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 2021. 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.

Managed Memory

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);
}
```

Using Managed Memory

- CUDA's opt-in approach to unified, automatically managed memory
- Define static variables in `.cu` files with new CUDA `__managed__` keyword
- Allocate managed memory dynamically: `cudaMallocManaged`
- Supported since CC 3.0 with 64-bit OS

```
__managed__ int foo;

__global__ void kernel(int* bar)
{
    printf("%d %x\n", foo, *bar);
}

int main()
{
    foo = 42;
    int* bar;
    cudaMallocManaged(&bar, 4);
    *bar = 0xcaffe;
    kernel<<<1, 1>>>(bar);
    cudaDeviceSynchronize();
}
```


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.

Concurrent Access by CPU and GPUs

- If kernels and CPU execution overlap, both may access same memory
- Concurrent access supported since CC 6.0, but not guaranteed
  - Even Turing GPUs and newer may not support concurrent access
  - Before attempting it, must check property `concurrentManagedAccess`
- If not supported, developer must ensure that managed memory is not accessed by the CPU while the GPU is running kernels
  - Applies to all managed memory, regardless of whether the GPU accesses it
  - `cudaDeviceSynchronize` to secure access from the CPU

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.

Concurrent Access by CPU and GPUs

- Also applies if GPU uses different memory or no memory at all

```
__managed__ int x, y=2;

__global__ void kernel() {
    printf("%d\n", x);
}

int main() {
    kernel<<< 1, 1 >>>();
    y = 20;   // Error on some GPUs, all CC < 6.0
    cudaDeviceSynchronize();
    return 0;
}
```

```
__managed__ int x, y=2;

__global__ void kernel() {
    printf("%d\n", x);
}

int main() {
    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();
    y = 20;
    return 0;
}
```


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.

Concurrent Access with Streams

- Possible to associate given ranges of memory with streams / processors
- `cudaStreamAttachMemAsync`
- Access to a memory range given to:
  - `cudaMemAttachHost` (CPU)
  - `cudaMemAttachGlobal` (all streams)
  - `cudaMemAttachSingle` (one stream)

```
__managed__ int x = 42, y = 2;

__global__ void kernel() {
    printf("%d\n", x);
}

int main() {
    cudaStream_t s1;
    cudaStreamCreate(&s1);
    unsigned int acc = cudaMemAttachHost;
    cudaStreamAttachMemAsync(s1, &y, 4, acc);
    kernel <<<1, 1 >>> ();
    y = 20;
    cudaDeviceSynchronize();
    return 0;
}
```


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.

Use Case: Simpler Multi-Threaded Access

- Multiple CPU threads with managed access
- Default stream would cause synchronization
- With streams, CPU threads can control exclusive access

```
void run_task(int *in, int *out, int length)
{
    int *data;
    cudaMallocManaged((void **)&data, length, cudaMemAttachHost);
    cudaStreamAttachMemAsync(stream, data);
    cudaStreamSynchronize(stream);

    for(int i=0; i<N; i++) {
        transform<<< 100, 256, 0, stream >>>(in, data, length);
        cudaStreamSynchronize(stream);
        host_process(data, length);
        convert<<< 100, 256, 0, stream >>>(out, data, length);
    }
}
```


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.

Tuning Managed Memory Performance

- Several issues that programs should consider with managed memory
  - Avoid excessive faulting: can cause data migration and page table updates
  - Keep data close to accessing processor: decrease latencies on access
  - Memory thrashing: memory is constantly migrated back and forth
- Developers can assist memory management with performance hints
  - Migrate a range of data to a specific location and map it to processor's page tables within a given stream with `cudaMemPrefetchAsync`
  - Additionally, can provide hints on the usage of data with `cudaMemAdvise`: preferred location, devices on which it should stay mapped, mostly read

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.

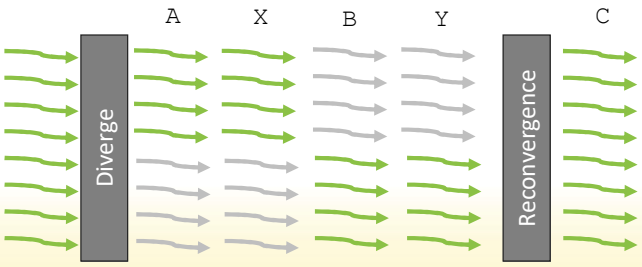
ITS – Opportunities & Pitfalls

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

Legacy Thread Scheduling

- Diverged threads will try to reach convergence point before switching
- Cannot get past convergence point until all involved threads arrive

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

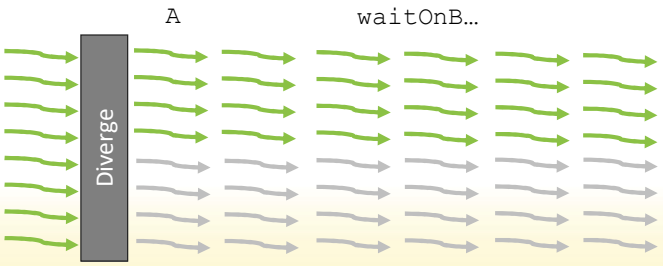


Let's quickly recap legacy scheduling. Consider for instance the branch given based on the thread ID. The lower four threads will enter one branch, the remaining threads will enter the other. However, once a branch has been chosen, it must be completed before the other branch can begin because the warp 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.

Legacy Thread Scheduling

- Scheduling dictates what algorithms are and aren't possible
- Actually, quite easy to get a deadlock between threads within a warp

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



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.

Independent Thread Scheduling (ITS)

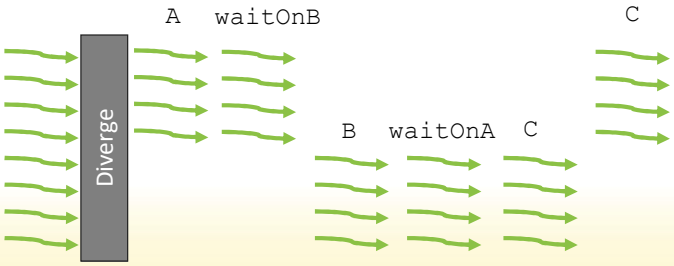
- Two registers reserved, each thread gets its own program counter
- Individual threads can now be at different points in the program
- Warp scheduler can (and does) advance warps on all possible fronts
  - Guaranteed progress for all resident threads
  - Enables thread-safe implementation of spinlocks, starvation-free algorithms
- Threads in a warp still can only do one instruction at a time

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)

- 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.

Enabling/Disabling ITS

- Currently, GPUs can still switch between legacy scheduling and ITS
- Compiler flags to enable ITS
  - `-arch=compute_70 -code=sm_70` for Volta
  - `-arch=compute_75 -code=sm_75` for Turing
- Compiler flags to disable ITS
  - `-arch=compute_60 -code=sm_70` for Volta
  - `-arch=compute_60 -code=sm_75` for Turing

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.

Use Case: Mutual Exclusion (Busy Wait)

- Minimalistic busy-wait loop implementation, run on Turing
- `threadfence` acts as barrier, can realize an acquire/release pattern in CUDA
- Hangs with ITS disabled, works with ITS enabled

