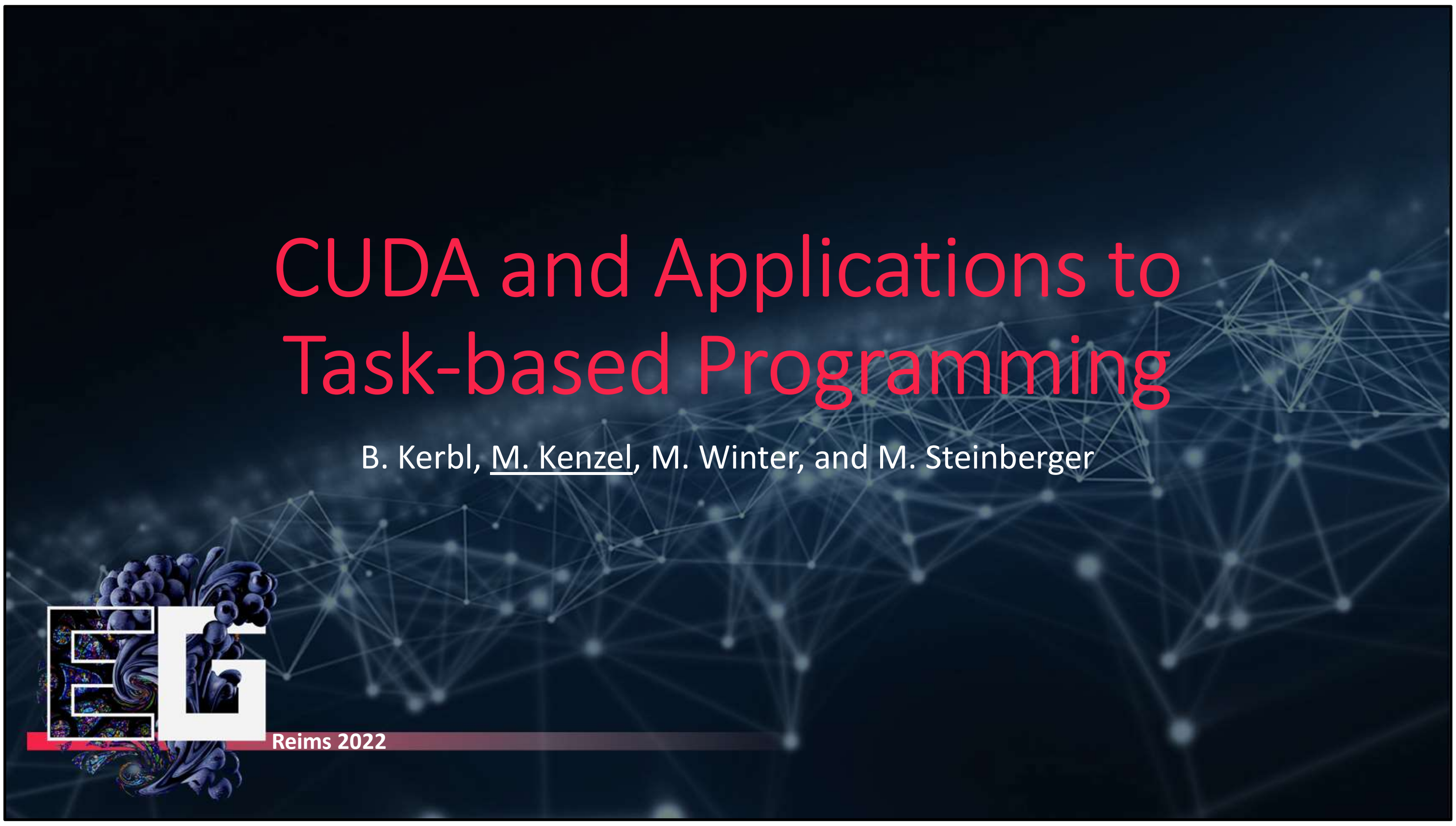
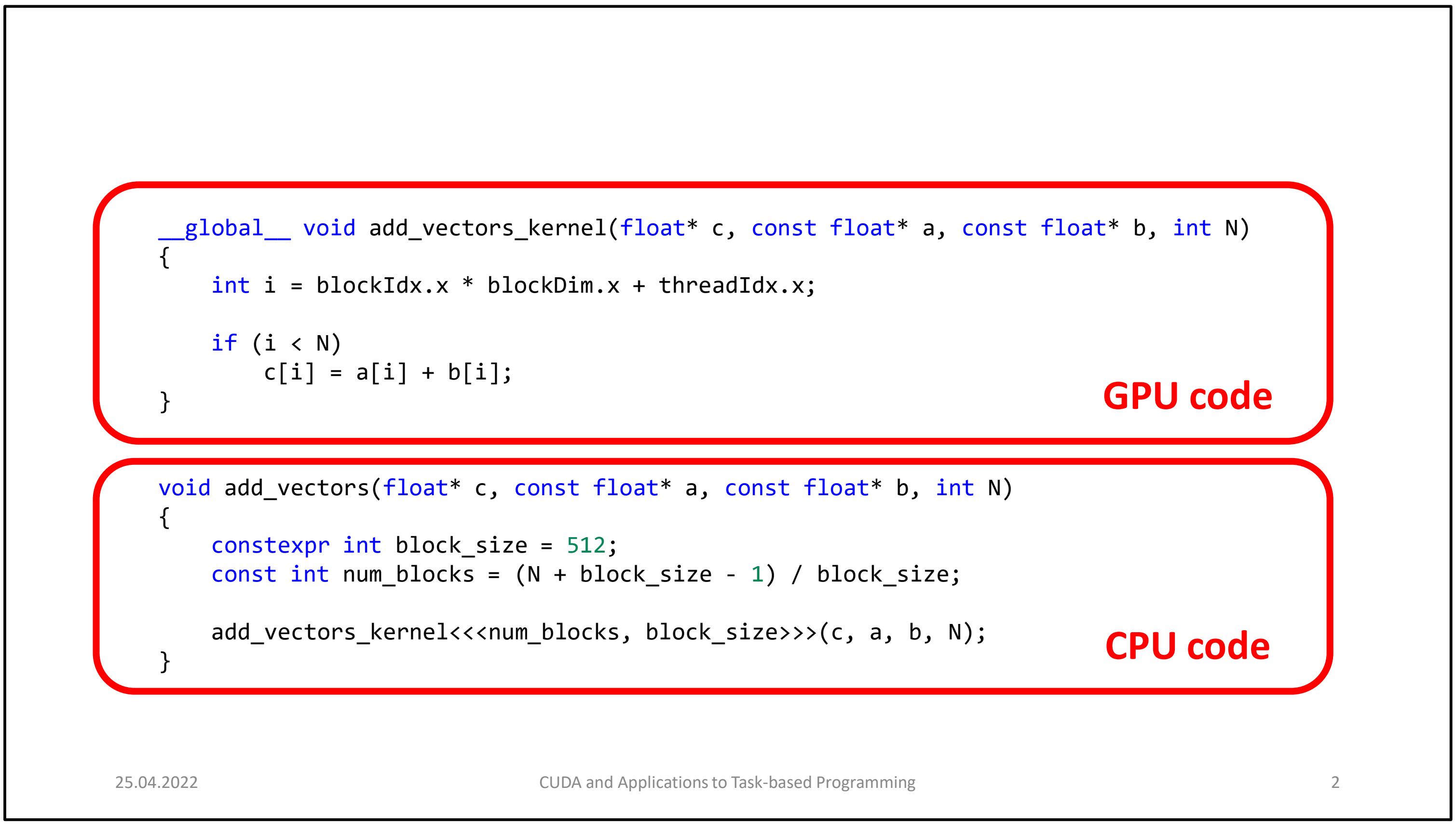


CUDA and Applications to Task-based Programming

B. Kerbl, M. Kenzel, M. Winter, and M. Steinberger

Reims 2022

In this second part of the tutorial, having explored the basic CUDA programming model in part one, we will now take a closer look at how the programming model maps to the underlying hardware architecture and discuss various low-level aspects that are crucial for performance.

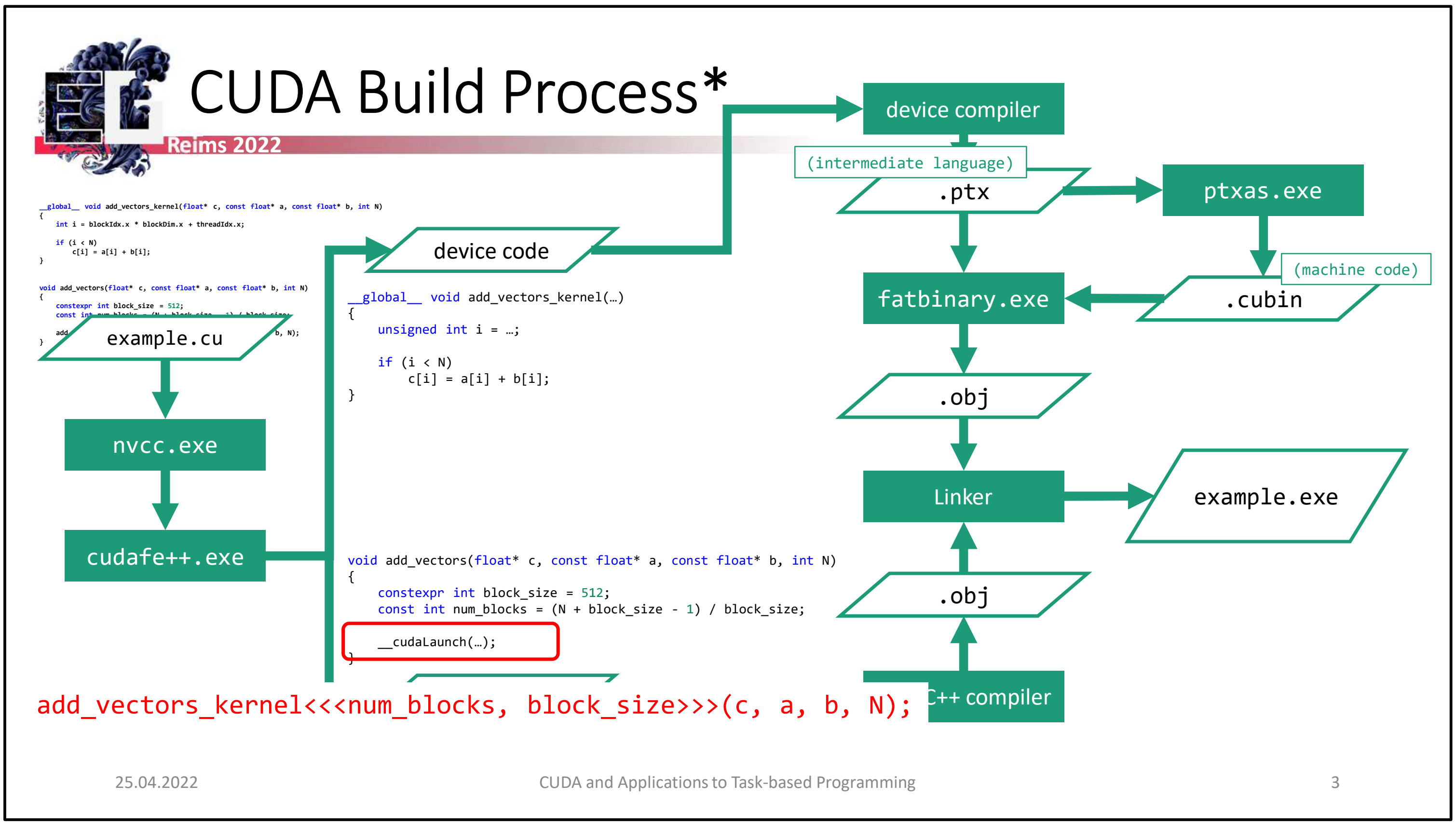


```
__global__ void add_vectors_kernel(float* c, const float* a, const float* b, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N)
        c[i] = a[i] + b[i];
}
```

```
void add_vectors(float* c, const float* a, const float* b, int N)
{
    constexpr int block_size = 512;
    const int num_blocks = (N + block_size - 1) / block_size;

    add_vectors_kernel<<<num_blocks, block_size>>>(c, a, b, N);
}
```

The first step in mapping CUDA to the underlying hardware is translating CUDA C++ code into machine code. So let's start by having a look at the process via which a CUDA program is compiled into an executable that runs code on the GPU. A good understanding of this process can be very helpful when it comes to diagnosing build problems or just deciding how to best structure a codebase that uses CUDA.

A typical CUDA program consists of two kinds of code: host code that's meant to run on the CPU, and device code that's meant to run on the GPU. The simple example program here computes the sum of two vectors. We have a kernel function that performs the actual computation on the GPU as well as a host function that can be called on the CPU side in order to kick off the computation on the GPU by launching the kernel.



CUDA Build Process*

```
__global__ void add_vectors_kernel(float* c, const float* a, const float* b, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N)
        c[i] = a[i] + b[i];
}


void add_vectors(float* c, const float* a, const float* b, int N)
{
    constexpr int block_size = 512;
    const int num_blocks = (N + block_size - 1) / block_size;

    add[illegible]b, N);
}
```

```
__global__ void add_vectors_kernel(…)
{
    unsigned int i = …;

    if (i < N)
        c[i] = a[i] + b[i];
}
```

```
void add_vectors(float* c, const float* a, const float* b, int N)
{
    constexpr int block_size = 512;
    const int num_blocks = (N + block_size - 1) / block_size;

    __cudaLaunch(…);
}
```

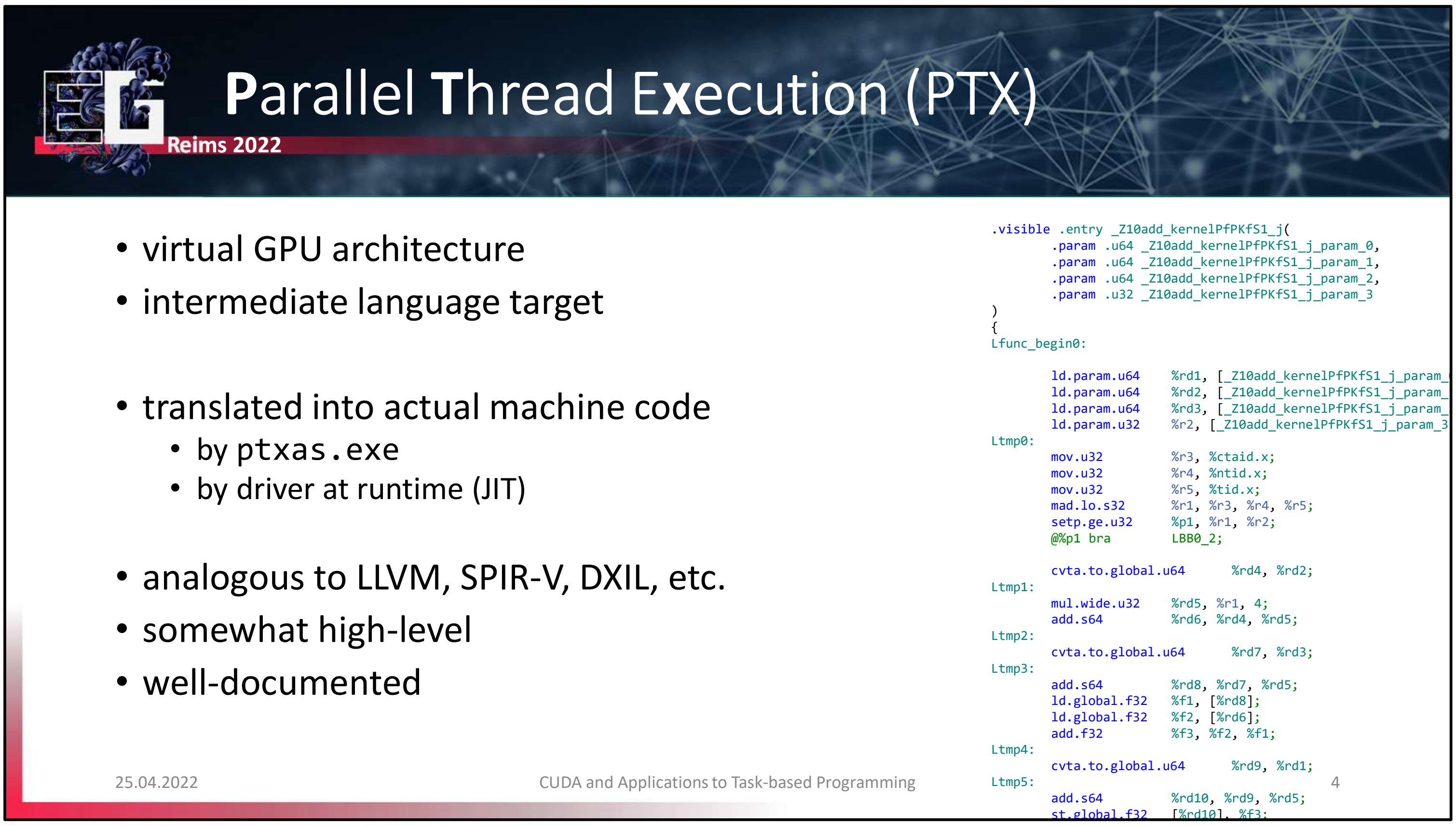
```
add_vectors_kernel<<<num_blocks, block_size>>>(c, a, b, N);
```

25.04.2022 CUDA and Applications to Task-based Programming 3

When compiling our example source file with nvcc, the first step nvcc will perform is to run the CUDA C++ front end cudafe++ on the input source file. This tool splits the input source code into separate source files, one with code to be compiled for the CPU and one with code to be compiled for GPU. It will also replace CUDA-specific syntax constructs like kernel launches with CUDA runtime library calls. The resulting CPU-side code is then simple standard C++ that can be fed into the normal host C++ compiler in order to obtain a normal object file.

Device code is instead fed into a device C++ compiler that generates PTX assembly, an intermediate representation for GPU code similar to, e.g., LLVM. This PTX code can then be assembled by the ptxas tool into a CUDA binary that contains actual machine code for a particular target GPU architecture. Both the PTX as well as precompiled binaries for various GPU architectures are then packed into an object file via the fatbinary tool.

Finally, the object file with the CPU-side code as well as the object file containing the GPU binaries can then be linked to form an executable. At runtime, the generated CPU-side code will call into the CUDA runtime libraries in order to perform work on the GPU. The CUDA runtime libraries take care of loading GPU binaries as needed using either one of the available precompiled GPU binaries or falling back to JIT compilation of the PTX if no suitable precompiled binary can be found.



Reims 2022

Parallel Thread Execution (PTX)

- virtual GPU architecture
- intermediate language target

- translated into actual machine code
  - by `ptxas.exe`
  - by driver at runtime (JIT)

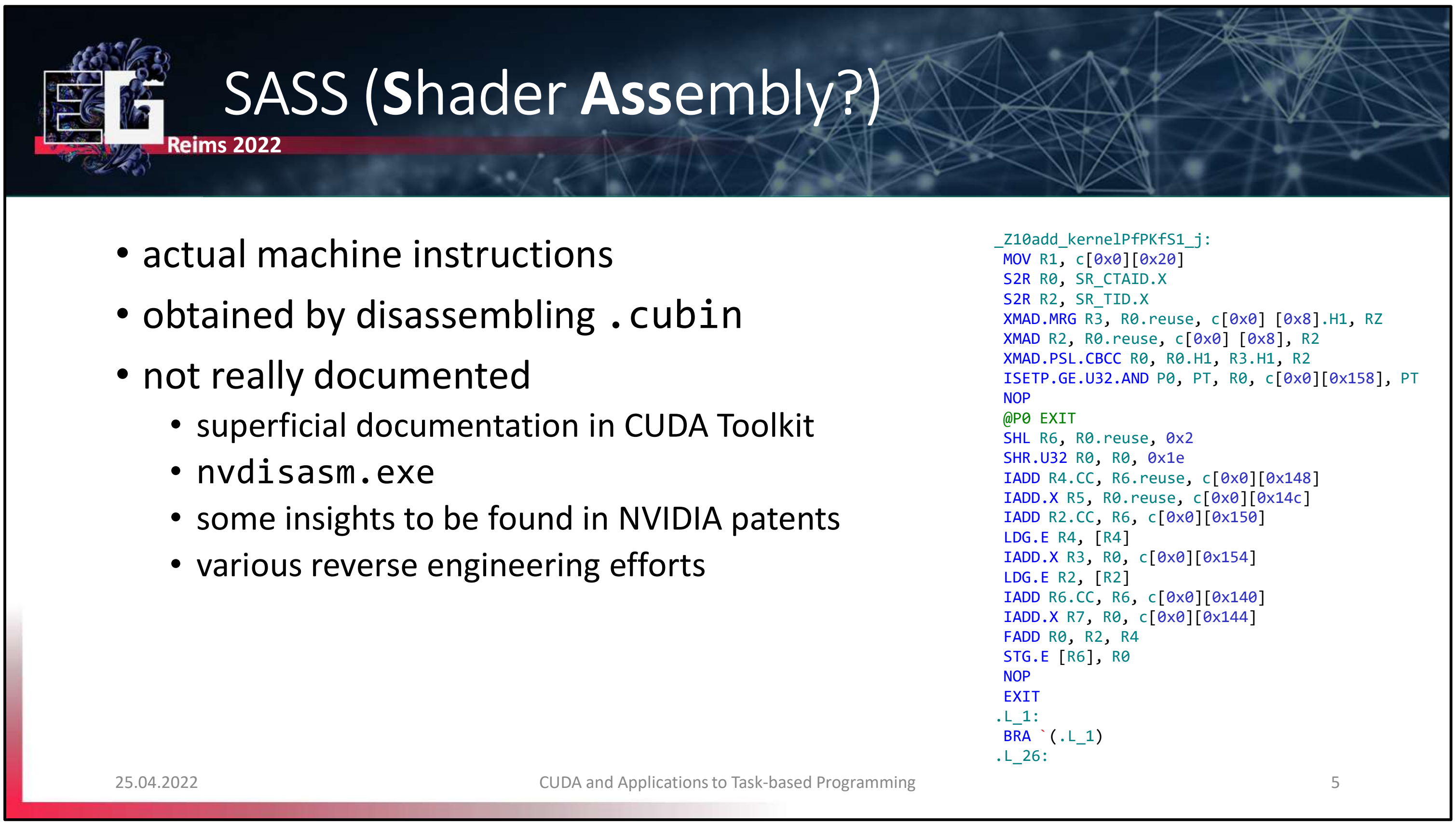
- analogous to LLVM, SPIR-V, DXIL, etc.
- somewhat high-level
- well-documented

```
.visible .entry _Z10add_kernelPfPKfS1_j(
        .param .u64 _Z10add_kernelPfPKfS1_j_param_0,
        .param .u64 _Z10add_kernelPfPKfS1_j_param_1,
        .param .u64 _Z10add_kernelPfPKfS1_j_param_2,
        .param .u32 _Z10add_kernelPfPKfS1_j_param_3
)
{
Lfunc_begin0:

        ld.param.u64    %rd1, [_Z10add_kernelPfPKfS1_j_param_
        ld.param.u64    %rd2, [_Z10add_kernelPfPKfS1_j_param_
        ld.param.u64    %rd3, [_Z10add_kernelPfPKfS1_j_param_
        ld.param.u32    %r2, [_Z10add_kernelPfPKfS1_j_param_3
Ltmp0:
        mov.u32         %r3, %ctaid.x;
        mov.u32         %r4, %ntid.x;
        mov.u32         %r5, %tid.x;
        mad.lo.s32      %r1, %r3, %r4, %r5;
        setp.ge.u32     %p1, %r1, %r2;
        @%p1 bra        LBB0_2;

        cvta.to.global.u64      %rd4, %rd2;
Ltmp1:
        mul.wide.u32    %rd5, %r1, 4;
        add.s64         %rd6, %rd4, %rd5;
Ltmp2:
        cvta.to.global.u64      %rd7, %rd3;
Ltmp3:
        add.s64         %rd8, %rd7, %rd5;
        ld.global.f32   %f1, [%rd8];
        ld.global.f32   %f2, [%rd6];
        add.f32         %f3, %f2, %f1;
Ltmp4:
        cvta.to.global.u64      %rd9, %rd1;
Ltmp5:
        add.s64         %rd10, %rd9, %rd5;
        st.global.f32   [%rd10], %f3;
```

25.04.2022 CUDA and Applications to Task-based Programming 4

There are two levels of assembly language one will encounter in CUDA. The first is Parallel Thread Execution (PTX) which serves as an intermediate representation for GPU code similar to, e.g., LLVM. PTX can be consumed by the CUDA driver directly or by the ptxas command line tool in order to be turned into executable machine code for a given GPU.

PTX is reasonably simple to read and well documented. While PTX is not what actually runs on a real GPU, it has similarities to the real GPU instruction set architectures. PTX does, however, contain higher-level constructs such as, e.g., functions, uses simplified control flow mechanisms, and many of its abstract instructions ultimately map to complex sequences of actual machine instructions. In order to fully understand the performance characteristics of a given piece of CUDA, e.g., as part of the profiling and optimization workflow, it is therefore often necessary to go further down to the actual machine code.



SASS (**S**hader **Ass**embly?)

Reims 2022

- actual machine instructions
- obtained by disassembling `.cubin`
- not really documented
  - superficial documentation in CUDA Toolkit
  - `nvdisasm.exe`
  - some insights to be found in NVIDIA patents
  - various reverse engineering efforts

```
_Z10add_kernelPfPKfS1_j:
 MOV R1, c[0x0][0x20]
 S2R R0, SR_CTAID.X
 S2R R2, SR_TID.X
 XMAD.MRG R3, R0.reuse, c[0x0] [0x8].H1, RZ
 XMAD R2, R0.reuse, c[0x0] [0x8], R2
 XMAD.PSL.CBCC R0, R0.H1, R3.H1, R2
 ISETP.GE.U32.AND P0, PT, R0, c[0x0][0x158], PT
 NOP
 @P0 EXIT
 SHL R6, R0.reuse, 0x2
 SHR.U32 R0, R0, 0x1e
 IADD R4.CC, R6.reuse, c[0x0][0x148]
 IADD.X R5, R0.reuse, c[0x0][0x14c]
 IADD R2.CC, R6, c[0x0][0x150]
 LDG.E R4, [R4]
 IADD.X R3, R0, c[0x0][0x154]
 LDG.E R2, [R2]
 IADD R6.CC, R6, c[0x0][0x140]
 IADD.X R7, R0, c[0x0][0x144]
 FADD R0, R2, R4
 STG.E [R6], R0
 NOP
 EXIT
.L_1:
 BRA `(.L_1)
.L_26:
```

25.04.2022 CUDA and Applications to Task-based Programming 5

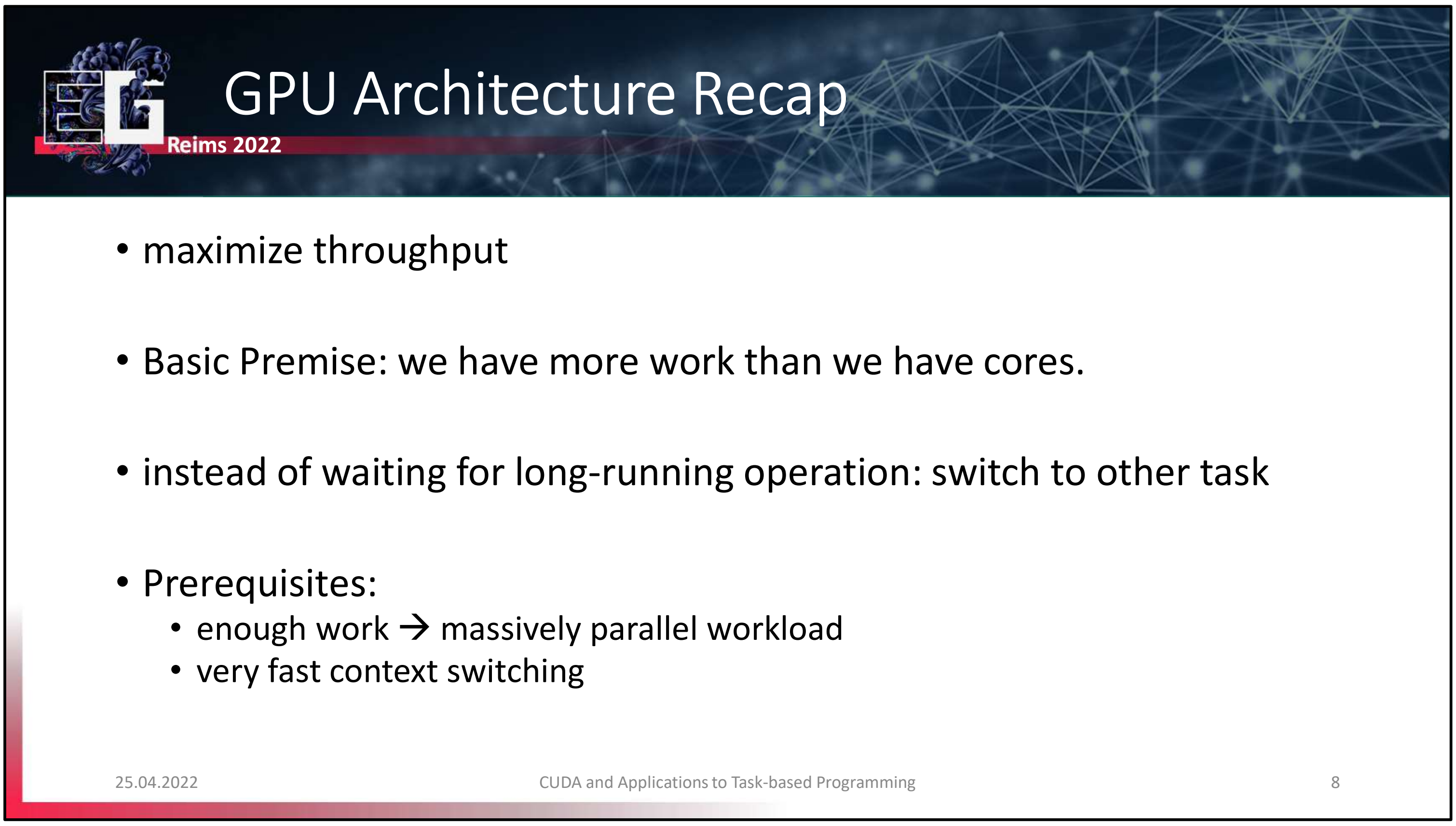
The second assembly language one will encounter in CUDA is known as SASS. This language is used to denote the actual machine instructions that run on a given GPU. The nvdisasm tool can be used to disassemble a CUDA binary into SASS. In contrast to PTX, SASS is unfortunately very poorly documented; even the name we can only assume to be an acronym for “shader assembly”. What little is publicly known about SASS is based on some very tight-lipped instruction listings found in the CUDA Toolkit documentation as well as reverse engineering efforts. Some insight into the workings of the machine instruction sets can also be gained from various NVIDIA patents relating to SIMT execution.

Nevertheless, PTX does not reflect various important aspects of how programs are executing on the GPU with a degree of accuracy sufficient for what some of the following discussion requires. Thus, for the remainder of this presentation, we will be looking at SASS code.

CUDA Build Process: Summary

- factor CUDA C++ into host and device parts
  - compiled separately
- generated host code
  - takes care of loading matching GPU binary stored in `.exe`
  - translate `kernel<<<…>>>(…)` syntax into API calls
- “Fat Binary” can contain both
  - PTX for various compute capabilities
    - allows the binary to target unknown architecture
  - precompiled machine code for specific GPU architectures
    - optimal performance on certain known devices

CUDA: Level 2



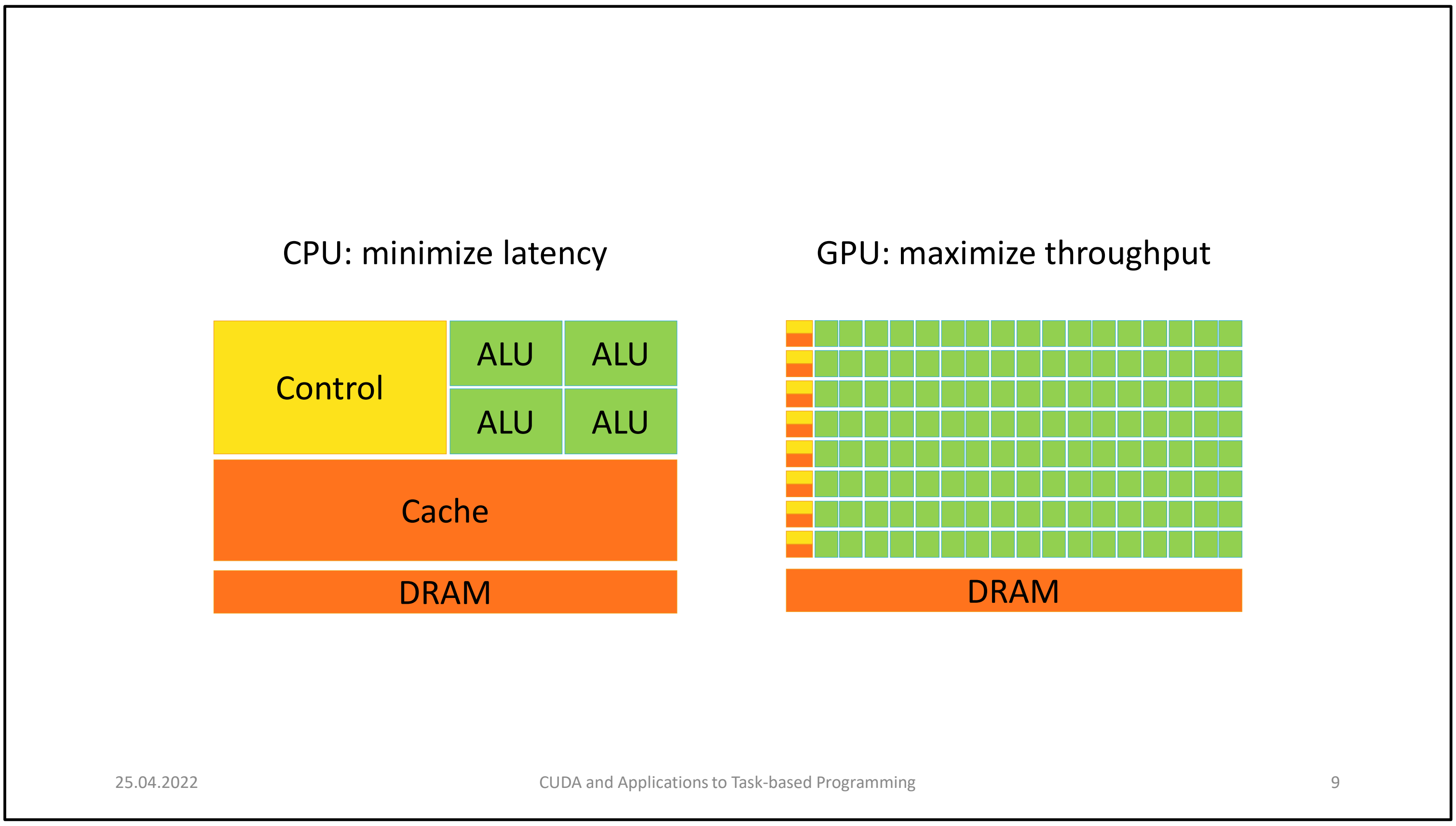
GPU Architecture Recap

Reims 2022

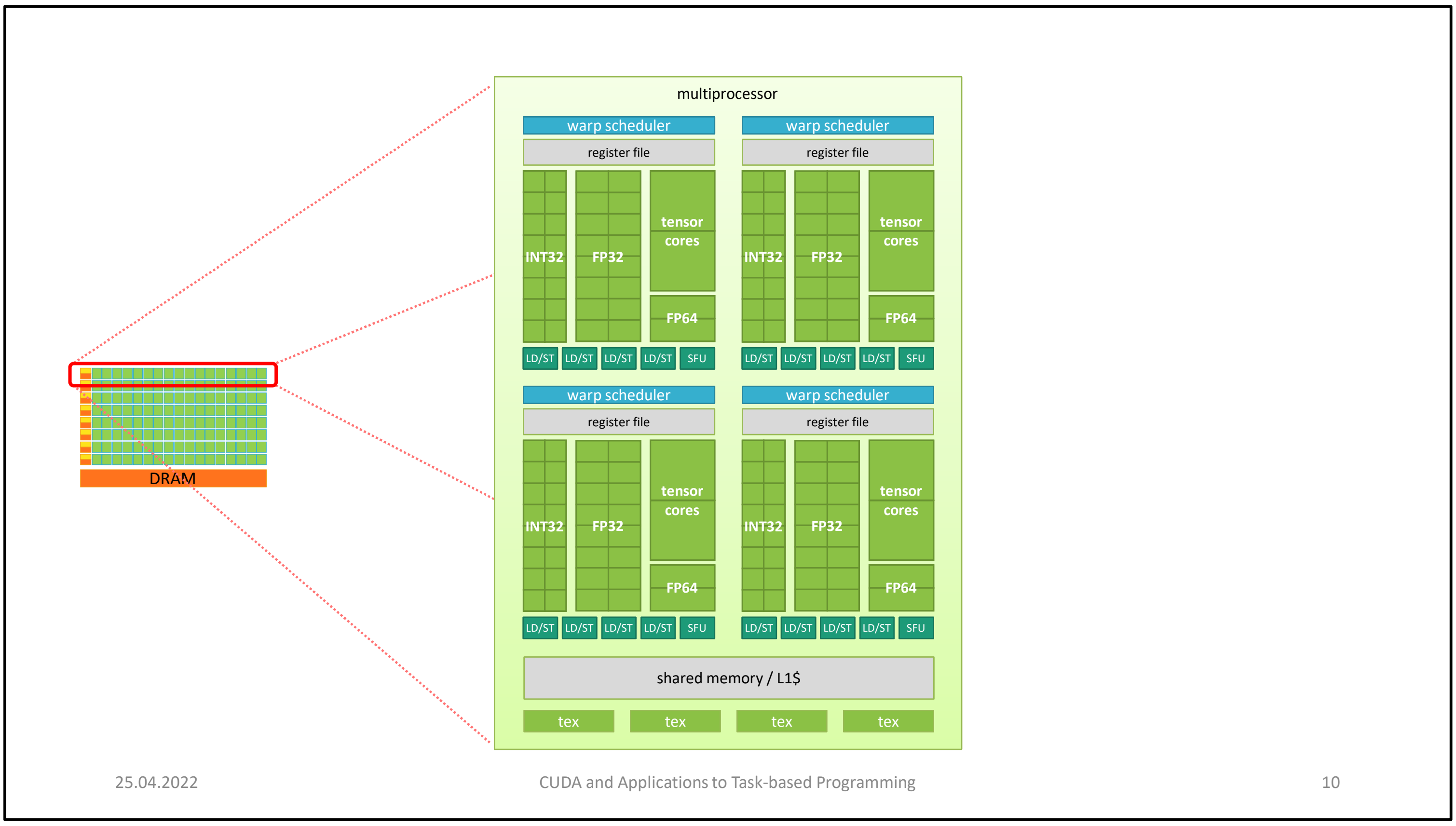
- maximize throughput
- Basic Premise: we have more work than we have cores.
- instead of waiting for long-running operation: switch to other task
- Prerequisites:
  - enough work → massively parallel workload
  - very fast context switching

25.04.2022 CUDA and Applications to Task-based Programming 8

From part 1, we remember that, contrary to CPU architectures which are designed to minimize latency, GPU architectures are designed to maximize throughput. The basic premise of GPU architecture is that we have so much independent work to get through that instead of waiting for any long-running operation to complete, we can always just switch to another task and, thus, hide the latency of long-running operations by doing other work until their results are available. For this approach to be successful, we need both a workload that is sufficiently large and parallelizable so that there is always enough independent work to switch between, and a means of switching between the execution contexts associated with different control flows very quickly.

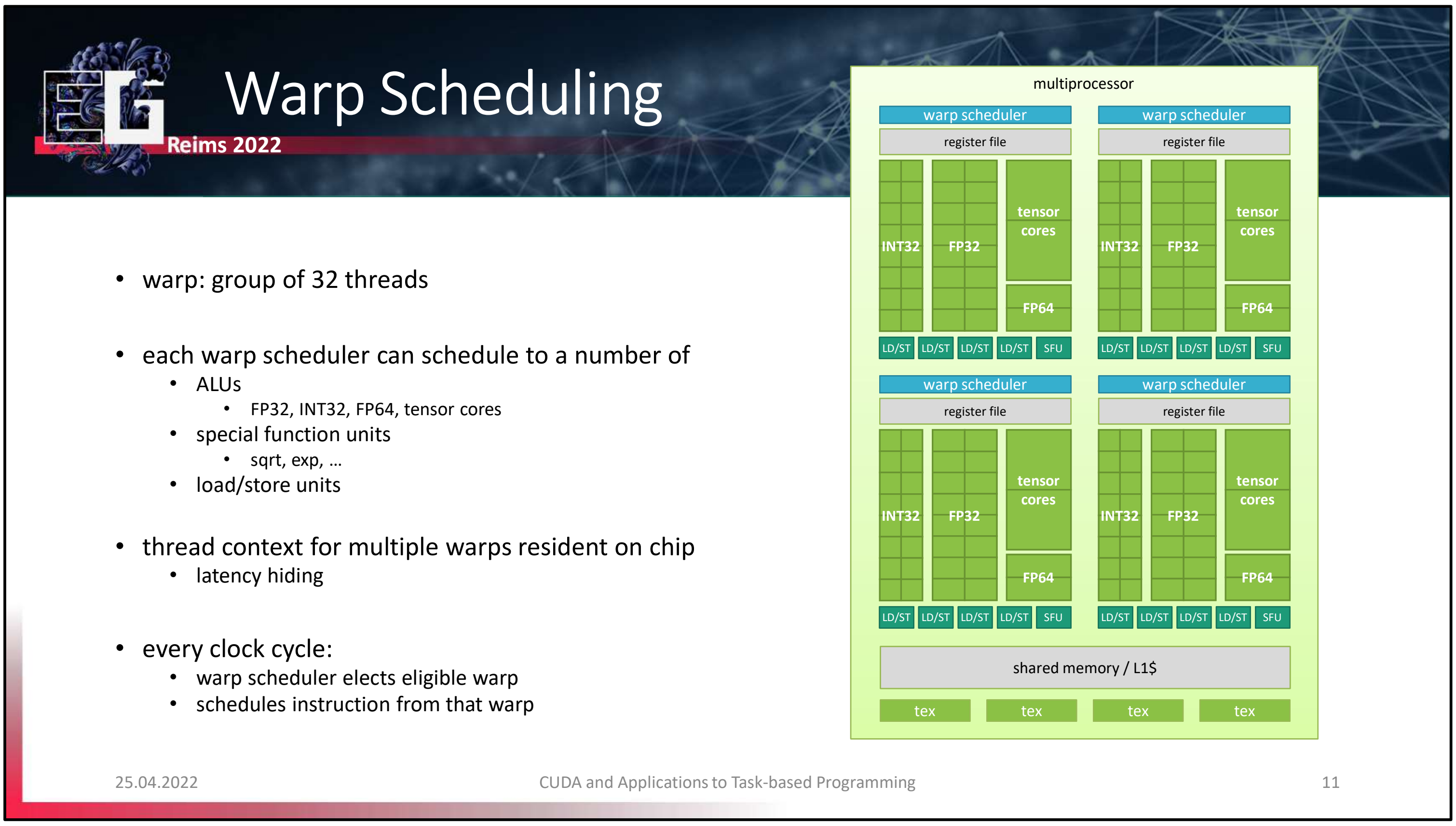


Instead of spending silicon on large caches, complex control logic for out-of-order execution, and other mechanisms that aim to speed up how quickly a single control flow can progress individually, GPU architectures spend silicon on building simple but many cores with large register files in order to keep the execution contexts of many independent control flows resident on chip to rapidly switch between. To further increase the fraction of silicon going towards computation as opposed to control logic, GPUs also rely on a SIMD approach. Each control logic is associated with multiple arithmetic logic units (ALUs), allowing for the parallel execution of multiple control flows as long as they all execute the same instructions just on different data.



