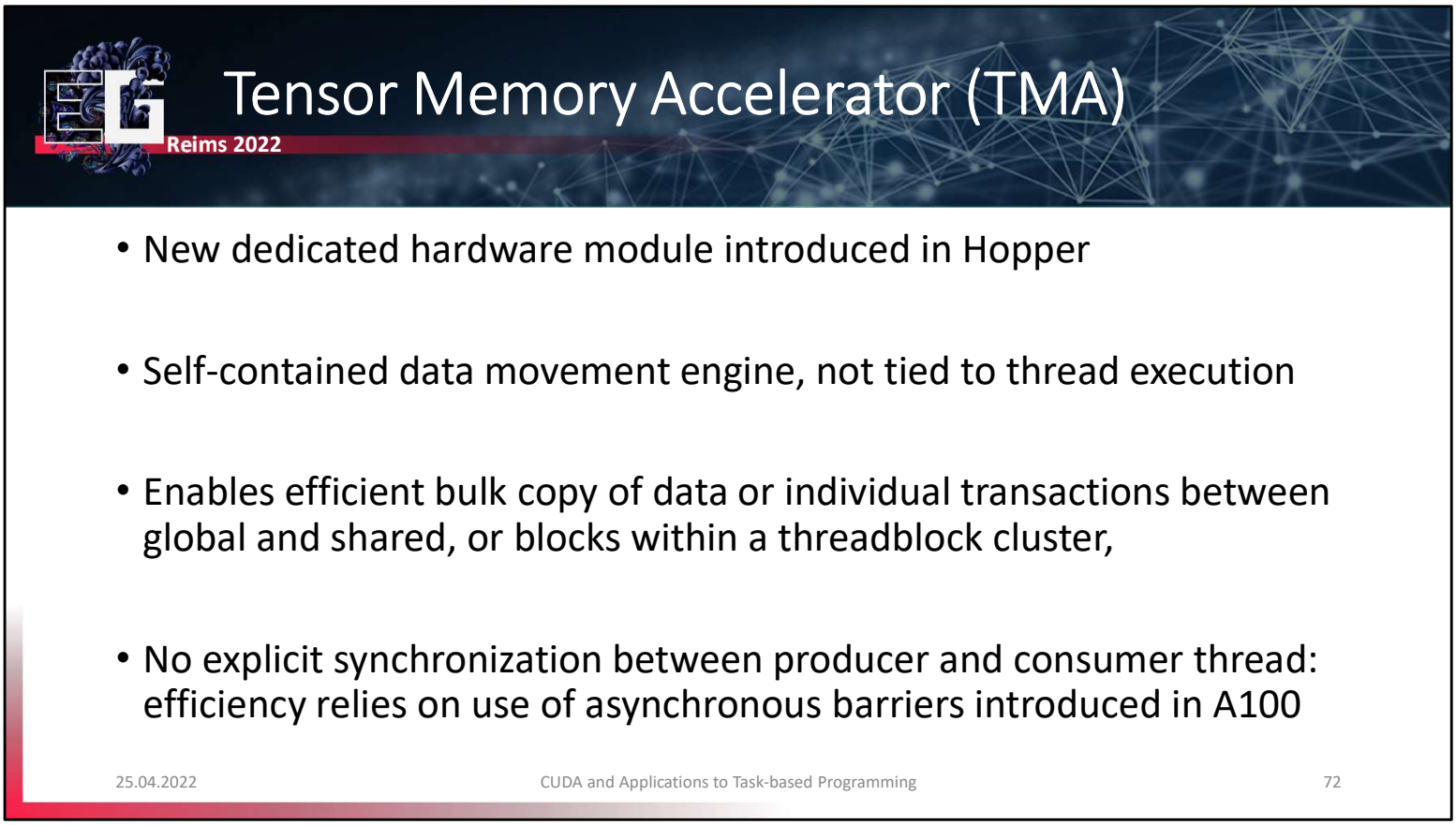
The hit ratio of a memory portion defines how much of it (chosen randomly) should comply with the defined hit property. The remainder will comply with the miss property. For instance, with a hit ratio of 50%, half of the memory associated will be treated with the hit property and the other half with miss. The properties can be set to encourage behavior for persistent data or streaming data, or the associated memory can be cleared of its persistent or streaming property to return to „normal“ caching behavior.

Finally, we want to provide with the most recent information that we have on the upcoming GPU generations at the time of preparing this tutorial: the professional data center-oriented Hopper, and the consumer-targeted Lovelace. Especially with regard to Lovelace, more detailed information will probably emerge by the time this tutorial is held, so we may update the tutorial notes, the newest version of which will be available on the tutorial website.



Hopper introduces the concept of threadblock clusters. Threadblock clusters take the key idea of transparent group creation directive of cooperative groups to the next level. Above the conventional thread block, developers can now opt to combine multiple blocks into a collaborative group at the Graphics Processing Cluster (GPC) level. Via methods exposed by cooperative groups, these new, larger collections of threads can communicate and manipulate data as one.

Traditionally, in the hardware rasterization pipeline, the GPC is used as a self-contained rasterization unit that is assigned to exclusive portions (tiles) of the framebuffer. The comparably low number of GPCs, compared to SMs, enables the use of simple static policies for assigning framebuffer tiles to GPCs, without causing severe workload imbalance. Up until now, if we attempted to replicate this behavior in a software rasterizer, we would be limited by the size of thread blocks/SMs and thus, in order to balance workload, needed more sophisticated tile assignment policies. Hence, this is good news for those out there trying to write efficient, sort-middle hierarchical software rasterizers on NVIDIA hardware. However, it raises the question of how the general-purpose exchange of data is enabled in Hopper: this is due to a new dedicated hardware module called the TMA.