```
__device__ int lock = 0;

__global__ void incrementCounter()
{
    while (atomicCAS(&lock, 0, 1) != 0);
    __threadfence();
    count++;
    __threadfence();
    atomicExch(&lock, 0);
}

int main()
{
    incrementCounter<<<256, 256>>>();
    return 0;
}
```


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.

Limitations and Caveats of ITS

- No amount of hardware scheduling can save you from live lock
- Only guaranteed progress for resident warps!
  - Threads will wait forever if their progress depends on non-resident warp
  - Number of concurrently resident warps can be retrieved with driver API
  - `cuOccupancyMaxActiveBlocksPerMultiprocessor` × #SMs
  - Computed based on resource requirements of kernel and hardware specs
- More care must be taken to ensure SIMD behavior of warps!

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.

ITS and the Importance of `__syncwarp`

- The concept of threads progressing in strict lockstep no longer applies
- `__syncwarp` is used to explicitly force synchronization, reconvergence
- Force executing threads to wait until all in mask hit **a** `__syncwarp`
  - Volta+: group of threads can synchronize from different points in the program
  - Masks of the called `__syncwarp` must match
- Extremely important for porting code to Volta and newer architectures!

__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.

Warp Synchronization (e.g., Reduction)

```
__shared__ shmem[blockDim.x];
unsigned tid = threadIdx.x;

shmem[tid] += shmem[tid+16];
__syncwarp();
shmem[tid] += shmem[tid+8];
__syncwarp();
shmem[tid] += shmem[tid+4];
__syncwarp();
shmem[tid] += shmem[tid+2];
__syncwarp();
shmem[tid] += shmem[tid+1];
__syncwarp();
```

```
__shared__ shmem[blockDim.x];
unsigned tid = threadIdx.x;
int v = shmem[tid];

v += shmem[tid+16]; __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+8];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+4];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+2];  __syncwarp();
shmem[tid] = v;     __syncwarp();
v += shmem[tid+1];  __syncwarp();
shmem[tid] = v;
```

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.

Accessing Tensor Cores

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.

Tensor Cores

- Volta architecture has prominently introduced **Tensor Cores**
  - Programmable Matrix-Multiply-and-Accumulate (MMA)
  - E.g., Titan V / Tesla V100 contain 8 Tensor Cores per SM

- Tensor core operates on matrices: $\mathrm{A}(\mathrm{M} \times \mathrm{K}), \mathrm{B}(\mathrm{K} \times \mathrm{N}), \mathrm{C}(\mathrm{M} \times \mathrm{N})$
  - $4 \times 4 \times 4$ $(\mathrm{M} \times \mathrm{N} \times \mathrm{K})$ matrix processing array, performs $D = A \cdot B + C$

$$\mathbf{D} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{bmatrix} + \begin{bmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{bmatrix}$$

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.

Tensor Cores

- Easily accessed through libraries
  - Primarily via TensorRT, cuDNN and cuBLAS
  - Recommended for highest performance in most use cases

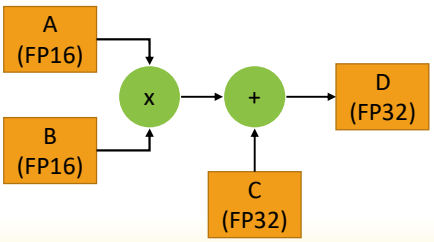
- Also exposed directly in CUDA kernel code
  - Exact data layout can be treated as blackbox, low-level definitions in CUDA 11
  - No specific instructions to be performed individually per thread
  - Warp matrix functions exposed to developers via `mma.h` header
  - Threads in a warp work together to collaboratively execute tensor operations
  - Each warp must uniformly perform the same `nvcuda::wmma` instructions

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

- 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. $16 \times 16 \times 16$ 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.

Using Tensor Cores in CUDA