25.04.2022 CUDA and Applications to Task-based Programming 10

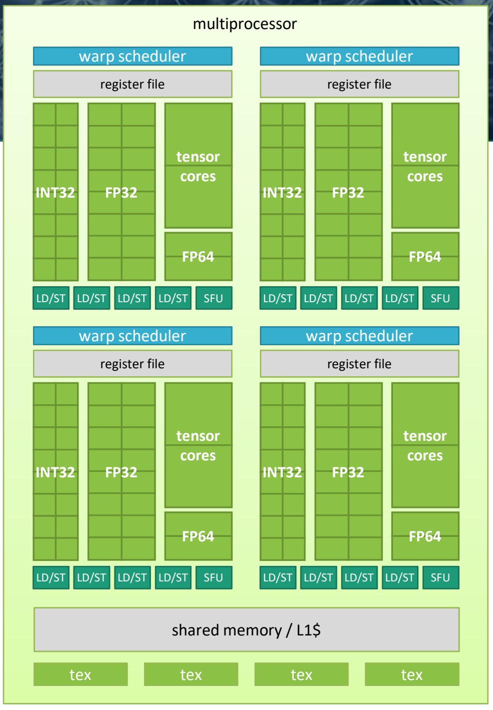
Taking a closer look at the organization of a typical GPU, we find what is essentially a two-level hierarchy that consists of groups of groups of SIMD cores. The GPU itself consists of multiprocessors. Each multiprocessor contains a number of warp schedulers. Associated with each warp scheduler is a register file, a set of 32-bit integer (INT32), 32-bit floating point (FP32), 64-bit floating point (FP64), and tensor core ALUs, load/store units (LD/ST), and a special function unit (SFU). Each multiprocessor also contains a local shared memory as well as a number of texture units that are shared among its warp schedulers. Part of the local shared memory is used as an L1 cache (L1$), the rest can be used to facilitate fast communication between threads that reside on the same multiprocessor.



Warp Scheduling

Reims 2022

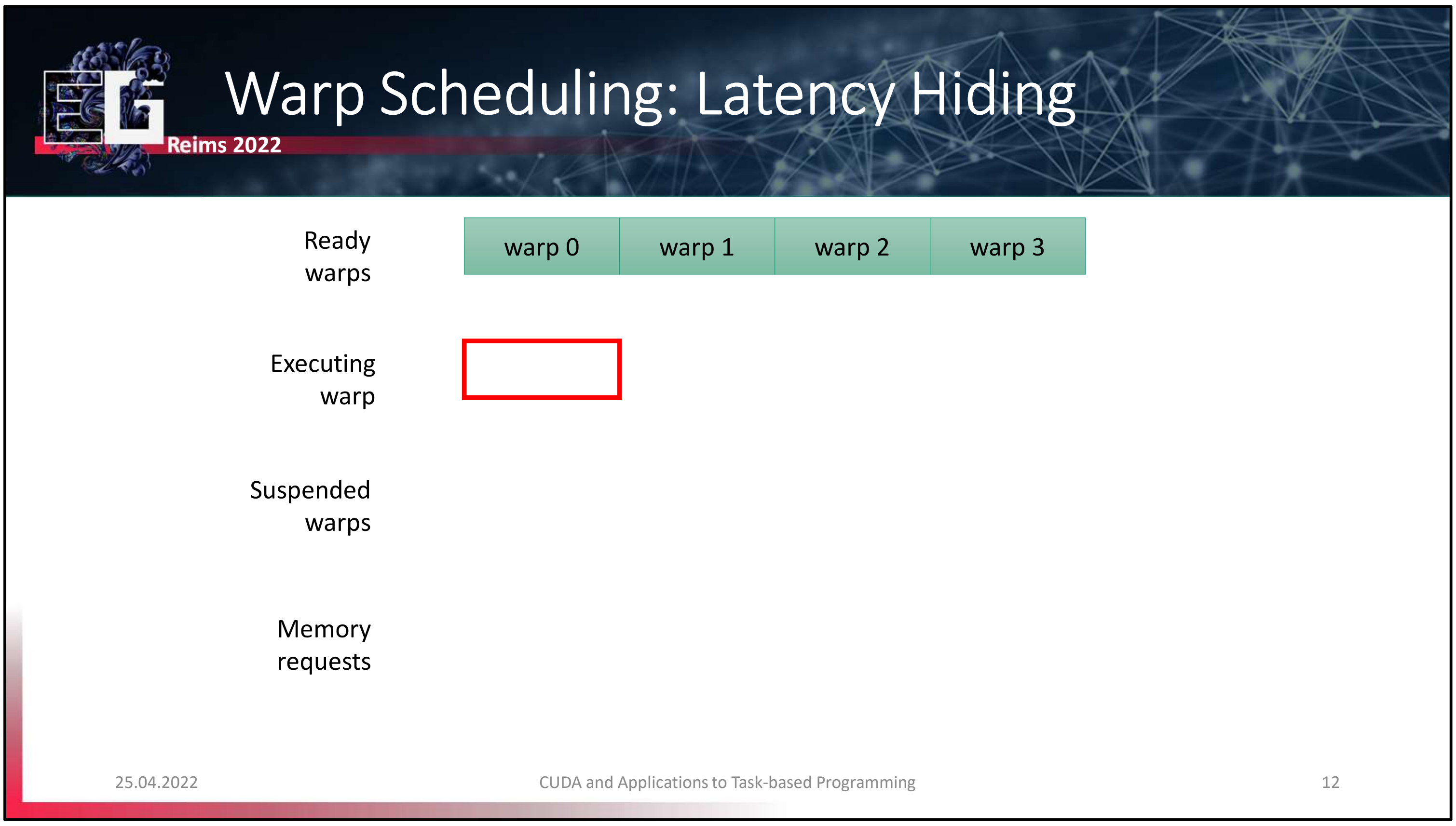
- warp: group of 32 threads
- each warp scheduler can schedule to a number of
  - ALUs
    - FP32, INT32, FP64, tensor cores
  - special function units
    - sqrt, exp, …
  - load/store units
- thread context for multiple warps resident on chip
  - latency hiding
- every clock cycle:
  - warp scheduler elects eligible warp
  - schedules instruction from that warp



25.04.2022 CUDA and Applications to Task-based Programming 11

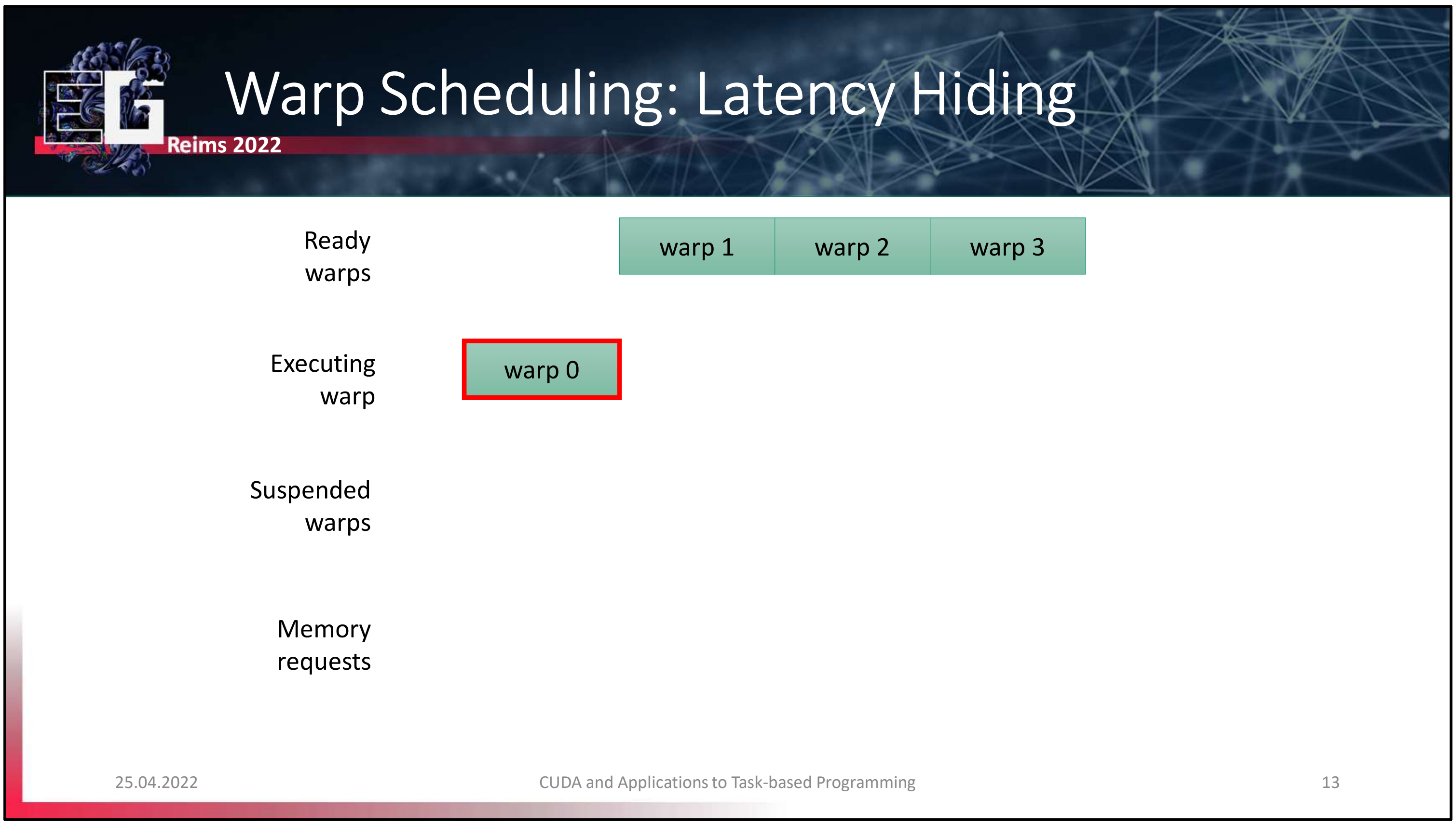
As already mentioned in part 1, the fundamental unit of execution on a GPU is a warp, a group of 32 threads. In order to run a kernel on the GPU, each warp scheduler is assigned a set of warps to execute. The execution context of all threads that are part of each warp is kept locally in the register file associated with the warp scheduler responsible for the warp. The warp scheduler’s job is to map execution of the threads in its warps to the hardware unit (ALUs, etc.) it’s responsible for.

Whenever it could issue a new instruction (e.g., each clock cycle unless all of the necessary resources are occupied), the warp scheduler will pick the next instruction from one of its warps that are ready to run (i.e., not waiting on the result of a previous operation) and schedule it to the appropriate hardware unit.



Warp Scheduling: Latency Hiding

Let’s look at a simple example. Suppose we have a warp scheduler that has four warps to execute. Initially, all four warps are ready to run.

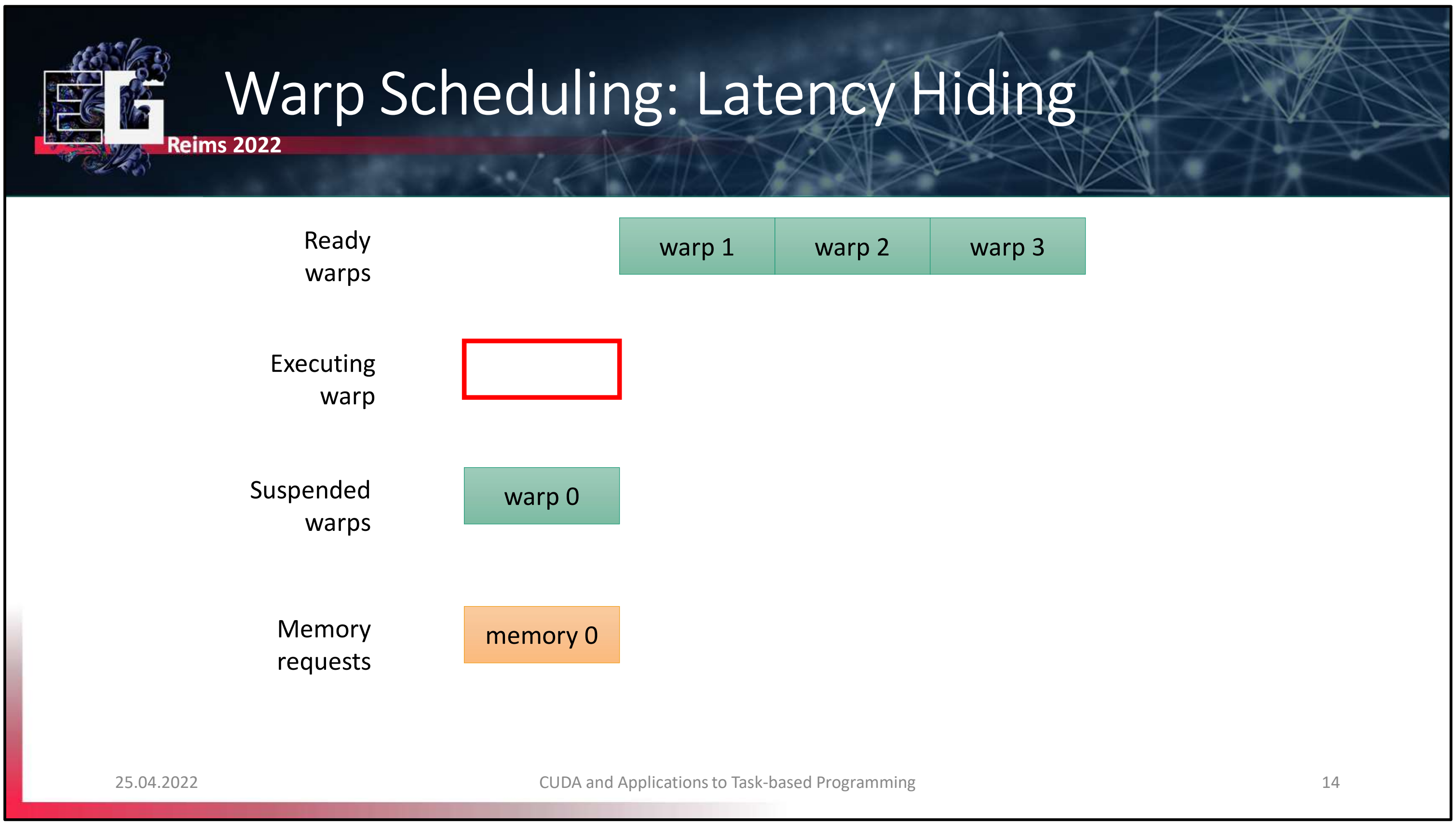


Warp Scheduling: Latency Hiding

Reims 2022

25.04.2022 CUDA and Applications to Task-based Programming 13

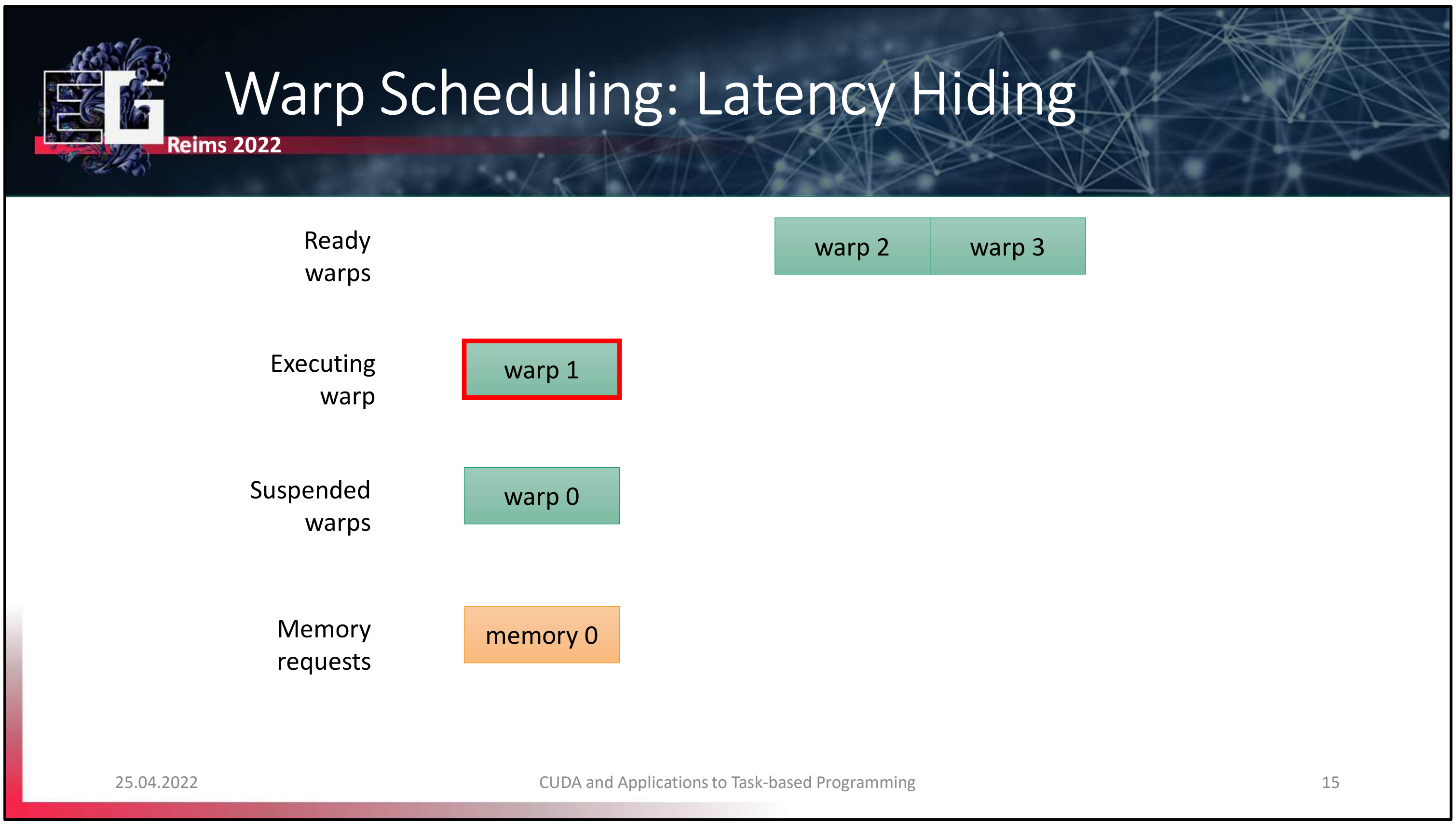
At instruction issue time, the warp scheduler simply picks any one of the four ready warps—in this case, warp 0—and schedules its next instruction.



Warp Scheduling: Latency Hiding

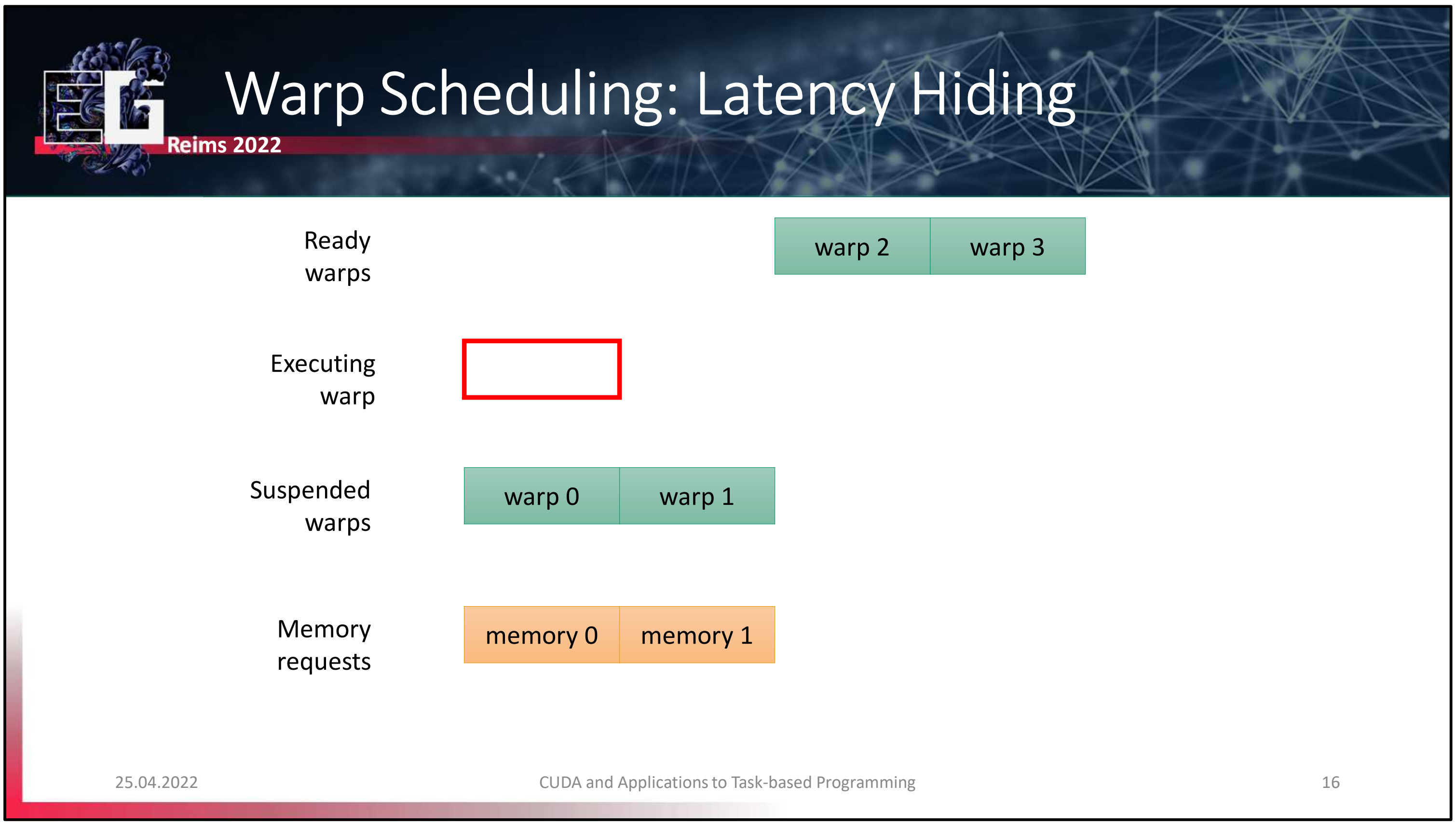
Reims 2022

In this example, the first thing the kernel program did was perform some memory access. Thus, execution of the first instruction resulted in a memory request. The next instruction in warp 0 also depends on the result of the memory request. Thus, warp 0 is now suspended as its next instruction cannot be issued until the memory request has been served.



Warp Scheduling: Latency Hiding

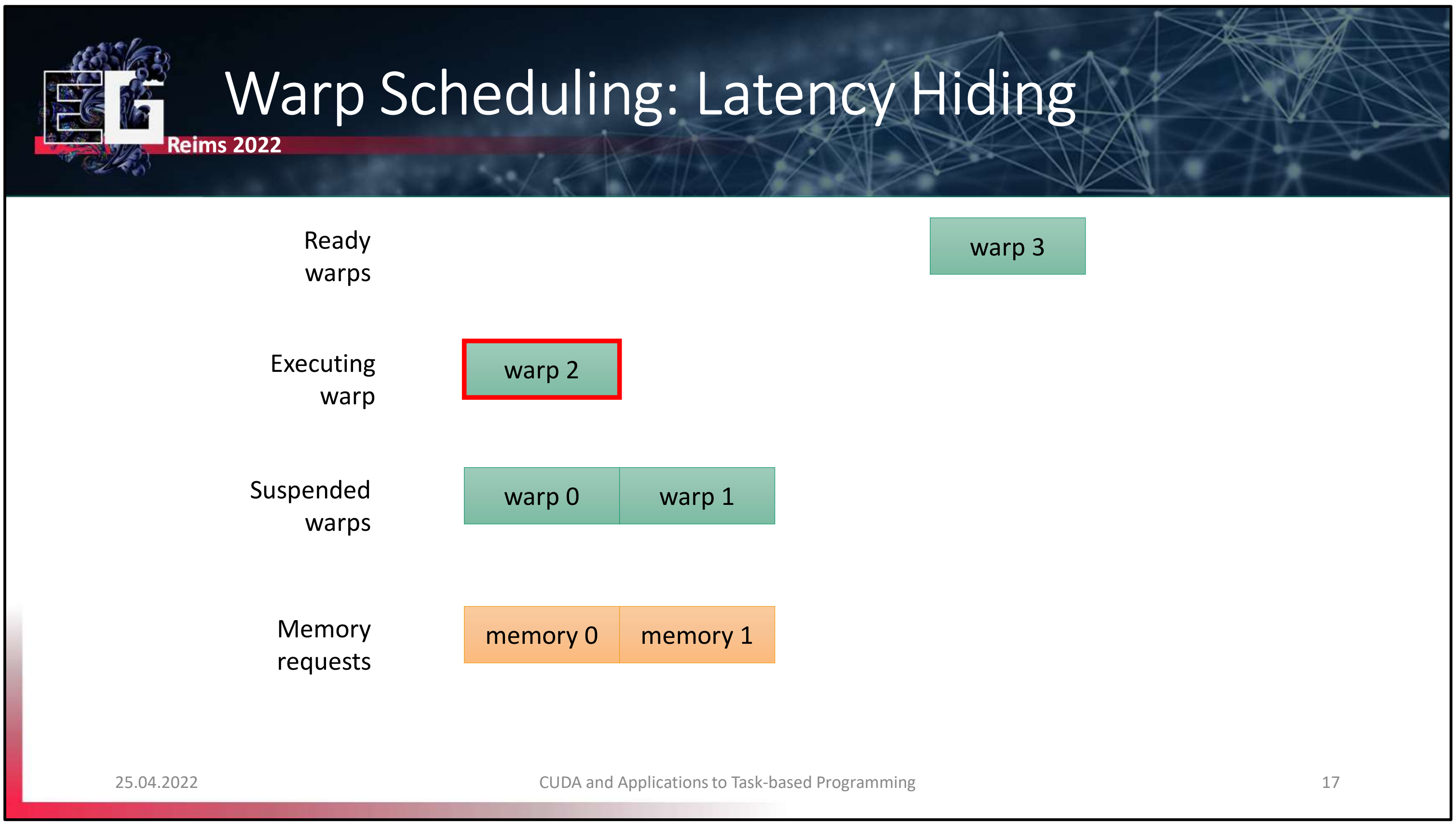
However, while warp 0 is suspended, there are still three other warps that are ready to run. Thus, in the next cycle, the warp scheduler can simply pick any of these three other warps to issue instructions from. Let’s say it picks warp 1 in this case.



Warp Scheduling: Latency Hiding

25.04.2022 CUDA and Applications to Task-based Programming 16

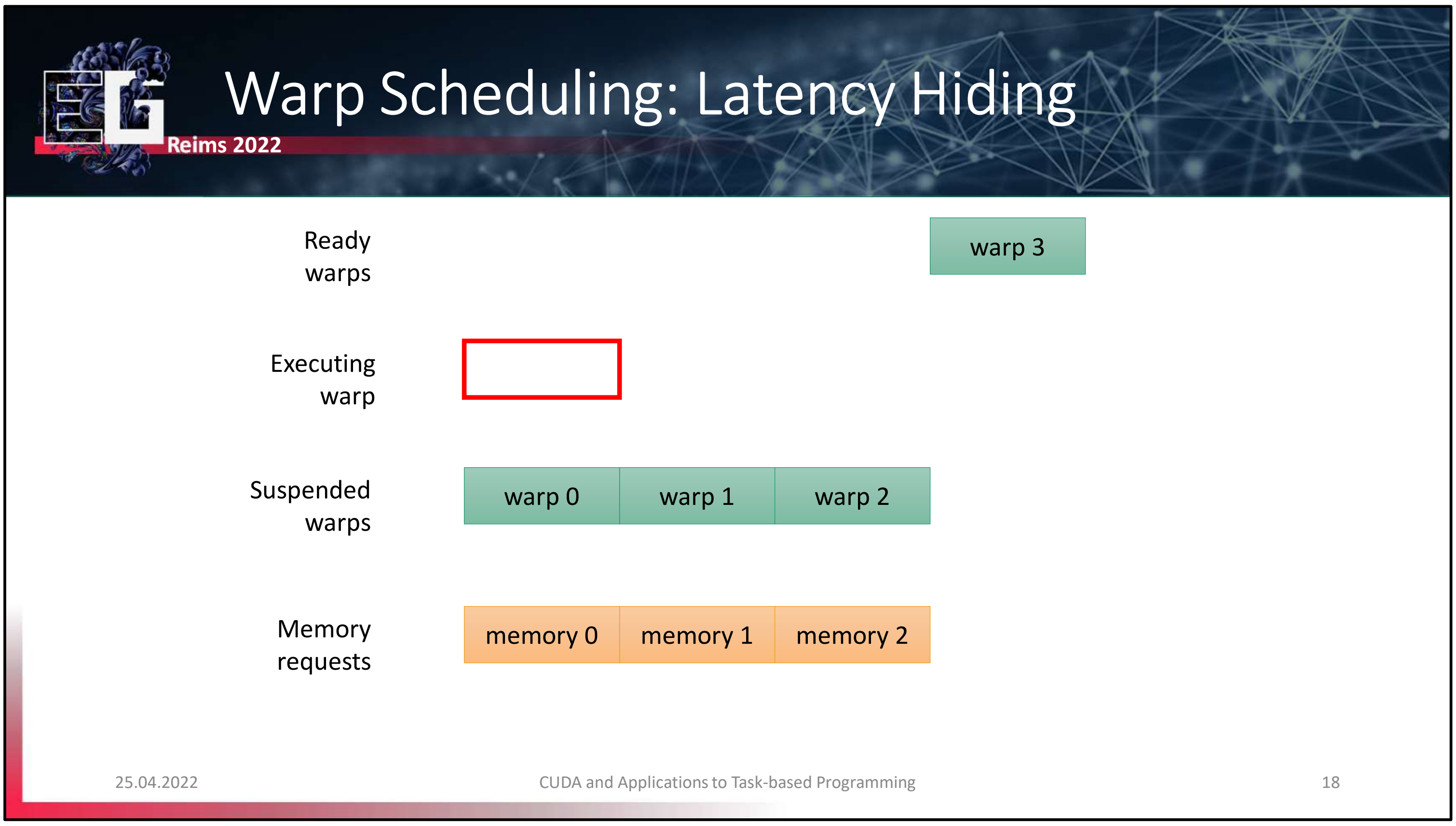
As all four warps are running the same kernel program, execution of the next instruction in warp 1 will result in a memory access as well. Thus, another memory request is now in flight and warp 1 is suspended as well.



Warp Scheduling: Latency Hiding

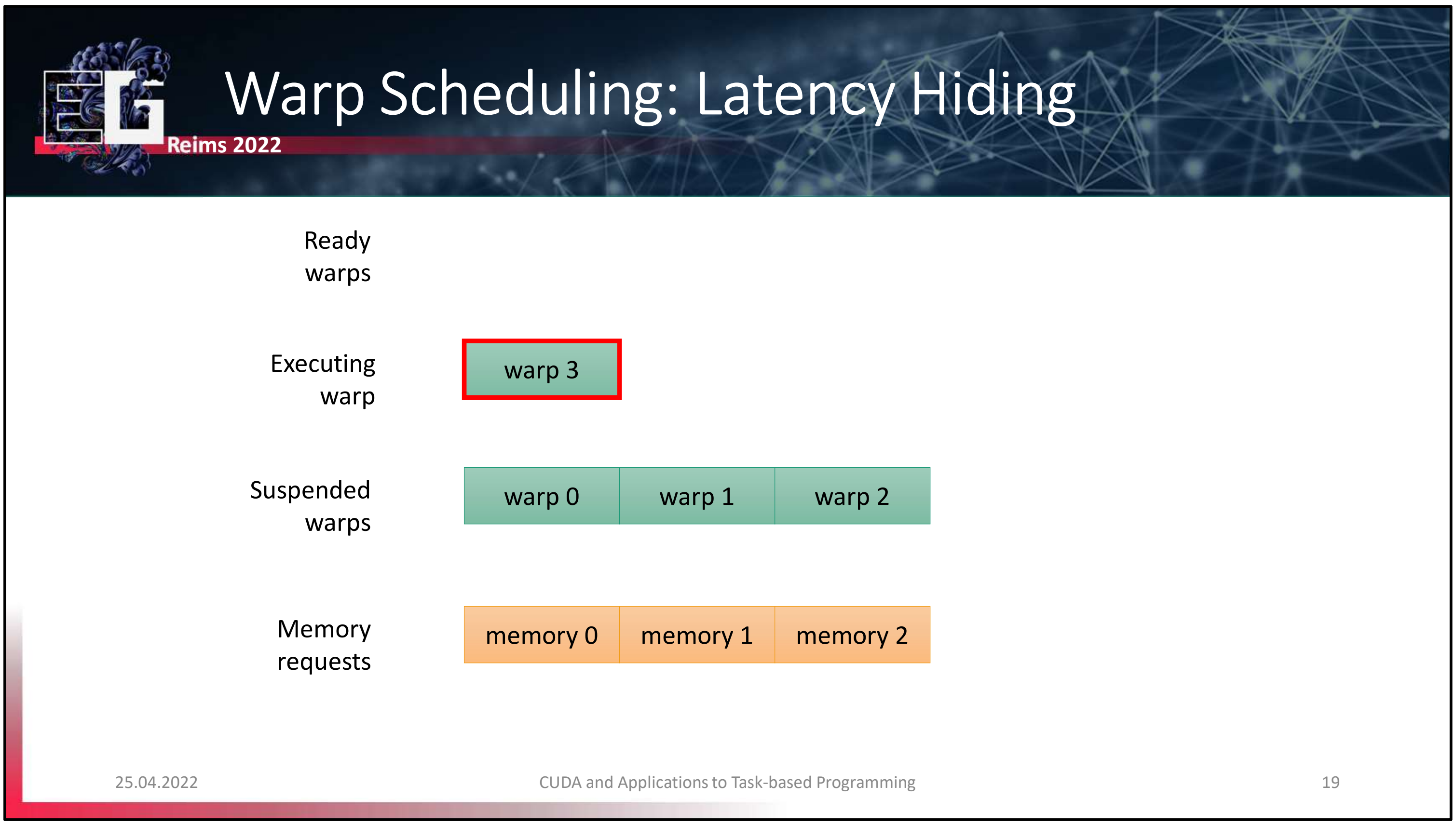
25.04.2022 CUDA and Applications to Task-based Programming 17

The same process repeats for the remaining two warps. We issue the next instruction from warp 2.



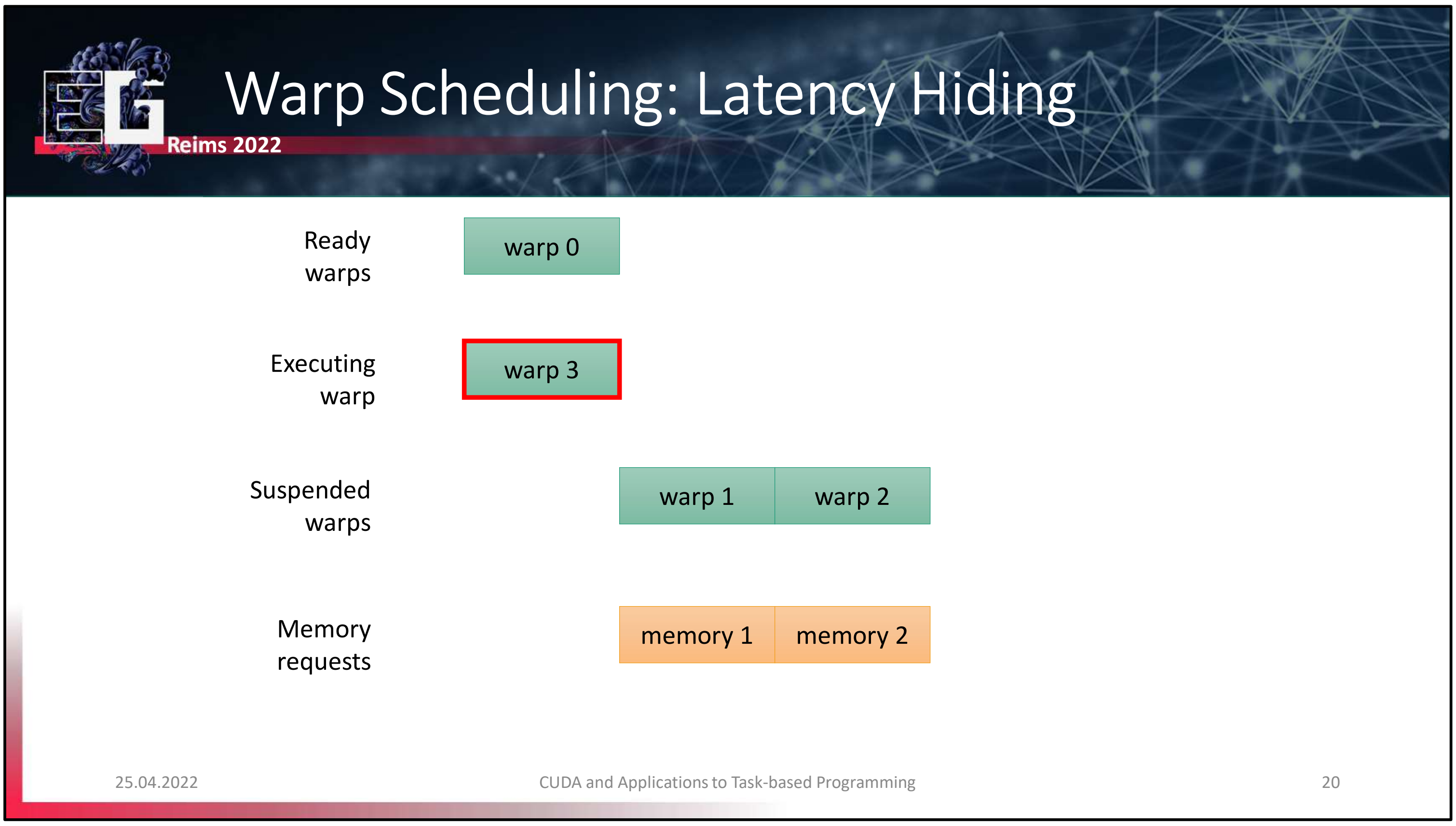
Warp Scheduling: Latency Hiding

The instruction, again, results in a memory request, the corresponding warp 2 is suspended.



Warp Scheduling: Latency Hiding

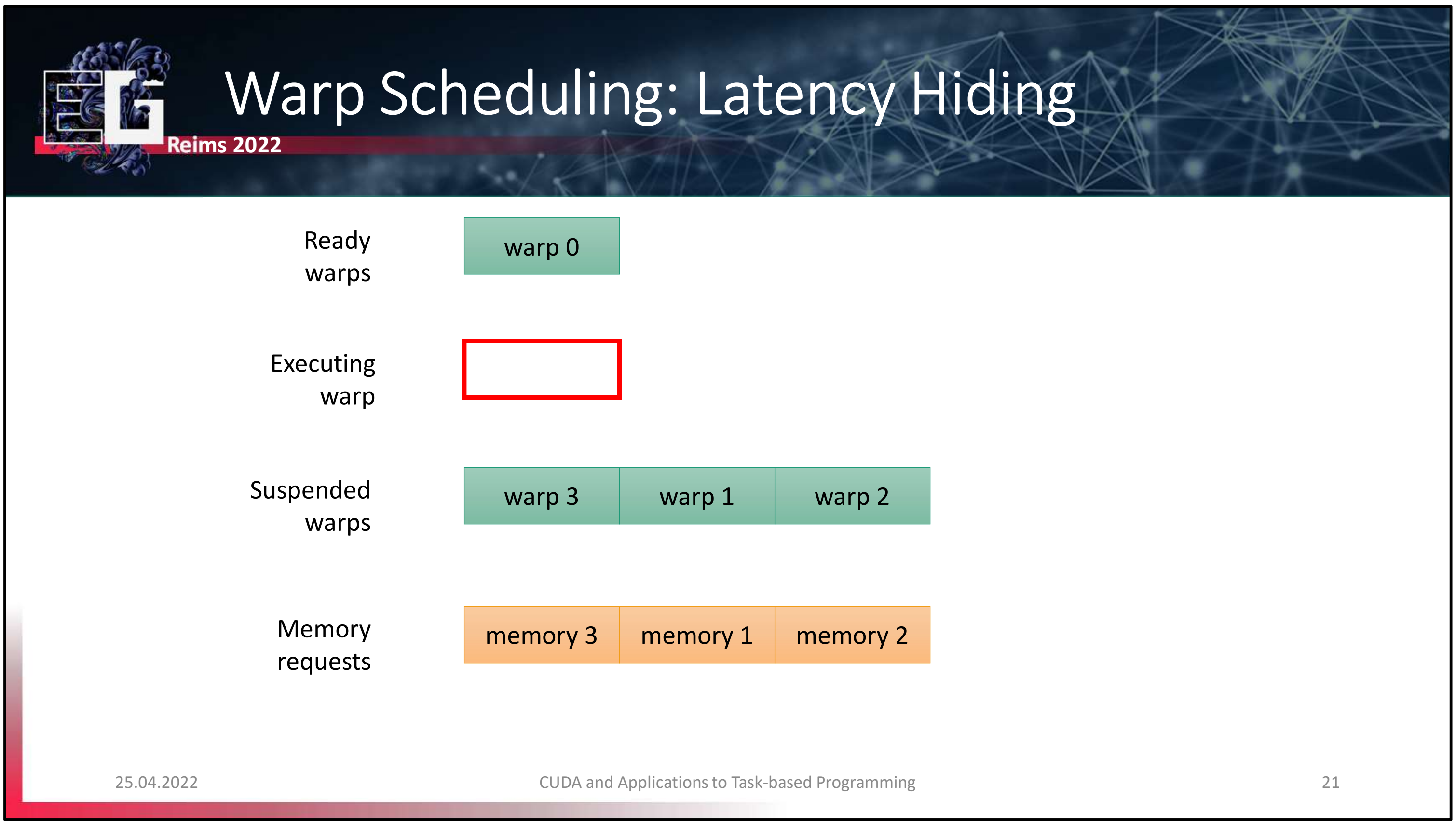
We still have one more ready warp, so we issue its next instruction.