In explicit terms, the TMA is the Tensor Memory Accelerator. It presents a new, dedicated hardware unit introduced with Hopper, which targets the efficient exchange of data. Its design is in part motivated by the high performance and demand for fast data transactions in applications that employ tensor cores. As such, it is capable of transferring bulks of 1/2/3-dimensional data blocks and above. Furthermore, it enables a number of new workflows for exchanging data between threads.

The TMA may be used for asynchronously copying data between shared and global memory (both ways) or between shared memory in different SMs inside a threadblock cluster. The TMA is its own self-contained unit. This means that writes initiated via the TMA can be “fire-and-forget”: the thread that initiates a transfer is not required to synchronize or perform a handshake with the potential receiver (e.g., by implementing a conventional release-acquire pattern via fences). The functionality to exploit the TMA in this way is provided by an extended set of asynchronous barrier methods.

Asynchronous Transaction Barriers

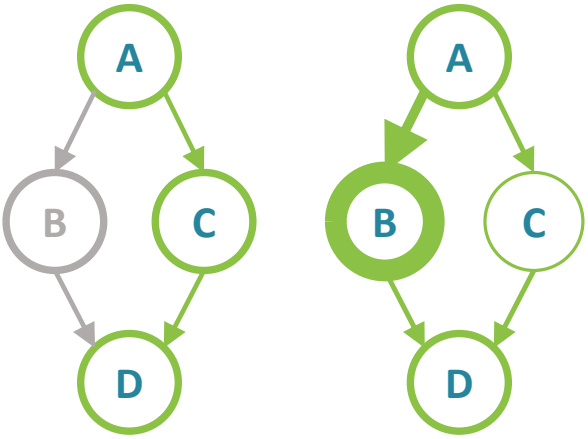
- libcu++ extends its `cuda::` variety of barriers, now can signal
  - Thread arrival
  - Initiated data transaction
  - Wait on threads or **data** to arrive
- Pure wait on `memcpy_async` enables "self-synchronizing writes"
  - No handshake to inform recipient that data has been released
- Extensions intertwine with the exploitation of new TMA features
  - `atomic_add_async, atomic_min_async…` (no CAS though)

Hopper extends the asynchronous barrier interface introduced with the A100, enabling the asynchronous initiation and waiting for data transfers rather than explicit threads. This leads to scenarios that NVIDIA refers to as "self-synchronizing writes", i.e., they avoid the necessity to explicitly ensure data consistency.

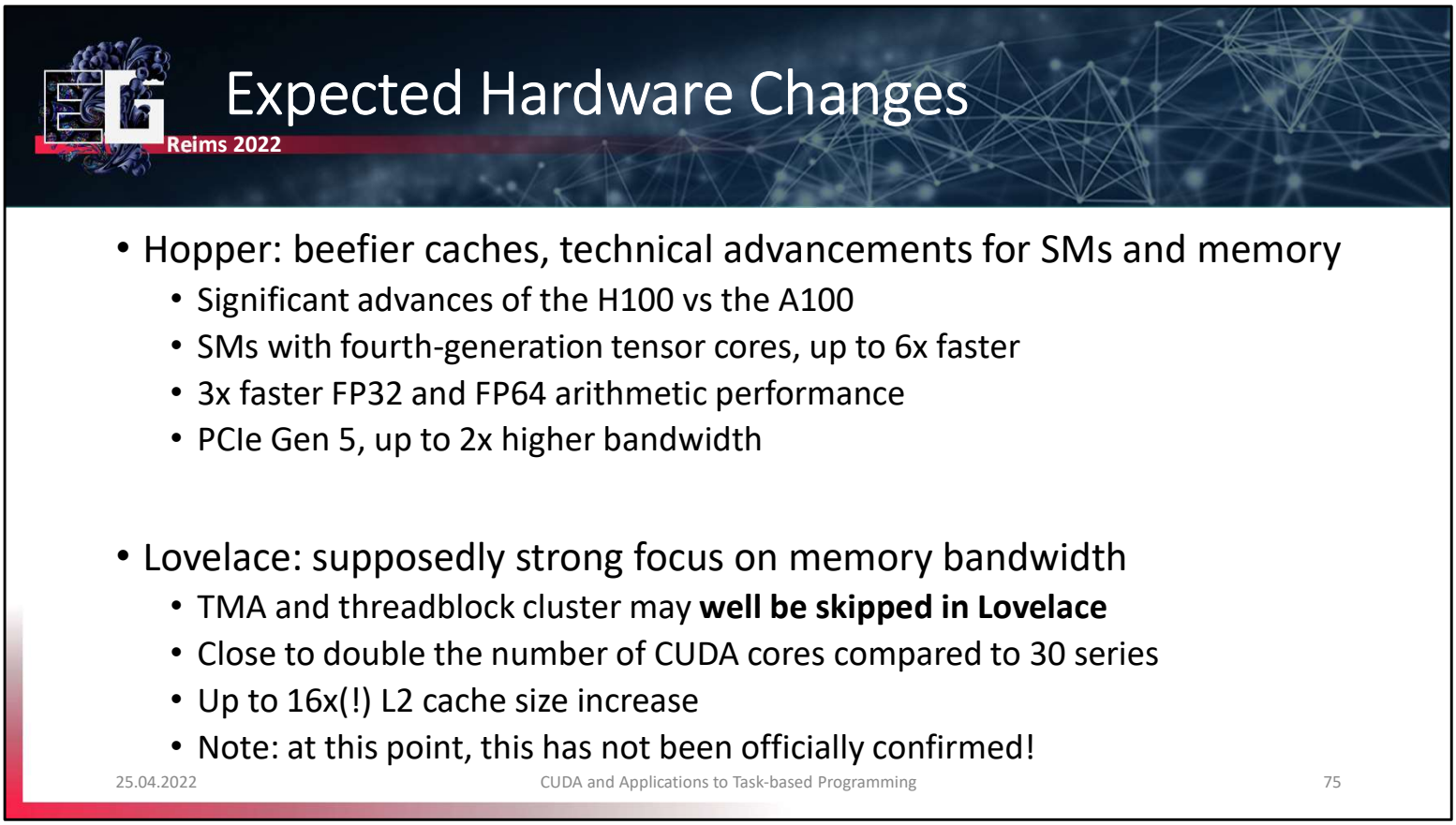
Beyond the standard copying method `memcpy_async`, the new barriers expose functionality to use the TMA with atomic functions. Hence, atomic counters and variables can be trivially shared by multiple SMs with Hopper via the TMA.