```
// Contains section of a matrix distributed across all threads in warp
template<typename Use, int m, int n, int k, typename T, typename Layout=void> class fragment;

// Waits until all warps are at load matrix and then loads matrix
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t layout);

// Waits until all warps are at store matrix and then stores matrix
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t layout);

// Fill fragment with constant value v
void fill_fragment(fragment<...> &a, const T& v);

// Perform warp-synchronous matrix multiply-accumulate d = a*b + c
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b,
const fragment<...> &c, bool satf=false);
```


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)

Multiplying two 16 × 16 Matrices

```cpp
using namespace nvcuda;

__global__ void wmma_example(half* a, half* b, float* c)
{
    // Declare the fragments
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::col_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> acc_frag;
    wmma::fill_fragment(acc_frag, 0.0f);
    // Load the inputs
    wmma::load_matrix_sync(a_frag, a, 16);
    wmma::load_matrix_sync(b_frag, b, 16);
    // Perform the matrix multiplication
    wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
    // Store the output
    wmma::store_matrix_sync(c, acc_frag, 16, wmma::mem_col_major);
}
```

Inputs

Output

Load/init input/output

Magic!

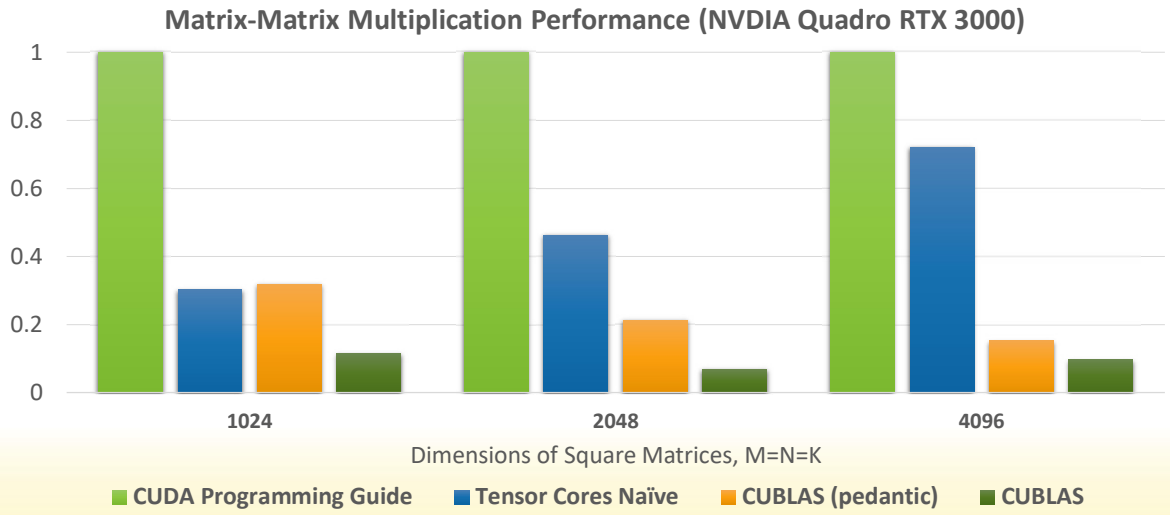
Store output

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

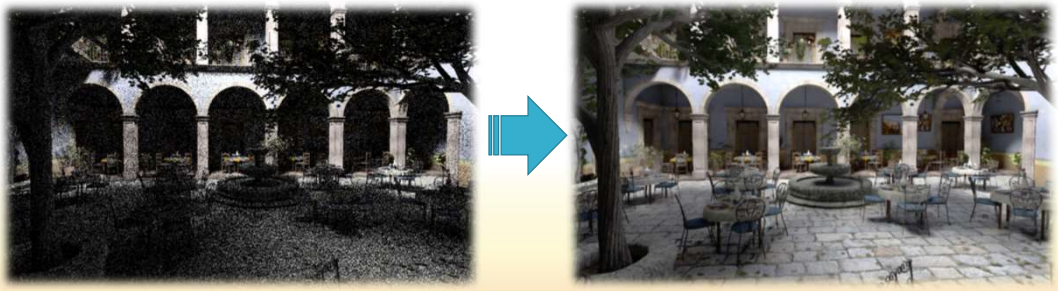
- Tensor cores are now used whenever possible by default
  - Can still be disabled using `CUBLAS_PEDANTIC_MATH`
  - `CUBLAS_TENSOR_OP_MATH` will soon be deprecated
- Copy matrix data from CPU to GPU (`cudaMemcpy` suffices)
- `cublasGemmEx(cublasHandle_t handle, cublasOperation_t transa, cublasOperation_t 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



Use Case: Denoising with CNNs

- Partial path-traced (1spp) results can be reconstructed using CNNs
- TensorRT enables directly using CUDA resources as input
- Sampling, inference, cleanup and visualization all on-chip
- Used, e.g., by Tatzgern et al. for “Stochastic Substitute Trees”[1]



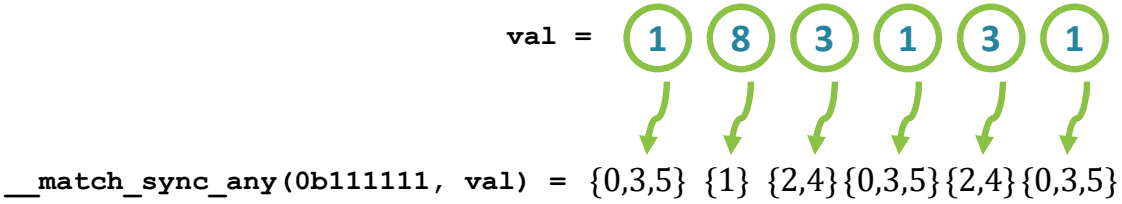
Although knowing the exact functionality of tensor cores is interesting, a much more practical approach for the most common use cases, like machine learning, is to use the available libraries, like TensorRT. The corresponding solutions support the loading and inference with network layouts in common 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.

New Warp-Level Primitives

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.

Match and Reduce

- `__match_sync` (new since compute capability 7.0, Volta)
  - Submit a value, return bitmask with threads that submitted the same value
  - E.g., identify threads that have the same value in a particular register



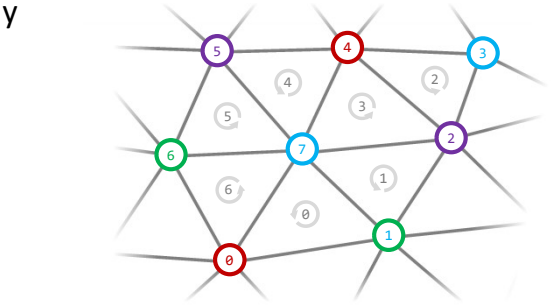
- `__reduce_sync` (new since compute capability 8.0, Ampere)
  - Perform warp-wide reduction (addition, OR, XOR, MIN, …)

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

- 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!



…, 0,1,7, 7,1,2, 3,4,2, 2,4,7, 7,4,5, 7,5,6, 7,6,0, …

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.

Use Case: Parallel Reduction Final Stage