Warp Scheduling: Latency Hiding

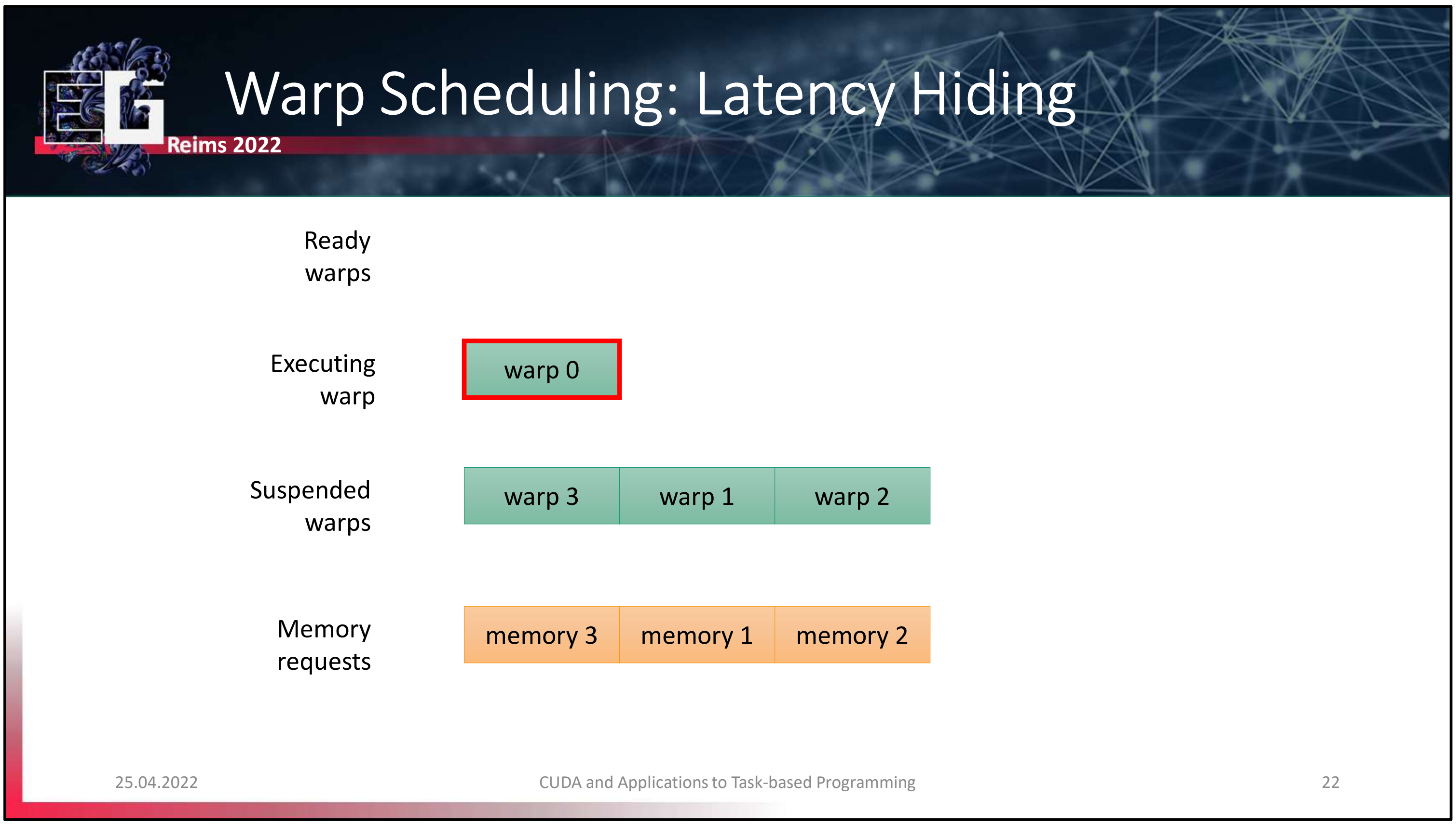
25.04.2022 CUDA and Applications to Task-based Programming 20

However, while we were busy executing instructions from the previous three warps, the initial memory request spawned by warp 0 completes. Thus, as we are executing the next instruction from warp 3, warp 0 becomes ready again since its next instruction now has its dependencies satisfied.



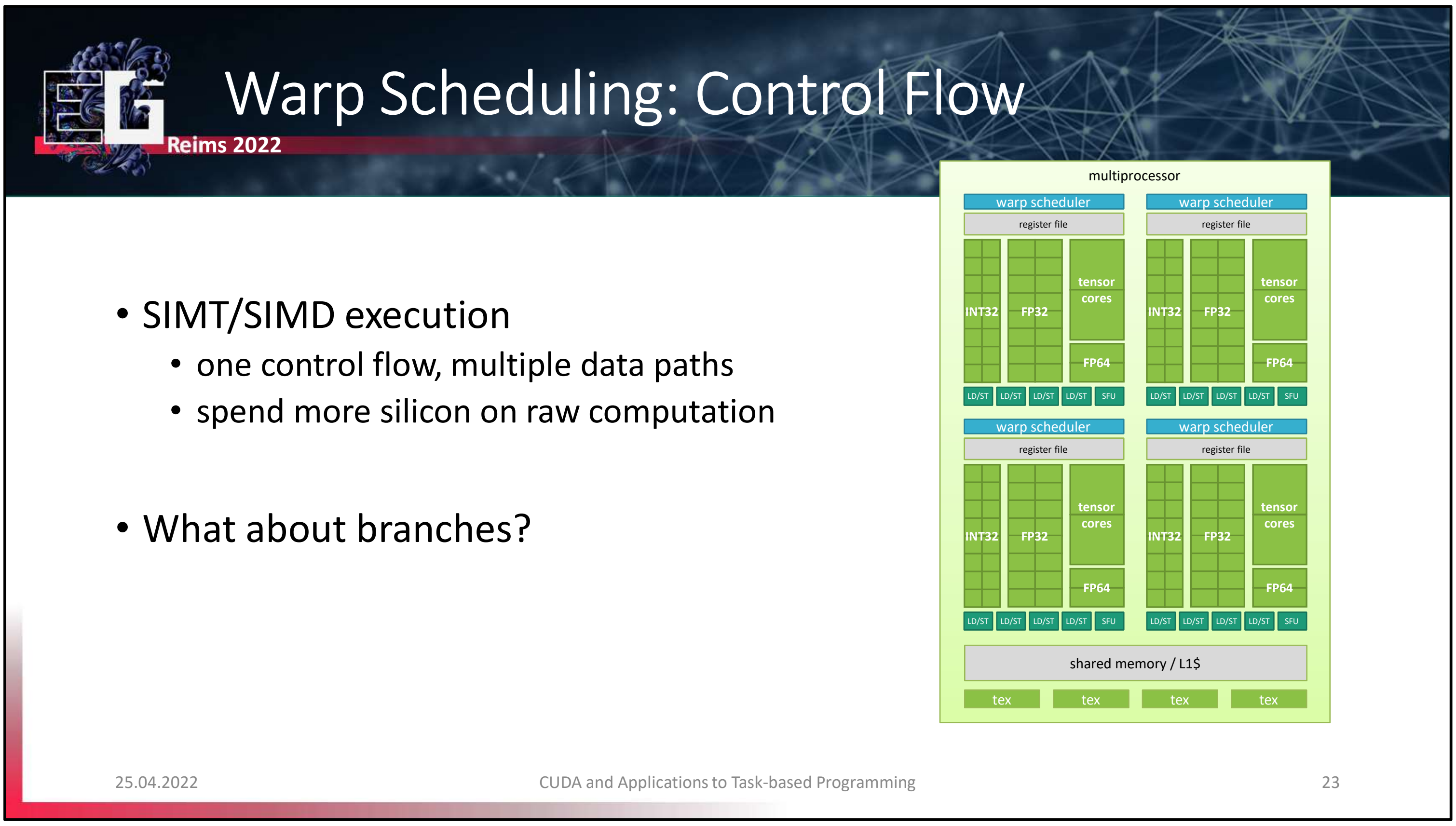
Warp Scheduling: Latency Hiding

As with the other three warps, execution of the first instruction in warp 3 results in yet another memory request and warp 3 is suspended.



Warp Scheduling: Latency Hiding

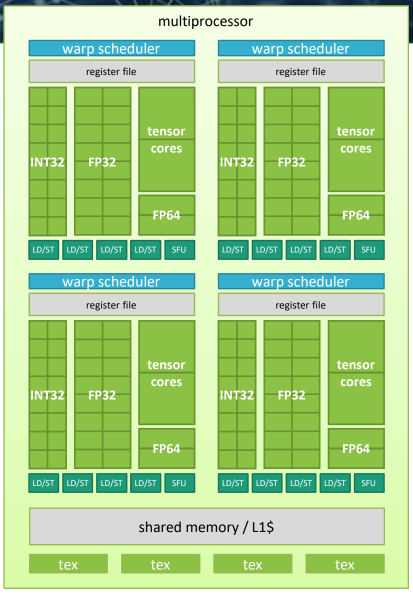
However, as warp 0 has become ready again in the meanwhile, we can issue its next instruction. Thus, we were able to hide the latency of the memory request from warp 0 by executing instructions from our other three warps while the memory request was being served. Assuming the other memory requests will complete within a similar amount of time, there is a good chance that at least warp 1 will be ready again by the next time warp 0 has to be suspended.



Warp Scheduling: Control Flow

Reims 2022

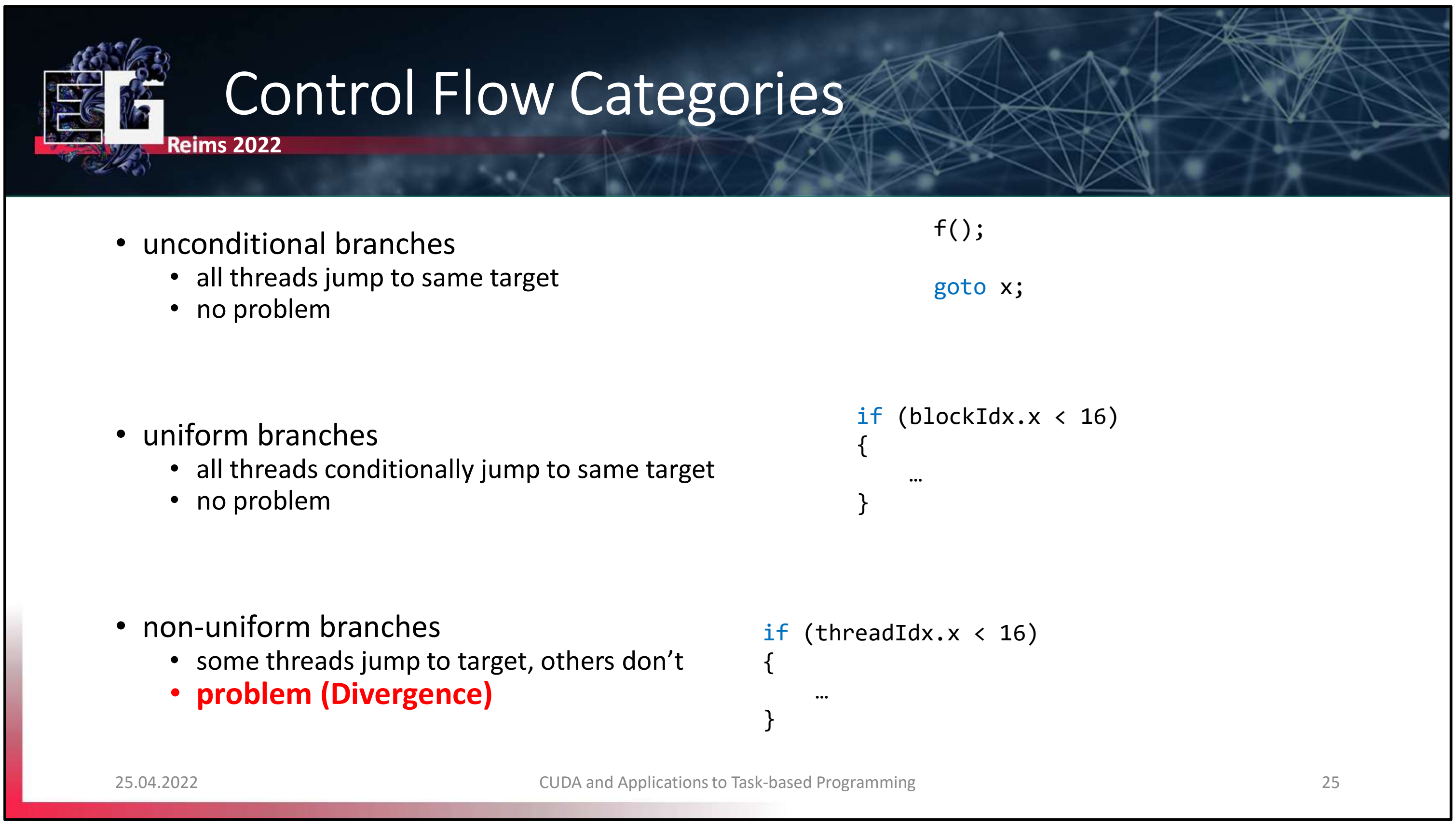
- SIMT/SIMD execution
  - one control flow, multiple data paths
  - spend more silicon on raw computation
- What about branches?



25.04.2022 CUDA and Applications to Task-based Programming 23

So far, we have simply assumed that all threads within a warp always need to run the same instruction next, allowing us to execute all threads of a warp in parallel in SIMD fashion. This is of course only true if we assume no complex control flow such as, e.g., branches in the individual threads. In general, we need to be able to handle such control flow, however.

DISCLAIMER: The following presents our best educated guesses based on what little information NVIDIA has disclosed about the features we're about to discuss in the form of documentation, technical reports, and various patents combined with what others have discovered, detailed on, and otherwise mentioned in various blog posts, articles, and forum posts. It does, for the most part, not constitute official information, there is some room for error. It should, however, serve sufficiently well for the purposes of forming a working understanding of how control flow is implemented at the ISA level.



Control Flow Categories

Reims 2022

- unconditional branches
  - all threads jump to same target
  - no problem

```
f();

goto x;
```

- uniform branches
  - all threads conditionally jump to same target
  - no problem

```
if (blockIdx.x < 16)
{
    …
}
```

- non-uniform branches
  - some threads jump to target, others don't
  - **problem (Divergence)**

```
if (threadIdx.x < 16)
{
    …
}
```

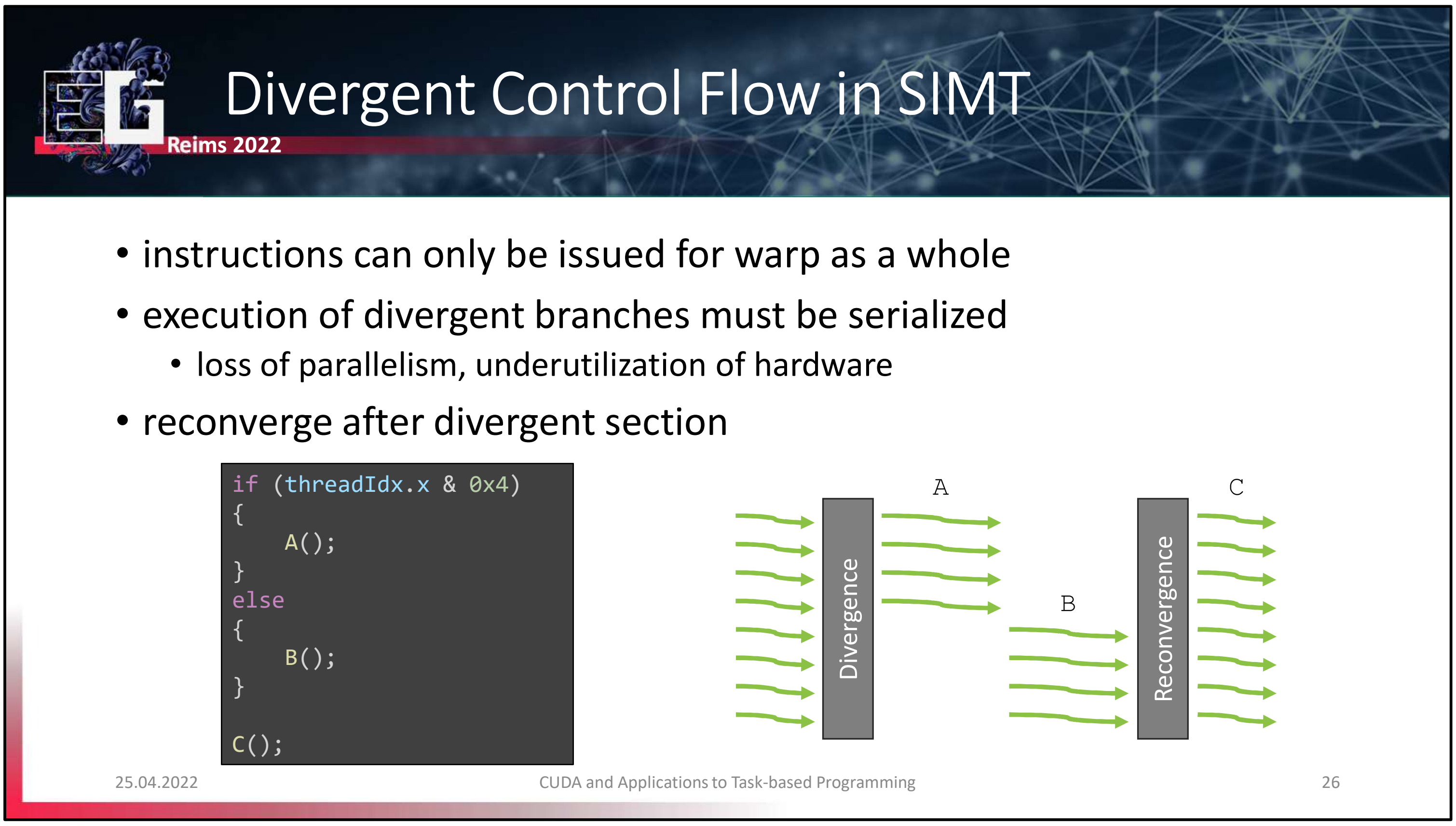
25.04.2022 CUDA and Applications to Task-based Programming 25

If we consider the different ways in which control flow may evolve within a warp, we find that there are basically three categories of branches. First, we have the case of an unconditional branch: all threads jump to the same target, e.g., a function call or `goto`. Such control flow poses no problem; before and after such a branch, the next instruction for all threads in the warp is the same.

Second, we have the case of a uniform branch: threads conditionally jump to a target, but the condition evaluates to the same value for all threads within the warp. Since all threads of the warp will go on into the same part of the branch, such control flow also poses no problem; before and after such a branch, the next instruction is once again the same for all threads in the warp.

Finally, we have the case of a non-uniform branch: threads conditionally jump to a target, but the condition does not evaluate to the same value for all threads within the warp. Thus, some threads jump while other threads don't. Such a branch gives rise to a condition known as divergence. Control flow no longer continues along the same path for all threads of the warp.

There are various further kinds of control flow one may want to consider such as the case of indirect branches where all threads jump to a target, but the target of the jump may be different for each thread. While the exact circumstances via which divergence may arise vary, what is ultimately of importance is simply whether control flow could diverge or not, and whether control flow does diverge or not.

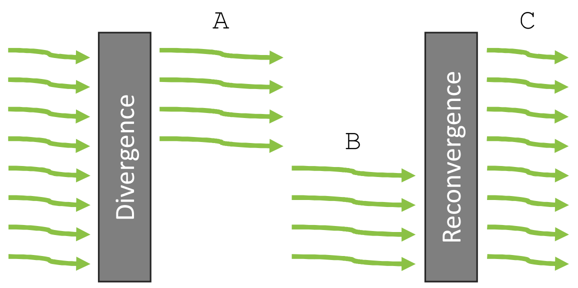


Divergent Control Flow in SIMT

Reims 2022

- instructions can only be issued for warp as a whole
- execution of divergent branches must be serialized
  - loss of parallelism, underutilization of hardware
- reconverge after divergent section

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

C();
```

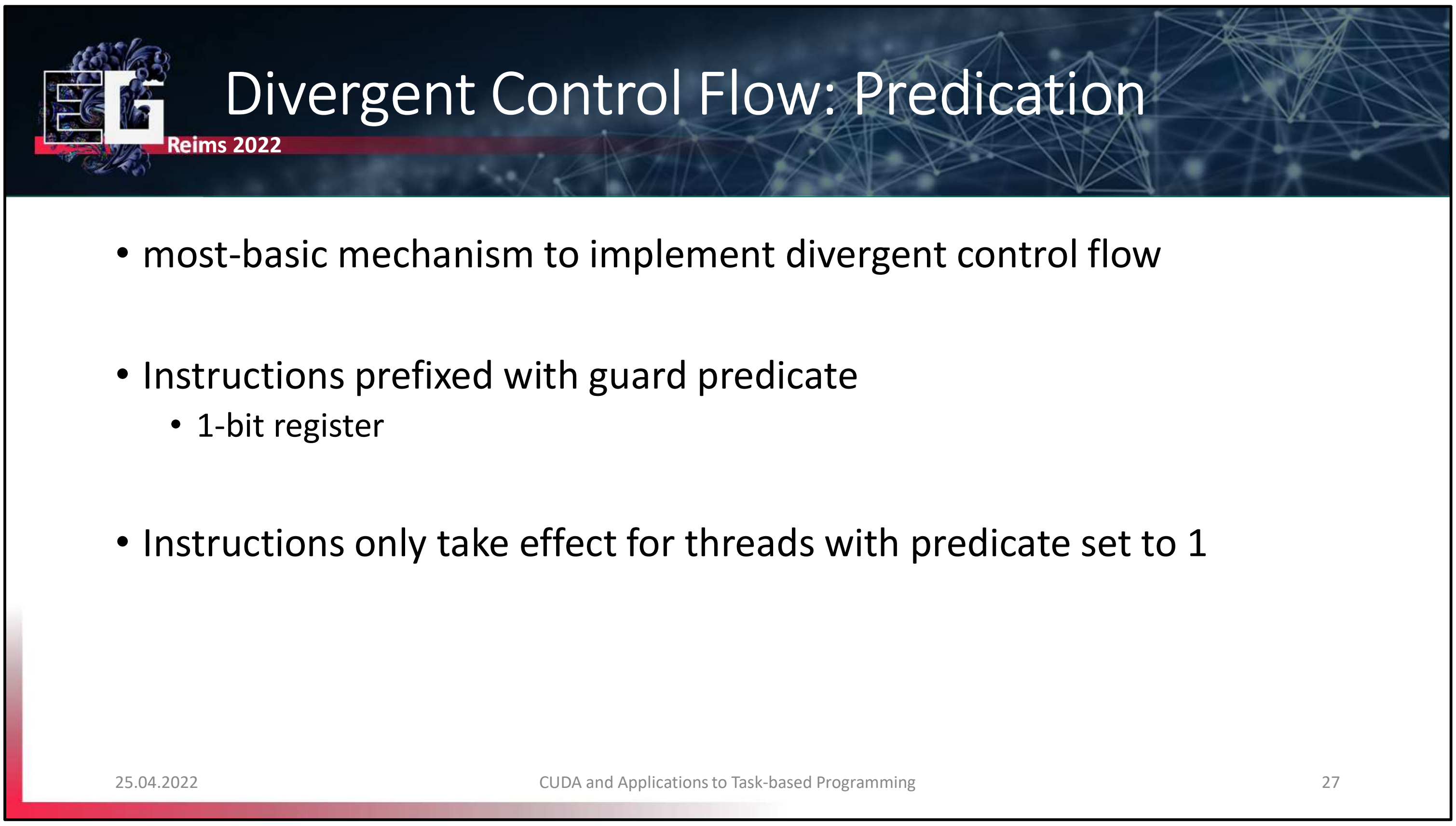


25.04.2022 CUDA and Applications to Task-based Programming 26

As previously discussed, the warp scheduler can only issue instructions for the entire warp as a whole. Thus, if control flow within a warp diverges, we are forced to serialize execution of the two divergent branches by, e.g., first running the instructions for one branch only for the threads that have taken this branch and then running the instructions for the other branch only for the threads in that other branch.

During execution of each branch, only a subset of threads will be active, which means that some of the hardware resources that could have been used to perform computation may effectively go unused. Thus, divergence results in a loss of parallelism and potential underutilization of the hardware. It is therefore generally desirable to reconverge the warp, i.e., move back to a common control flow as soon as possible.

For the CUDA programmer, this means that divergent code should generally be avoided and, where unavoidable, sections of code that need to be run in divergence should be kept as short as possible.



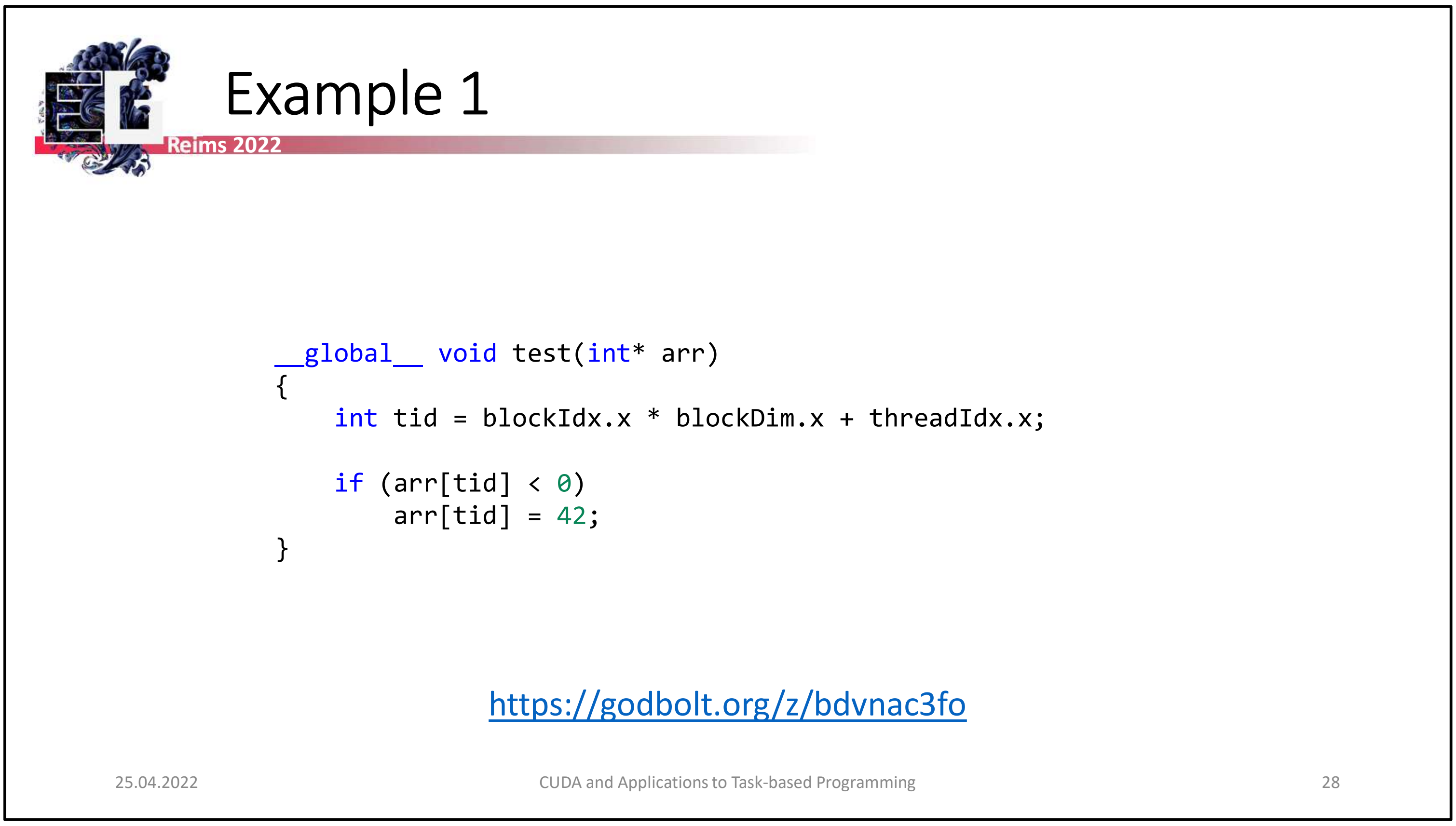
Divergent Control Flow: Predication

Reims 2022

- most-basic mechanism to implement divergent control flow
- Instructions prefixed with guard predicate
  - 1-bit register
- Instructions only take effect for threads with predicate set to 1

25.04.2022 CUDA and Applications to Task-based Programming 27

The most basic mechanism by which we can implement divergent control flow within a warp is predication. Instructions can be prefixed with a guard predicate that nominates a 1-bit predicate register. When a predicated instruction is executed, its effects are only materialized for the threads in which the corresponding predicate register contains a 1.



Example 1

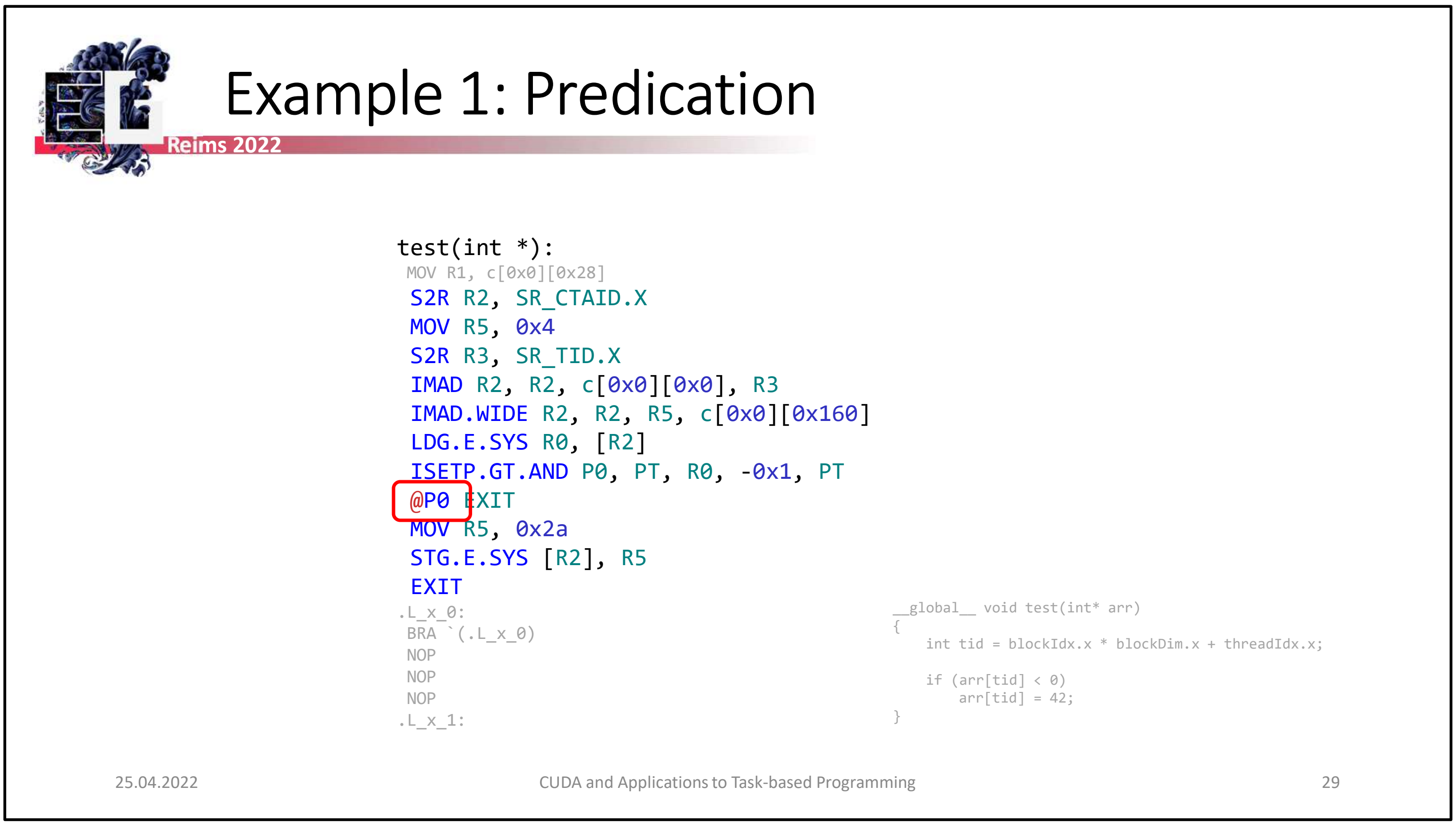
Reims 2022

```
__global__ void test(int* arr)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (arr[tid] < 0)
        arr[tid] = 42;
}
```

https://godbolt.org/z/bdvnac3fo

25.04.2022 CUDA and Applications to Task-based Programming 28

Let’s look at a simple example. The kernel here replaces all negative elements in a given array with the value 42. Since, in general, not all threads will be writing to their corresponding elements in the array, we are indeed dealing with a divergent branch.



Example 1: Predication

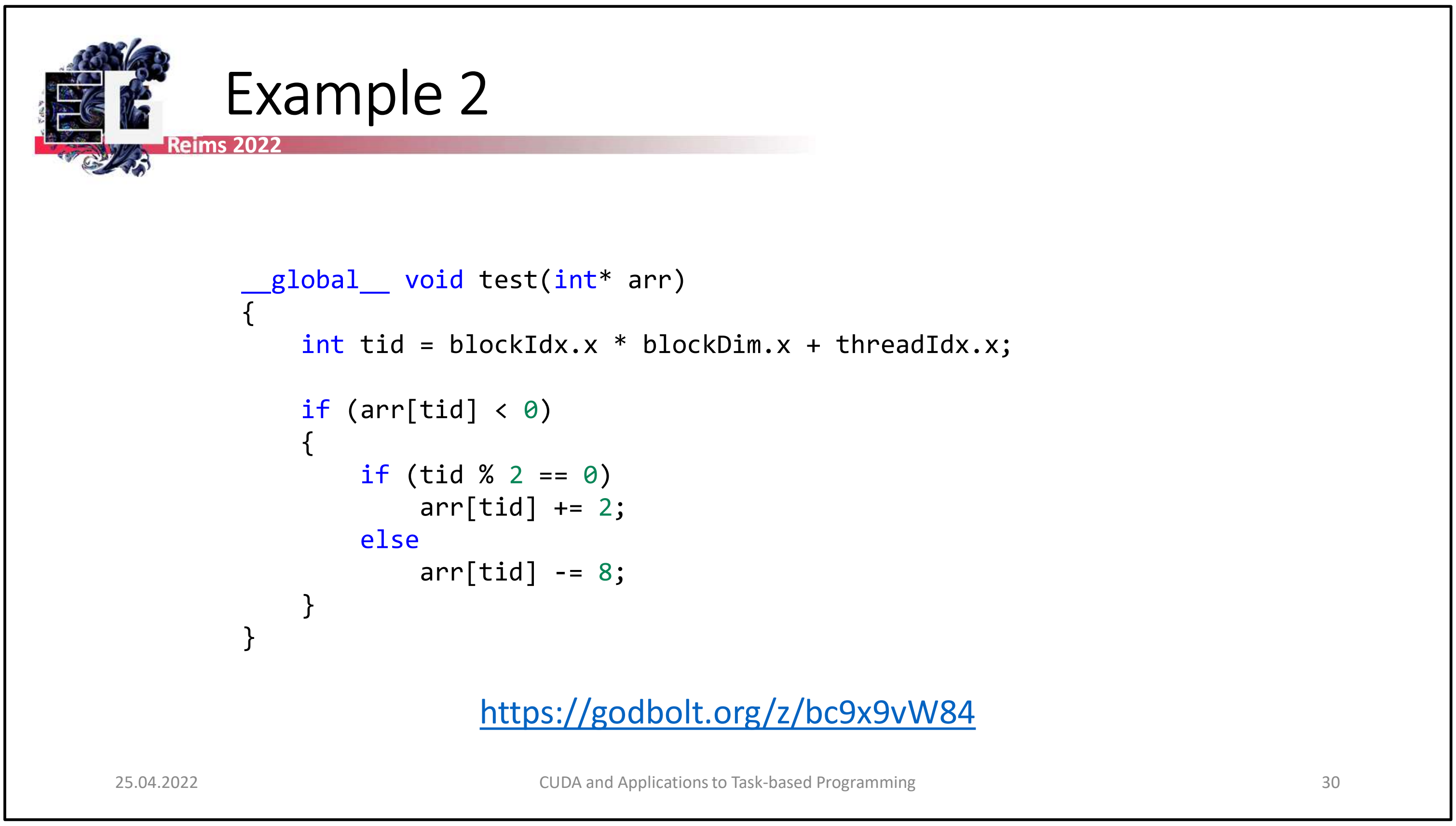
Reims 2022

```
test(int *):
 MOV R1, c[0x0][0x28]
 S2R R2, SR_CTAID.X
 MOV R5, 0x4
 S2R R3, SR_TID.X
 IMAD R2, R2, c[0x0][0x0], R3
 IMAD.WIDE R2, R2, R5, c[0x0][0x160]
 LDG.E.SYS R0, [R2]
 ISETP.GT.AND P0, PT, R0, -0x1, PT
 @P0 EXIT
 MOV R5, 0x2a
 STG.E.SYS [R2], R5
 EXIT
.L_x_0:
 BRA `(.L_x_0)
 NOP
 NOP
 NOP
.L_x_1:
```

```
__global__ void test(int* arr)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (arr[tid] < 0)
        arr[tid] = 42;
}
```

25.04.2022 CUDA and Applications to Task-based Programming 29

Looking at the assembly for the generated machine code, we note the EXIT instruction prefixed with predicate P0. Threads for which the predicate register P0 contains a 1 will stop executing when this instruction is reached. All other threads will go on, move the value 42 into register R5, and store it from there to the global memory location at the address previously computed in R2. The value of the predicate register is set up by the `ISETP` instruction, which sets a given predicate register based on the result of an integer comparison, in this case, a greater-than (`.GT`) comparison against the value `-1`.



Example 2

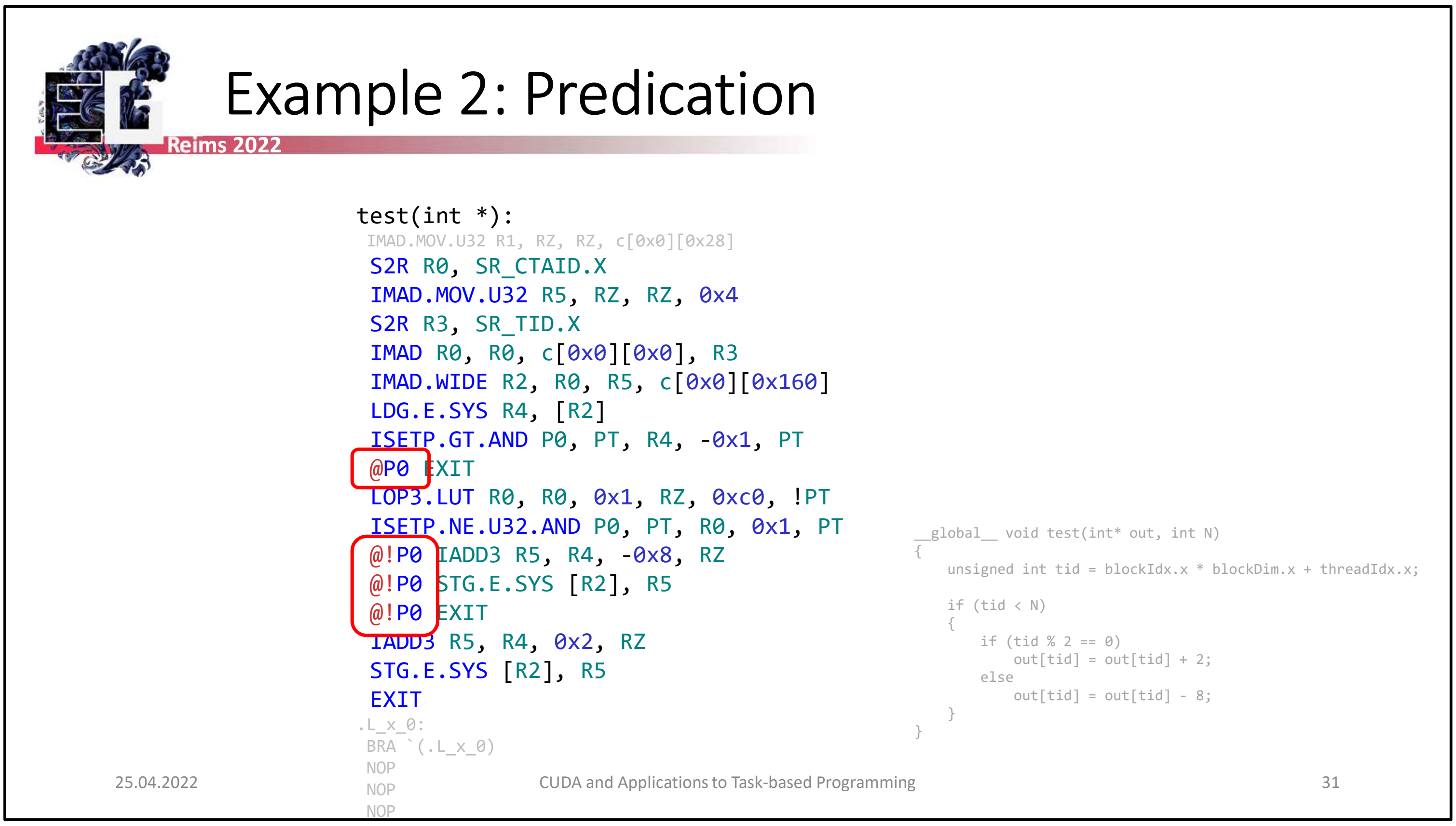
Reims 2022

```
__global__ void test(int* arr)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (arr[tid] < 0)
    {
        if (tid % 2 == 0)
            arr[tid] += 2;
        else
            arr[tid] -= 8;
    }
}
```

https://godbolt.org/z/bc9x9vW84

25.04.2022 CUDA and Applications to Task-based Programming 30

This kernel presents a slightly more complex example. We are either adding 2 or subtracting 8 from all negative elements in an array depending on whether the element appears at an even or odd index within the array.



Example 2: Predication

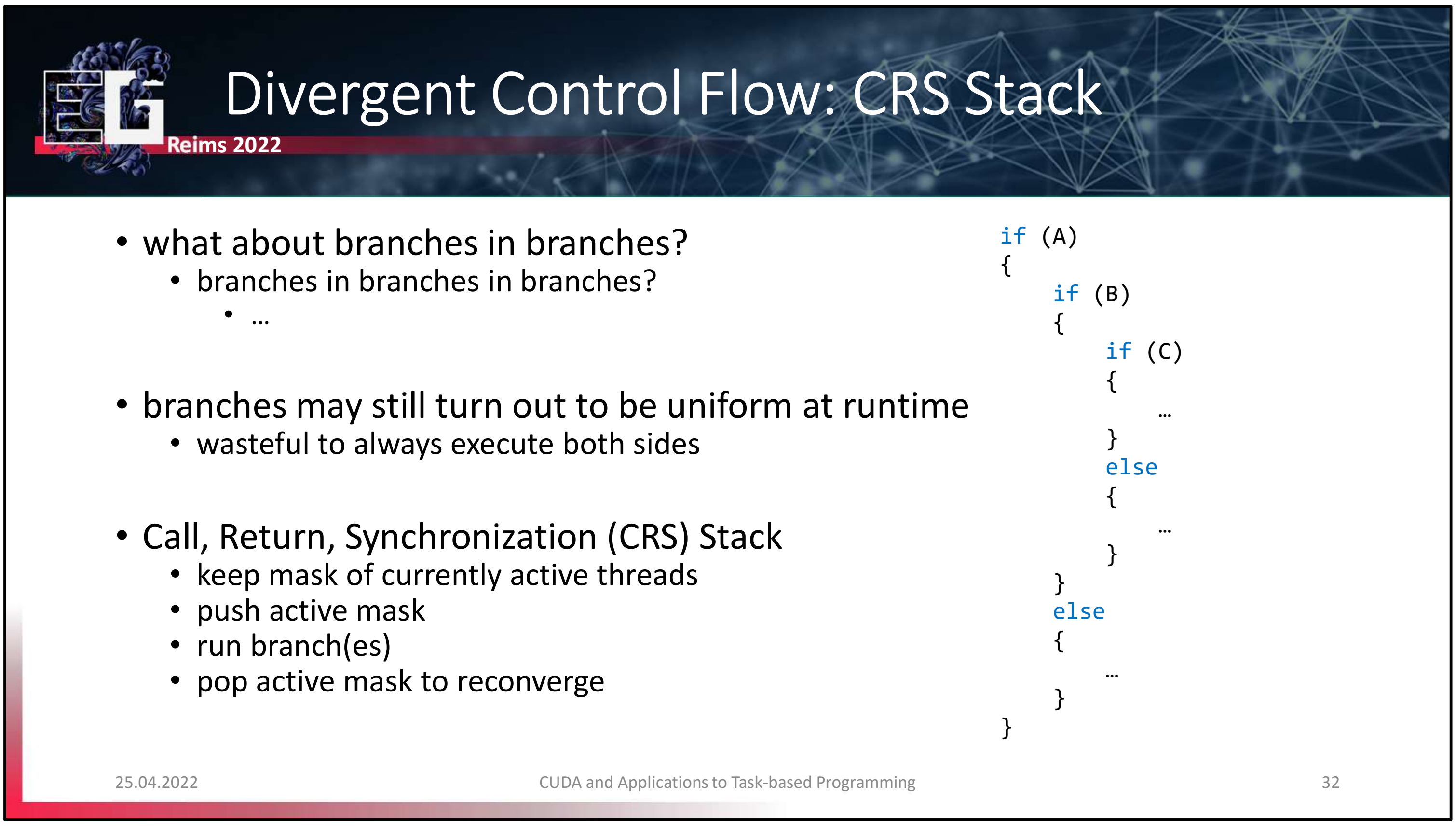
```
test(int *):
 IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28]
 S2R R0, SR_CTAID.X
 IMAD.MOV.U32 R5, RZ, RZ, 0x4
 S2R R3, SR_TID.X
 IMAD R0, R0, c[0x0][0x0], R3
 IMAD.WIDE R2, R0, R5, c[0x0][0x160]
 LDG.E.SYS R4, [R2]
 ISETP.GT.AND P0, PT, R4, -0x1, PT
 @P0 EXIT
 LOP3.LUT R0, R0, 0x1, RZ, 0xc0, !PT
 ISETP.NE.U32.AND P0, PT, R0, 0x1, PT
 @!P0 IADD3 R5, R4, -0x8, RZ
 @!P0 STG.E.SYS [R2], R5
 @!P0 EXIT
 IADD3 R5, R4, 0x2, RZ
 STG.E.SYS [R2], R5
 EXIT
.L_x_0:
 BRA `(.L_x_0)
 NOP
 NOP
 NOP
```