More Dynamic CUDA Graphs

- CUDA Graph API is continuously adapted for better flexibility
- Recent features facilitate prioritization
- Individual subgraphs or nodes can now be temporarily disabled
- Subgraphs or nodes can be assigned a priority for scheduling workloads



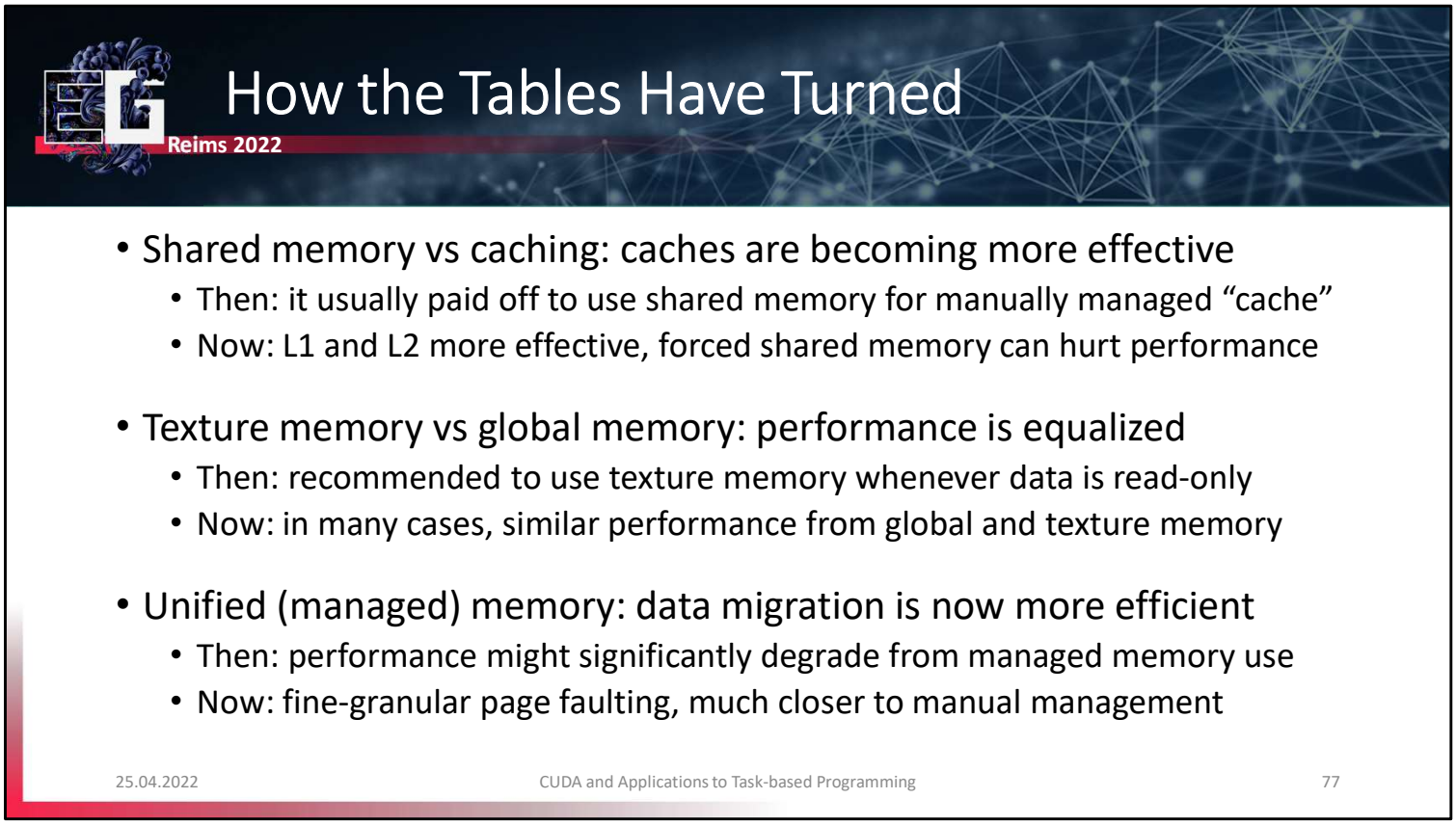
The CUDA graph API has continuously been extended with new convenience features for improved flexibility. Most recently, two new changes have been introduced that enable selective prioritization of nodes: subgraphs and nodes may now be disabled/enabled and hence temporarily ignored in the execution of completed graphs. Furthermore, subgraphs may now be assigned execution priorities. These will be considered for the scheduling policies of instantiated graphs and enable adaptive behavior, especially for complex program flows and graphs.