```
__global__ void reduceSharedShuffle(const float* input, float* result, int N)
{
    …
    x = data[threadIdx.x];
    if (threadIdx.x < 32)
    {


        x = __reduce_add_sync(0xFFFFFFFF, x);


    }
    if (threadIdx.x == 0)
        atomicAdd(result, x);
}
```


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.

Opportunistic Warp-Level Programming

- Due to ITS, threads no longer progress in lockstep
- At any point of a kernel, an arbitrary set of threads may be active
- New primitive `__activemask` returns a bitmask of current threads
  - Does not include warp synchronization!
  - Threads can simply let each other know if they are at the same instruction
- Enables set of threads to quickly collaborate anywhere in the program

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.

Use Case: Aggreate Atomics in Warp

- Use `__activemask` to combine increments before writing data

```
{
    unsigned int writemask = __activemask();
    unsigned int total = __popc(writemask);
    unsigned int prefix = __popc(writemask & __lanemask_lt());
    int elected_lane = __ffs(writemask) - 1;
    int base_offset = 0;
    if (prefix == 0) {
        base_offset = atomicAdd(p, total);
    }
    base_offset = __shfl_sync(writemask, base_offset, elected_lane);
    int thread_offset = prefix + base_offset;
    return thread_offset;
}
```

Which threads are active?

How many are there?

Which one am I?

Who is the leader?

Thread 0 adds atomically to get offset

Write data there!

Share offset with other threads

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.

Outlook

- Opportunistic programming depends on correct use of mask
  - Use of `__activemask` is easy to get wrong
  - Due to ITS, can result in computation of incomplete results
- The list of special functions to remember is getting longer
  - Increasing number of warp-level primitives to remember and apply
  - Raise performance, but are often restricted to specific architectures
  - Complicates generation of portable code
- Better: use cooperative thread groups (also available since CUDA 9.0)

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.

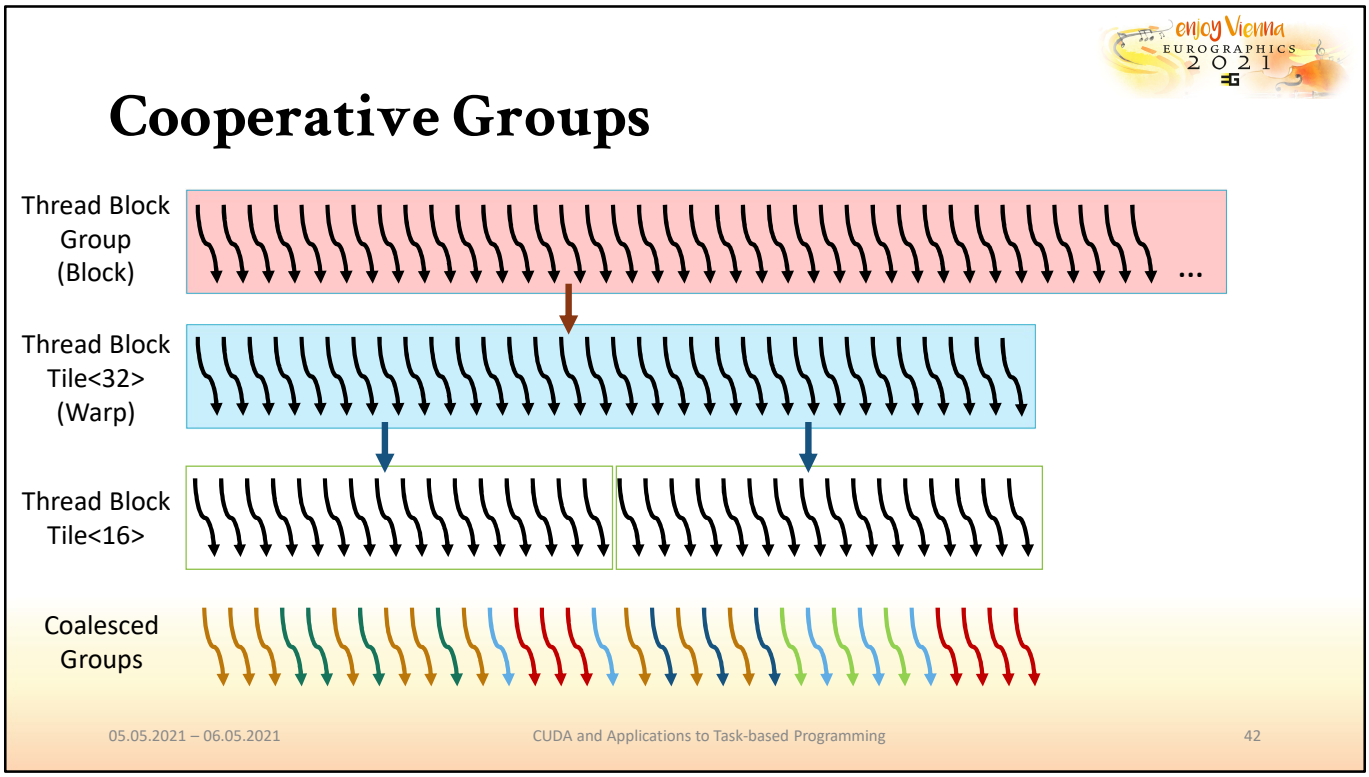
Cooperative Groups

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

Cooperative Groups

- To best exploit the GPU, threads may want to cooperate at any scope
  - A few threads
  - An entire warp
  - An entire block
  - All blocks in a grid
- Cooperative groups hide the details of collaboration between threads