```
__global__ void test(int* out, int N)
{
    unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < N)
    {
        if (tid % 2 == 0)
            out[tid] = out[tid] + 2;
        else
            out[tid] = out[tid] - 8;
    }
}
```

25.04.2022 CUDA and Applications to Task-based Programming 31

Just like in the previous example, the generated machine code starts off by computing the address of the array element to operate on, loads its value, and sets the predicate register P0 depending on whether the value is larger than -1. If the array element was non-negative, we exit. Otherwise, we go on, compute the bitwise logical AND of the array index in R0 and the number 1 via the `LOP3` instruction (this effectively computes the index modulo 2), and set (`.ISETP`) predicate register P0 based on whether the result is not equal (`.NE`) to 1.

We then add -8 to our element value, store the result back to memory, and exit, all predicated on the logical complement of the value in the predicate register P0. Next, we add 2 to our element value, store the result back to memory, and exit. This effectively implements the serialized execution of the two halves of the branch associated with the inner if in the original C++ code. We first run code that implements the case of the odd index and exit. The only threads that have not yet exited after this part of the sequence are the threads with negative elements at even indices. The instructions implementing this last part of our two nested branches can therefore simply be run without requiring a predicate.



Divergent Control Flow: CRS Stack

Reims 2022

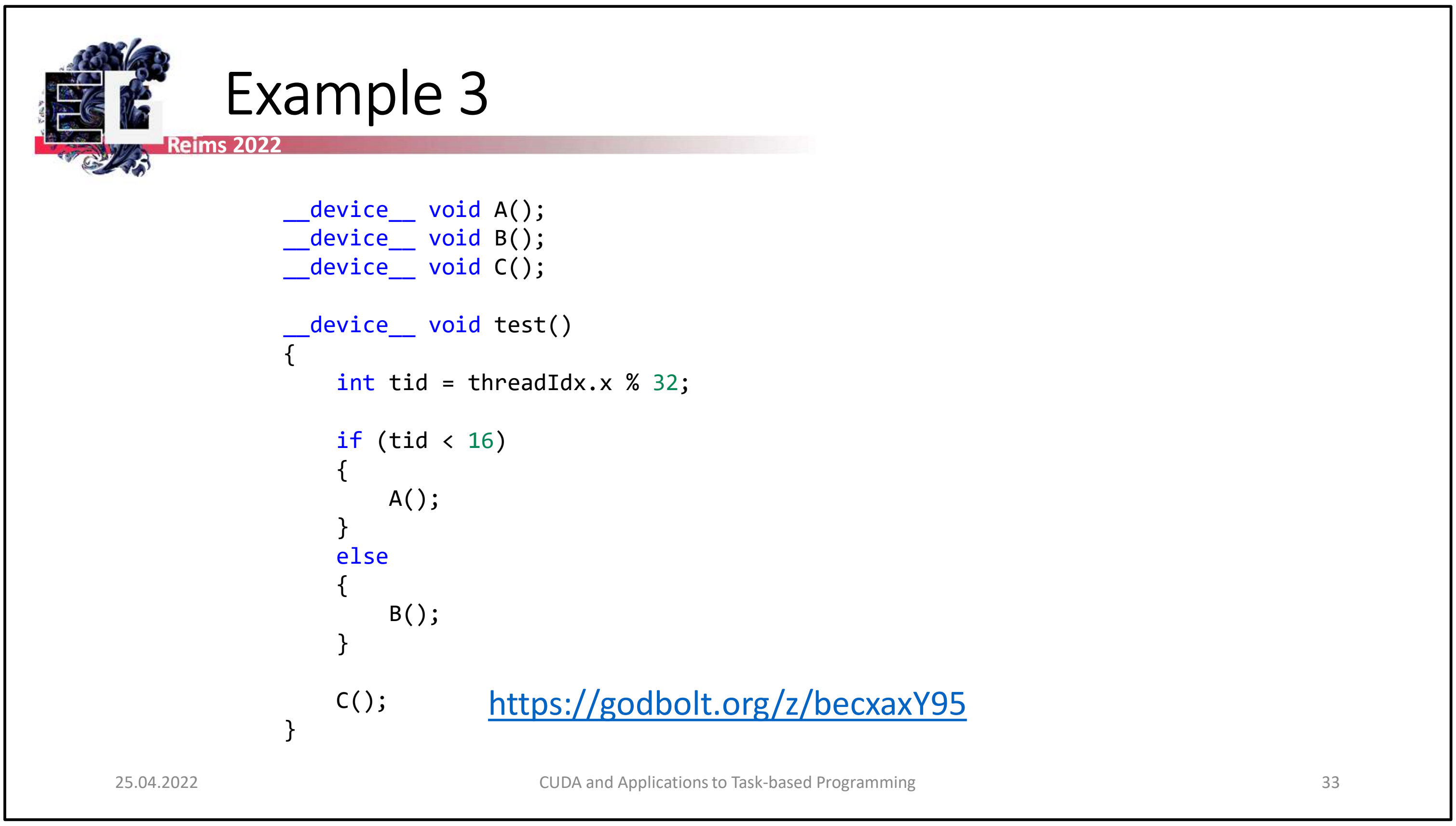
- what about branches in branches?
  - branches in branches in branches?
    - …
- branches may still turn out to be uniform at runtime
  - wasteful to always execute both sides
- Call, Return, Synchronization (CRS) Stack
  - keep mask of currently active threads
  - push active mask
  - run branch(es)
  - pop active mask to reconverge

```
if (A)
{
    if (B)
    {
        if (C)
        {
            …
        }
        else
        {
            …
        }
    }
    else
    {
        …
    }
}
```

25.04.2022 CUDA and Applications to Task-based Programming 32

With divergent branches implemented based on predication, the program simply executes as one linear stream of instructions for the entire warp. However, control flow is forced to pass over all instructions on both sides of every branch. Many branches, while potentially divergent, will often actually be taken in unison by all threads in a warp. Having to still pass over all instructions on the other side of a branch even if no thread in the warp has actually taken this path is wasteful and can introduce significant overhead especially as branches are nested more and more deeply.

In order to solve this problem, the hardware keeps track of a bitmask of the currently active threads within a warp. Like with predication, instructions only take effect for the threads that are marked as active in the current active mask. A Call, Return, Synchronization (CRS) Stack is used to save and restore the set of active threads before and after processing a branch. This allows for dynamic handling of nested branches and also doubles as a call stack for function calls.



Example 3

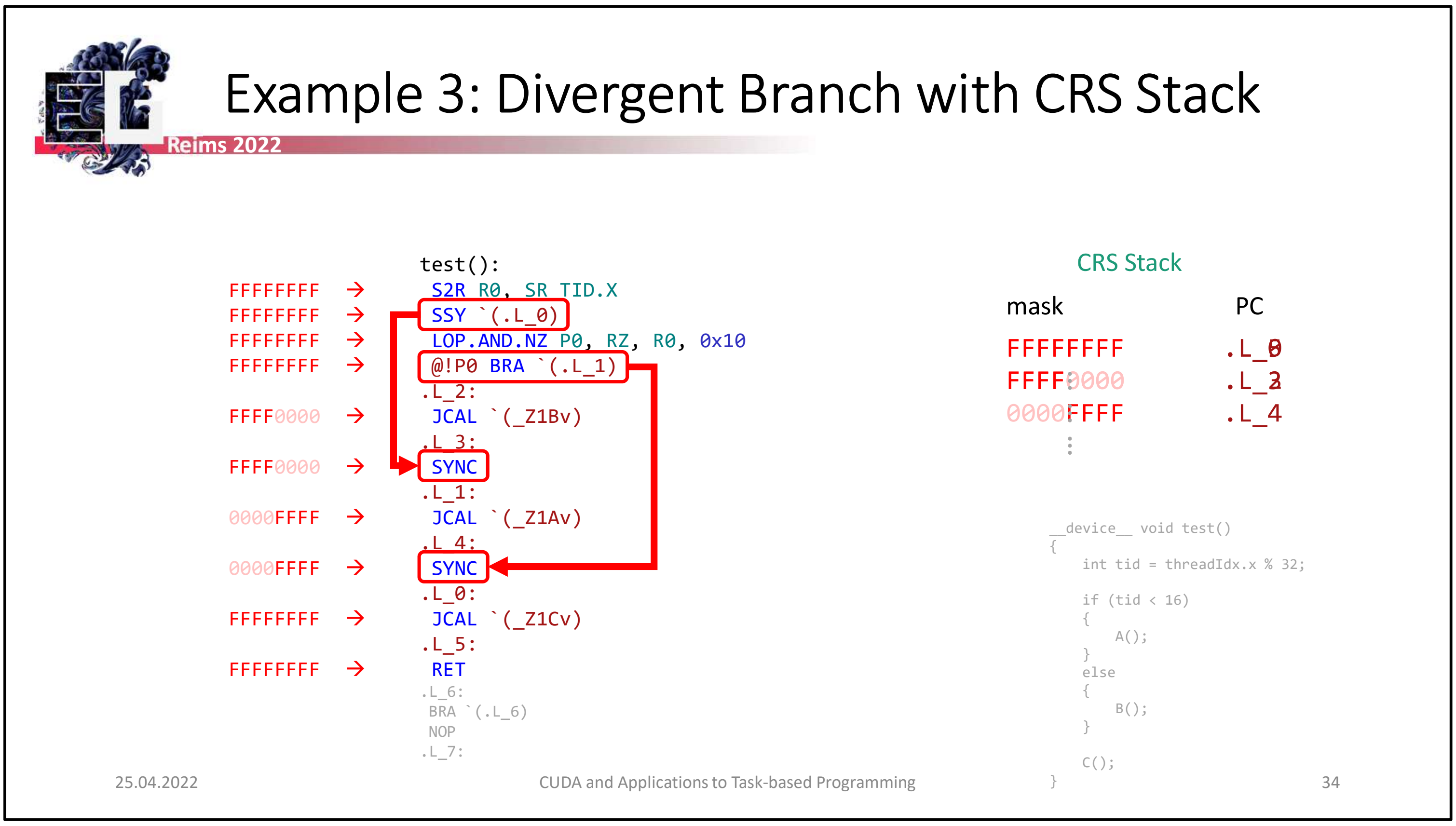
Reims 2022

```
__device__ void A();
__device__ void B();
__device__ void C();

__device__ void test()
{
    int tid = threadIdx.x % 32;

    if (tid < 16)
    {
        A();
    }
    else
    {
        B();
    }

    C();
}
```

https://godbolt.org/z/becxaxY95

25.04.2022 CUDA and Applications to Task-based Programming 33

To illustrate how more complex control flow can be implemented efficiently via a CRS stack, we will be using this example of a function that contains a divergent branch which itself contains calls to other functions. The lower half of the threads calls function A, the upper half calls function B, and, finally, all threads call function C.



Example 3: Divergent Branch with CRS Stack

```
                      test():
FFFFFFFF  →            S2R R0, SR_TID.X
FFFFFFFF  →            SSY `(.L_0)
FFFFFFFF  →            LOP.AND.NZ P0, RZ, R0, 0x10
FFFFFFFF  →            @!P0 BRA `(.L_1)
                      .L_2:
FFFF0000  →            JCAL `(_Z1Bv)
                      .L_3:
FFFF0000  →            SYNC
                      .L_1:
0000FFFF  →            JCAL `(_Z1Av)
                      .L_4:
0000FFFF  →            SYNC
                      .L_0:
FFFFFFFF  →            JCAL `(_Z1Cv)
                      .L_5:
FFFFFFFF  →            RET
                      .L_6:
                       BRA `(.L_6)
                       NOP
                      .L_7:
```

CRS Stack

mask	PC
FFFFFFFF	.L_0
FFFF0000	.L_2
0000FFFF	.L_4
⋮	

```
__device__ void test()
{
    int tid = threadIdx.x % 32;

    if (tid < 16)
    {
        A();
    }
    else
    {
        B();
    }

    C();
}
```

25.04.2022   CUDA and Applications to Task-based Programming   34

When called, the generated machine code for this function proceeds as follows: All threads load their thread index into R0. The next instruction is a set synchronization (SSY) instruction. This instruction pushes the current active mask as well as the instruction pointer for where an upcoming branch will reconverge onto the CRS stack. The code then sets predicate register P0 based on which half of the warp the thread is in (false for threads 0..15, true for threads 16..31). The following BRA instruction then branches to .L_1 predicated on P0. The branch instruction pushes the mask of the threads that did *not* take the branch as well as the instruction pointer for where these threads would continue onto the CRS stack. Only the lower half of the threads executes the branch instruction. The active mask is updated accordingly and control is transferred to .L_1.

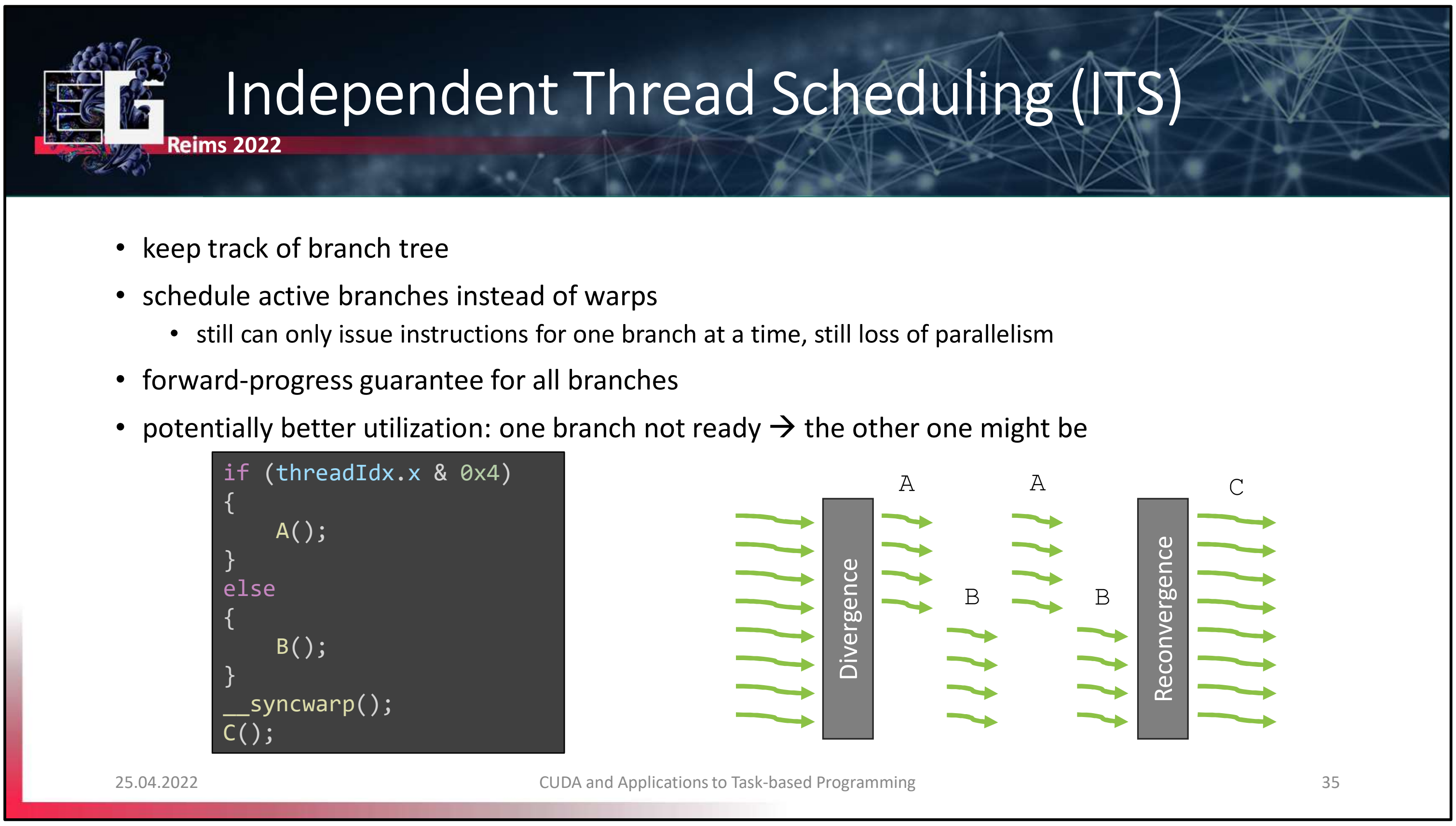
The next instruction at that point in the program is the call to function A. The current active mask as well as the address of the instruction with which to continue after the call are pushed onto the CRS stack. Control is transferred into the function. Eventually, the function will return, popping the corresponding entry off the CRS stack. The active mask will be restored to what it was before the call and control transferred to the return address saved on the stack, i.e., the next instruction after the call.

After calling function A, we execute a `SYNC` instruction. The purpose of a `SYNC` instruction is to reconverge the warp. In order to do so, it pops the topmost entry off the CRS stack, restoring the active mask to the mask previously saved on the stack and transferring control to the associated instruction pointer. In this case, the entry popped off the stack is the one put there by the initial branch instruction. The active mask is updated to the mask corresponding to the threads that did not take the branch, and control is transferred to where these threads would have continued. Thus, this `SYNC` instruction effectively now switches to executing the other half of the warp that did not take the branch.

The reactivated threads of the other branch now execute their next instruction, which is the call to function B. As before, the call will save the active mask and return address onto the CRS stack before transferring control into function B. Once the function call returns, the active mask will be restored and the next instruction after the call executed, which is another `SYNC` instruction. Since the topmost element on the CRS stack is now the one pushed by the `SSY` instruction at the very beginning, this `SYNC` now has the effect of reconverging the warp by restoring the active mask to full and transferring control to the convergence point just after the code for both branches.

Once reconverged, all threads call function C and then finally return, restoring the active mask and transferring control to where the function itself was originally called from.

Note: Older architectures used a pop sync flag (.`S`) that could be placed on instructions (e.g.: `NOP.S`, `MUL`.S, etc.) instead of an explicit `SYNC` instruction.

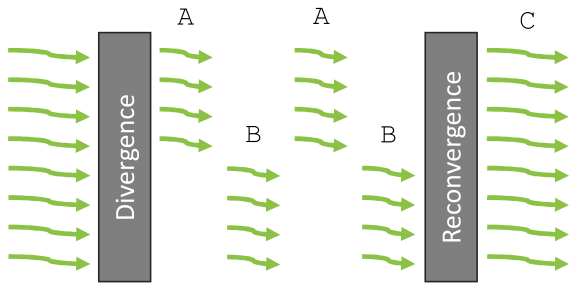


Independent Thread Scheduling (ITS)

Reims 2022

- keep track of branch tree
- schedule active branches instead of warps
  - still can only issue instructions for one branch at a time, still loss of parallelism
- forward-progress guarantee for all branches
- potentially better utilization: one branch not ready → the other one might be

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



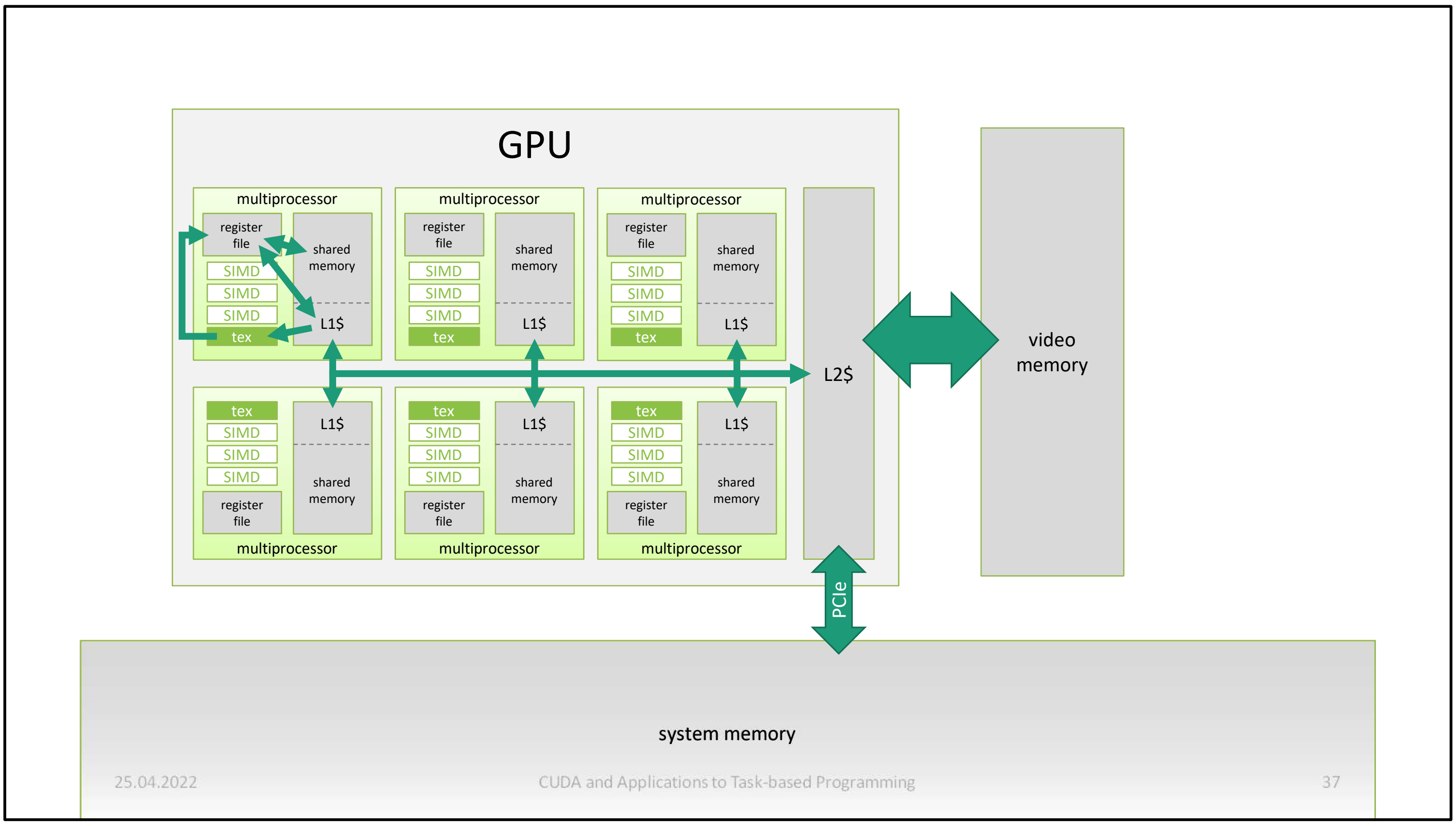
25.04.2022 CUDA and Applications to Task-based Programming 35

With a CRS stack, information about active branches is implicit/hidden in the structure on the stack. Active branches could appear arbitrarily far from the current top of the stack. Even if we could discover other branches on the stack, we could not continue running another branch past a point where it itself branches any further before the current stack has been dealt with. Thus, in an approach using a CRS stack, we generally have to run one half of a branch to completion before we can switch to the other half.

As a result, classic warp scheduling based on a CRS stack imposes restrictions such as that threads in two separate parts of a branch in the same warp cannot synchronize with one another. Many concurrent algorithms cannot be used, the SIMT programming model becomes a leaky abstraction. Ideally, all threads would be able to make progress independently from other threads, divergence would only be a matter of performance but not correctness. Recent CUDA GPUs with Independent Thread Scheduling (ITS) explicitly track the entire branch tree for a warp. Instead of just scheduling warps as a whole, the warp scheduler can switch between active branches of warps. While execution of divergent branches still has to be serialized, it can be interleaved. This not only removes the restrictions concerning intra-warp synchronization by providing a forward-progress guarantee for each thread but also unlocks additional opportunities for latency hiding (larger set of potential work to choose from).

The Memory Hierarchy

Memory tends to be *the* bottleneck in any modern system. And GPUs are no exception. In fact, given the massive amount of parallel computation that is typically taking place on a GPU at any given moment, the effect of memory on performance is arguably even more pronounced on the GPU than we might be used to from working on the CPU. Thus, in many ways, GPU programming really is all about making the most of the GPU's memory subsystem.

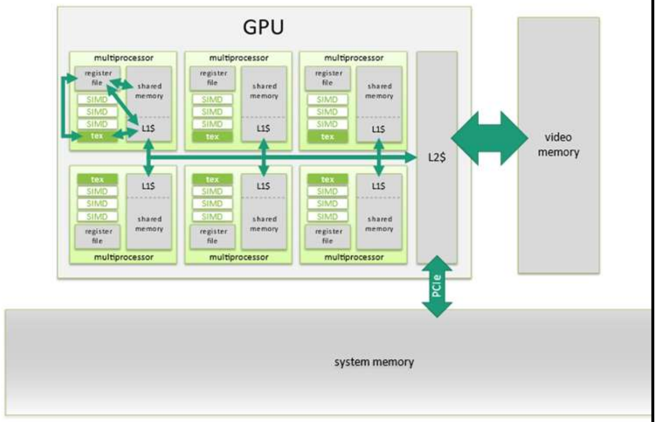


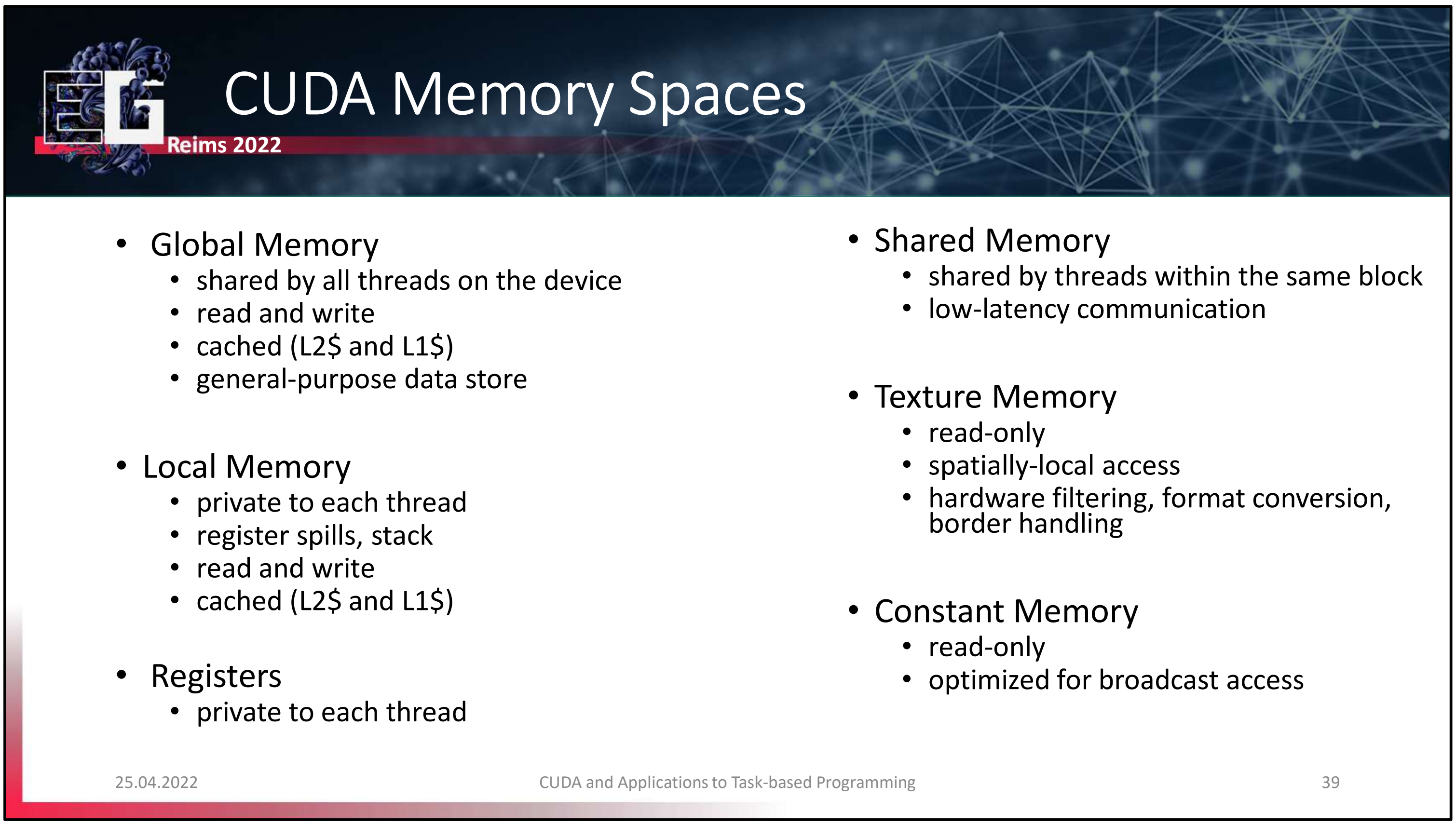
As we've already seen, the GPU consists of a number of multiprocessors. On each multiprocessor, we find SIMD cores as well as local shared memory and texture units. The GPU is connected to video memory as well as system memory via PCIe. A level-2 cache (L2$) is used between the chip and external memory. A portion of the local shared memory on each multiprocessor is used as level-1 data cache (L1$). Data can be fetched into registers to feed the cores via L1$ as well as from local shared memory and from texture units. Data can be written out from registers either through L1$ or to shared memory.

GPU Memory Hierarchy

- multiple paths to fetch from memory
- different trade-offs
- special-purpose hardware

- exposed in CUDA in the form of memory spaces




CUDA Memory Spaces

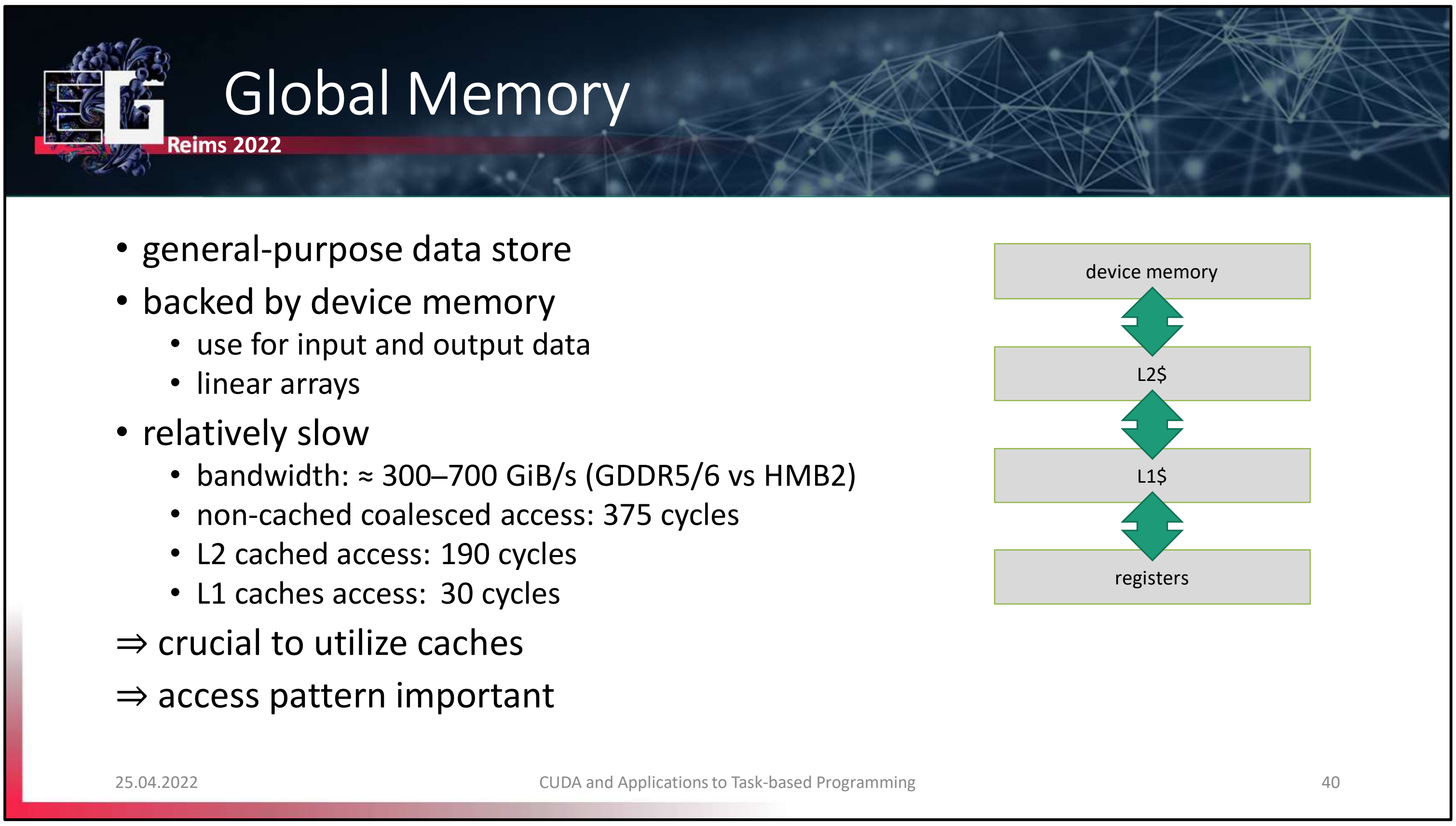
Reims 2022

- Global Memory
  - shared by all threads on the device
  - read and write
  - cached (L2$ and L1$)
  - general-purpose data store
- Local Memory
  - private to each thread
  - register spills, stack
  - read and write
  - cached (L2$ and L1$)
- Registers
  - private to each thread
- Shared Memory
  - shared by threads within the same block
  - low-latency communication
- Texture Memory
  - read-only
  - spatially-local access
  - hardware filtering, format conversion, border handling
- Constant Memory
  - read-only
  - optimized for broadcast access

25.04.2022 CUDA and Applications to Task-based Programming 39

In CUDA, these hardware resources are exposed in the form of a number of memory spaces, each with different properties designed for different kinds of access patterns. CUDA applications take advantage of the various capabilities found within the GPU's memory system by placing and accessing data in the corresponding CUDA memory space.

We will now have a more detailed look at each one of these memory spaces and what they have to offer.



Global Memory

Reims 2022

- general-purpose data store
- backed by device memory
  - use for input and output data
  - linear arrays
- relatively slow
  - bandwidth: ≈ 300–700 GiB/s (GDDR5/6 vs HMB2)
  - non-cached coalesced access: 375 cycles
  - L2 cached access: 190 cycles
  - L1 caches access: 30 cycles

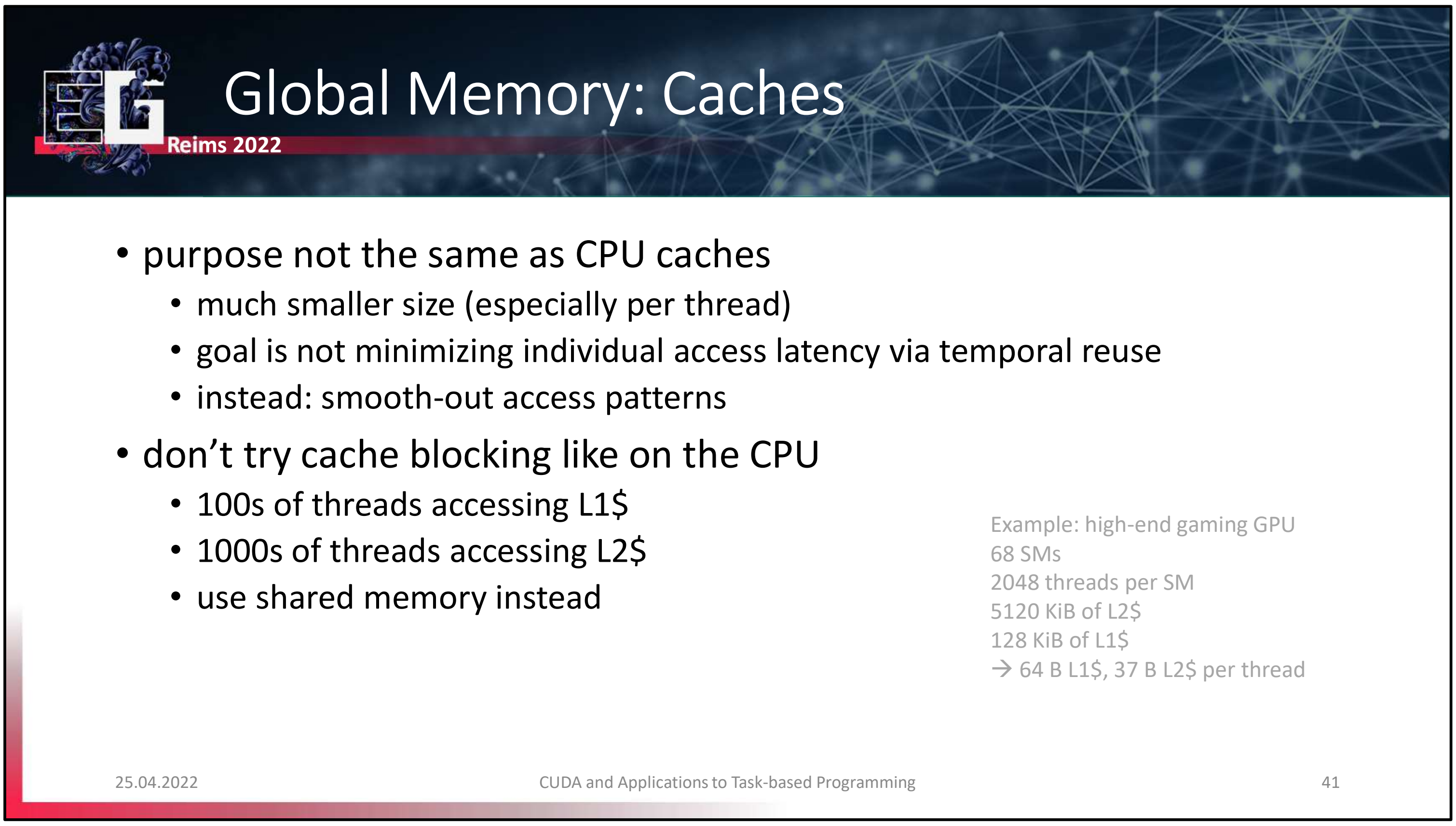
⇒ crucial to utilize caches

⇒ access pattern important

device memory
L2$
L1$
registers

25.04.2022 CUDA and Applications to Task-based Programming 40

The most important memory space is global memory. It corresponds to device memory and is accessible to all threads running on the GPU. As a result of this wide scope, memory accesses potentially have to bubble all the way up to, or all the way down from device memory. Making use of the available caches is, thus, vital for performance.



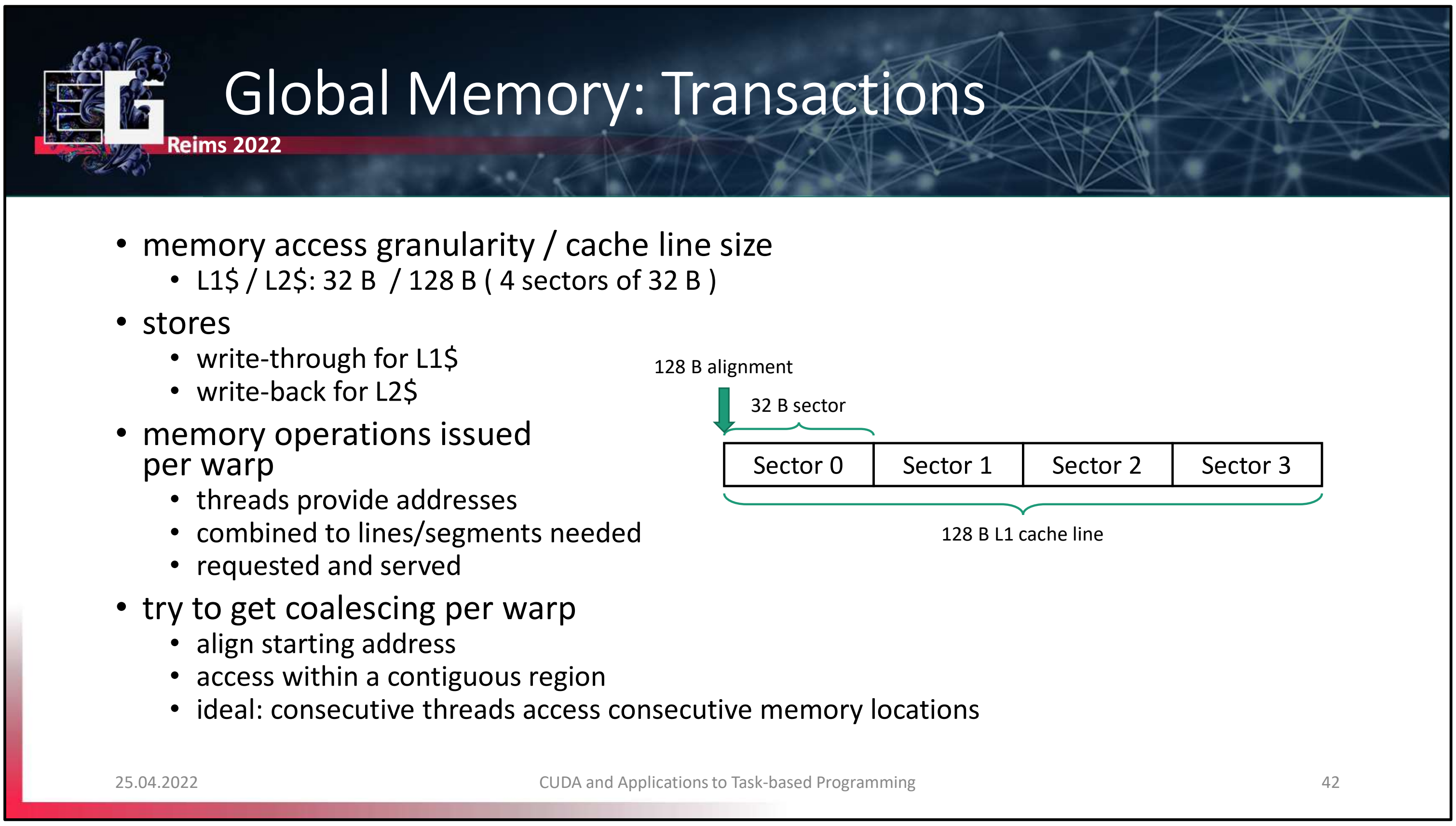
Global Memory: Caches

Reims 2022

- purpose not the same as CPU caches
  - much smaller size (especially per thread)
  - goal is not minimizing individual access latency via temporal reuse
  - instead: smooth-out access patterns
- don’t try cache blocking like on the CPU
  - 100s of threads accessing L1$
  - 1000s of threads accessing L2$
  - use shared memory instead

Example: high-end gaming GPU
68 SMs
2048 threads per SM
5120 KiB of L2$
128 KiB of L1$
→ 64 B L1$, 37 B L2$ per thread

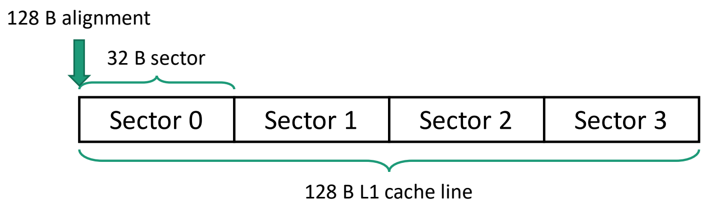
25.04.2022 CUDA and Applications to Task-based Programming 41



Global Memory: Transactions

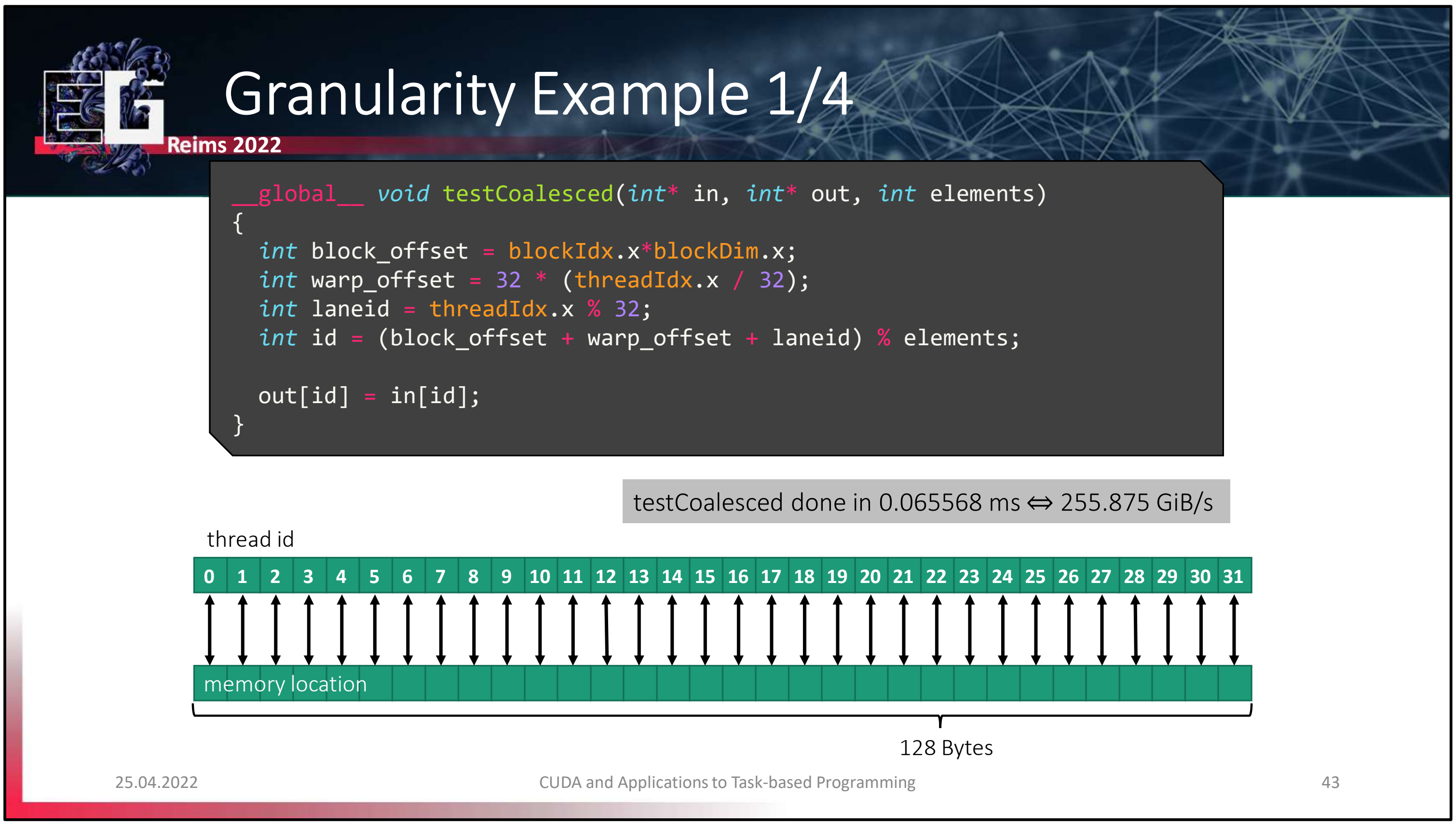
Reims 2022

- memory access granularity / cache line size
  - L1$ / L2$: 32 B / 128 B (4 sectors of 32 B)
- stores
  - write-through for L1$
  - write-back for L2$
- memory operations issued per warp
  - threads provide addresses
  - combined to lines/segments needed
  - requested and served
- try to get coalescing per warp
  - align starting address
  - access within a contiguous region
  - ideal: consecutive threads access consecutive memory locations



25.04.2022 CUDA and Applications to Task-based Programming 42

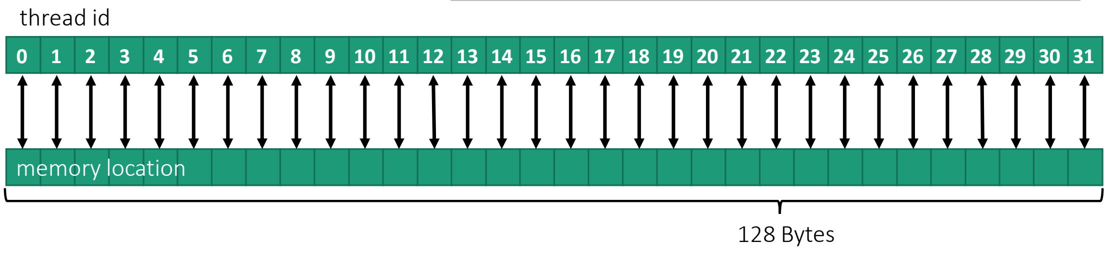
One cache line is 128 bytes, which is split into 4 sectors of Size 32 bytes. Memory Transactions are 32 byte long and only actually requested 32 byte sectors are read from memory.



Granularity Example 1/4

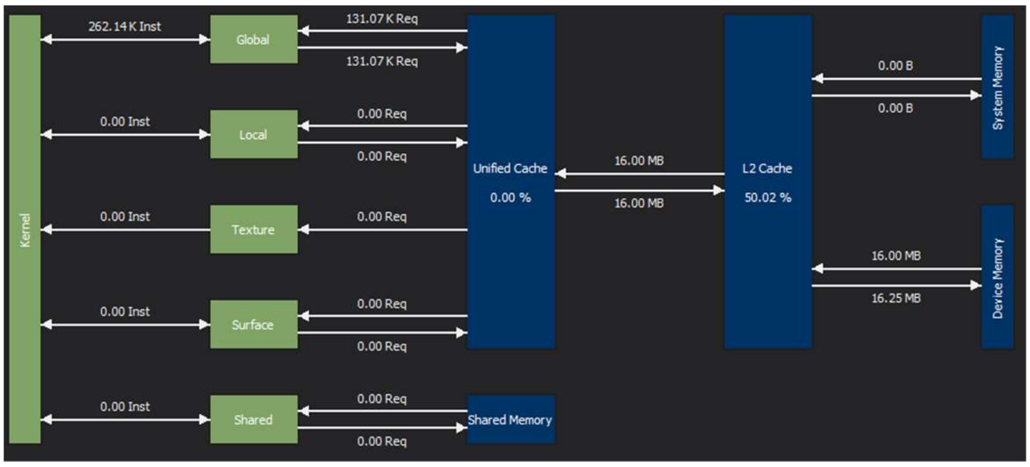
```
__global__ void testCoalesced(int* in, int* out, int elements)
{
  int block_offset = blockIdx.x*blockDim.x;
  int warp_offset = 32 * (threadIdx.x / 32);
  int laneid = threadIdx.x % 32;
  int id = (block_offset + warp_offset + laneid) % elements;