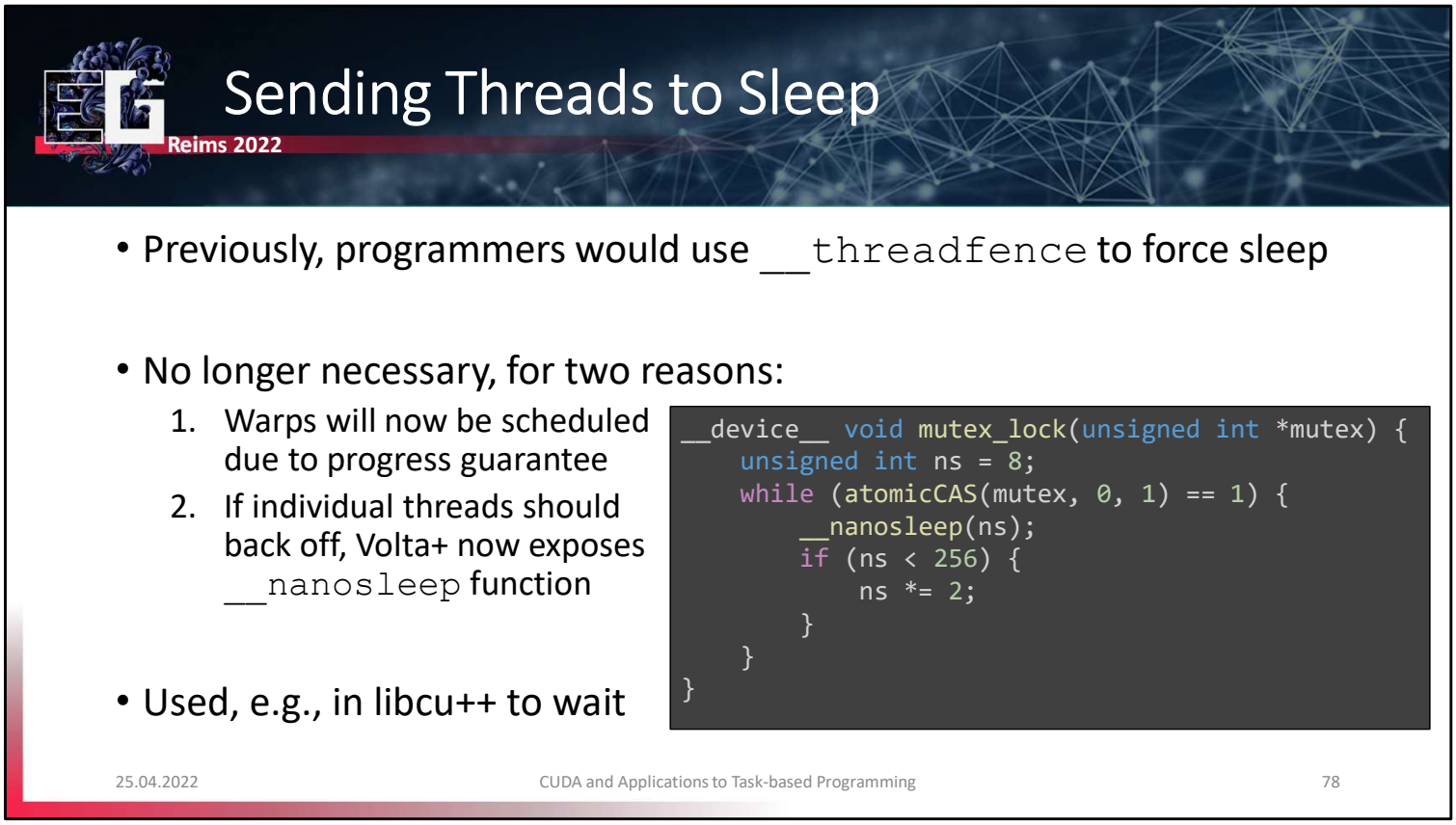
Lastly, let's talk numbers: for Hopper, we can draw from the most recent specifications of the H100, and compare them to the previous generation (A100). SMs will be fitted with advanced components, including fourth-generation tensor cores, promising a performance increase of up to 6x. Conventional floating point operation performance is also projected to increase by approx. 3x, for both 32- and 64-bit precision. Hopper will provide support for generation 5 PCIe, enabling up to 2x the bandwidth of the A100.

We do not have confirmed specifications for the consumer-grade Lovelace models at this point. Individual parties have suggested that it may feature some of the cutting-edge Hopper features, such as the TMA. However, at this point, it is just as likely that Lovelace will be more similar to Ampere, albeit with significantly increased memory capabilities: inofficial statements suggest an increase in L2 cache size by up to 16x. This could, in part, be due to addressing the demands for bandwidth of close to double the size of CUDA cores that Lovelace is projected to feature in comparison to the 30 series. If confirmed, this change would affect the way compute-heavy applications are written on NVIDIA hardware, triggering a much stronger focus on exploiting the large L2 cache, especially for persistent-threads solutions.