  - Efficient cooperation between threads in block/warp via primitives
  - Require careful handling, correct masking, controlled synchronization
  - Cooperative groups simplify the code structure, abstract low-level commands

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).

Cooperative Groups

- Not built-in, extra features included via `cooperative_groups.h`

- Cooperative groups functionalities include:
  - Data structures and types for groups of different sizes
  - Methods to create new groups from implicit scopes or larger groups
  - Methods to synchronize threads in a group
  - Algorithms to collaboratively perform more complex operations
  - Operations to inspect group properties
    - Total group size
    - Thread ID within a given group

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.

Creating Cooperative Groups

```
// Obtain a group for the current thread block
auto threadblock = cooperative_groups::this_thread_block();
// Obtain a group for each warp in the thread block
auto warpgroup = cooperative_groups::tiled_partition<32>(threadblock);
// Obtain a group for each warp in the thread block
auto subwarp16 = cooperative_groups::tiled_partition<16>(threadblock);
// Obtain a group for all currently coalesced threads in the warp
auto active = cooperative_groups::coalesced_threads();

// Thread block groups can sync, reflect
threadblock.sync();
printf("Size: %d Id: %d\n", threadblock.size(), threadblock.thread_rank());

// Explicit groups are smaller than warps - can use warp-level primitives!
uint answer = active.ballot(foo == 42);
uint neighbor_answer = active.shfl_down(answer, 1);
```


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.

Use Case: Updating Reduction Final Stage

- Cooperative groups also provide reduction functions on CC < 8.0

```
__global__ void reduceSharedShuffle(const float* input, float* result, int N)
{
    …
    auto warp = cooperative_groups::tiled_partition<32>(threadblock);
    if (warp.meta_group_rank() == 0) // First warp group only
    {
        int warpLane = warp.thread_rank();
        float v = values[warpLane] + values[warpLane + 32];
        v = cooperative_groups::reduce(warp, v, cooperative_groups::plus<float>());
        if (warpLane == 0)
            atomicAdd(&result, v);
    }
}
```


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.

Use Case: Opportunistic Group Creation

- Revisit aggregation of atomic increments with warp-level primitives