  out[id] = in[id];
}
```

```
testCoalesced done in 0.065568 ms ⇔ 255.875 GiB/s
```

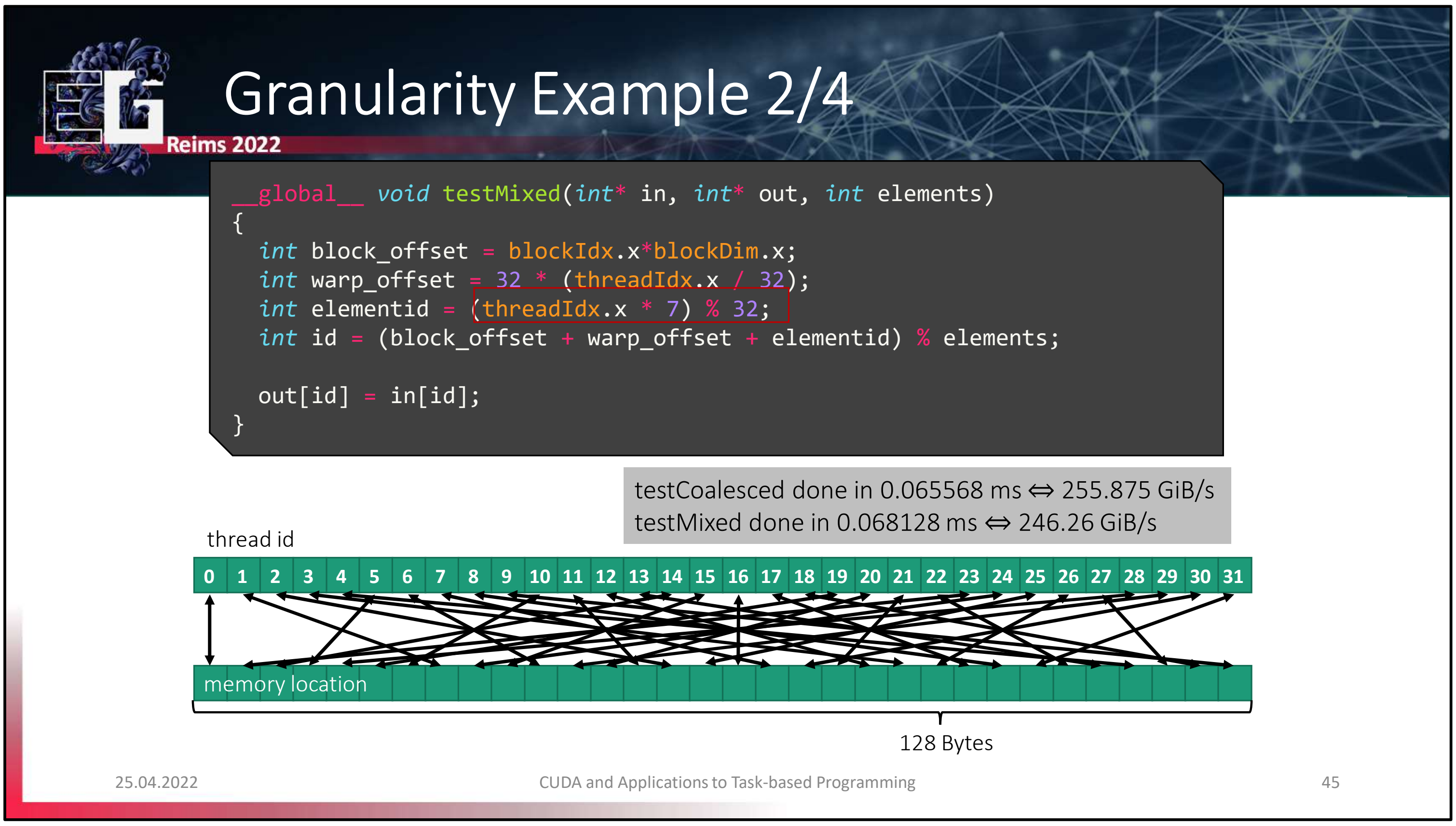


Using this example of a simple kernel that copies data from one array to another, we illustrate the importance of coalescing. In this first test, all threads of a warp access consecutive memory locations.

Granularity Example 1/4



Coalesced

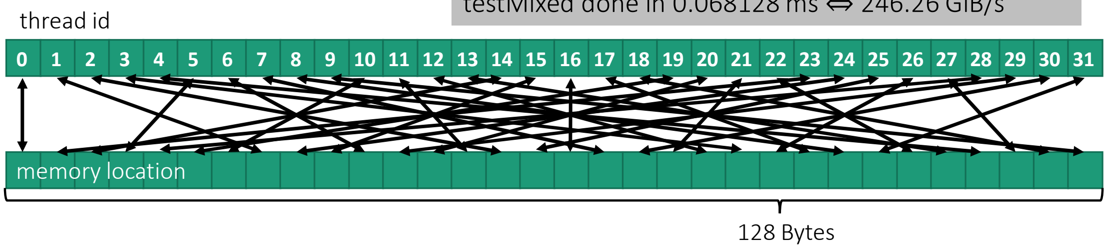


Granularity Example 2/4

Reims 2022

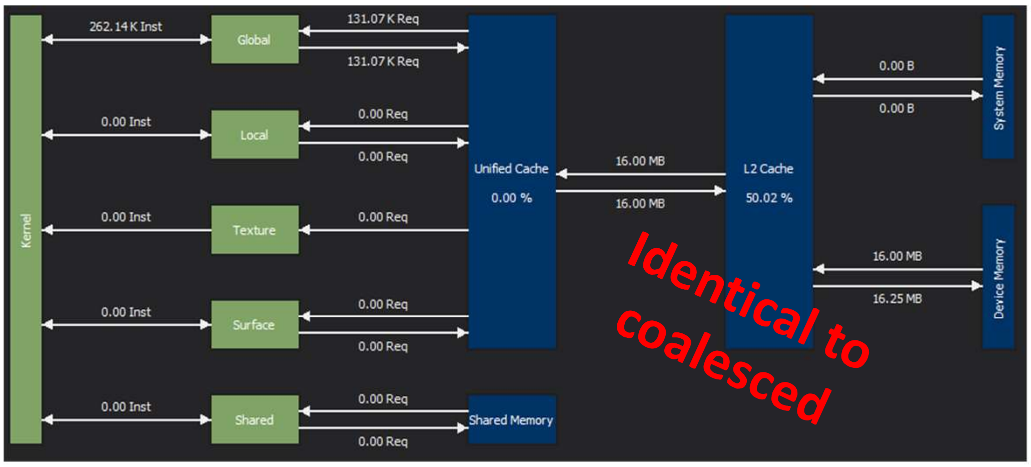
```
__global__ void testMixed(int* in, int* out, int elements)
{
  int block_offset = blockIdx.x*blockDim.x;
  int warp_offset = 32 * (threadIdx.x / 32);
  int elementid = (threadIdx.x * 7) % 32;
  int id = (block_offset + warp_offset + elementid) % elements;

  out[id] = in[id];
}
```

```
testCoalesced done in 0.065568 ms ⇔ 255.875 GiB/s
testMixed done in 0.068128 ms ⇔ 246.26 GiB/s
```

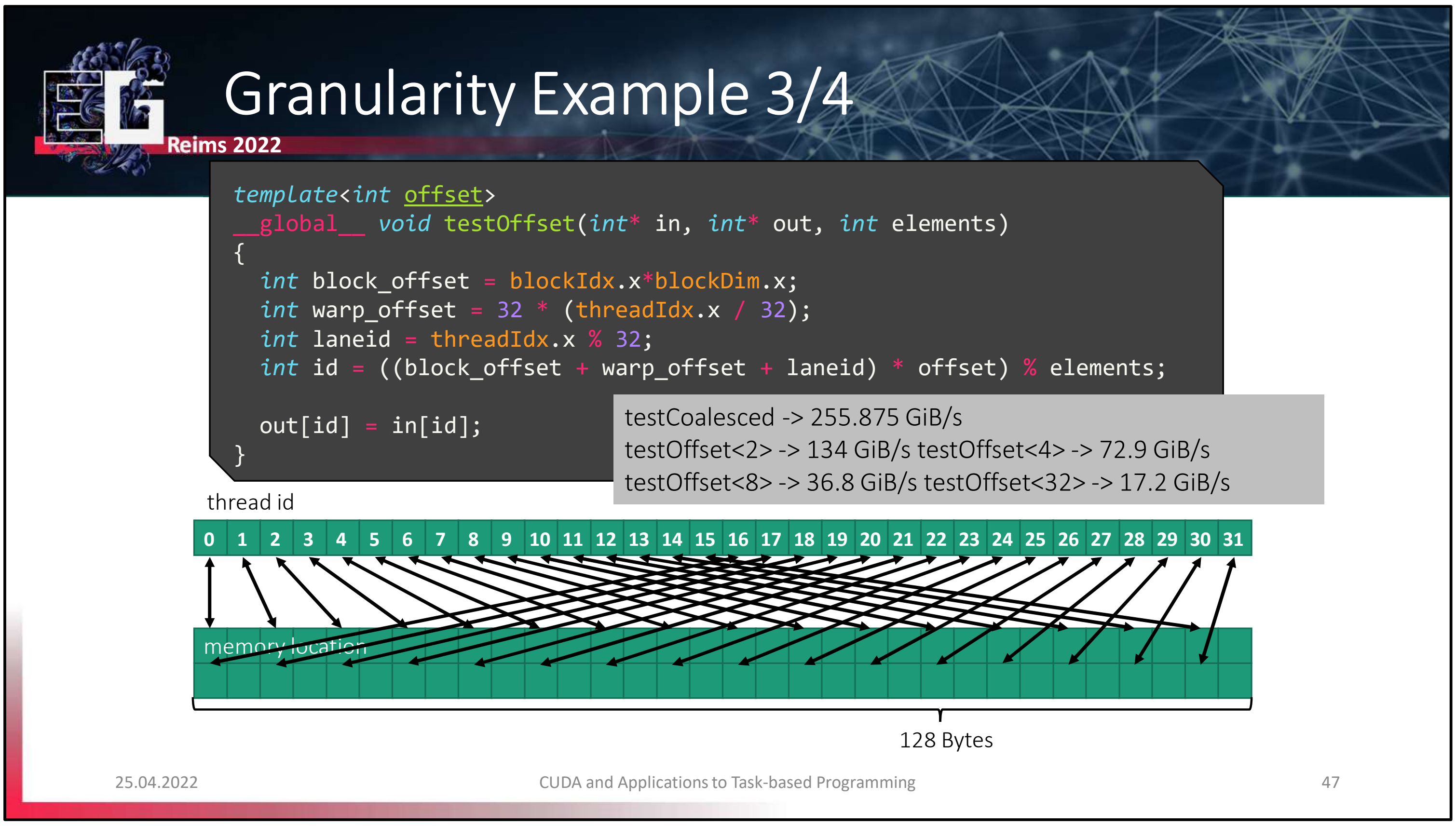


In this second test, consecutive threads within a warp don’t access consecutive memory locations anymore. However, memory accesses within the warp still stay within a 128 B cache line. We see that performance is comparable to the previous test.

Granularity Example 2/4



Mixed



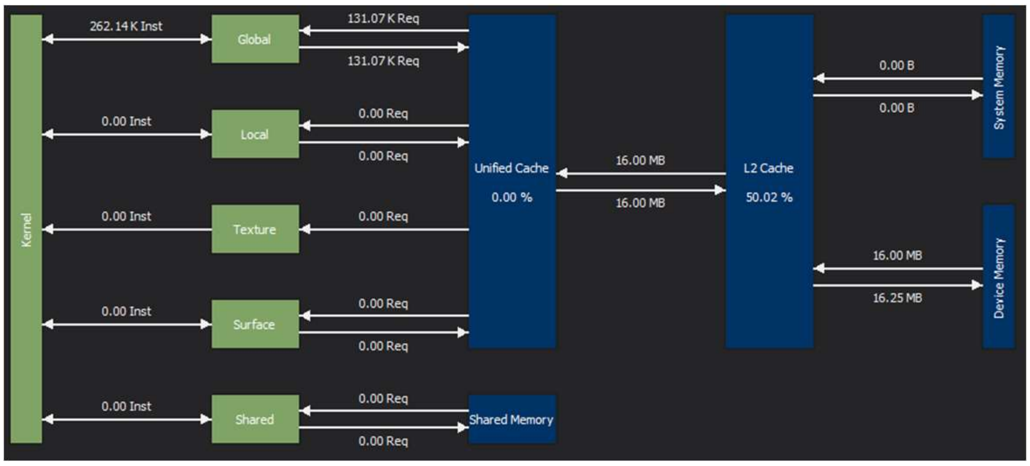
Granularity Example 3/4

```
template<int offset>
__global__ void testOffset(int* in, int* out, int elements)
{
  int block_offset = blockIdx.x*blockDim.x;
  int warp_offset = 32 * (threadIdx.x / 32);
  int laneid = threadIdx.x % 32;
  int id = ((block_offset + warp_offset + laneid) * offset) % elements;

  out[id] = in[id];
}
```

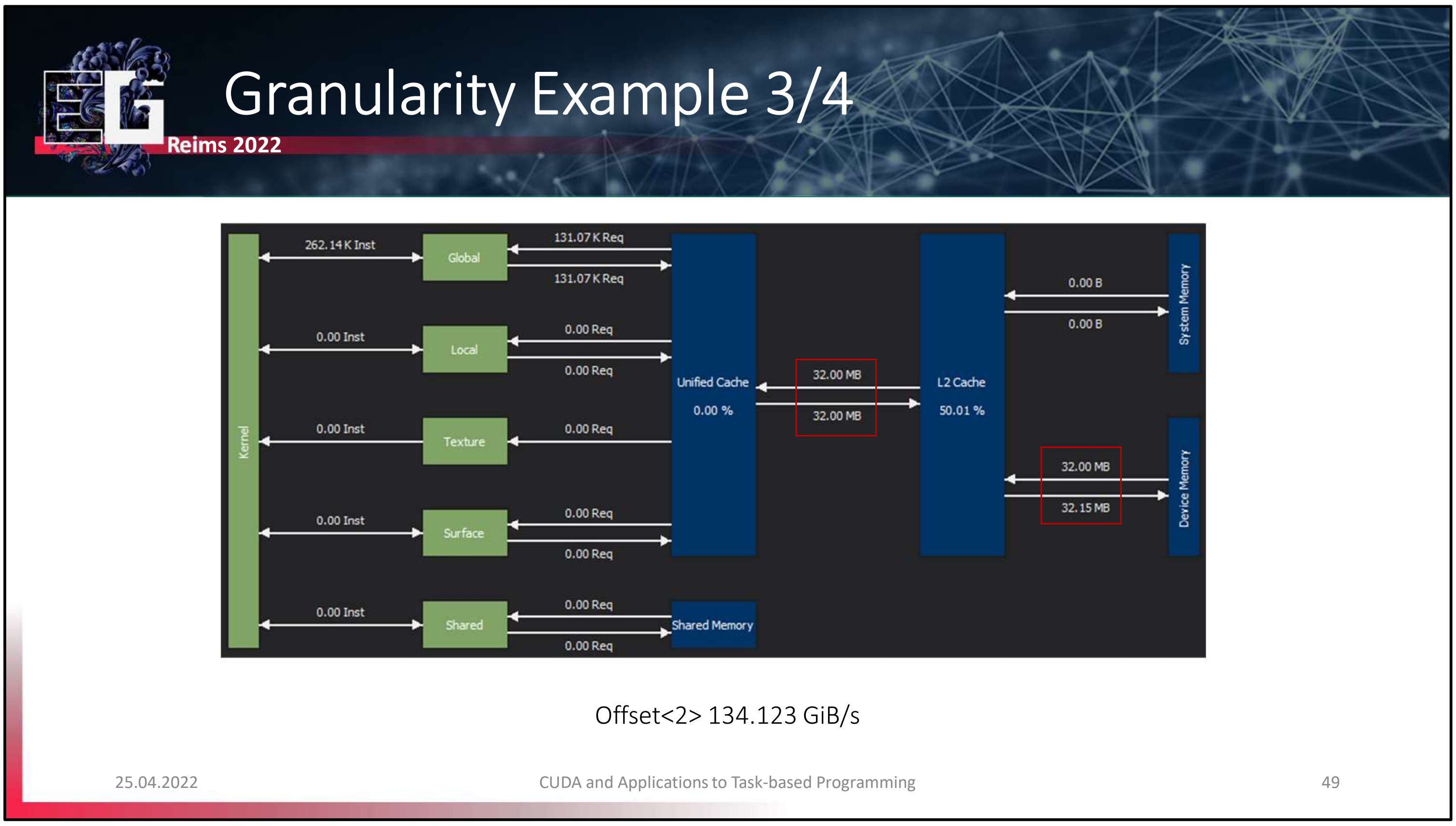
testCoalesced -> 255.875 GiB/s
testOffset<2> -> 134 GiB/s testOffset<4> -> 72.9 GiB/s
testOffset<8> -> 36.8 GiB/s testOffset<32> -> 17.2 GiB/s

By introducing a stride between the memory locations that consecutive threads access, we spread the memory access for each warp out to cover multiple cache lines. As we increase the stride, performance degrades roughly linearly with the number of cache lines each warp has to fetch.

Granularity Example 3/4

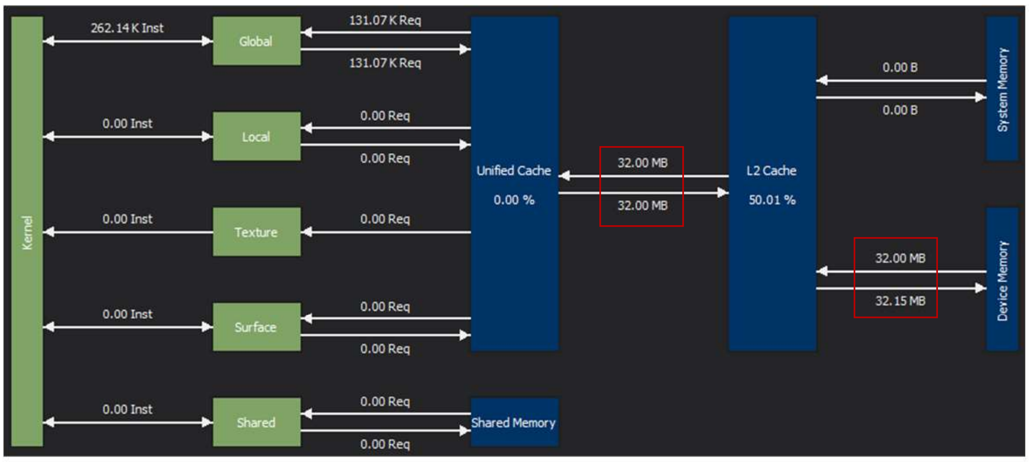


Coalesced 255.875 GB/s



Granularity Example 3/4

Reims 2022

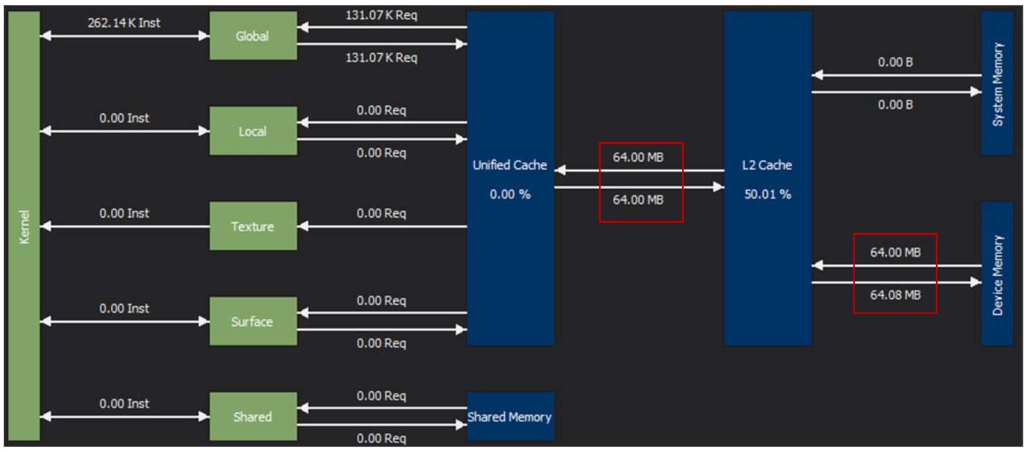


Offset<2> 134.123 GiB/s

Comparing to the baseline of perfectly coalesced access, we find that the amount of memory transfer between the SM and L2$ does in fact increase exactly linearly with the number of cache lines that need to be fetched. Despite each kernel accessing the same amount of data, due to the holes in the memory access pattern, the efficiency of the memory access degrades significantly.

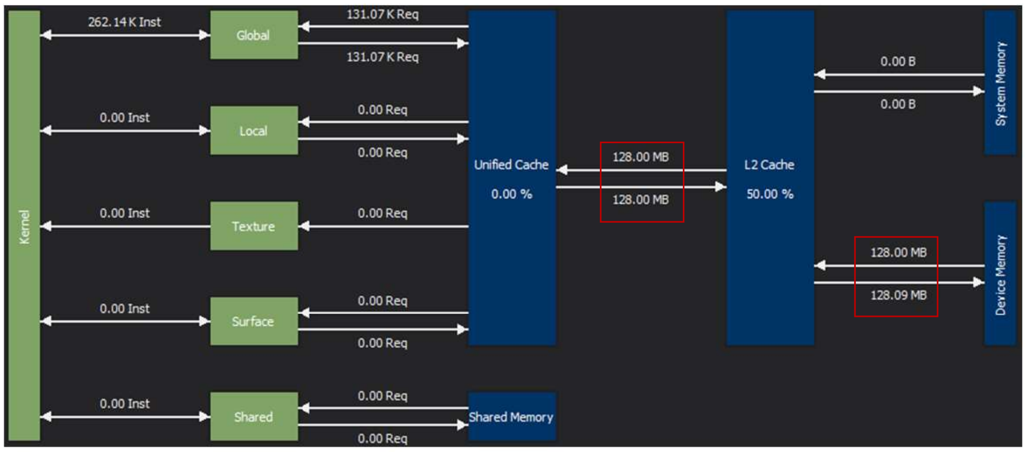
Granularity Example 3/4

Reims 2022



Offset<4> 72.9 GiB/s

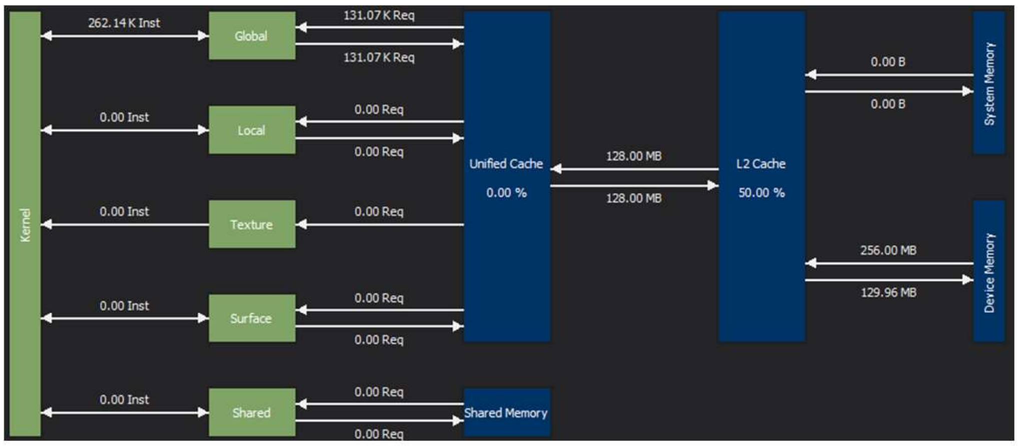
Granularity Example 3/4



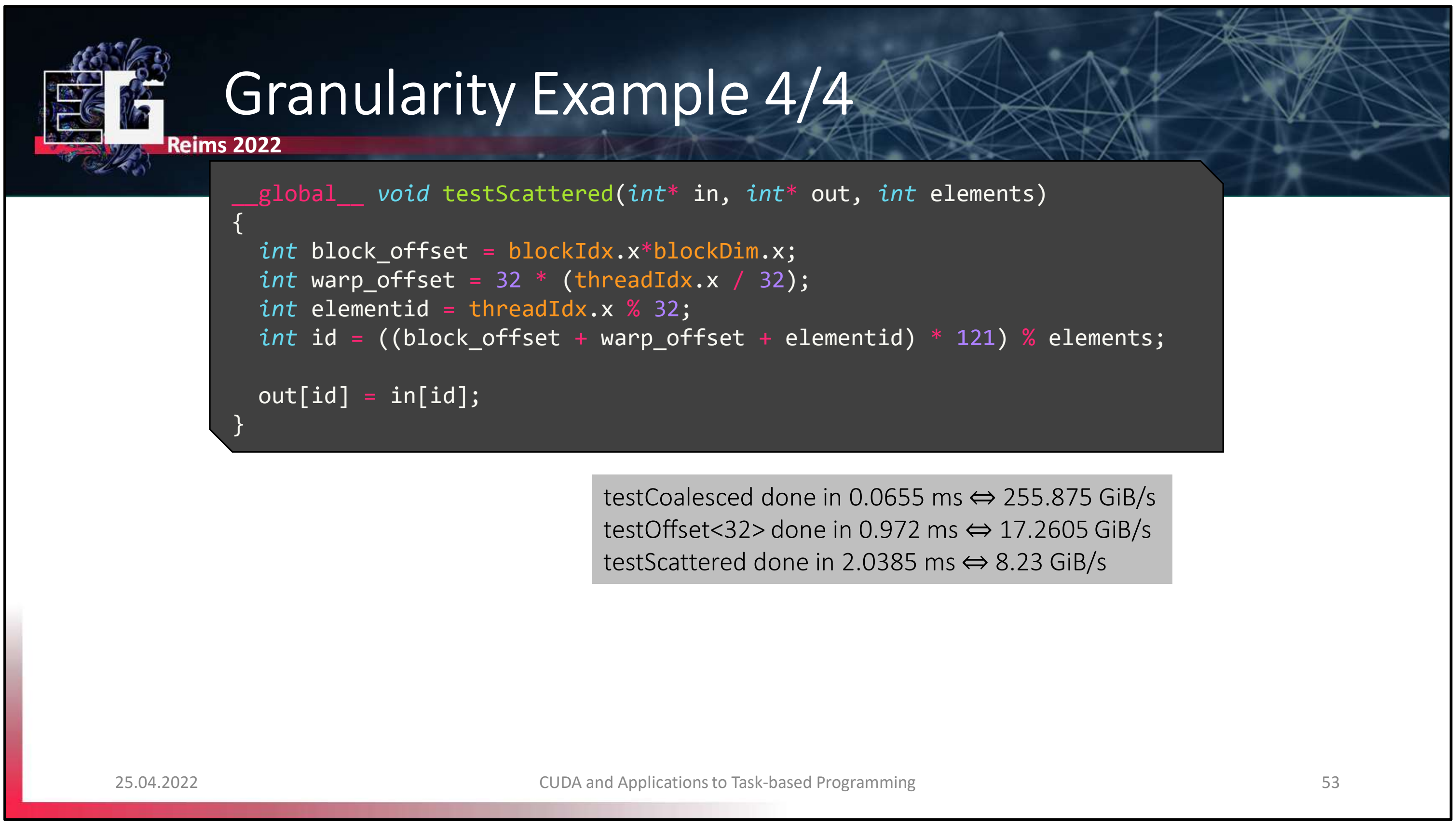
Offset<8> 36.81 GiB/s

Reims 2022

Granularity Example 3/4



Offset<32> 17.26 GiB/s



Granularity Example 4/4

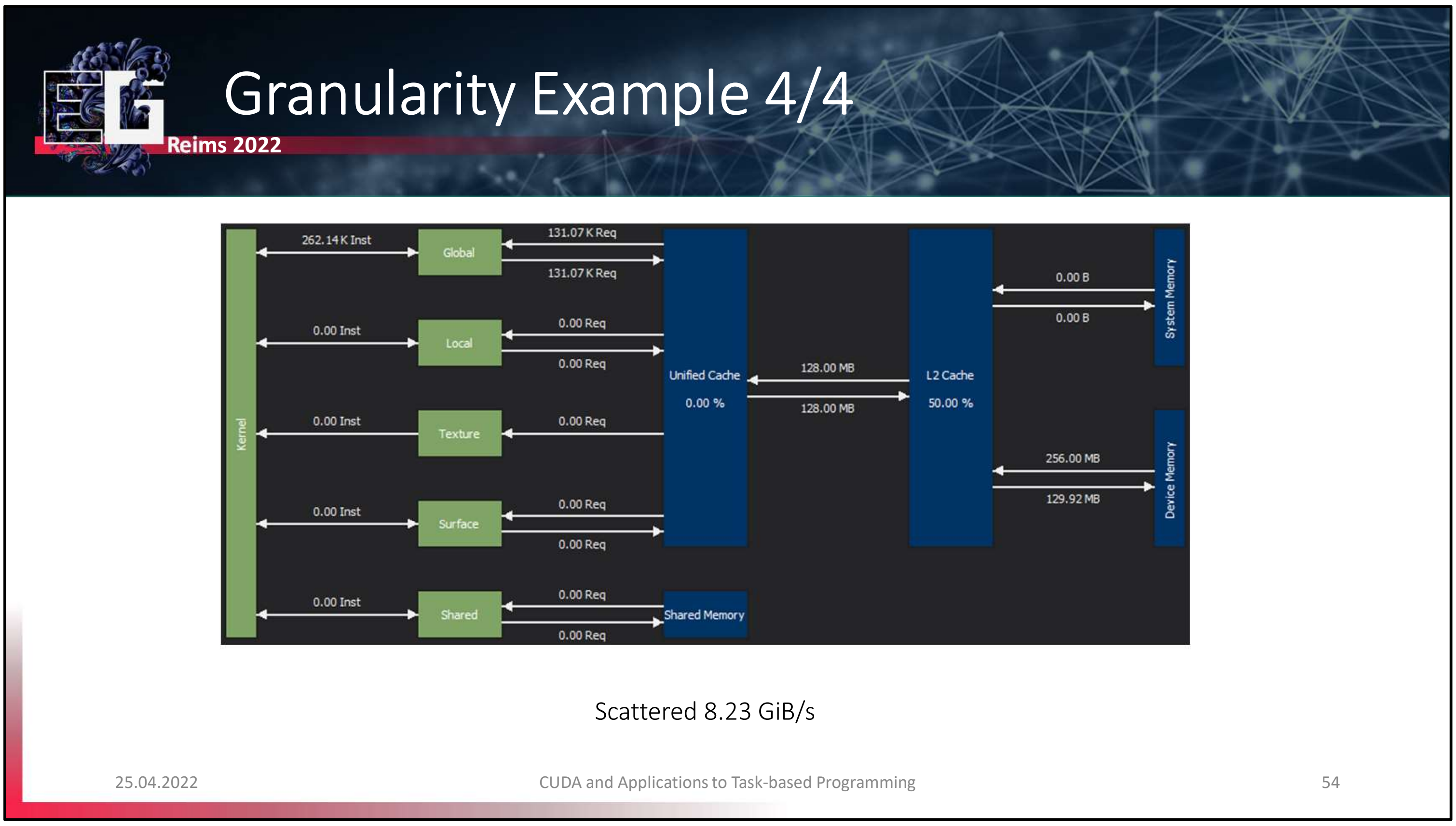
Reims 2022

```
__global__ void testScattered(int* in, int* out, int elements)
{
  int block_offset = blockIdx.x*blockDim.x;
  int warp_offset = 32 * (threadIdx.x / 32);
  int elementid = threadIdx.x % 32;
  int id = ((block_offset + warp_offset + elementid) * 121) % elements;

  out[id] = in[id];
}
```

```
testCoalesced done in 0.0655 ms ⇔ 255.875 GiB/s
testOffset<32> done in 0.972 ms ⇔ 17.2605 GiB/s
testScattered done in 2.0385 ms ⇔ 8.23 GiB/s
```

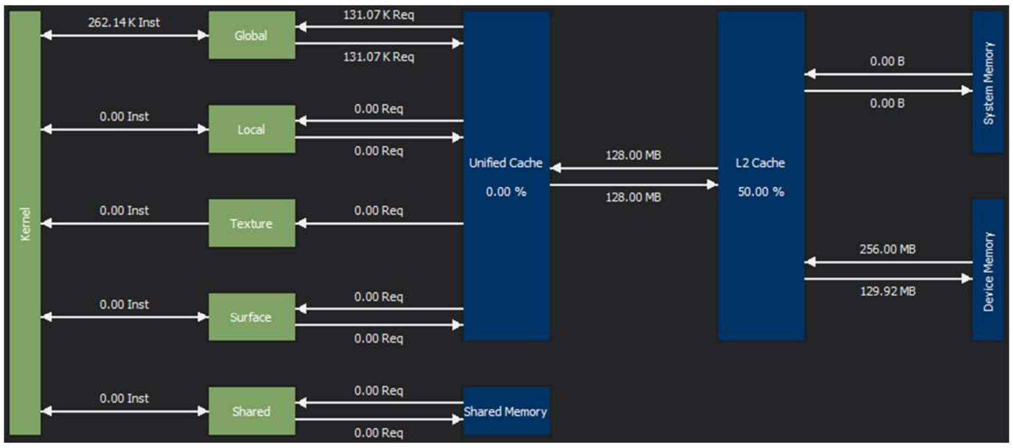
25.04.2022 CUDA and Applications to Task-based Programming 53

A more or less random access pattern using a simple linear congruential generator to compute the indices results in the worst performance.



Granularity Example 4/4

Reims 2022



Scattered 8.23 GiB/s

25.04.2022 CUDA and Applications to Task-based Programming 54

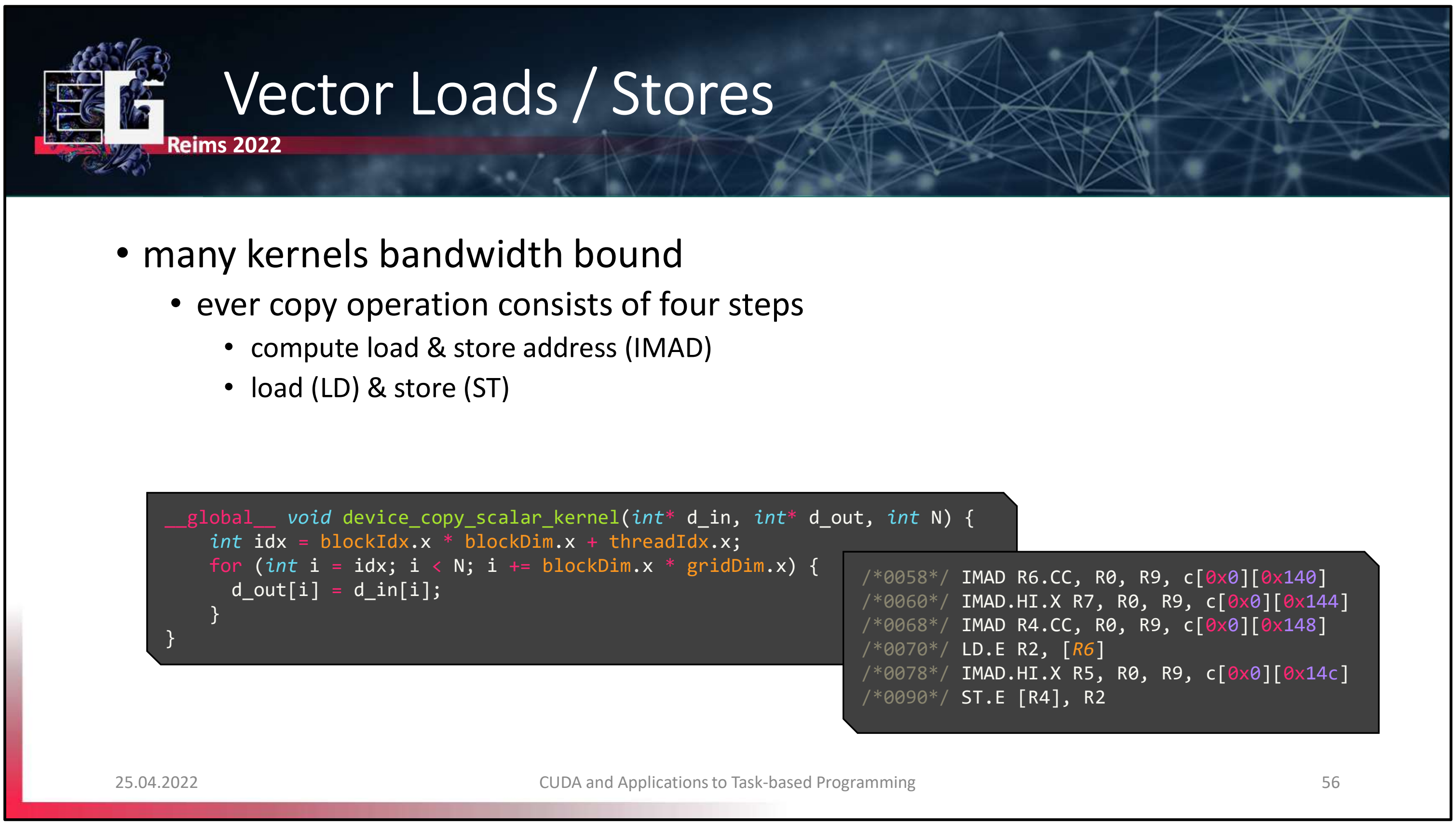
However, the overall memory bandwidth usage is comparable to the version with 32 element stride. This is not surprising as in the stride 32 version, every memory access already gave rise to a separate transaction, which is the worst possible scenario. The fact that the random access pattern version is even slower than the stride 32 version is explained by the more complex index computation used in the kernel.

Granularity Example Summary

Reims 2022

• testCoalesced	0.066 ms	⇔	255.875 GiB/s
testMixed	0.068 ms	⇔	246.260 GiB/s
testOffset<2>	0.125 ms	⇔	134.123 GiB/s
testOffset<4>	0.230 ms	⇔	72.979 GiB/s
testOffset<8>	0.456 ms	⇔	36.812 GiB/s
testOffset<32>	0.972 ms	⇔	17.260 GiB/s
testScattered	2.039 ms	⇔	8.230 GiB/s

- access pattern within 128 Byte segment does not matter
- offset between data → more requests need to be handled
- peak performance not met due to computation overhead
- more scattered data access slower with GDDR RAM



Vector Loads / Stores

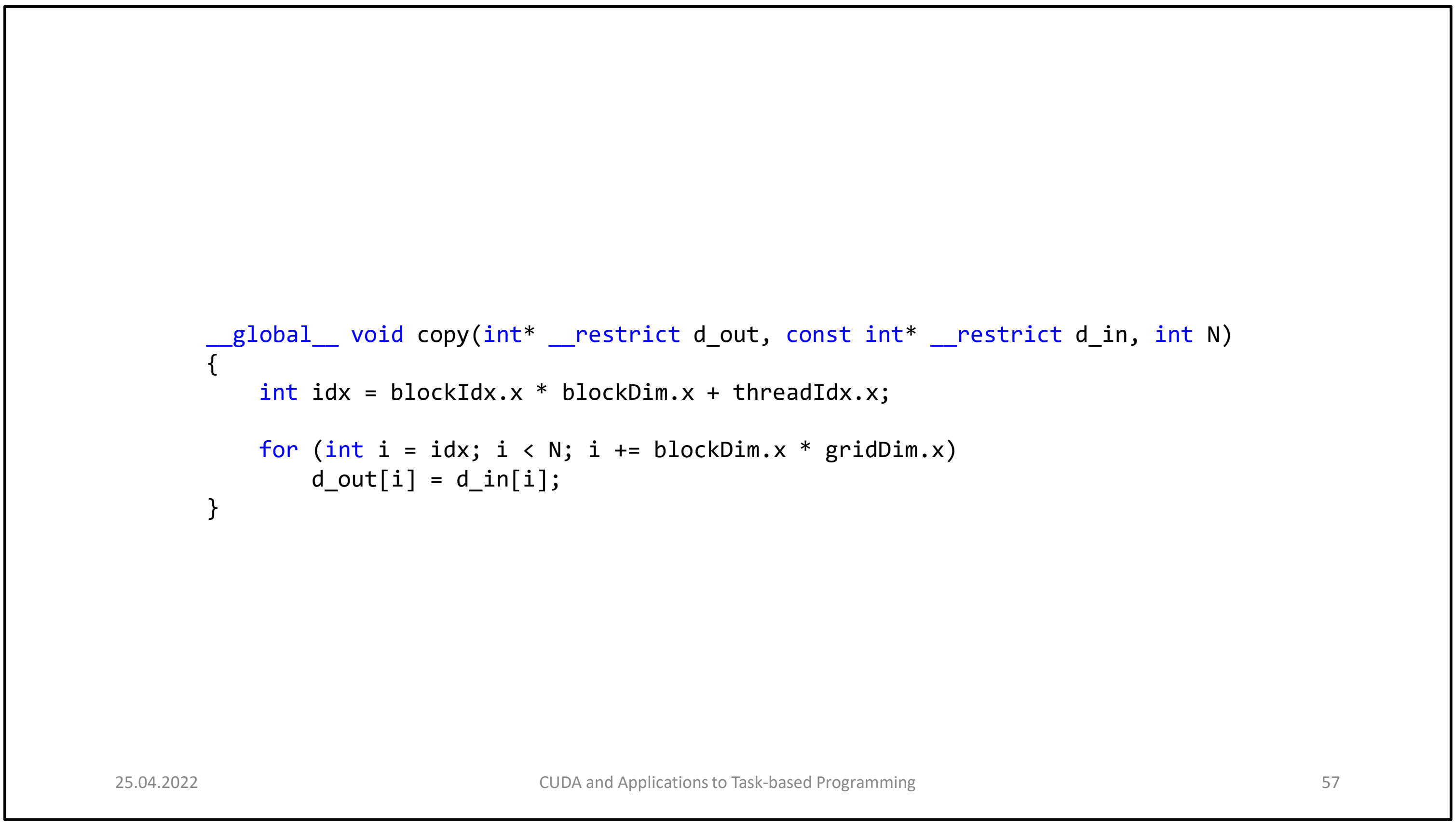
- many kernels bandwidth bound
  - ever copy operation consists of four steps
    - compute load & store address (IMAD)
    - load (LD) & store (ST)