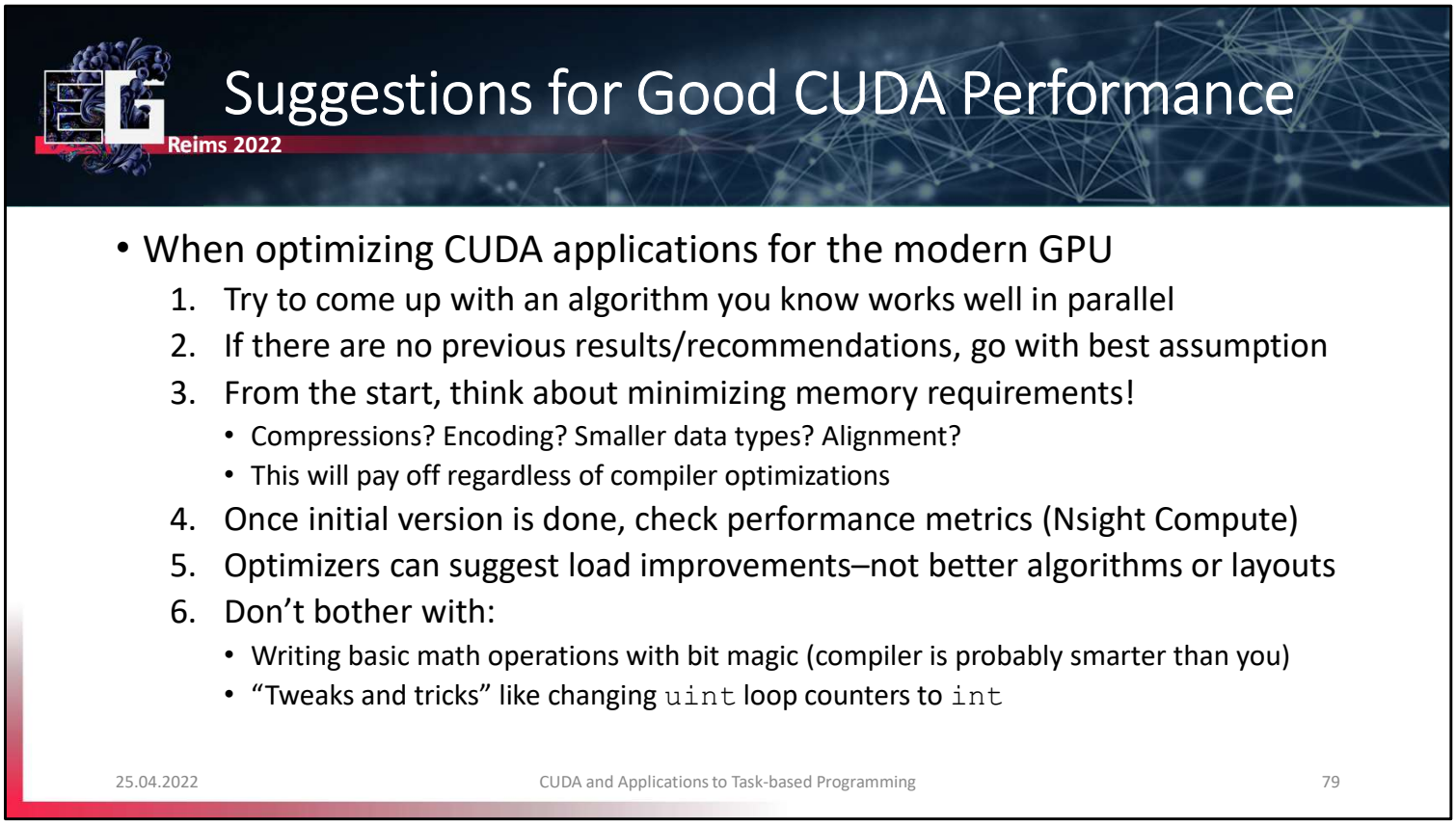
Finally, we would like point out general cavetas, trends, things that changed from how they used to be and our personal suggestions for working with CUDA.



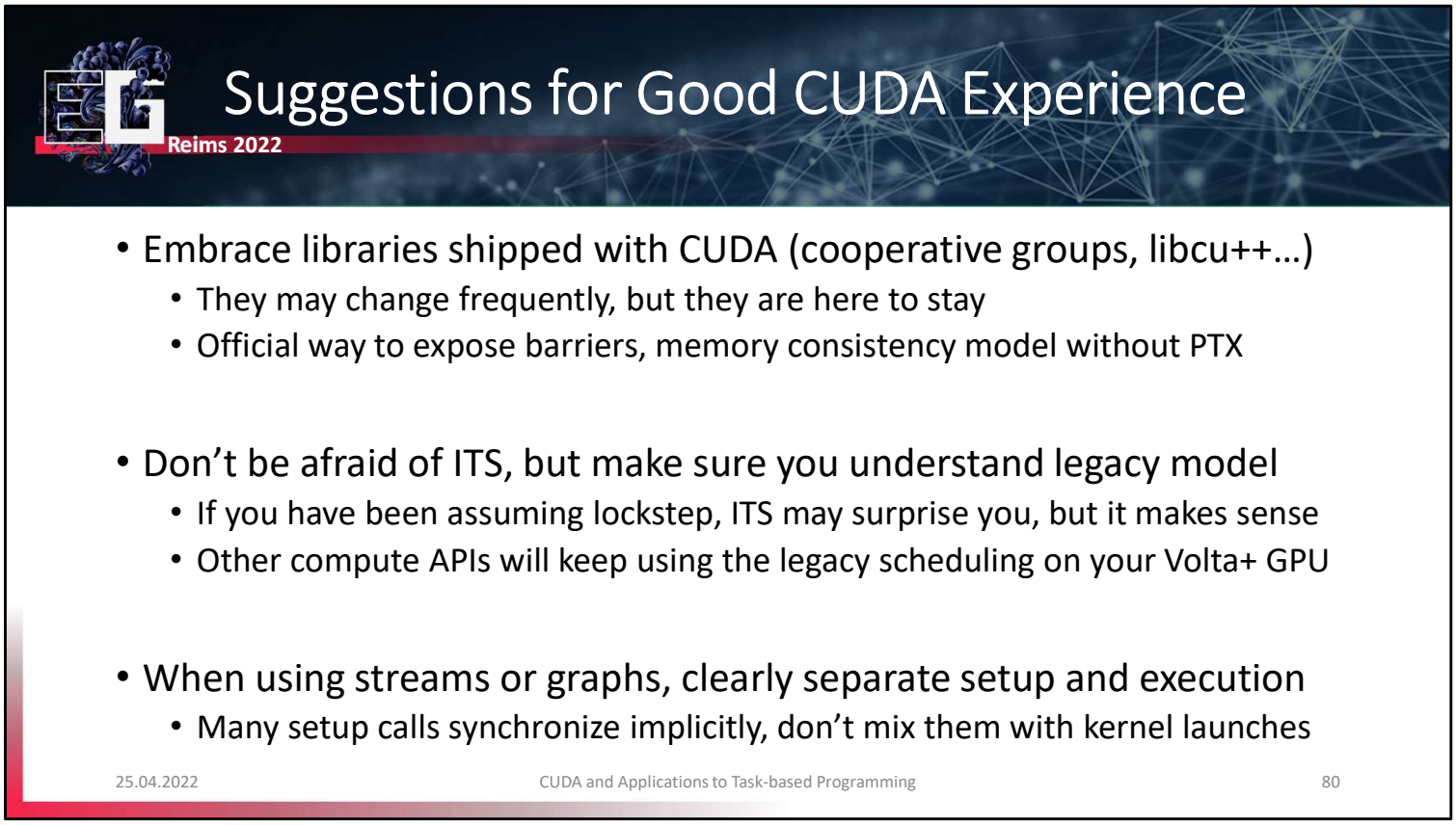
First, there a few things that used to be go-to solutions for increased performance, which are no longer universally true. For instance, caches are catching up with the benefits of shared memory. The L2 and L1 cache have adapted caching policies, to the point where it is no longer always smarter to prefer manual handling of shared memory over an automatically managed L1 cache.
Second, texture memory was long promoted as an immediate boost to performance for read-only data. This property too seems to be less evident than it used to be. Performance of global and texture memory is, for a wide range of patterns, mostly similar, except for random access patterns within a small, spatially confined window. Of course, the additional functionality of texture memory (filtering, sampling) remains.
Lastly, as we already pointed out, unified (managed memory) is no longer the performance hog it used to be. Thanks to fine-granular page faulting mechanisms and better migration policies, it has become a viable alternative in many applications.