```
{
    cg::coalesced_group g = cg::coalesced_threads();
    int prev;
    if (g.thread_rank() == 0)
    {
        prev = atomicAdd(p, g.size());
    }
    prev = g.thread_rank() + g.shfl(prev, 0);
    return prev;
}
```


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.

CUDA Standard Library libcu++

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

A Unified Standard Library for CUDA

- Previously, thrust to use `std::`-like containers, sorting, scanning…
- libcu++ brings the functionality of the standard library to the **device**
- Incremental integration of features (`chrono`, `complex`, `atomic`, …)
- Introduce two new namespaces that may be used on host **and** device
  - `cuda::std::` for standard API functionality according to specification
  - `cuda::` for extended features to exploit device-side optimization

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.

Time, Numerics and Utility

- `cuda::std::chrono`: provides system and high-resolution clock
  - May be used on either CPU and GPU, but using different system clocks: `%globaltimer` (PTX) for device, `gettimeofday` or equivalent on CPU
  - Logical discrepancies between readouts may occur (accepted by standard)
- Numerics library: ratios, complex numbers, cstdint, cfloat and climits
- Utility library: tuples, pairs, functional and version

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.

Memory Coherency Model Recap

- We can enforce memory coherency with basic barriers…
- But the PTX ISA defines common coherency model

```
__device__ void signal_flag(volatile int& flag)
{
    __threadfence();
    flag = 1;
}

__device__ void poll_flag_then_read(volatile int& flag, int& data)
{
    while (flag != 1);
    __threadfence();
    return data;
}
```

- A lot of effort spent on enforcing coherent memory model: since Volta, PTX exposes `acquire, release, relaxed, acq_rel,…!`

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.

libcu++ Atomics

- `cuda::std::atomic` and `cuda::atomic` expose most parts of the coherent memory model through atomic variables and operations
- Support for `std::atomic`, extensions to exploit device peculiarities
- Maps instructions of `std::` library to underlying PTX commands
- Reduce code complexity, improve code reuse, avoid common pitfalls

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.

Use Case: Flags with libcu++ Atomics

```
__host__ __device__ void signal_flag(atomic<bool>& flag) { // Works on CPU and GPU!
    flag = true;
}

__host__ __device__ int poll_flag_then_read(atomic<bool>& flag, int& data) {
    while (flag != 1);
    return data;
}

__host__ __device__ void signal_flag(atomic<bool>& flag) {
    flag.store(true, memory_order_release);
    flag.notify_all();
}

__host__ __device__ int poll_flag_then_read(atomic<bool>& flag, int& data) {
    flag.wait(false, memory_order_acquire);
    return data;
}
```


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

- 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.

libcu++ Caveats

- When porting, `std::` needs more verbose code for same behavior
  - E.g., atomics default to strongest memory coherence (sequentially consistent)
  - Also, default scope of synchronization primitives is `system` (host+device)

```
// Before
__device__ void increment_old(int* val)
{
    atomicAdd(val, 1);
}
// Now
__device__ void increment_new(cuda::atomic<int, cuda::thread_scope_block>* val)
{
    val->fetch_add(1, cuda::memory_order_relaxed);
}
```


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).

libcu++ Caveats

- Expect very different compilation results!
  - If you frequently inspect your code, this may need some getting used to
  - Minor differences, e.g., atomic operations may not convert to reductions

```
// Before
__device__ void increment_old(int* val)
{
    atomicAdd(val, 1);
}
// Now
__device__ void increment_new(cuda::atomic<int, cuda::thread_scope_device>* val)
{
    val->fetch_add(1, cuda::memory_order_relaxed);
}
```

RED.E.ADD.STRONG.GPU [R2.64], R5

ATOM.E.ADD.STRONG.GPU PT, RZ, [R2.64], R5

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

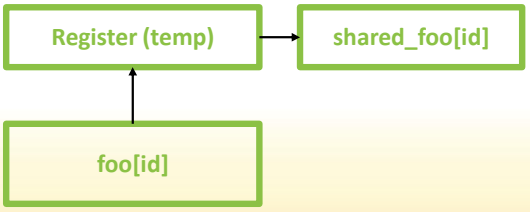
- Features are still being added incrementally
  - Key motivator: expose memory coherency model through `std::atomic`
  - Aim to improve portability, but majority of `std::` features still missing
  - A full replica of the standard library may not be practical or even feasible
  - Cannot just copy existing 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: Asynchronous Data Copies

- First step in many algorithms is copying data from global to shared
- Until now, no direct line of communication between the two
  - Data has to move via intermediate register, visible after `__syncthreads`
  - Ampere architecture offers hardware acceleration for this type of transfer

```
__global__ void kernel(int* foo)
{
    __shared__ int shared_foo[256];
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    shared_foo[id] = foo[i];
     __syncthreads();
    ...
}
```

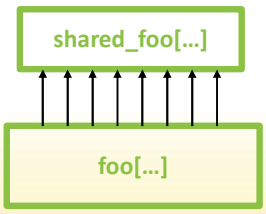


Libcu++ also includes definitions for CUDA barriers, which largely mimic the behavior of std::barrier types. These become important to exploit, e.g., a new feature of the Ampere architecture for efficiently transferring data from global to shared memory. Until now, such transfers, which are very common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

Use Case: Asynchronous Data Copies

- Transfer directly from global memory to shared, no register needed
- May synchronize with the availability of data only at a later time
  - Copy operation can be tied to a barrier
  - Data becomes visible eventually when threads wait on barrier