```
__global__ void device_copy_scalar_kernel(int* d_in, int* d_out, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = idx; i < N; i += blockDim.x * gridDim.x) {
      d_out[i] = d_in[i];
    }
}
```

```
/*0058*/ IMAD R6.CC, R0, R9, c[0x0][0x140]
/*0060*/ IMAD.HI.X R7, R0, R9, c[0x0][0x144]
/*0068*/ IMAD R4.CC, R0, R9, c[0x0][0x148]
/*0070*/ LD.E R2, [R6]
/*0078*/ IMAD.HI.X R5, R0, R9, c[0x0][0x14c]
/*0090*/ ST.E [R4], R2
```

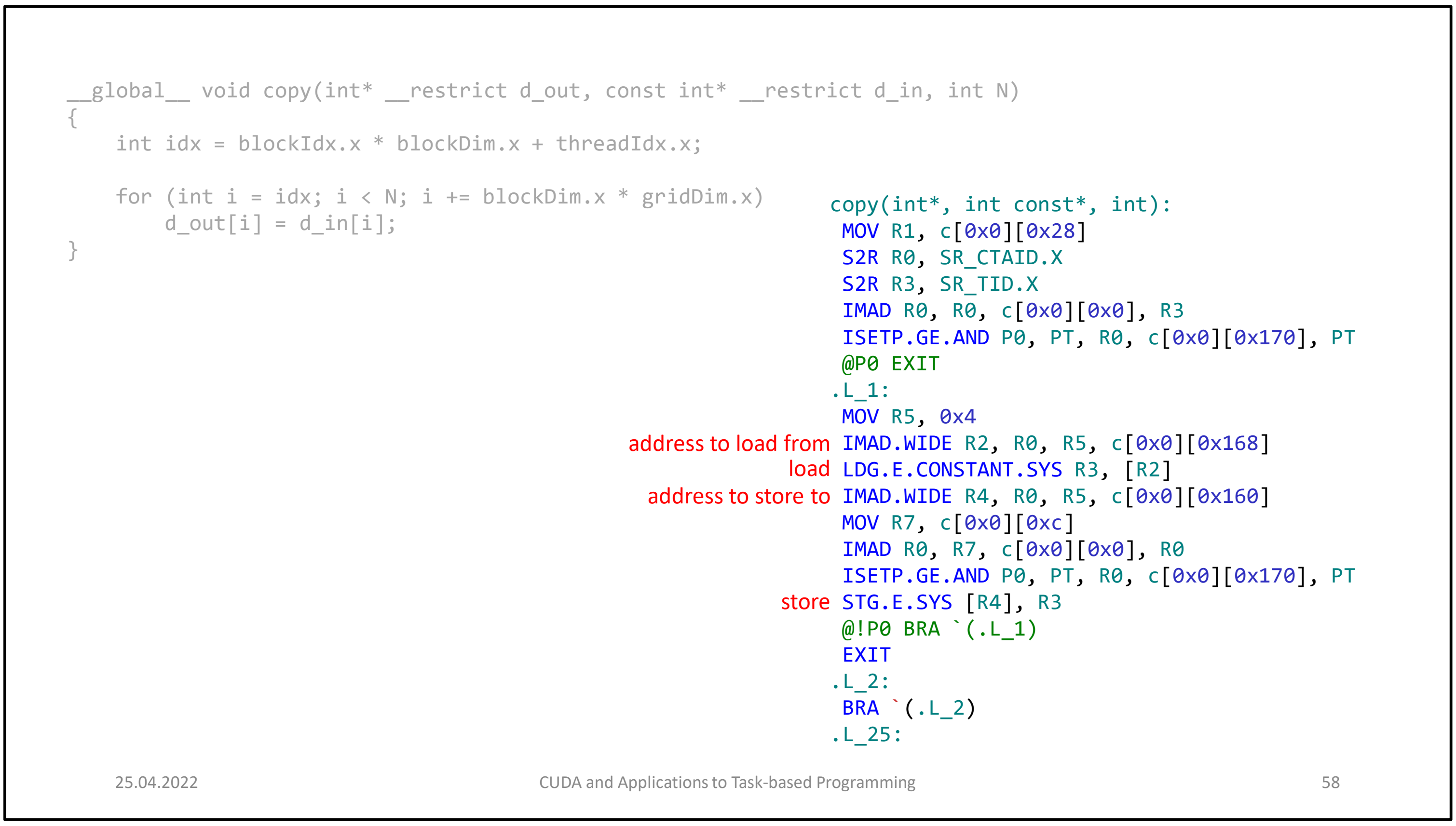
25.04.2022 CUDA and Applications to Task-based Programming 56

While many kernels are ultimately bandwidth bound, as we've seen before, scheduling overhead can add significant cost on top of what should be a purely bandwidth-limited operation. Before the memory request itself can be issued, we must first compute the addresses to load from and store to, which adds latency that cannot easily be hidden since the memory operations have a data dependency on their results while, at the same time, all warps are most-likely waiting either on their memory requests to be served or their address computations to be completed.



```
__global__ void copy(int* __restrict d_out, const int* __restrict d_in, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    for (int i = idx; i < N; i += blockDim.x * gridDim.x)
        d_out[i] = d_in[i];
}
```

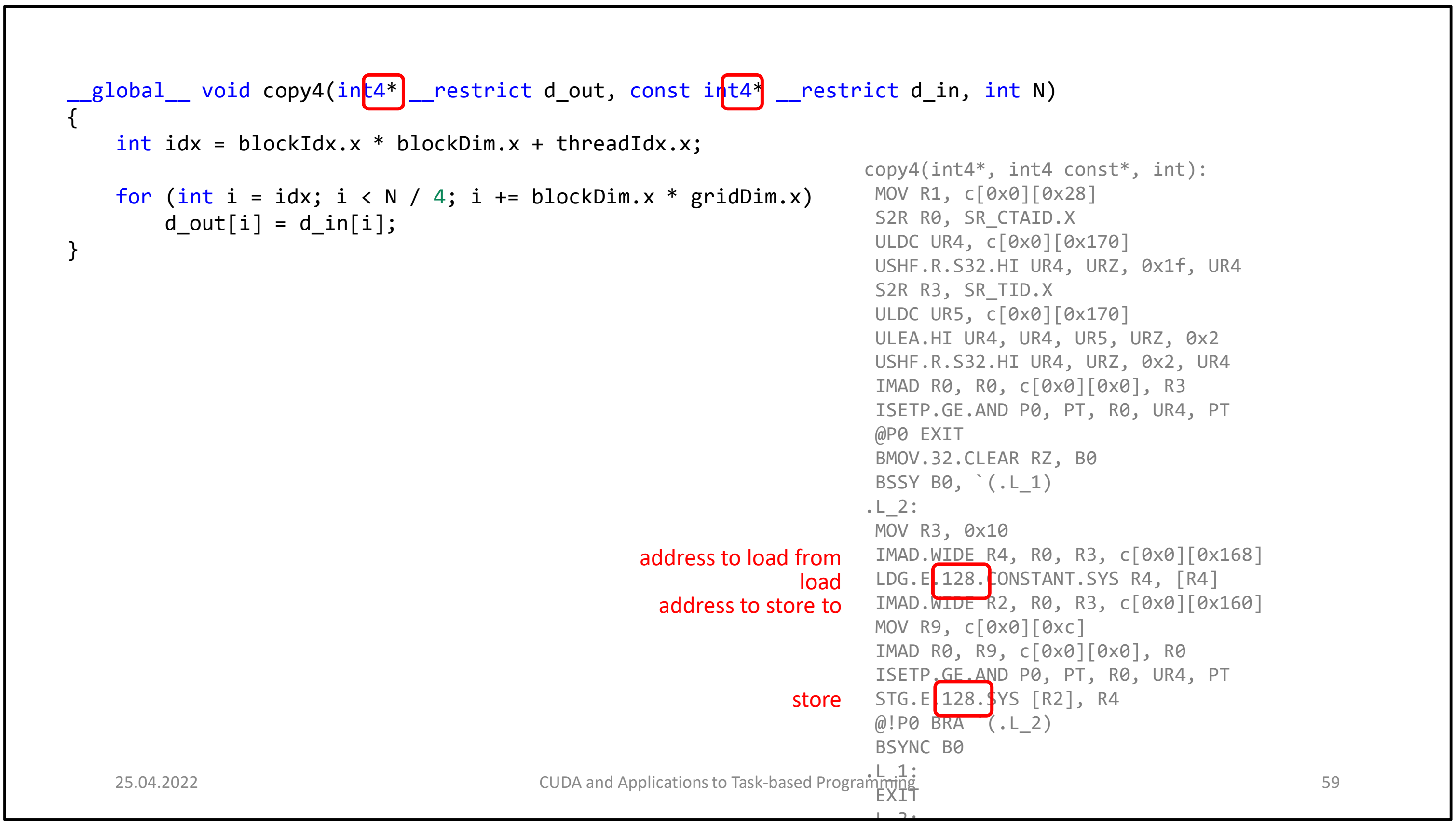
A fist optimization we can make to our simple copy kernel is to annotate the input and output pointer parameters with the `__restrict` qualifier. This instructs the compiler to assume that the input and output will never point to overlapping memory regions. The implication of this is that the writes through `d_out` can never affect the values that any of the reads through `d_in` will produce. As a result, the compiler can potentially, e.g., fetch input data through faster non-coherent caches.



```
__global__ void copy(int* __restrict d_out, const int* __restrict d_in, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    for (int i = idx; i < N; i += blockDim.x * gridDim.x)
        d_out[i] = d_in[i];
}
```

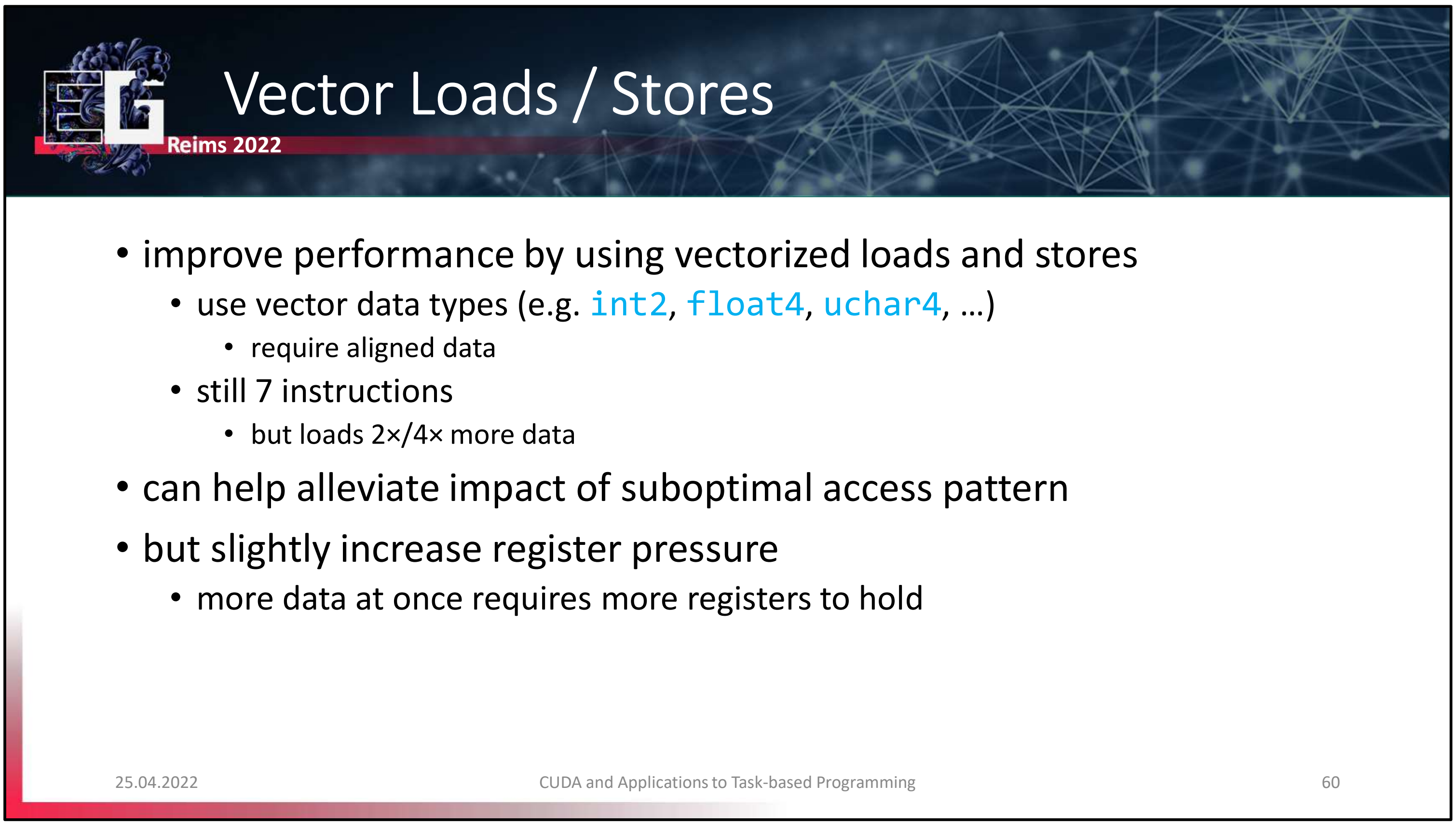
```
                    copy(int*, int const*, int):
                     MOV R1, c[0x0][0x28]
                     S2R R0, SR_CTAID.X
                     S2R R3, SR_TID.X
                     IMAD R0, R0, c[0x0][0x0], R3
                     ISETP.GE.AND P0, PT, R0, c[0x0][0x170], PT
                     @P0 EXIT
                    .L_1:
                     MOV R5, 0x4
address to load from IMAD.WIDE R2, R0, R5, c[0x0][0x168]
                load LDG.E.CONSTANT.SYS R3, [R2]
 address to store to IMAD.WIDE R4, R0, R5, c[0x0][0x160]
                     MOV R7, c[0x0][0xc]
                     IMAD R0, R7, c[0x0][0x0], R0
                     ISETP.GE.AND P0, PT, R0, c[0x0][0x170], PT
               store STG.E.SYS [R4], R3
                     @!P0 BRA `(.L_1)
                     EXIT
                    .L_2:
                     BRA `(.L_2)
                    .L_25:
```

Looking at the SASS for our simple copy kernel, we find that there are 7 instructions that have to complete before each store can be issued.


```
__global__ void copy4(int4* __restrict d_out, const int4* __restrict d_in, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    for (int i = idx; i < N / 4; i += blockDim.x * gridDim.x)
        d_out[i] = d_in[i];
}
```

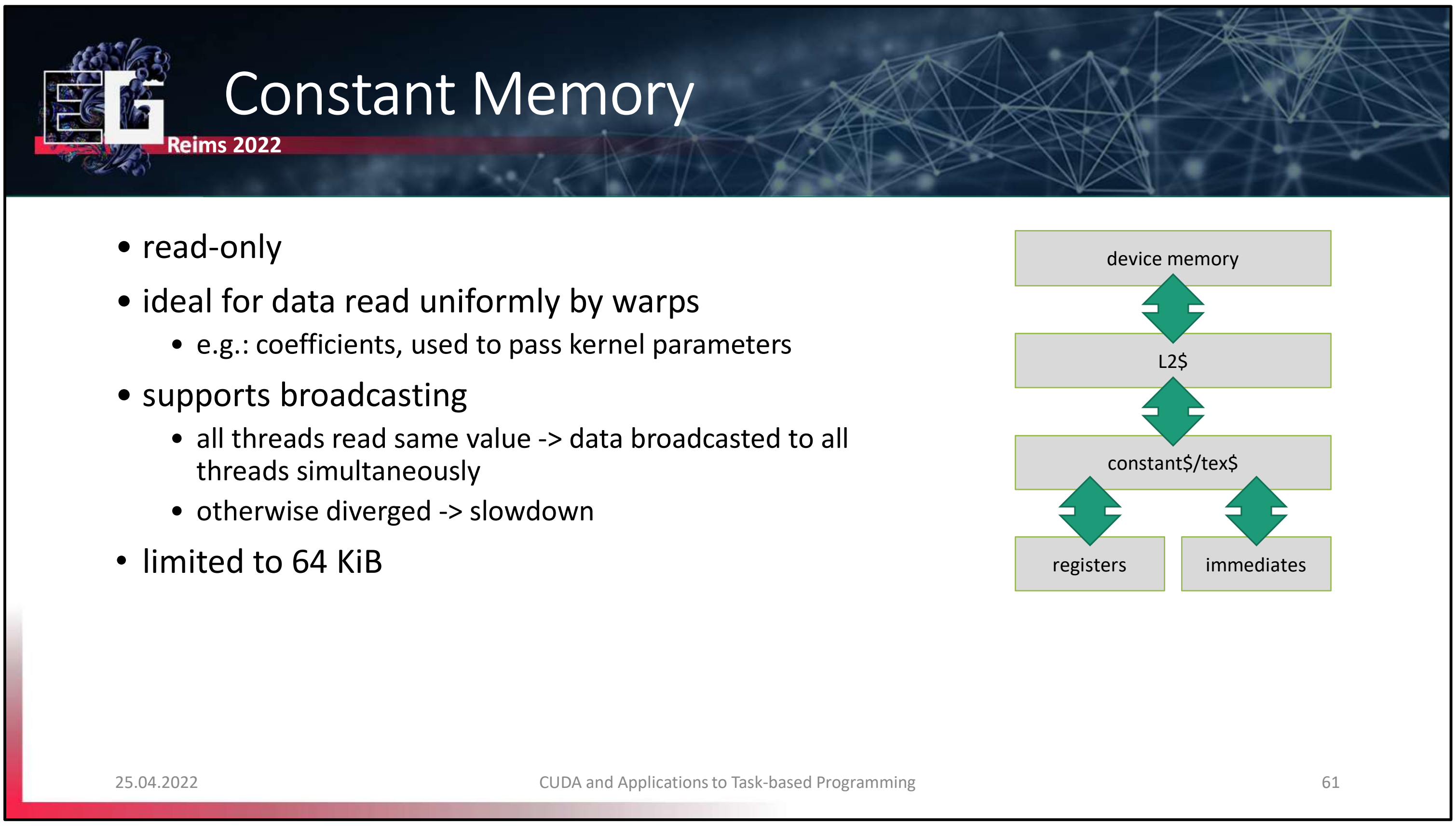
```
copy4(int4*, int4 const*, int):
 MOV R1, c[0x0][0x28]
 S2R R0, SR_CTAID.X
 ULDC UR4, c[0x0][0x170]
 USHF.R.S32.HI UR4, URZ, 0x1f, UR4
 S2R R3, SR_TID.X
 ULDC UR5, c[0x0][0x170]
 ULEA.HI UR4, UR4, UR5, URZ, 0x2
 USHF.R.S32.HI UR4, URZ, 0x2, UR4
 IMAD R0, R0, c[0x0][0x0], R3
 ISETP.GE.AND P0, PT, R0, UR4, PT
 @P0 EXIT
 BMOV.32.CLEAR RZ, B0
 BSSY B0, `(.L_1)
.L_2:
 MOV R3, 0x10
 IMAD.WIDE R4, R0, R3, c[0x0][0x168]
 LDG.E.128.CONSTANT.SYS R4, [R4]
 IMAD.WIDE R2, R0, R3, c[0x0][0x160]
 MOV R9, c[0x0][0xc]
 IMAD R0, R9, c[0x0][0x0], R0
 ISETP.GE.AND P0, PT, R0, UR4, PT
 STG.E.128.SYS [R2], R4
 @!P0 BRA `(.L_2)
 BSYNC B0
.L_1:
 EXIT
```

One way of improving this situation is by using vector loads and stores. These can load and store up to 16 bytes at once (assuming the data is suitably aligned). While there are still 7 instructions to complete before every store, since every store now moves a larger amount of memory, the instruction overhead per byte moved is significantly reduced.



Vector Loads / Stores

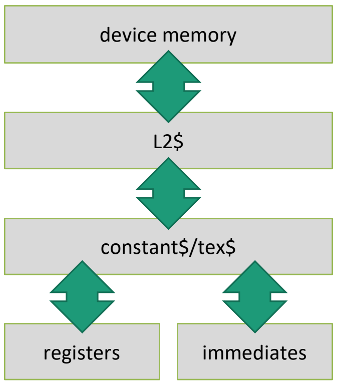
- improve performance by using vectorized loads and stores
  - use vector data types (e.g. `int2`, `float4`, `uchar4`, …)
    - require aligned data
  - still 7 instructions
    - but loads 2×/4× more data
- can help alleviate impact of suboptimal access pattern
- but slightly increase register pressure
  - more data at once requires more registers to hold



Constant Memory

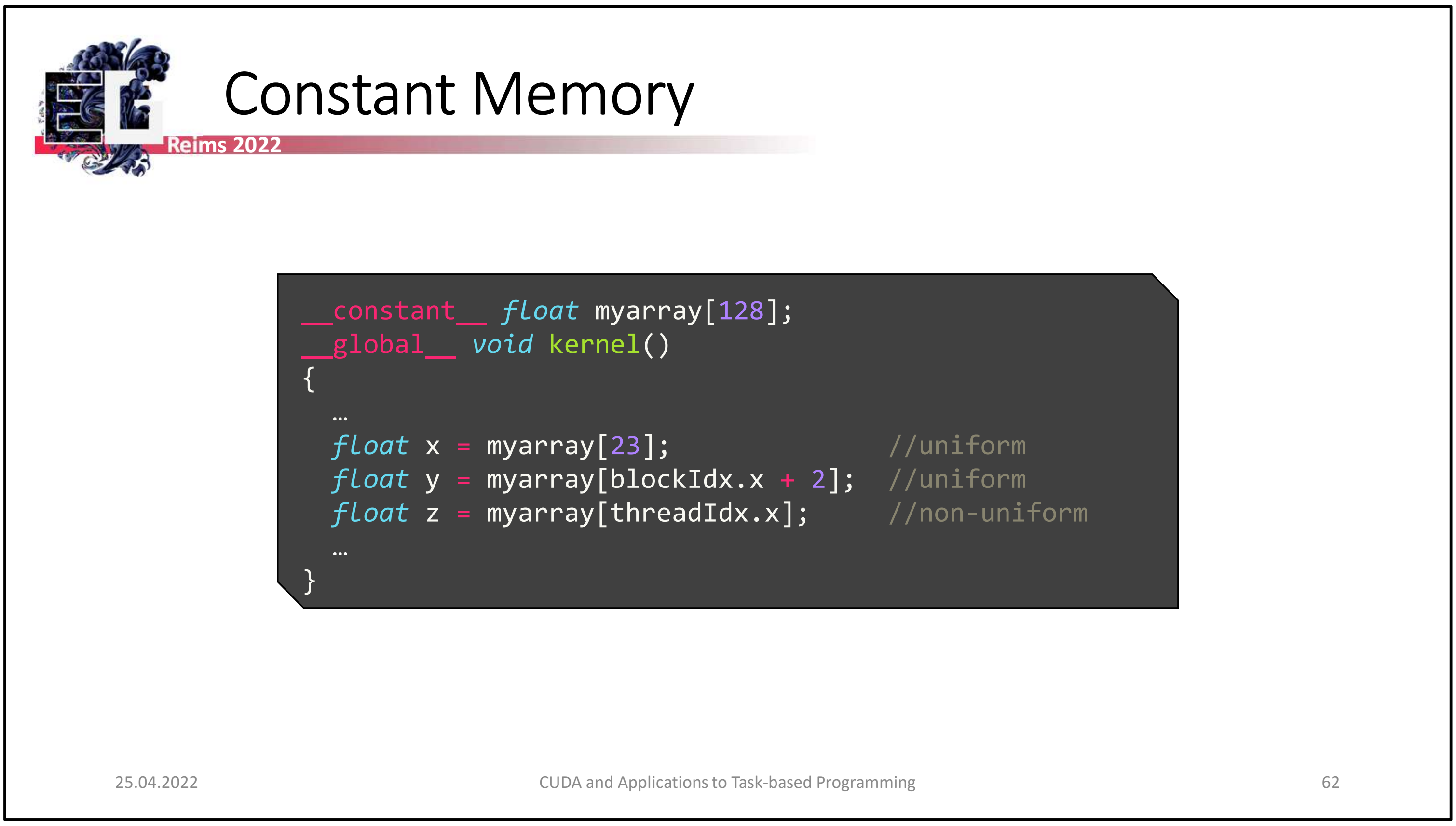
Reims 2022

- read-only
- ideal for data read uniformly by warps
  - e.g.: coefficients, used to pass kernel parameters
- supports broadcasting
  - all threads read same value -> data broadcasted to all threads simultaneously
  - otherwise diverged -> slowdown
- limited to 64 KiB



25.04.2022 CUDA and Applications to Task-based Programming 61

Kernel programs often need to access small amounts of read-only data such as coefficients/parameters where, at every point, all threads always need to access the same data element. In order to optimize for this common need, the GPU offers a dedicated system of small caches that are optimized for read-only broadcast access. This part of the memory hierarchy is exposed in CUDA as constant memory (analogous to uniform/constant buffers in graphics APIs).



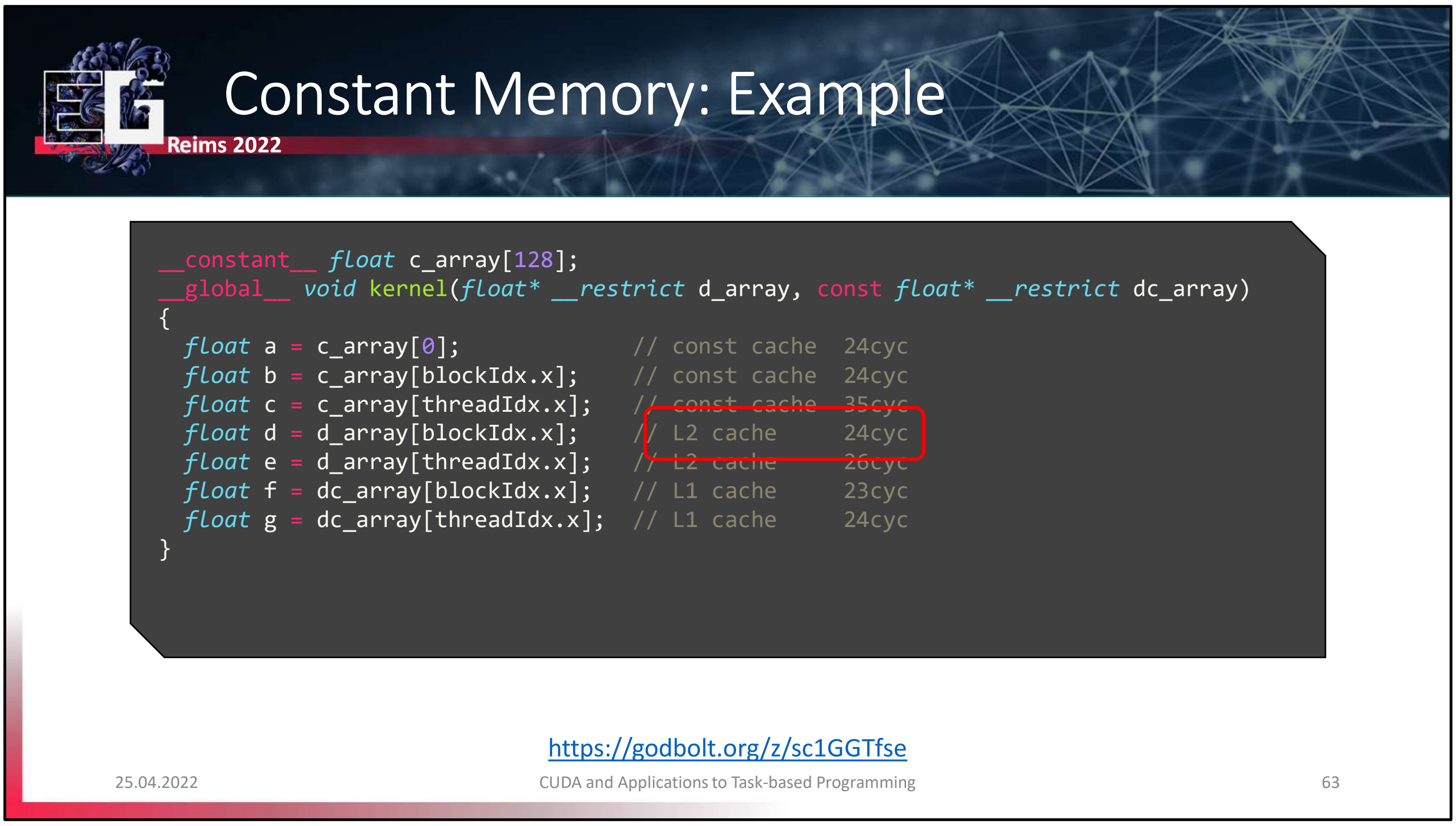
Constant Memory

Reims 2022

```
__constant__ float myarray[128];
__global__ void kernel()
{
  …
  float x = myarray[23];              //uniform
  float y = myarray[blockIdx.x + 2];  //uniform
  float z = myarray[threadIdx.x];     //non-uniform
  …
}
```

25.04.2022 CUDA and Applications to Task-based Programming 62

When using constant memory, it is important to make sure that access to elements in constant memory does indeed happen uniformly. The constant caches are designed for broad cast access. Divergent access will result in a performance penalty.



Constant Memory: Example

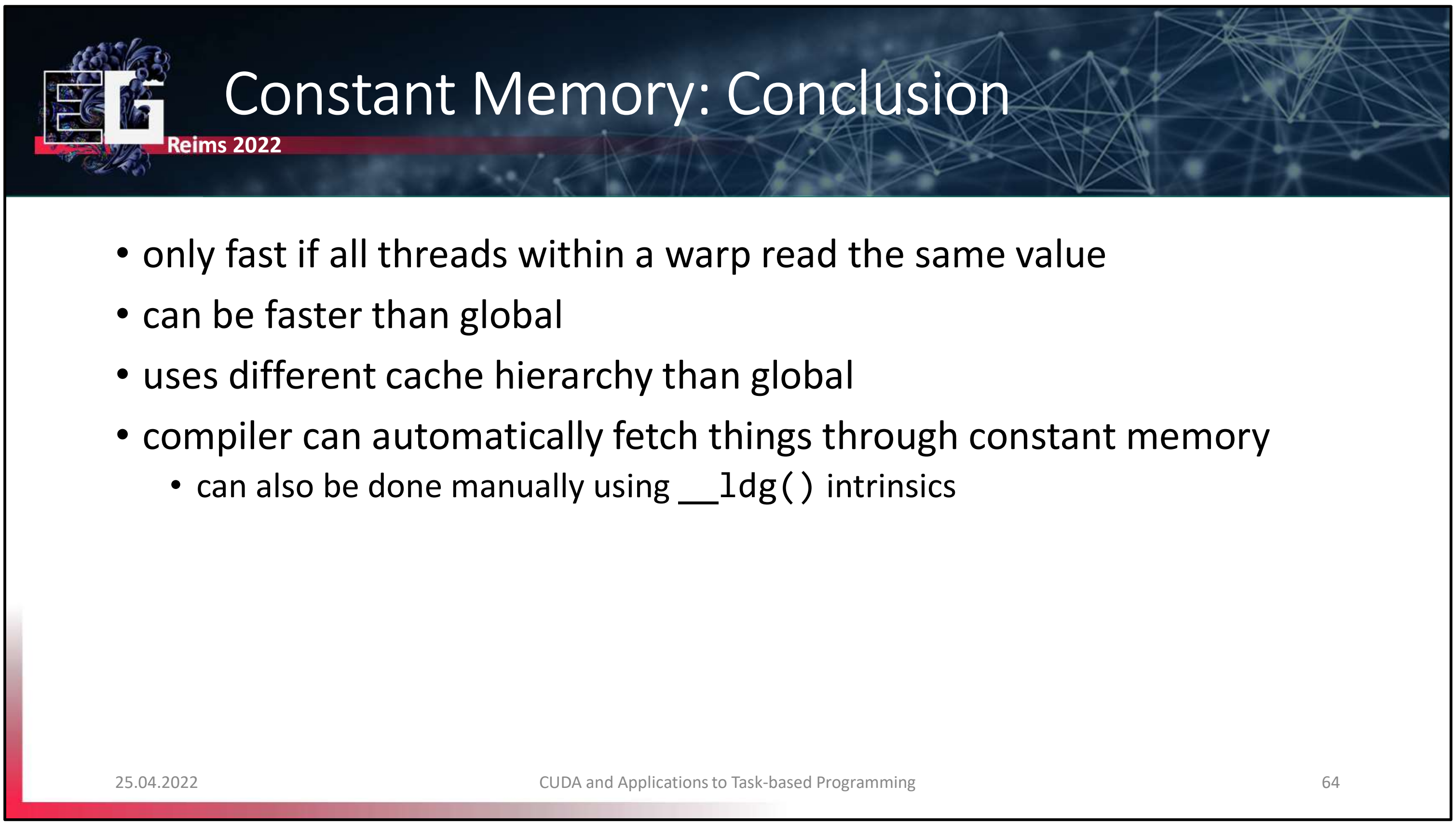
Reims 2022