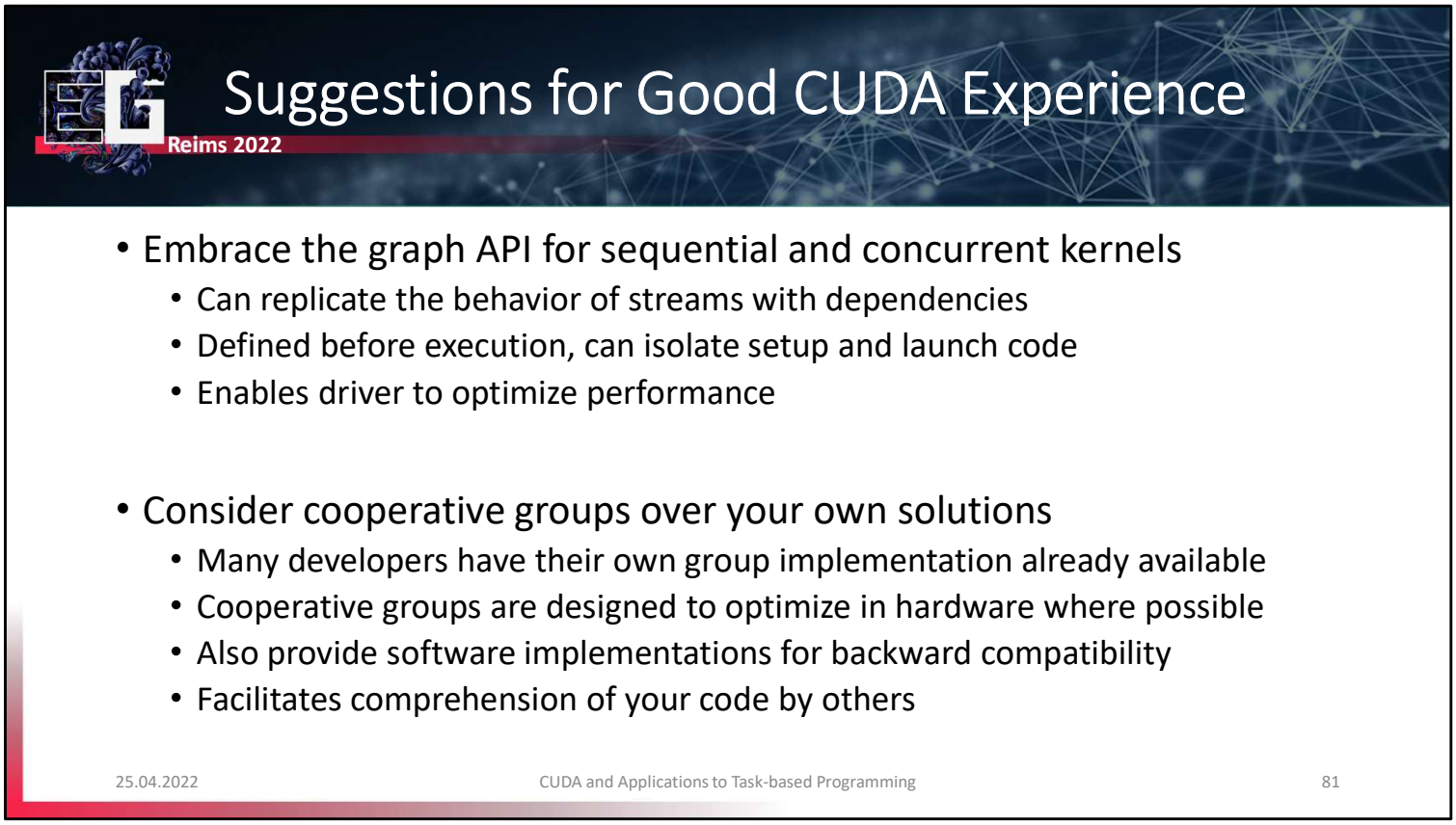
Developers used to abuse the __threadfence operation to force threads to back off and release certain resources. It was also a common requirement for blocking algorithms where threads depend on the progress of other threads. A thread fence would cause the calling warp to yield and let other warps progress before being scheduled again. These hacks are no longer necessary, since ITS guarantees progress for all resident threads. In case where it is still desired that threads back off and releaes resources, like locks, Volta introduced the __nanosleep function for that very purpose.