```
…
__shared__ int shared_foo[256];
__shared__ cuda::barrier<cuda::thread_scope::thread_scope_block> barrier;
…
auto block = cooperative_groups::this_thread_block();
size_t blockID = block.group_index().x * block.size();
cuda::memcpy_async(block, shared_foo, foo + blockID, 4 * 256, barrier);
barrier.arrive_and_wait();
…
```



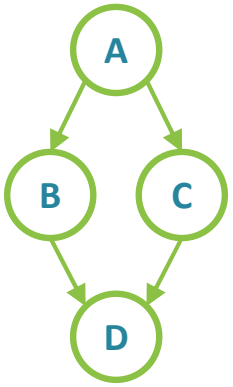
With libcu++, we can use barriers and the new cooperative memcpy_async functionality, which enables us to kick off an asynchronous copy of data from global to shared memory and, at some later point in the program, wait for that transfer to finish before progressing. The true benefit of this new functionality, 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.

CUDA Graph API

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

CUDA Graphs

- Many HPC applications build on iterative structure
  - Work submitted for every iteration
  - Repetitive in nature
  - E.g., physics simulations, learning or inference
- Modeling CUDA applications as graphs
  - Typical HPC applications are strongly pipelined
  - Series of stages, e.g., memory copies, kernel launches, …
  - Connected by dependencies
  - Often don't change frequently or not at all



Workflow Graph

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.

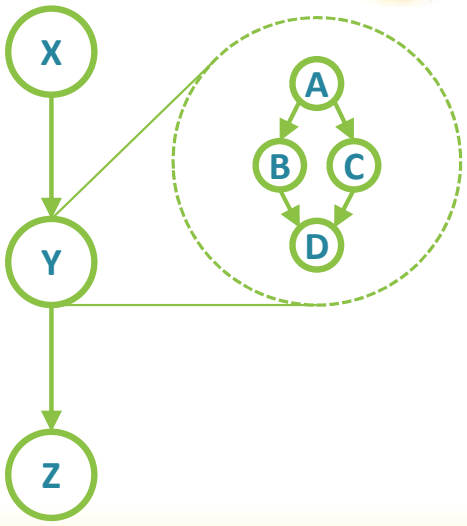
Benefits

- Overhead of CUDA operations can be significant
  - CUDA graphs allow to define or record execution ahead of time
  - Reuse same launch schedule many times
  - Separation of definition and execution reduces overall overhead
- Given a clearly defined schedule, driver can make optimizations
  - As whole workflow is visible, including
    - Kernel execution
    - CPU-side functions
    - Data movement
    - …

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

- 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.

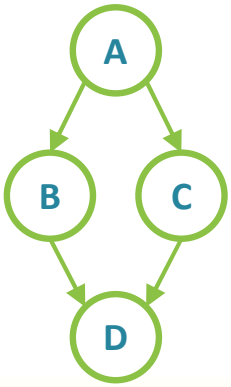
Create CUDA Graph from Scratch

```
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.

Record Existing CUDA Code as Graph

```
if (!recorded)
{
      // Define a graph and record CUDA instructions
      cudaGraphCreate(&graph);
      cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
      // Call your 100 kernels with unchanging parameters
      for(int i=0; i<100; i++)
            iterationKernel<<< …, stream >>>(i)
      // End capture and make graph executable
      cudaStreamEndCapture(stream, &graph);
      cudaGraphInstantiate(&instance, graph, 0, 0, 0);
      recorded = true;
}
else
      cudaGraphLaunch(instance, stream);
```


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.

Streams and Graph Dependencies

- When constructing graphs from scratch, no dependencies assumed
  - Need to manually add them (compare Vulkan/DX12)
- When recording existing code, standard CUDA dependencies apply
  - Events are assumed to depend on previous events in the same stream (strict!)
  - No dependencies across different recorded streams in the same graph
- It is possible to record multiple streams into the same CUDA graph
  - However, only one stream, the „origin“ stream, must start the recording
  - To capture other streams, add dependencies on origin (e.g., event waits)

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.

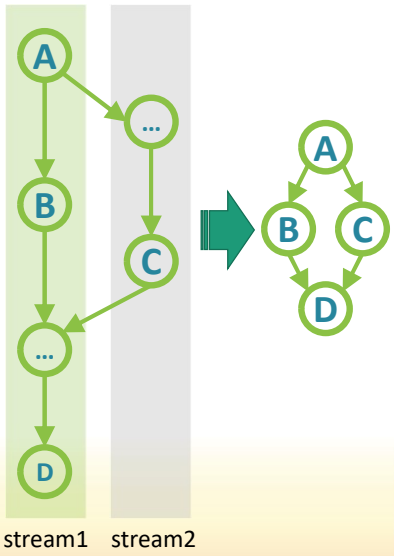
Example

```
// 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…
```

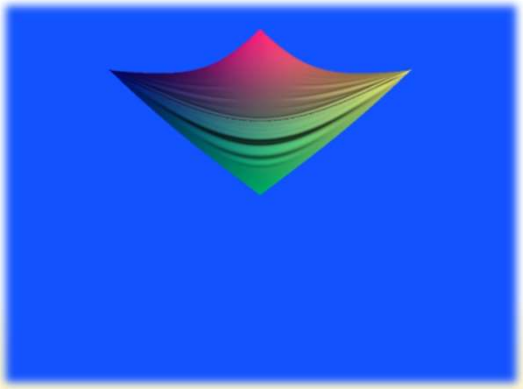
Event required to initiate recording another stream



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

- 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%.