```
__constant__ float c_array[128];
__global__ void kernel(float* __restrict d_array, const float* __restrict dc_array)
{
  float a = c_array[0];           // const cache  24cyc
  float b = c_array[blockIdx.x];   // const cache  24cyc
  float c = c_array[threadIdx.x];  // const cache  35cyc
  float d = d_array[blockIdx.x];   // L2 cache     24cyc
  float e = d_array[threadIdx.x];  // L2 cache     26cyc
  float f = dc_array[blockIdx.x];  // L1 cache     23cyc
  float g = dc_array[threadIdx.x]; // L1 cache     24cyc
}
```

https://godbolt.org/z/sc1GGTfse

25.04.2022 CUDA and Applications to Task-based Programming 63

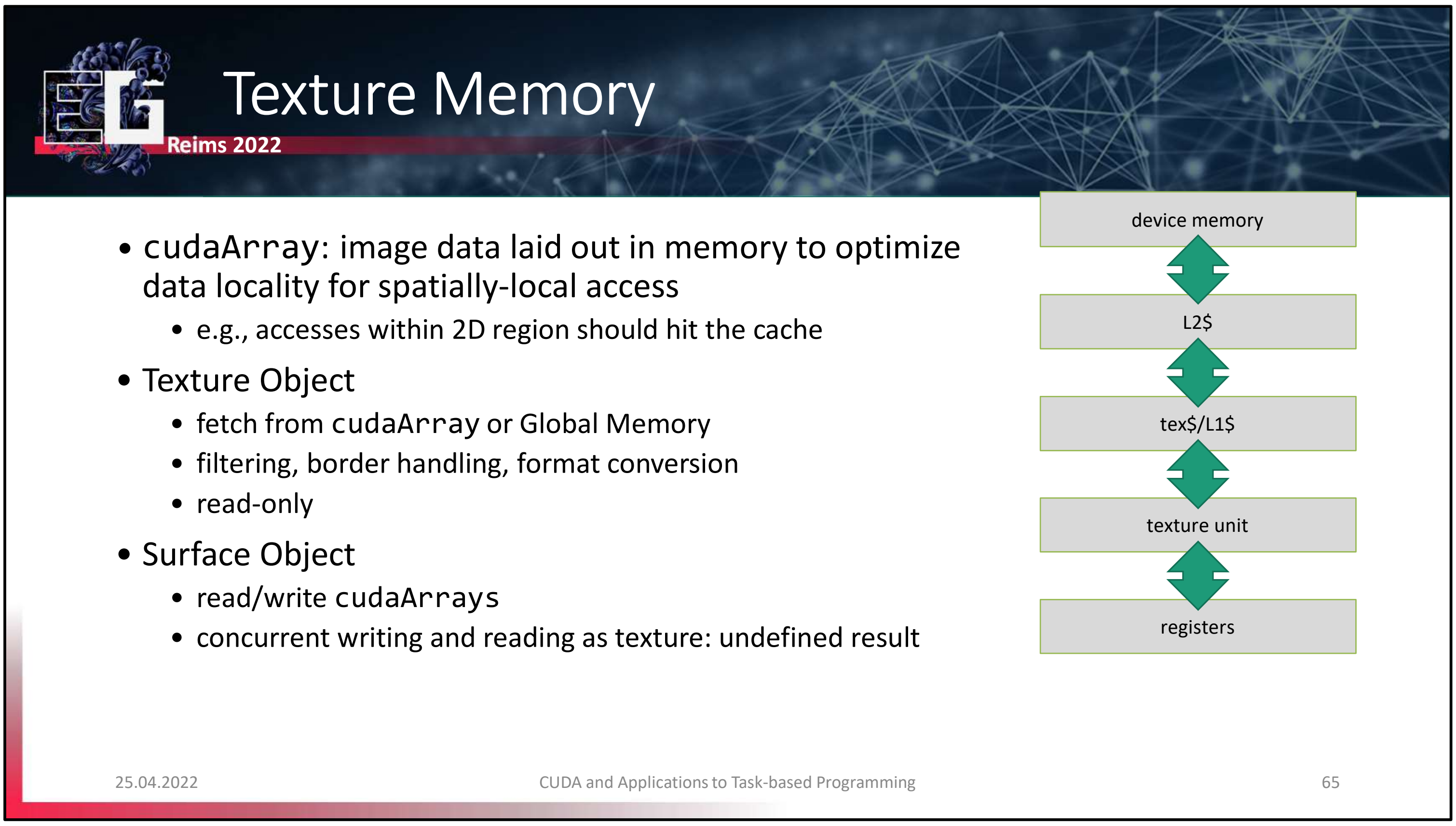
By measuring access times to elements in both constant memory as well as global memory, we demonstrate the importance of accessing constant memory uniformly. As long as access happens uniformly, constant memory can provide very low access latencies. However, as soon as the memory access diverges, a slowdown occurs. For divergent access patterns, normal global memory can typically provide better access latencies due to its different cache design (as long as the access hits the cache). We can also see that the compiler will automatically fetch data through the read-only cache if it can be sure that the data will not be modified during the run time of the kernel.



Constant Memory: Conclusion

Reims 2022

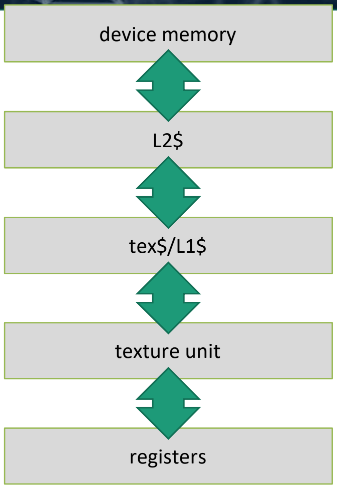
- only fast if all threads within a warp read the same value
- can be faster than global
- uses different cache hierarchy than global
- compiler can automatically fetch things through constant memory
  - can also be done manually using `__ldg()` intrinsics



Texture Memory

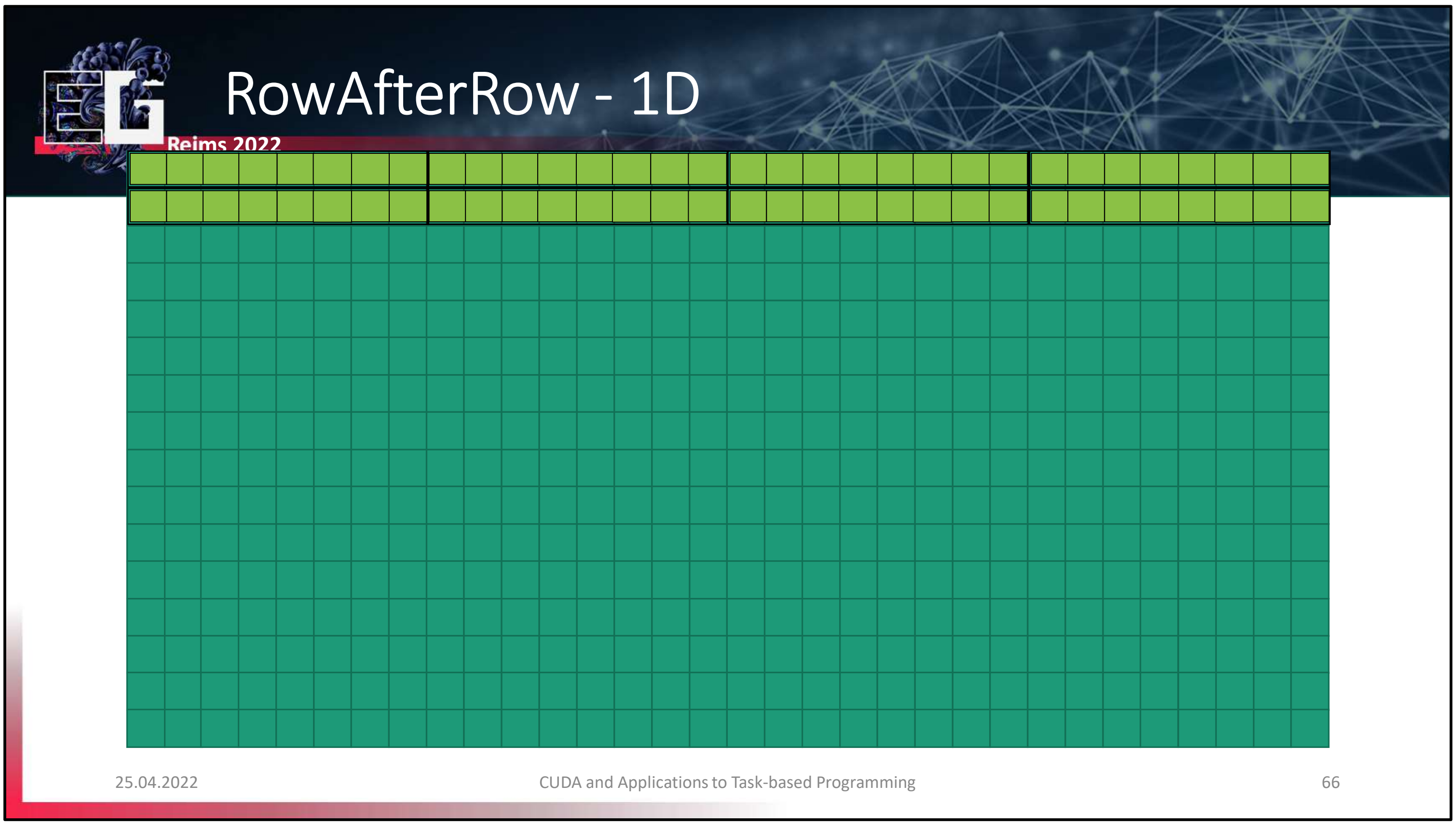
Reims 2022

- `cudaArray`: image data laid out in memory to optimize data locality for spatially-local access
  - e.g., accesses within 2D region should hit the cache
- Texture Object
  - fetch from `cudaArray` or Global Memory
  - filtering, border handling, format conversion
  - read-only
- Surface Object
  - read/write `cudaArrays`
  - concurrent writing and reading as texture: undefined result



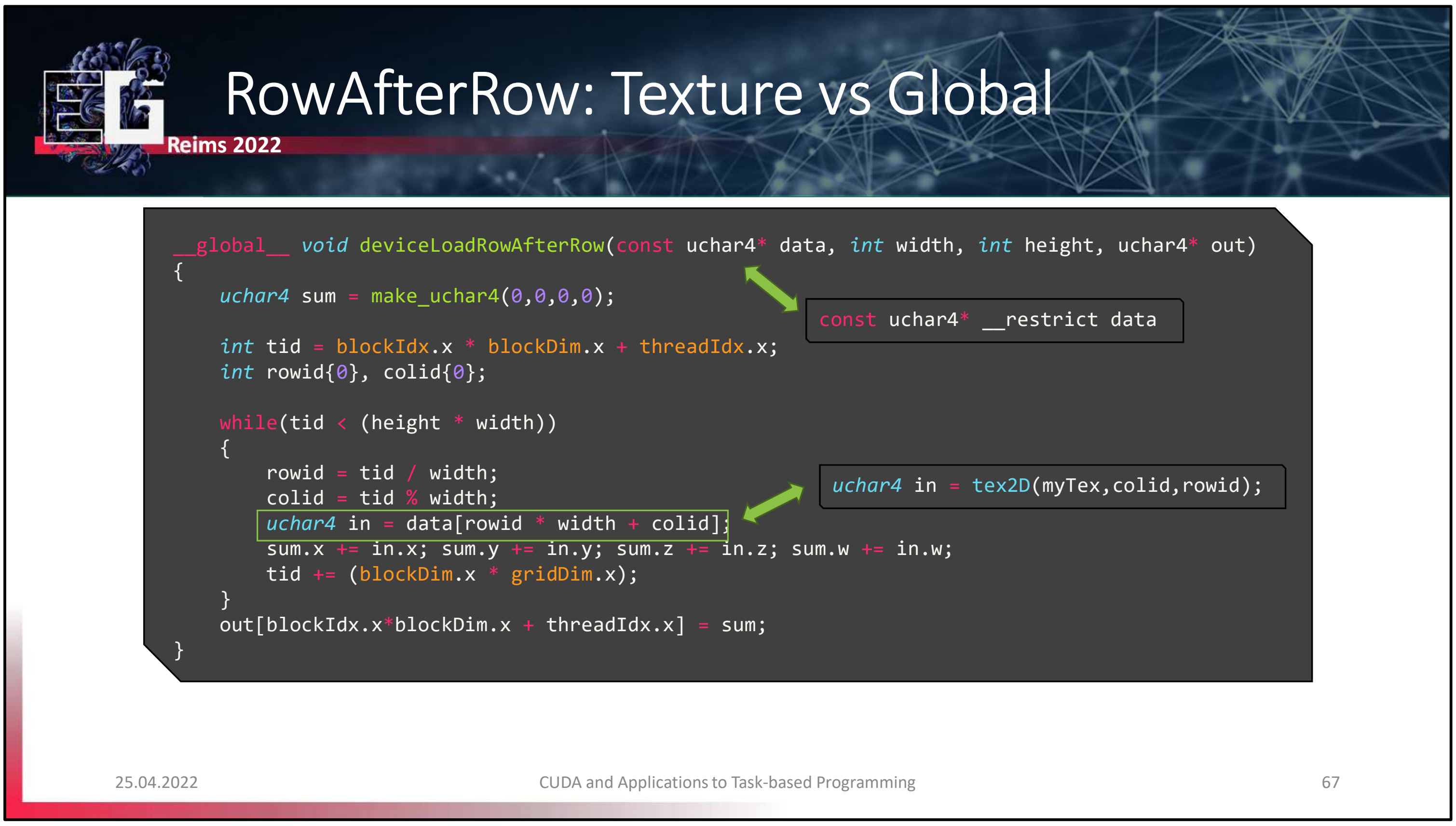
25.04.2022 CUDA and Applications to Task-based Programming 65

In order to cater to their original purpose of accelerating rendering, GPUs come with a sophisticated subsystem for accessing image data. This subsystem is exposed in CUDA in the form of cudaArrays, Texture Objects as well as Surface Objects. A cudaArray represents raw image data laid out in memory in a way that optimizes for cache utilization under spatially local access patterns. Texture Objects provide the ability to sample image data from a cudaArray (or Global Memory) with the hardware applying filtering as well as border handling and format conversions. Surface Objects provide read/write access to the image data in a cudaArray.



RowAfterRow - 1D

In order to illustrate the trade-offs of texture memory, we will perform a couple of simple tests. In this first example, we simply launch blocks that read image data in such a way that consecutive threads access consecutive pixels in a row-wise fashion.



RowAfterRow: Texture vs Global

Reims 2022

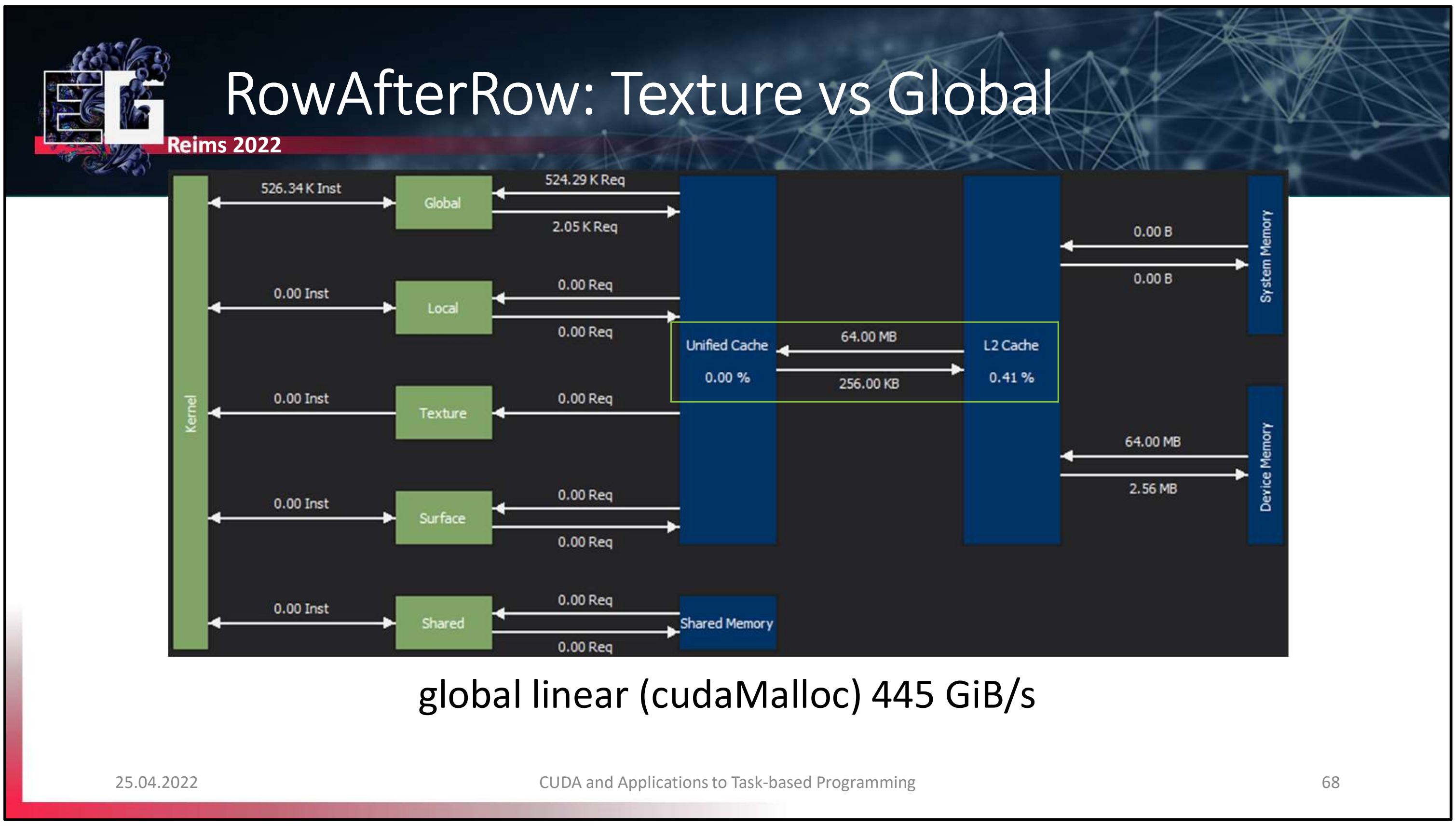
```
__global__ void deviceLoadRowAfterRow(const uchar4* data, int width, int height, uchar4* out)
{
    uchar4 sum = make_uchar4(0,0,0,0);

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int rowid{0}, colid{0};

    while(tid < (height * width))
    {
        rowid = tid / width;
        colid = tid % width;
        uchar4 in = data[rowid * width + colid];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
        tid += (blockDim.x * gridDim.x);
    }
    out[blockIdx.x*blockDim.x + threadIdx.x] = sum;
}
```

```
const uchar4* __restrict data
```

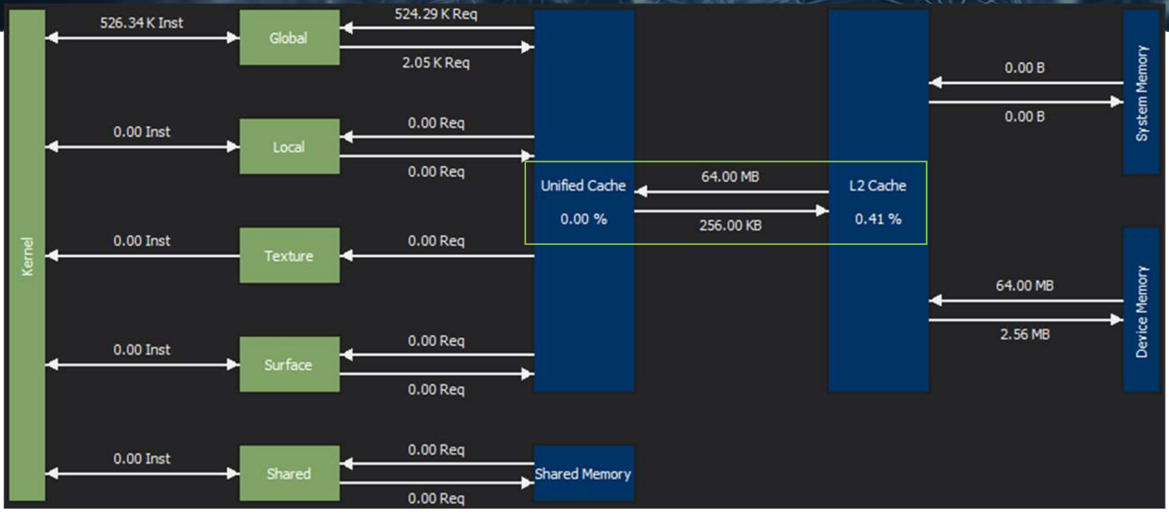
```
uchar4 in = tex2D(myTex,colid,rowid);
```

25.04.2022 CUDA and Applications to Task-based Programming 67

Both kernels are the same except for the fact that one accesses the image data via a pointer into a global memory buffer while the other accesses the image data by sampling it through a texture object. We also test one version that accesses the image data in global memory through a `__restrict` pointer to `const`, which will trigger reading through the read-only data cache.



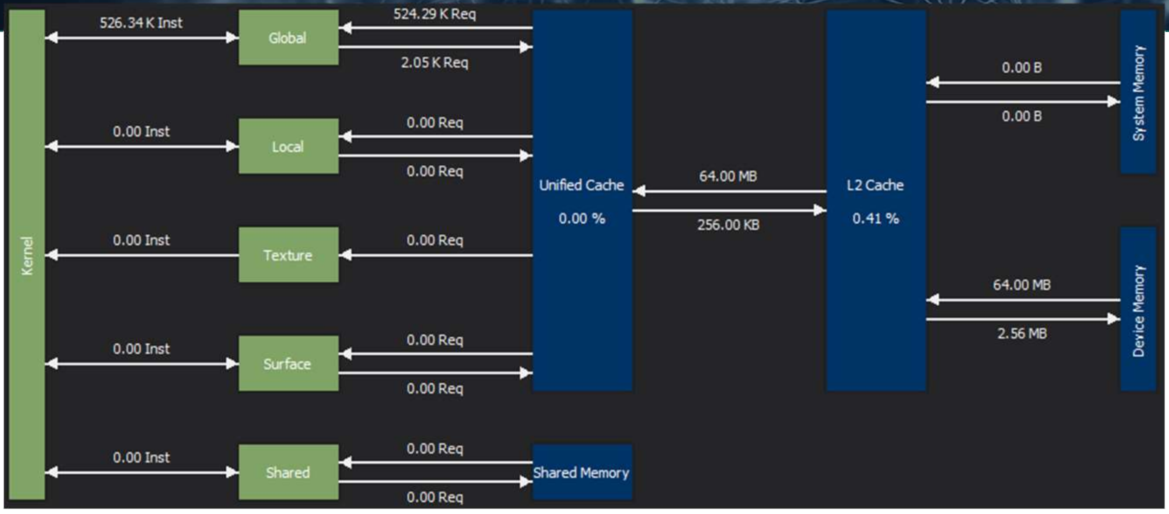
RowAfterRow: Texture vs Global



global linear (cudaMalloc) 445 GiB/s

As we can see, access via global memory performs significantly better than access via texture memory; access through the read-only data cache performs slightly better than access through plain global memory. The reason for this is to be found in the memory layout. Our image data in global memory is laid out in row-major fashion which, in this case, results in a perfectly coalesced memory access pattern. The texture, on the other hand, is laid out for a 2D spatially local access pattern, which results in a less optimal cache utilization with a purely row-wise access pattern like in this example.

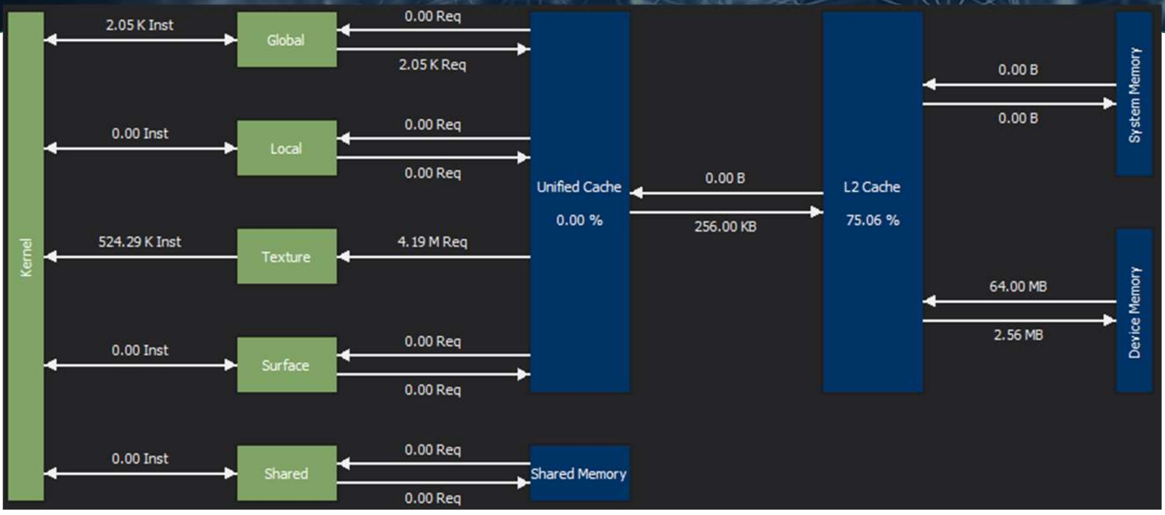
RowAfterRow: Texture vs Global



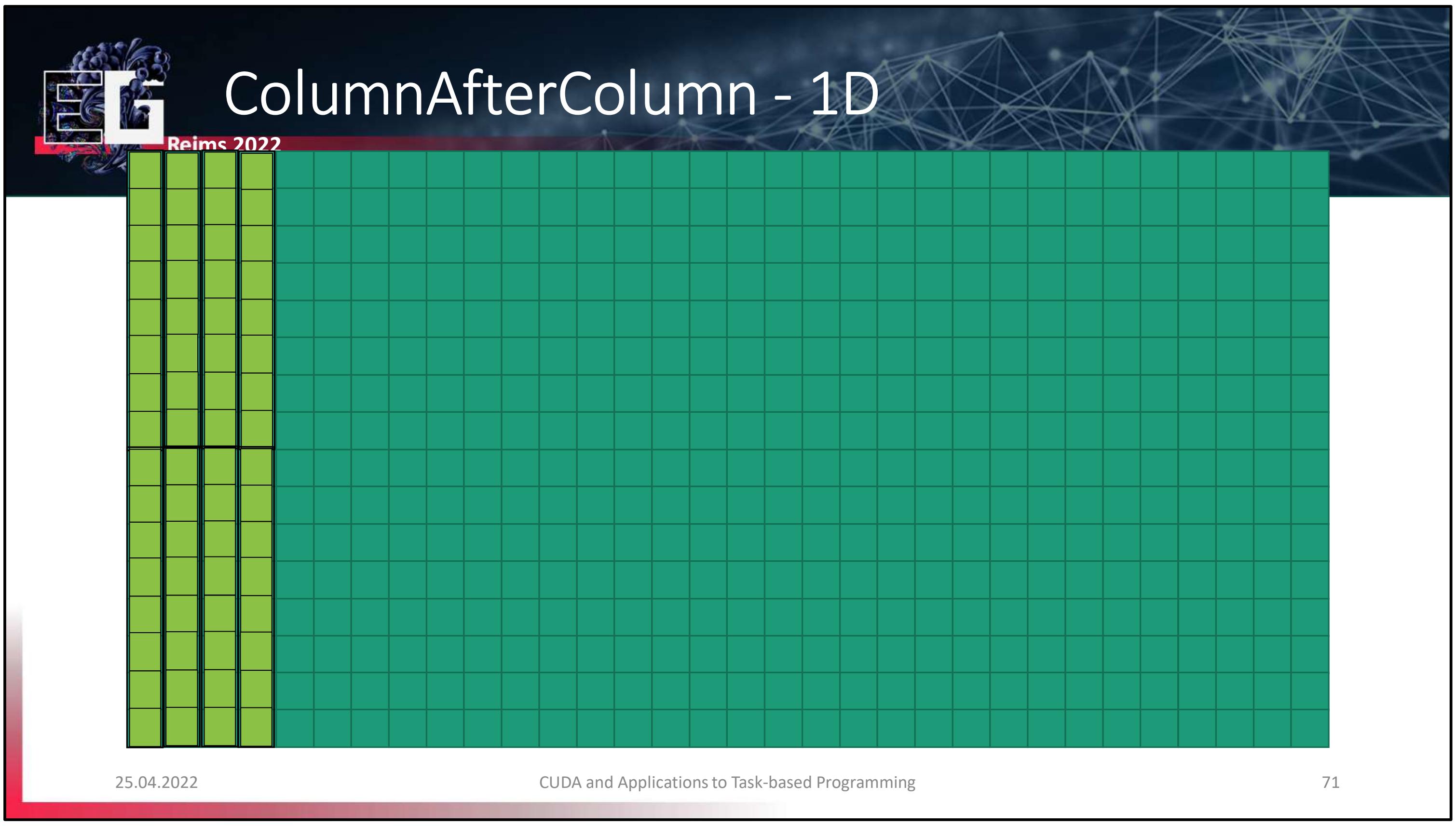
global restrict (cudaMalloc) 464 GiB/s

RowAfterRow: Texture vs Global

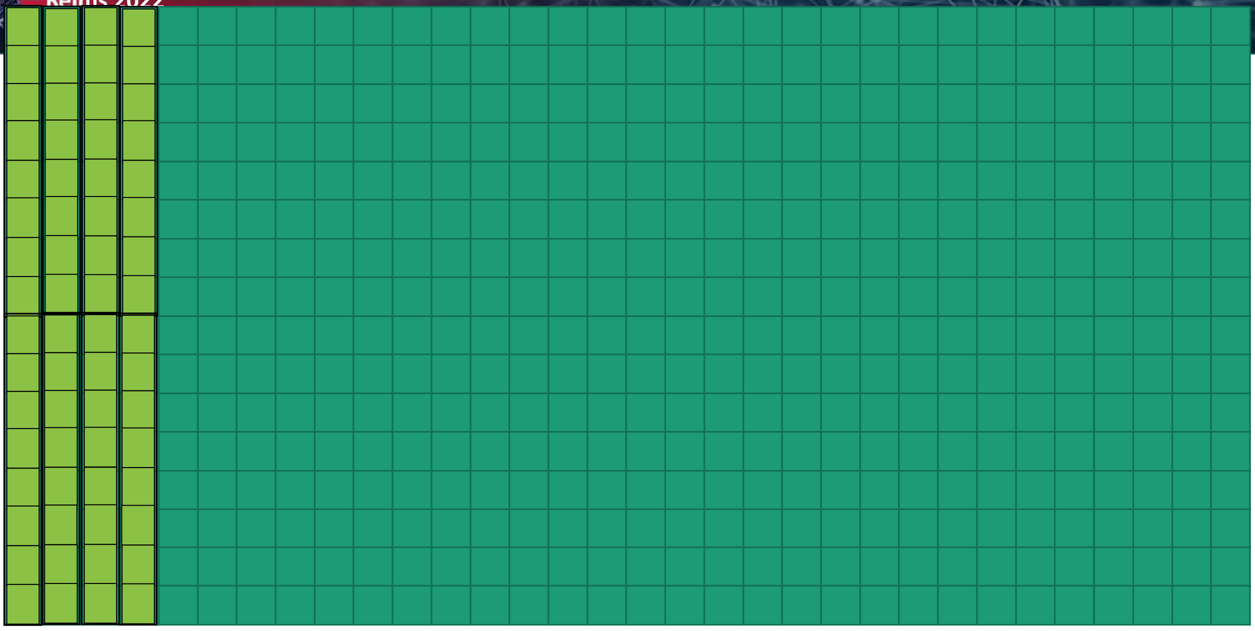
Reims 2022



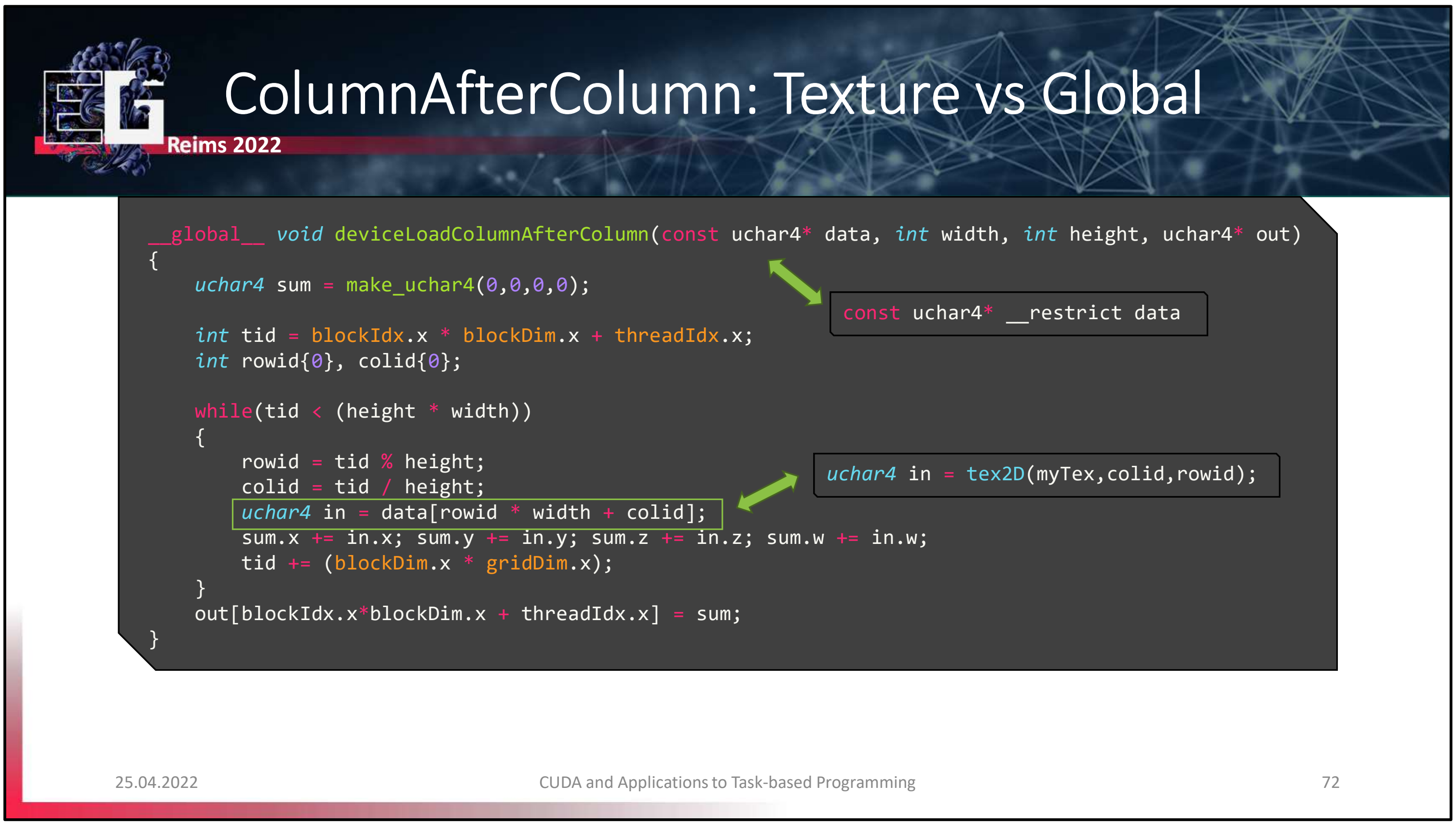
texture (cudaArray) 387 GiB/s



ColumnAfterColumn - 1D



In this second example, we launch blocks that read image data just like before, except that we now perform the access in such a way that consecutive threads access consecutive pixels in a column-wise fashion.



Reims 2022

ColumnAfterColumn: Texture vs Global

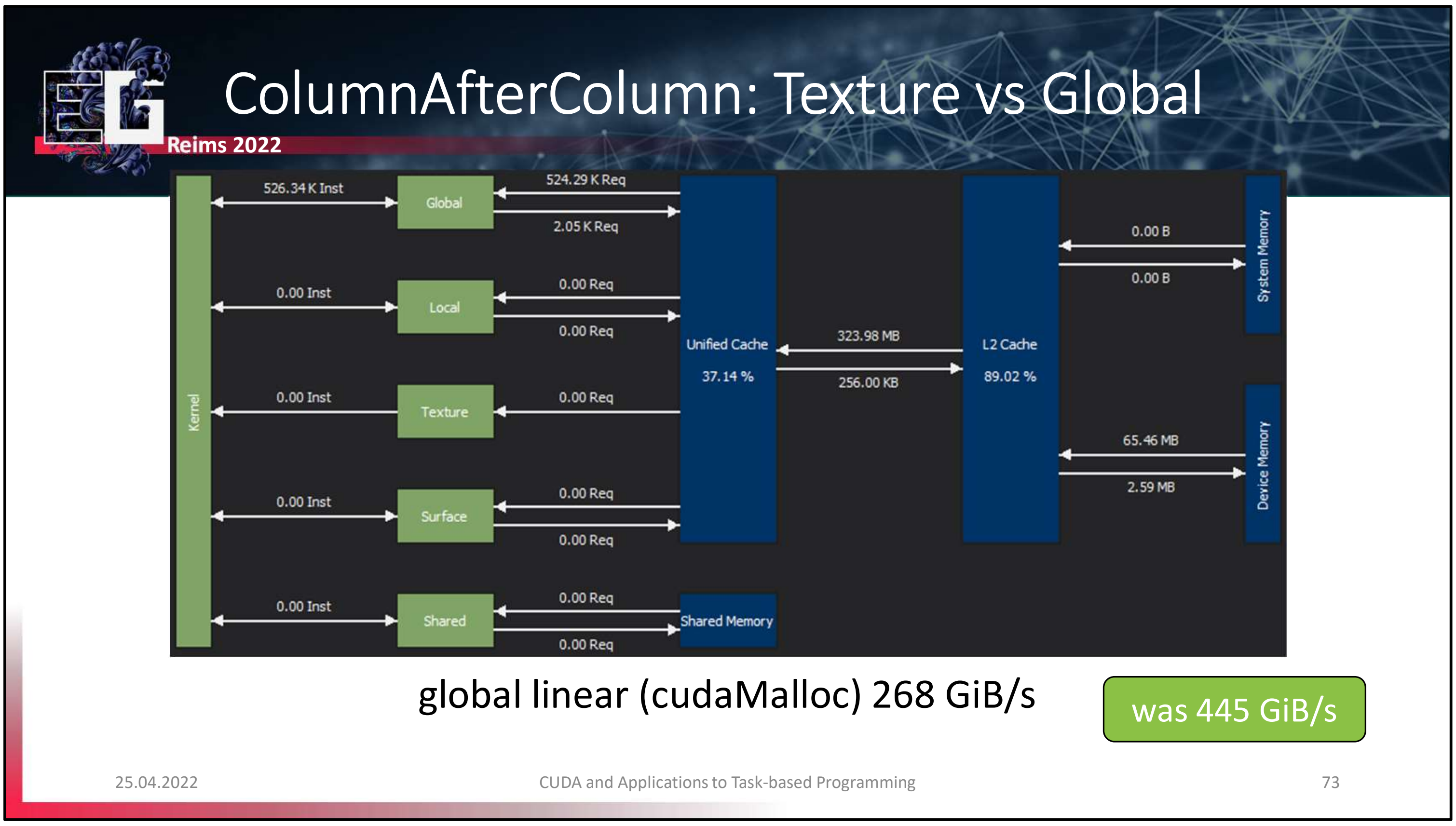
```
__global__ void deviceLoadColumnAfterColumn(const uchar4* data, int width, int height, uchar4* out)
{
    uchar4 sum = make_uchar4(0,0,0,0);

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int rowid{0}, colid{0};

    while(tid < (height * width))
    {
        rowid = tid % height;
        colid = tid / height;
        uchar4 in = data[rowid * width + colid];
        sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
        tid += (blockDim.x * gridDim.x);
    }
    out[blockIdx.x*blockDim.x + threadIdx.x] = sum;
}
```