Given the recent developments and expected trends, we offer our personal recommendation of the above steps to be followed in this order for developing algorithms with modern CUDA.

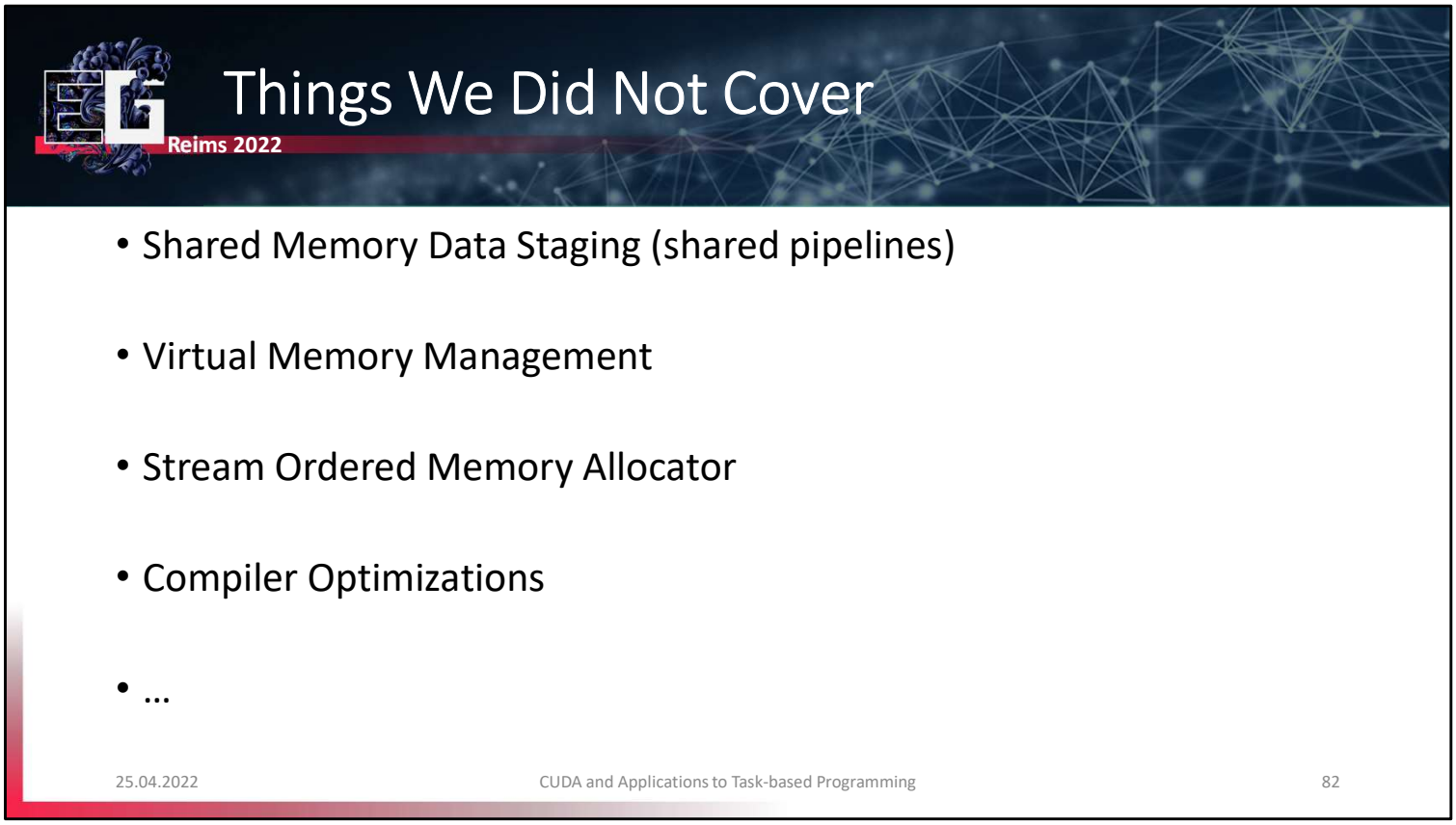


Finally, we want to highlight key ideas that we believe will make development more stable, secure and efficient moving forward. One would be the adoption of the libraries shipped with CUDA, specifically cooperative groups and the standard library, particularly if one intends to write CUDA code in C++ rather than PTX. Second, we would like to encourage developers to embrace the ITS. While it is a significant change and breaks many of the previously used optimization patterns, Volta, in general, was a great step towards bringing the CPU and GPU closer together and enabling more portable and stable code. Another sign of this development is the effort to introduce the new memory coherency model, which makes special solutions, like combining the volatile decorator with __threadfence no longer necessary. The GPU takes care to ensure the new coherency model, and its behavior has changed accordingly, making these special cases largely unnecessary.



We also recommend embracing the graph API and considering its use over the conventional solution with streams. Graphs that are directly designed from scratch have clear and easily understood dependencies that can be extrapolated from a few lines that define a CUDA graph. But the main benefit of creating graphs is the performance gain, which can be obtained regardless of whether graphs are built from scratch or capture from code.