Set-Aside L2 Cache

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

Controlling the Residency of Data in L2

- Data read from global memory have different access frequency
  - Data that is accessed frequently – persistent
  - Data that is accessed rarely (perhaps only once) – streaming
- For best performance, L2 should ensure persistent data remains
  - Fewer accesses to slower global memory
  - Impossible to predict, L2 behavior is reactive, eviction randomized
- With CUDA 11 and CC 8.0+, it becomes possible to define a set-aside region of the L2 cache that can be freely managed by the developer

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.

Set-Aside L2 Cache Region

- Define desired set-aside region (must be smaller than total L2 size)
  - `cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize,…)`
  - Size must be less than `persistingL2CacheMaxSize` limit

- Once defined, set-aside region can be associated with data

```
cudaStreamAttrValue attrib;      // Stream level attributes data structure
cudaAccessPolicyWindow apw;
apw.base_ptr = reinterpret_cast<void*>(data); // Global Memory data pointer
apw.num_bytes = window_size;                  // Number of bytes for persistence
apw.hitRatio = 0.6;                           // Hint for cache hit ratio
apw.hitProp = cudaAccessPropertyPersisting;   // Persistence Property
apw.missProp = cudaAccessPropertyStreaming;
attrib.accessPolicyWindow = apw;
cudaStreamSetAttribute(s1, cudaStreamAttributeAccessPolicyWindow, &attrib);
```


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.

Set-Aside L2 Cache Region Access Policy

- `hitRatio`: Portion of given data that will receive hit/miss property
  - E.g., 32 KB window size, 50% hit ratio:
    - 16 KB (random) will receive property `hitProp`
    - Remaining 16 KB will receive property `missProp`

- `hitProp/missProp`: What happens in case of a hit/miss
  - `cudaAccessPropertyStream` – data less likely to remain in L2 cache
  - `cudaAccessPropertyPersisting` – data more likely to remain in L2
  - `cudaAccessPropertyNormal` – restore usual, „normal“ L2 behavior

  (also important to evict cache lines from earlier kernels that may still remain!)

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.

Misconceptions and Hints

"It is better to know nothing than to know what ain't so" – Josh Billings

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.

How the Tables Have Turned

- Shared memory vs caching: caches are becoming more effective
  - Then: it usually paid off to use shared memory for manually managed “cache”
  - Now: L1 and L2 more effective, forced shared memory can hurt performance
- Texture memory vs global memory: performance is equalized
  - Then: recommended to use texture memory whenever data is read-only
  - Now: in many cases, similar performance from global and texture memory
- Unified (managed) memory: data migration is now more efficient
  - Then: performance might significantly degrade from managed memory use
  - Now: fine-granular page faulting, much closer to manual management

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.

Sending Threads to Sleep

- Previously, programmers would use `__threadfence` to force sleep
- No longer necessary, for two reasons:
  1. Warps will now be scheduled due to progress guarantee
  2. If individual threads should back off, Volta+ now exposes `__nanosleep` function
- Used, e.g., in libcu++ to wait

```
__device__ void mutex_lock(unsigned int *mutex) {
    unsigned int ns = 8;
    while (atomicCAS(mutex, 0, 1) == 1) {
        __nanosleep(ns);
        if (ns < 256) {
            ns *= 2;
        }
    }
}
```


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.

Suggestions for Good CUDA Performance

- When optimizing CUDA applications for the modern GPU
  1. Try to come up with an algorithm you know works well in parallel
  2. If there are no previous results/recommendations, go with best assumption
  3. From the start, think about minimizing memory requirements!
     - Compressions? Encoding? Smaller data types? Alignment?
     - This will pay off regardless of compiler optimizations
  4. Once initial version is done, check performance metrics (Nsight Compute)
  5. Optimizers can suggest load improvements–not better algorithms or layouts
  6. Don't bother with:
     - Writing basic math operations with bit magic (compiler is probably smarter than you)
     - "Tweaks and tricks" like changing `uint` loop counters to `int`

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.

Suggestions for Good CUDA Experience

- Embrace libraries shipped with CUDA (cooperative groups, libcu++…)
  - They may change frequently, but they are here to stay
  - Official way to expose barriers, memory consistency model without PTX
- Don’t be afraid of ITS, but make sure you understand legacy model
  - If you have been assuming lockstep, ITS may surprise you, but it makes sense
  - Other compute APIs will keep using the legacy scheduling on your Volta+ GPU
- When using streams or graphs, clearly separate setup and execution
  - Many setup calls synchronize implicitly, don’t mix them with kernel launches

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.

Suggestions for Good CUDA Experience

- Embrace the graph API for sequential and concurrent kernels
  - Can replicate the behavior of streams with dependencies
  - Defined before execution, can isolate setup and launch code
  - Enables driver to optimize performance
- Consider cooperative groups over your own solutions
  - Many developers have their own group implementation already available
  - Cooperative groups are designed to optimize in hardware where possible
  - Also provide software implementations for backward compatibility
  - Facilitates comprehension of your code by others

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.

Things We Did Not Cover

- Shared Memory Data Staging (shared pipelines)
- Virtual Memory Management
- Stream Ordered Memory Allocator
- Compiler Optimizations
- …

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?

- 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

[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