```
const uchar4* __restrict data
```

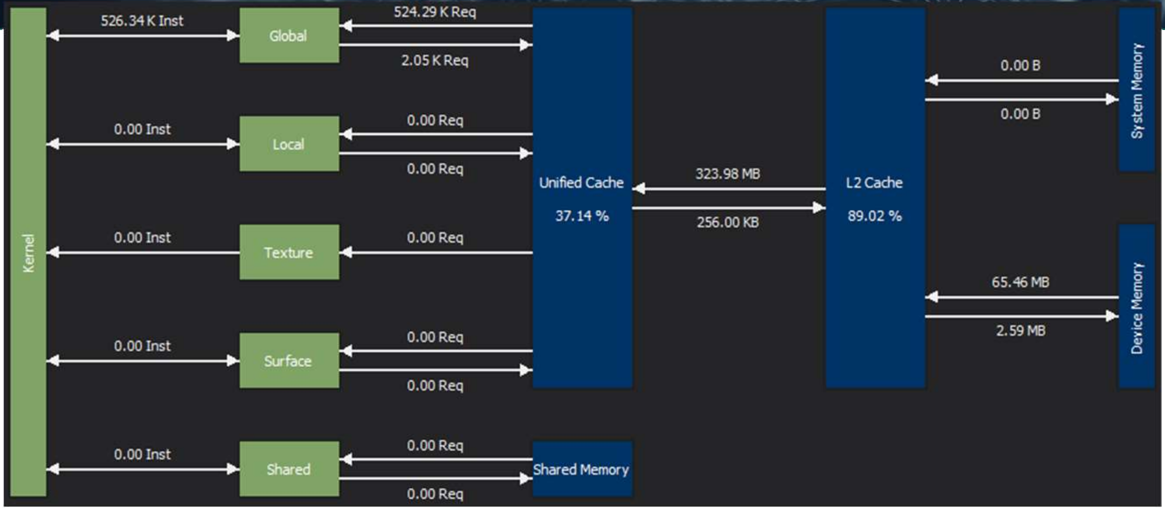
```
uchar4 in = tex2D(myTex,colid,rowid);
```

25.04.2022 CUDA and Applications to Task-based Programming 72

The kernel is virtually the same as before except for the index computation that now traverses the image in a column-wise manner.



ColumnAfterColumn: Texture vs Global



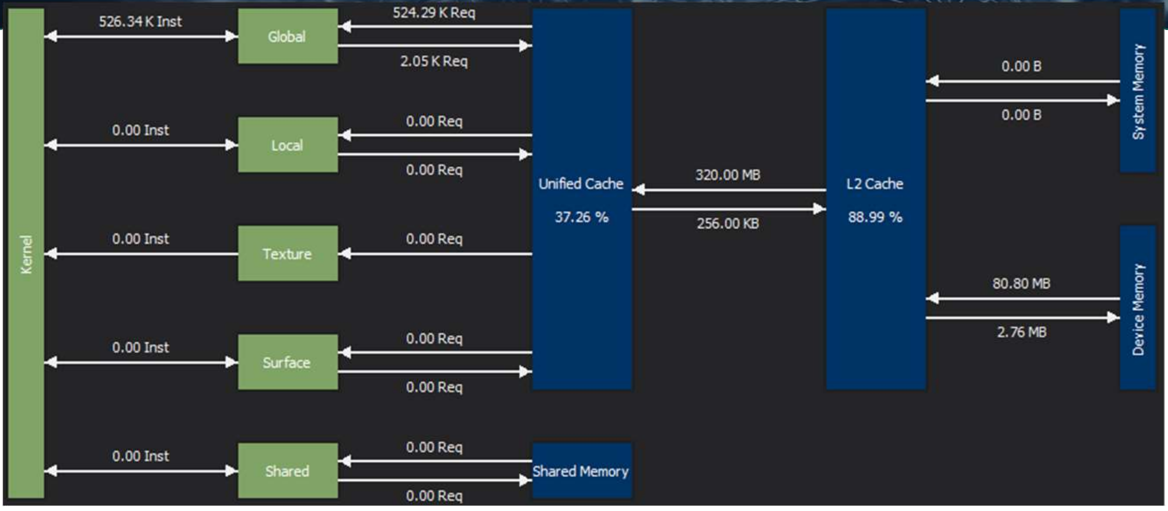
global linear (cudaMalloc) 268 GiB/s

was 445 GiB/s

25.04.2022 CUDA and Applications to Task-based Programming 73

As we can see, global memory, which used to perform best in the previous test case, now performs far worse than texture memory. The reason for this can again be found in the access pattern. Given the row-major layout of the global memory image data, the column-wise access in this example is the pathological worst case. The result is an incredibly low transaction efficiency which manifests in the fact that memory transfer between the SMs and L2$ is multiple times larger than the image itself. Texture memory, on the other hand, performs virtually identical as in the previous test case. Again, this is owed to its memory layout being optimized for 2D spatial access. While optimal for neither row-wise nor column-wise access, it can handle both with similar efficiency.

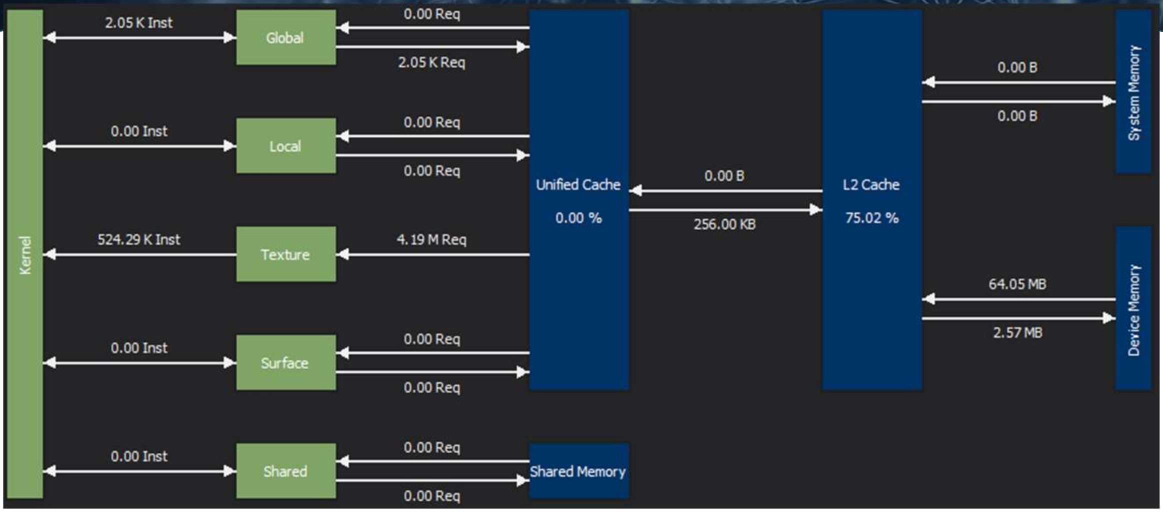
ColumnAfterColumn: Texture vs Global



global restrict (cudaMalloc) 270 GiB/s

was 464 GiB/s

ColumnAfterColumn: Texture vs Global

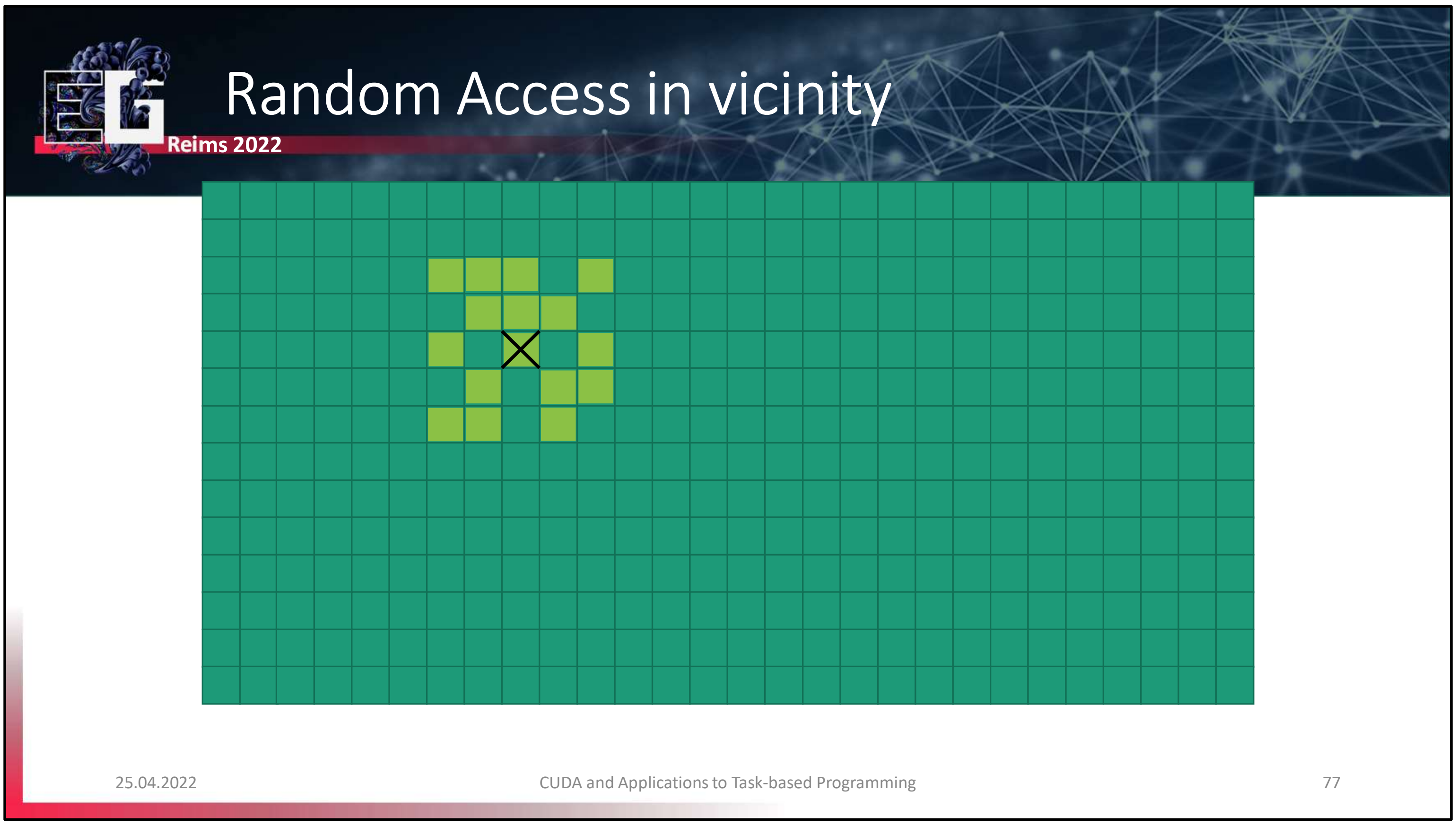


texture (cudaArray) 368 GiB/s

was 387 GiB/s

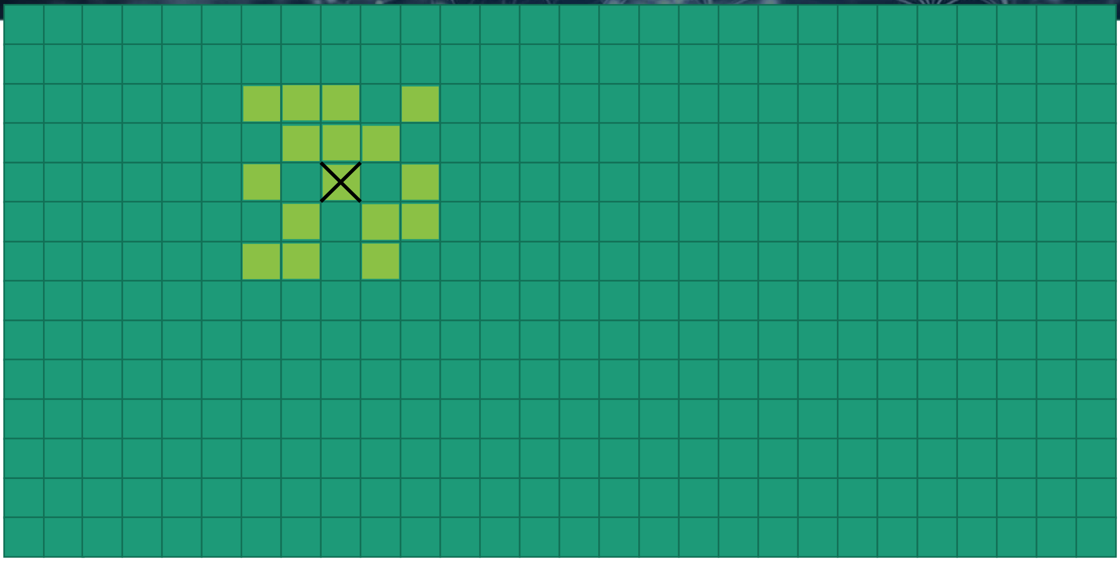
Texture vs Global: 1D Access

- row access is similarly efficient with all types of memory
  - virtually no cache usage
  - slighty better with linear memory layout vs textures
- column access significantly slower for global vs texture
  - texture only slightly slower than row access

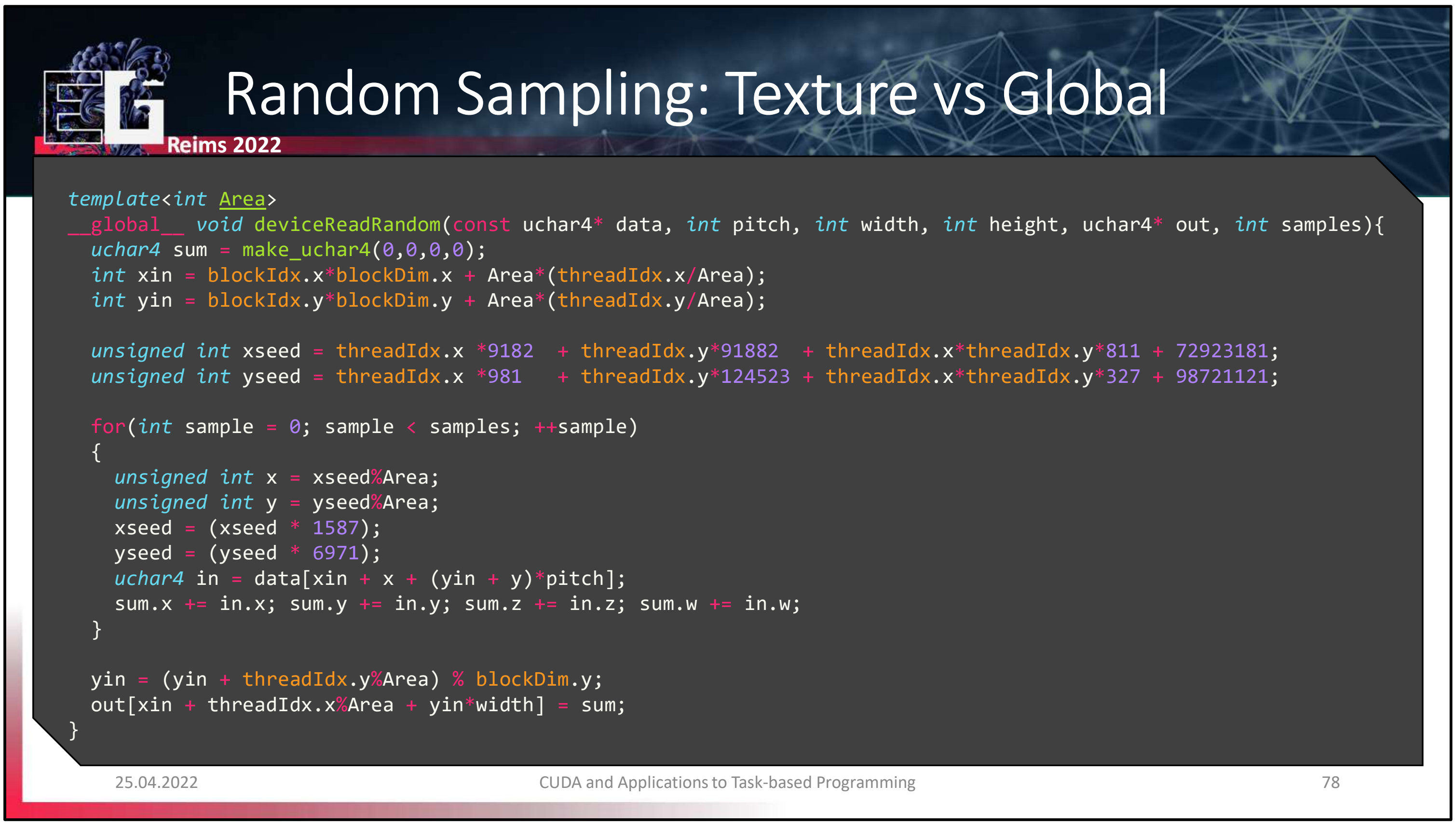


Random Access in vicinity

Reims 2022

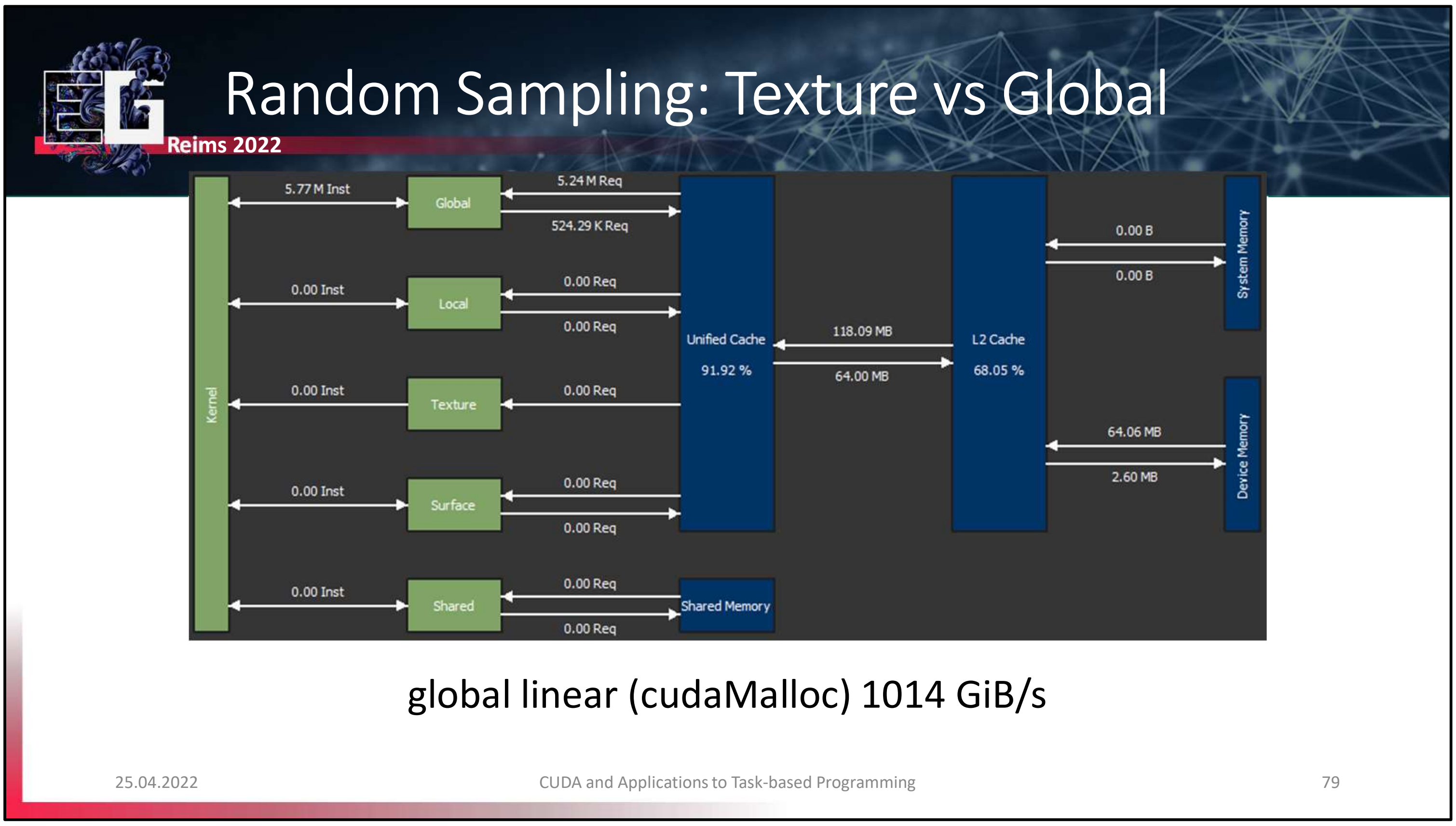


Finally, we look at an example where each thread sums up a number of randomly-chosen pixels within a certain neighborhood around each pixel of the image.



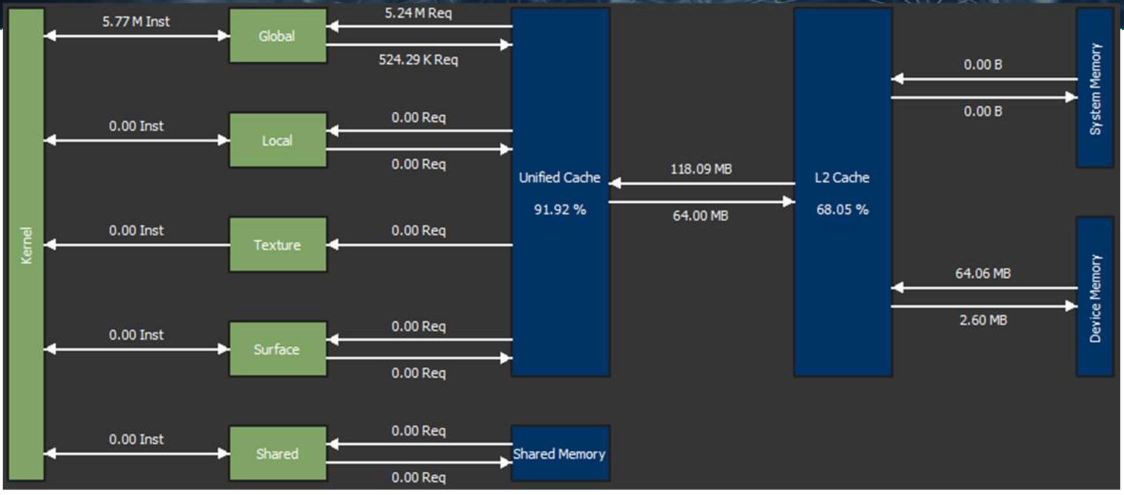
Random Sampling: Texture vs Global

```
template<int Area>
__global__ void deviceReadRandom(const uchar4* data, int pitch, int width, int height, uchar4* out, int samples){
  uchar4 sum = make_uchar4(0,0,0,0);
  int xin = blockIdx.x*blockDim.x + Area*(threadIdx.x/Area);
  int yin = blockIdx.y*blockDim.y + Area*(threadIdx.y/Area);

  unsigned int xseed = threadIdx.x *9182  + threadIdx.y*91882  + threadIdx.x*threadIdx.y*811 + 72923181;
  unsigned int yseed = threadIdx.x *981   + threadIdx.y*124523 + threadIdx.x*threadIdx.y*327 + 98721121;

  for(int sample = 0; sample < samples; ++sample)
  {
    unsigned int x = xseed%Area;
    unsigned int y = yseed%Area;
    xseed = (xseed * 1587);
    yseed = (yseed * 6971);
    uchar4 in = data[xin + x + (yin + y)*pitch];
    sum.x += in.x; sum.y += in.y; sum.z += in.z; sum.w += in.w;
  }

  yin = (yin + threadIdx.y%Area) % blockDim.y;
  out[xin + threadIdx.x%Area + yin*width] = sum;
}
```



Random Sampling: Texture vs Global

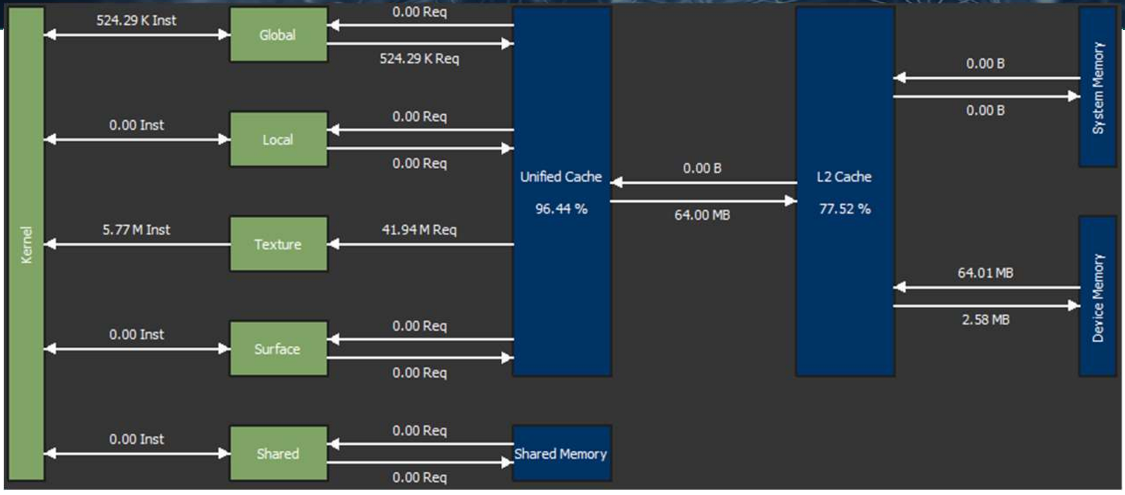
Reims 2022



global linear (cudaMalloc) 1014 GiB/s

As one would have expected, texture memory outperforms global memory in this example since the memory layout of texture data is optimized precisely for this kind of access pattern. The difference, however, is somewhat surprisingly small, which can be attributed to a very effective L1$ on modern GPUs.

Random Sampling: Texture vs Global



texture (cudaArray) 1170 GiB/s

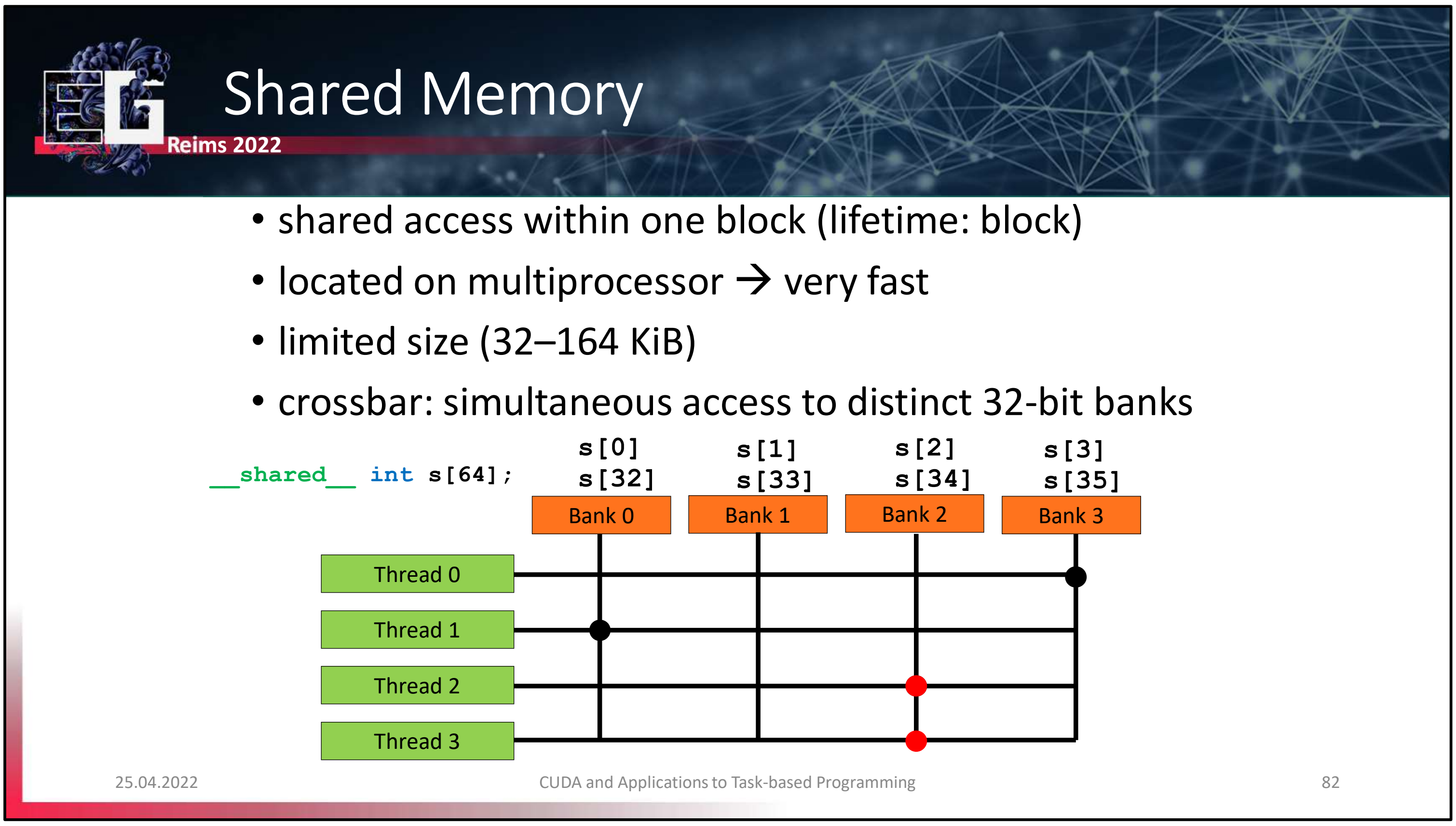
Reims 2022

Texture vs Global: Conclusion

- prior to Volta
  - Textures tend to perform at least as good, sometimes better
  - put less stress on L2 cache
  - L1 cache free for other tasks

- now
  - advanced Unified Cache (L1 + Tex)
  - Textures still perform best for spatially local access pattern
  - but can also be slower if access pattern and cache hits favor linear memory

- unique features: filtering, border handling, format conversion



Shared Memory

Reims 2022

- shared access within one block (lifetime: block)
- located on multiprocessor → very fast
- limited size (32–164 KiB)
- crossbar: simultaneous access to distinct 32-bit banks

```
__shared__ int s[64];
```

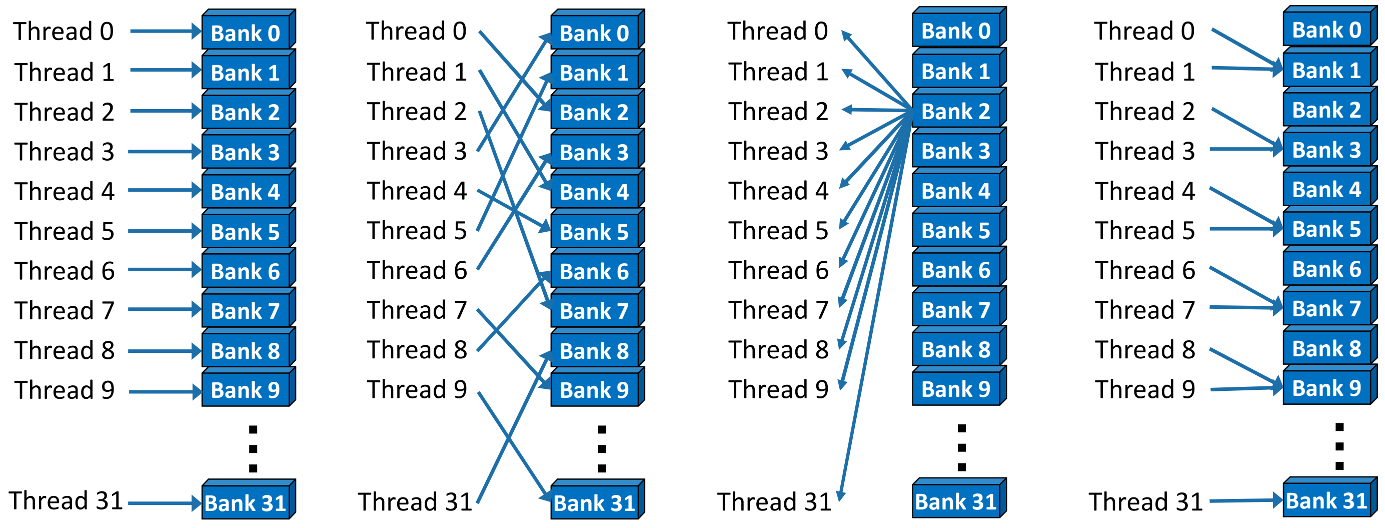
s[0]
s[32]
Bank 0
s[1]
s[33]
Bank 1
s[2]
s[34]
Bank 2
s[3]
s[35]
Bank 3
Thread 0
Thread 1
Thread 2
Thread 3

25.04.2022 CUDA and Applications to Task-based Programming 82

Shared Memory exposes the local on-chip memory that can be found on every SM to CUDA programs. Shared Memory is limited in size but very fast to access and shared by all threads within the same block, thus, facilitating low-latency local inter-thread communication.However, in order to utilize Shared Memory effectively, one must once again be careful about the memory access pattern. Shared memory is organized into 32 memory banks such that every 32 consecutive 4-byte elements are distributed among the 32 consecutive memory banks. Each memory bank can serve 4-byte loads and stores from and to the addresses it’s responsible for. The 32 threads of a warp can access shared memory through a crossbar connecting each thread to each memory bank. However, only one thread can be connected to the same memory bank at the same time. If two or more threads were to attempt to access addresses that are served by the same memory bank at the same time, their accesses have to be serialized, resulting in reduced performance. Such a situation where N threads are trying to access addresses served by the same memory bank at the same time is called an N-way bank conflict. One notable exception is the case where multiple threads try to read not just any address served by the same bank but all the same address. In this case, they can all be served at once by broadcasting the result.

Reims 2022

Shared Memory: Bank Conflicts

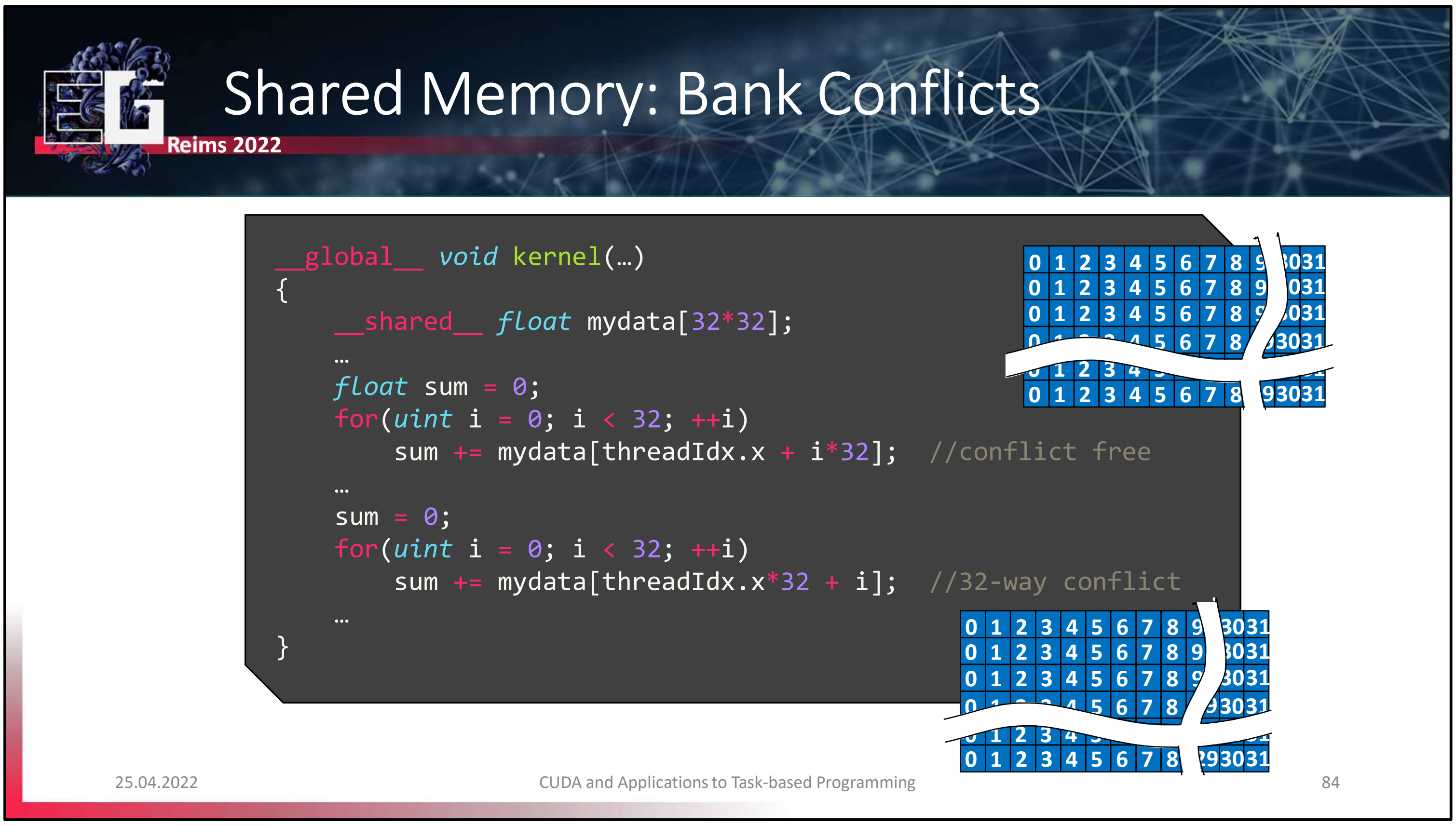


conflict free access

conflict free access

broadcast

two-way bank conflict (different words)



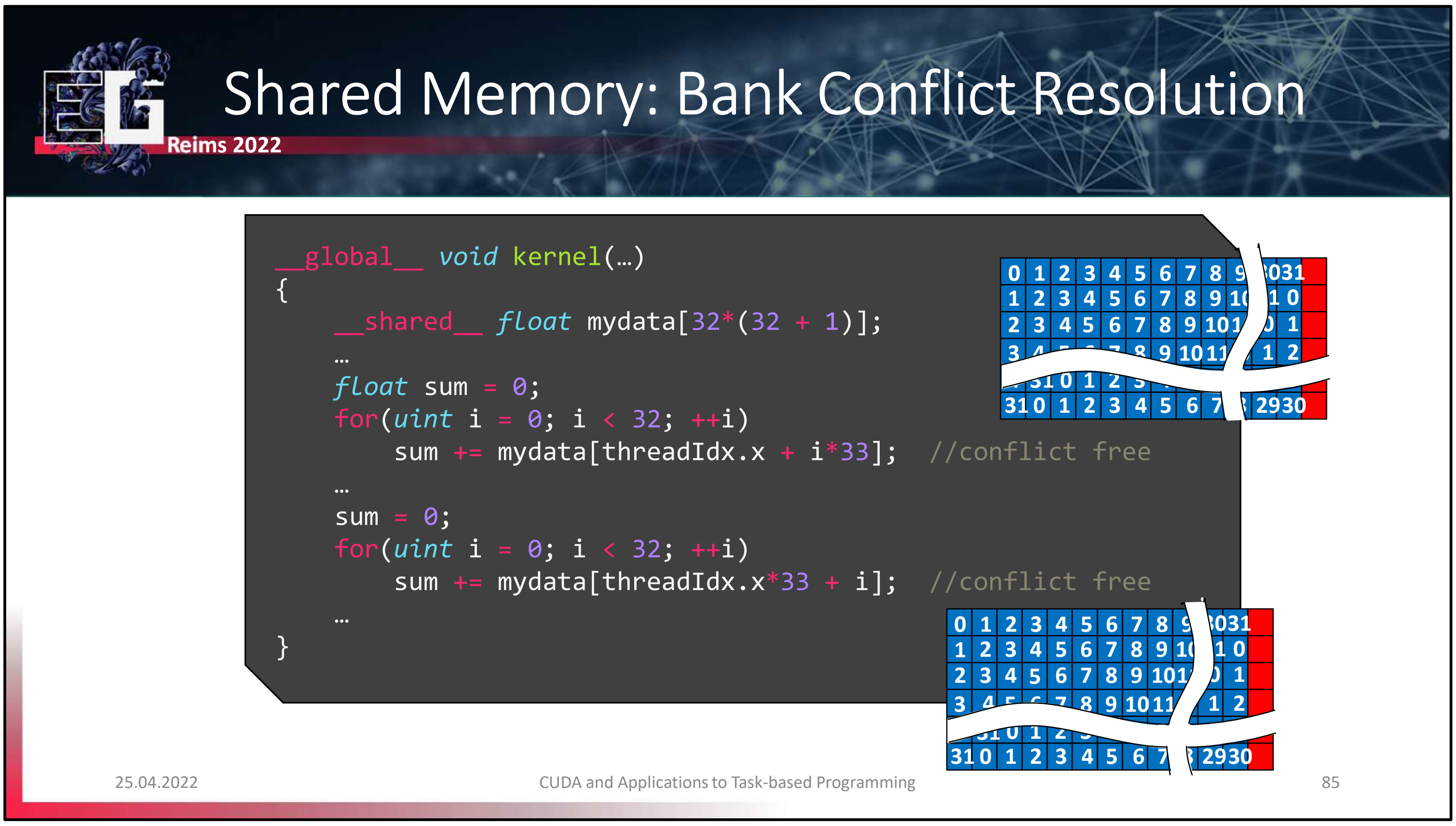
Shared Memory: Bank Conflicts

Reims 2022

```
__global__ void kernel(…)
{
    __shared__ float mydata[32*32];
    …
    float sum = 0;
    for(uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x + i*32];  //conflict free
    …
    sum = 0;
    for(uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x*32 + i];  //32-way conflict
    …
}
```

25.04.2022 CUDA and Applications to Task-based Programming 84

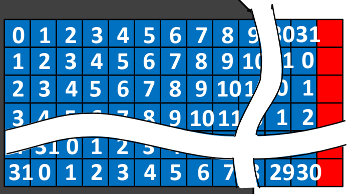
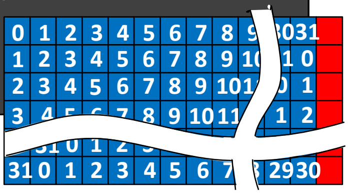
To illustrate the use of Shared Memory, we look at a simple example. At two points within its operation, the given kernel needs to compute the sum once over the columns and, at a later point, over the rows of a 32×32 element matrix stored in row-major layout. It does so using the 32 threads of a warp in order to compute the sum for each column/row in parallel. To compute the sum over the columns, the kernel proceeds row-by-row, each thread adding the element in the corresponding column to its running total. The access pattern in this case is such that each thread accesses an element served by a different bank. To compute the sum over the rows, the kernel proceeds column-by-column, each thread adding the element in the corresponding row to its running total. The access pattern in this case is such that the elements accessed by all threads are served by the same bank, resulting in the worst possible case of a 32-way bank conflict.



Shared Memory: Bank Conflict Resolution

Reims 2022

```
__global__ void kernel(…)
{
    __shared__ float mydata[32*(32 + 1)];
    …
    float sum = 0;
    for(uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x + i*33];  //conflict free
    …
    sum = 0;
    for(uint i = 0; i < 32; ++i)
        sum += mydata[threadIdx.x*33 + i];  //conflict free
    …
}
```

25.04.2022 CUDA and Applications to Task-based Programming 85

A trick to resolve this issue is to pad the matrix by adding a dummy column. As a result of this padding, the access pattern in both scenarios works out such that no bank conflicts arise at any point.

Shared Memory: Use Cases

Reims 2022

- inter-thread communication

```
__global__ void kernel(…)
{
    __shared__ bool run;
    run = true;
    while(run)
    {
        __syncthreads();
        if(found_it())
            run = false;
        __syncthreads();
    }
}
```

Reims 2022

Shared Memory: Use Cases

- inter-thread communication
- reduce global memory access → manual cache

```
__global__ void kernel(float* global_data, …)
{
    extern __shared__ float data[];
    uint linid = blockIdx.x*blockDim.x + threadIdx.x;
    //load
    data[threadIdx.x] = global_data[linid];
    __syncthreads();
    for(uint it = 0; it < max_it; ++it)
        calc_iteration(data); //calc
    __syncthreads();
    //write back
    global_data[linid] = data[threadIdx.x];
}
```

Shared Memory: Use Cases

- inter-thread communication
- reduce global memory access → manual cache
- adjust global memory access pattern

```
__global__ void transp(float* global_data, float* global_data2)
{
    extern __shared__ float data[];
    uint linid1 = blockIdx.x*32 + threadIdx.x + blockIdx.y*32*width;
    uint linid2 = blockIdx.x*32*width + threadIdx.x + blockIdx.y*32;
    for(uint i = 0; i < 32; ++i)
        data[threadIdx.x + i*33] = global_data[linid1 + i*width];
    __syncthreads();
    for(uint j = 0; j < 32; ++j)
        global_data2[linid2 + j*width] = data[threadIdx.x*33 + j] ;
}
```

Reims 2022

Shared Memory: Use Cases

- inter-thread communication
- reduce global memory access → manual cache
- adjust global memory access pattern
- indexed access

```
__global__ void kernel(…)
{
    uint mydata[8]; //will be spilled to local memory
    for(uint i = 0; i < 8; ++i)
        mydata[i] = complexfunc(i, threadIdx.x);
    uint res = 0;
    for(uint i = 0; i < 64; ++i)
        res += secondfunc(mydata[(threadIdx.x + i) % 8],
                          mydata[i*threadIdx.x % 8]);
}
```

Reims 2022

Shared Memory use cases

- inter-thread communication
- reduce global memory access → manual cache
- adjust global memory access pattern
- indexed access

```
__global__ void kernel(…)
{
    __shared__ uint allmydata[8*BlockSize];
    uint *mydata = allmydata + 8*threadIdx.x;
    for(uint i = 0; i < 8; ++i)
        mydata[i] = complexfunc(i, threadIdx.x);
    uint res = 0;
    for(uint i = 0; i < 64; ++i)
        res += secondfunc(mydata[(threadIdx.x + i) % 8],
                          mydata[i*threadIdx.x % 8]);
}
```

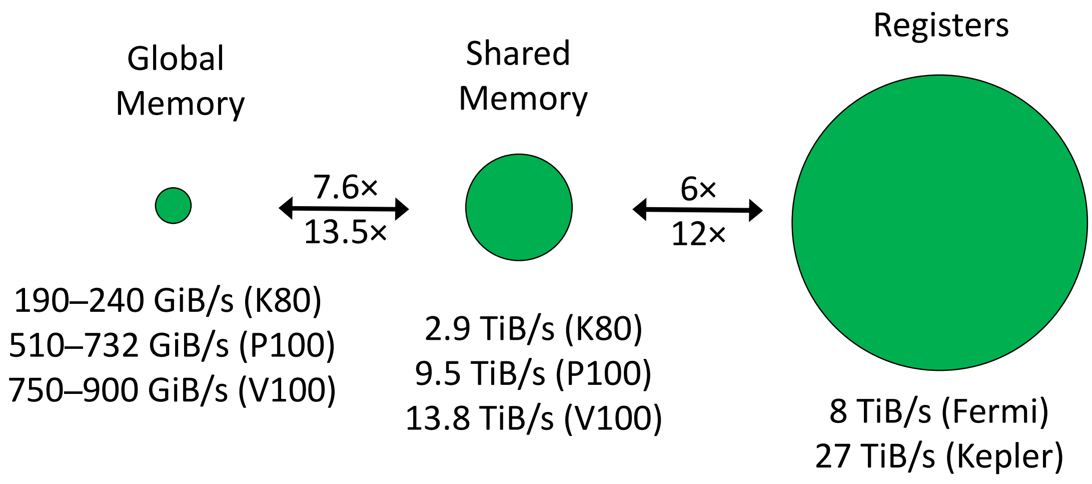
Shared Memory: Use Cases

- inter-thread communication
- reduce global memory access → manual cache
- adjust global memory access pattern
- indexed access
- combine costly operations

```
__global__ void kernel(uint *global_count, …)
{
    __shared__ uint blockcount;
    blockcount = 0;
    __syncthreads();
    uint myoffset = atomicAdd(&blockcount, myadd);
    __syncthreads();
    if(threadIdx.x == 0)
        blockcount = atomicAdd(global_count, blockcount);
    __syncthreads();
    myoffset += blockcount;
}
```

Reims 2022

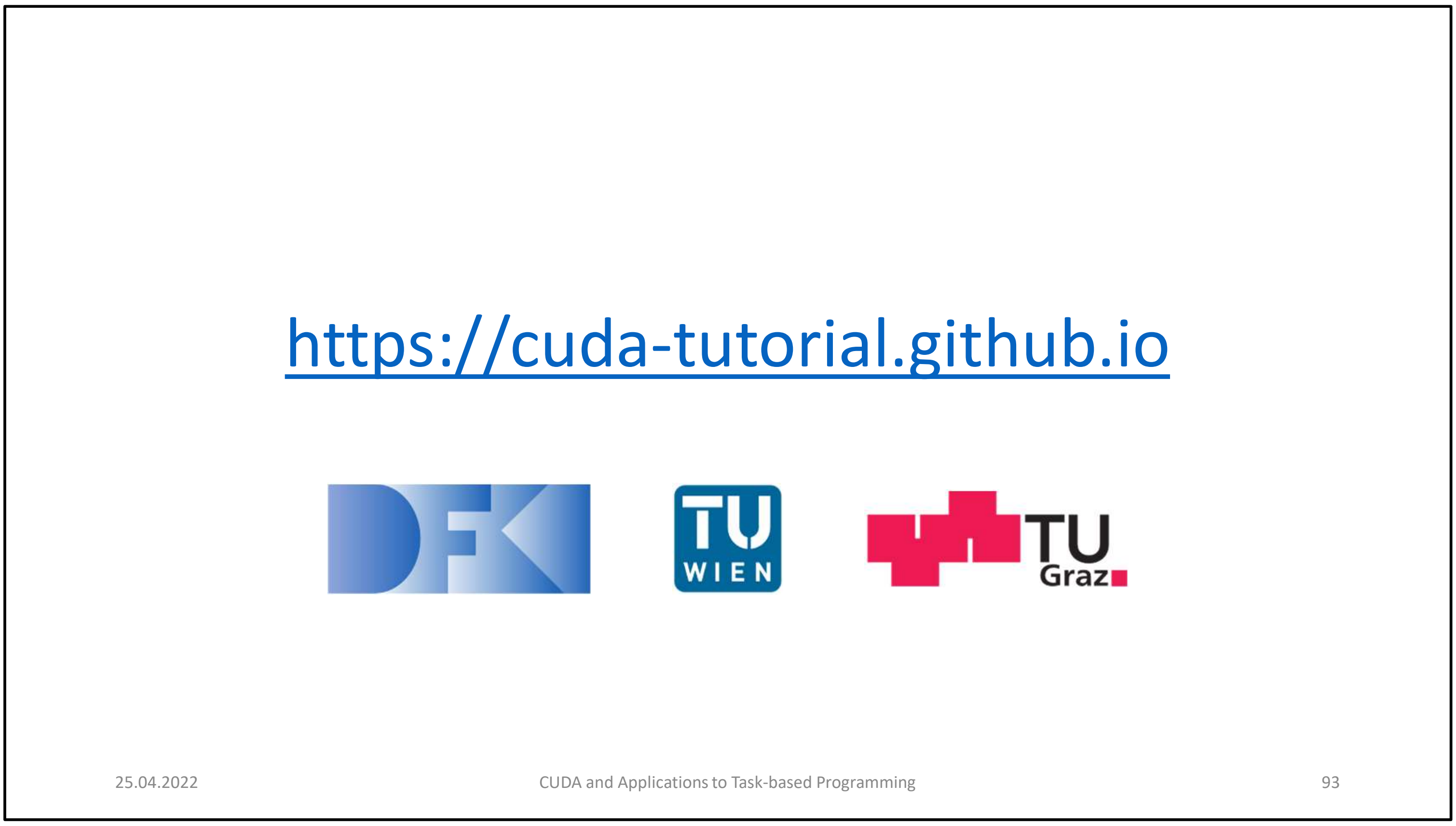
Registers vs Shared vs Global Memory



190–240 GiB/s (K80)
510–732 GiB/s (P100)
750–900 GiB/s (V100)

2.9 TiB/s (K80)
9.5 TiB/s (P100)
13.8 TiB/s (V100)

8 TiB/s (Fermi)
27 TiB/s (Kepler)



https://cuda-tutorial.github.io

Reims 2022

References

- NVIDIA, *CUDA Programming Guide*
- NVIDIA, *NVCC Documentation*
- NVIDIA, *PTX ISA*
- NVIDIA, *CUDA Binary Utilities*

- NVIDIA, *Ampere GA102 GPU Architecture Whitepaper*, 2020

- E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, *NVIDIA Tesla: A Unified Graphics and Computing Architecture*, in IEEE Micro, vol. 28, no. 2, pp. 39–55, 2008, doi: 10.1109/MM.2008.31.

- Z. Jia, M. Maggioni, B. Staiger, D. P. Scarpazza, *Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking*, arXiv, 2018, arXiv: 1804.06826

References

- B. W. Coon, J. R. Nickolls, J. E. Lindholm, S. D. Tzvetkov, *Structured programming control flow in a SIMD architecture*, 2007, US Patent 7877585B1
- B. W. Coon, J. E. Lindholm, P. C. Mills, J. R. Nickolls, *Processing an indirect branch instruction in a SIMD architecture*, 2006, US Patent 7761697B1
- B. W. Coon, J. E. Lindholm, *Systems and method for managing divergent threads in a SIMD architecture*, 2005, US Patent 8667256B1
- J. E. Lindholm, M. C. Shebanow, *Tree-based thread management*, 2014, US Patent 9830161B2