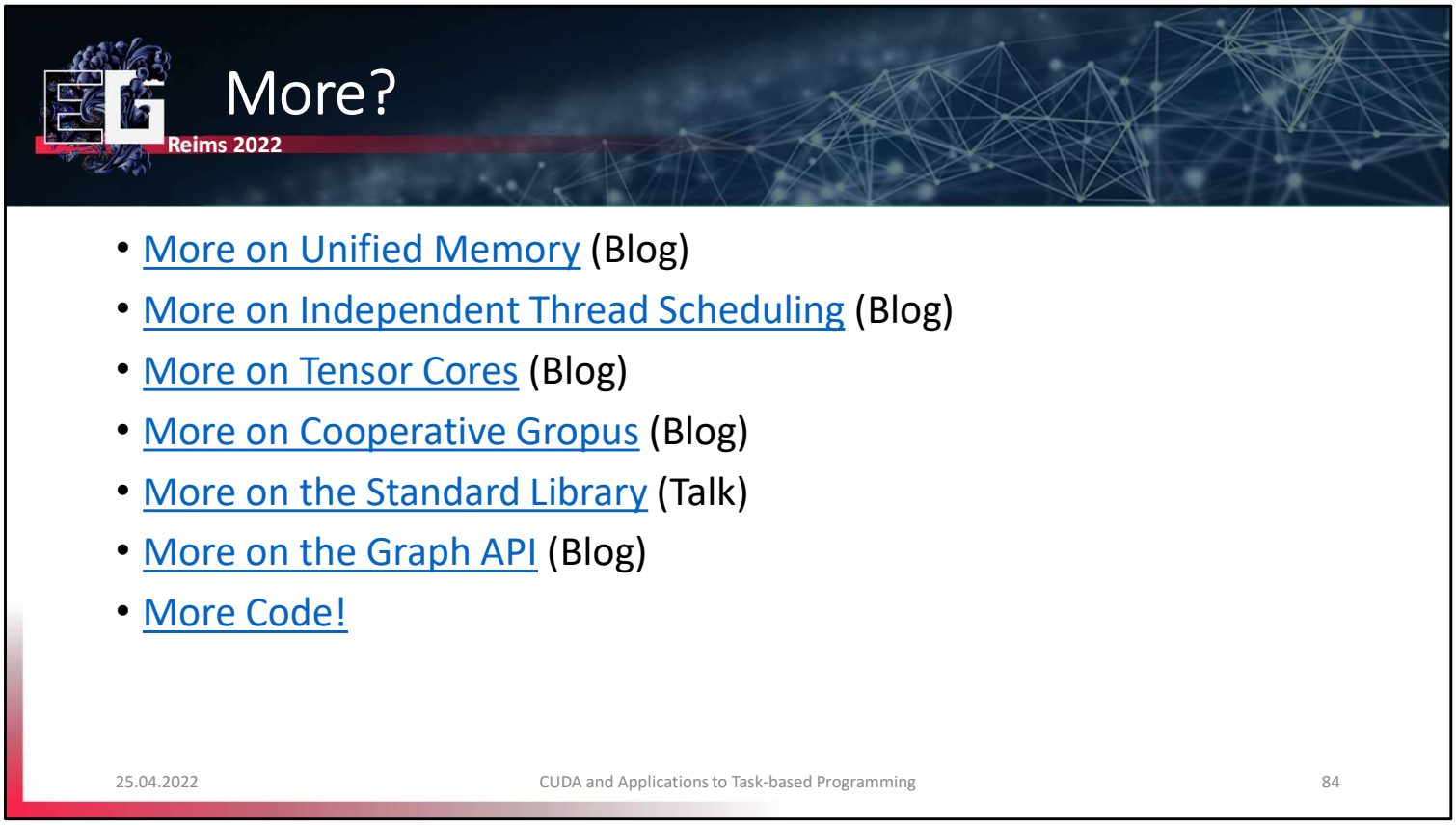
Many CUDA developers out there will have noticed that they themselves have something similar to the cooperative groups implementation. We recommend that it should be attempted to switch to cooperative groups instead or integrate them into custom solutions to benefit from the clean design and the architecture-agnostic patterns they provide.



Lastly, there are few important things that we did not manage to treat in this tutorial (and perhaps a few more that we didn‘t think of), which are nonetheless exciting and worthy of you looking into them if you are aiming to advance your CUDA expertise. Examples and detailed explanations for these can be found in the list of recommended reading material that we provided in the first part of the tutorial. We hope that during the course of this tutorial, you either confirmed or discovered that CUDA has a vast amount of great features to offer and plan to pursue it on your own from here on out.

Where to Go From Here?

- Work through the recommended reading list from Part 1
- CUDA Samples
  - C/C++ projects, provided with CUDA Toolkit
  - Great range of different and advanced applications
- Books?
  - CUDA changes quickly, advanced books don't stay relevant for too long…
  - Good choice: *David B. Kirk und Wen-mei W. Hwu*, **Programming Massively Parallel Processors: A Hands-on Approach** (3rd edition, 2016)



- More on Unified Memory (Blog)
- More on Independent Thread Scheduling (Blog)
- More on Tensor Cores (Blog)
- More on Cooperative Gropus (Blog)
- More on the Standard Library (Talk)
- More on the Graph API (Blog)
- More Code!

References

Reims 2022

[1] Wolfgang Tatzgern, Benedikt Mayr, Bernhard Kerbl, and Markus Steinberger. 2020. Stochastic Substitute Trees for Real-Time Global Illumination. In *Symposium on Interactive 3D Graphics and Games* (*I3D '20*). Association for Computing Machinery, New York, NY, USA, Article 2, 1–9. DOI:https://doi.org/10.1145/3384382.3384521

[2] Michael Kenzel, Bernhard Kerbl, Wolfgang Tatzgern, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. 2018. On-the-fly Vertex Reuse for Massively-Parallel Software Geometry Processing. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2, Article 28 (August 2018), 17 pages. DOI:https://doi.org/10.1145/3233303