



CUDA and Applications to Task-based Programming

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

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.

About These Course Notes

- Practically-oriented portions rely on ability to maintain code samples
- For the full version on the fundamentals of CUDA, GPU hardware and recent developments, please refer to the tutorial's web page at: <https://cuda-tutorial.github.io>
- The **full version** of these course notes includes additional slides, auxiliary media and code samples

In order to ensure compliance with applicable copyright and enable continuous maintenance of slides and relevant code samples, we have decided to create two separate versions of these course notes.

The version at hand was prepared for a one-time electronic distribution among the Eurographics 2021 conference participants ahead of the presentation itself and includes the documentation of previous and ongoing research into task-based programming with CUDA, as per April 2021.

For the full, extended version of the course notes including an easily approachable introduction, up-to-date code samples, and descriptions of recently enabled features in CUDA, please see the tutorial's web page.

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.

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.

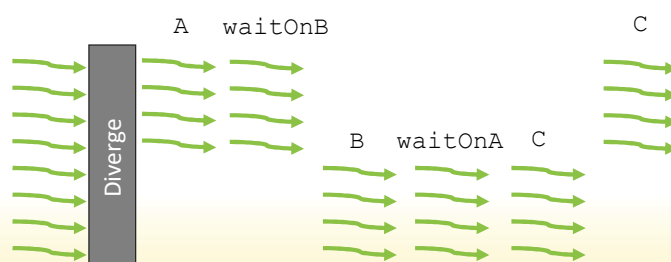
Independent Thread Scheduling (ITS)

- 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();

```



We previously discussed the behavior of ITS, and how it enables for instance use cases where threads in the same warp may wait on each other, which would have caused a deadlock with legacy scheduling. However, with guaranteed progress, such algorithms are now safe to implement in CUDA.

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 access 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.

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.

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.

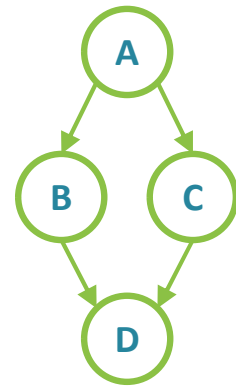
CUDA Graph API

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.

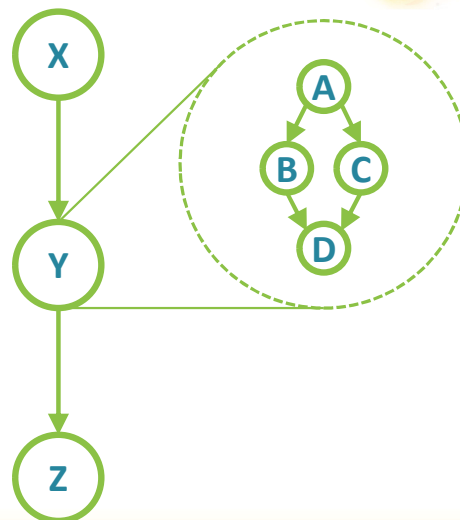
Benefits

- 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

```

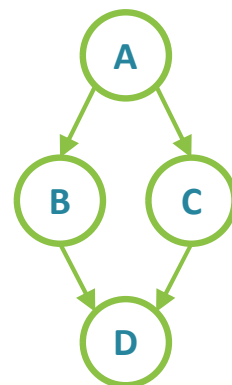
cudaGraph_t graph;

// Define graph of work + dependencies
cudaGraphCreate(&graph);
cudaGraphAddKernelNode(kernel_A, graph, ...);
cudaGraphAddKernelNode(kernel_B, graph, ...);
cudaGraphAddKernelNode(kernel_C, graph, ...);
cudaGraphAddKernelNode(kernel_D, graph, ...);

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

// Launch executable graph 100 times
for(int i=0; i<100; i++)
    cudaGraphLaunch(instance, stream);

```



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.

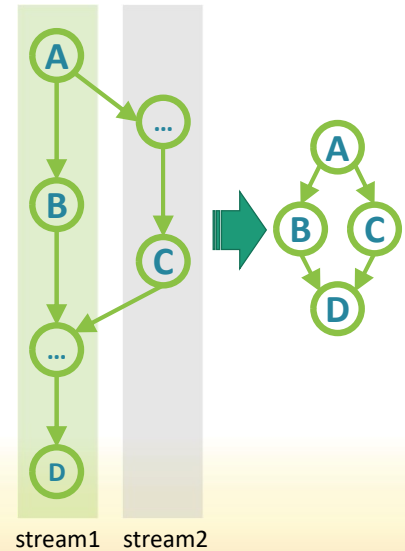
Example

```
// Start by initiating 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...
```

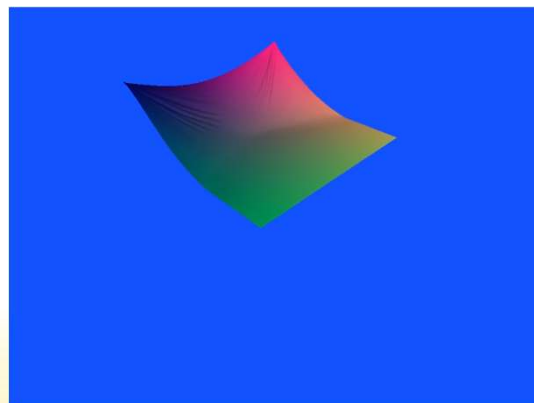
Event required to initiate recording another stream



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%.

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: $A(M \times K)$, $B(K \times N)$, $C(M \times N)$
 - $4 \times 4 \times 4$ ($M \times N \times K$) matrix processing array, performs $D = A \cdot B + C$

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

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.

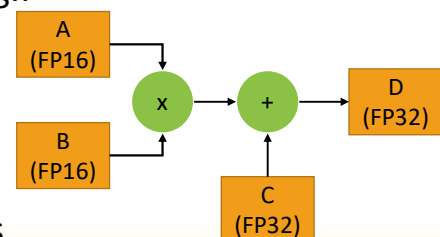
However, the access to tensor cores is also exposed in CUDA directly via a separate header for matrix multiplication and accumulation of small matrices, which are usually only a part of the total input. These matrix tiles, or „fragments“, can be larger than 4×4 if threads in a warp cooperate. The MMA 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 \rightarrow 1024 operations/cycle

- Restrictions on format for input fragments, e.g.:

- A = __half (16bit float), B = __half, C = float $\rightarrow\rightarrow\rightarrow$
- 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);

```

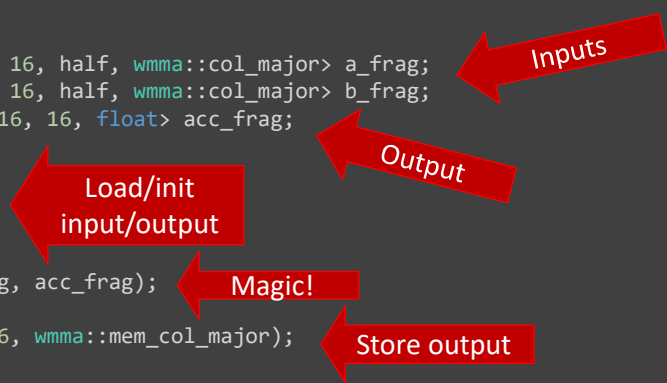
Here, we list the relevant types and functions that are exposed to warps for performing tensor core operations:

- `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

```
using namespace nvcuda;

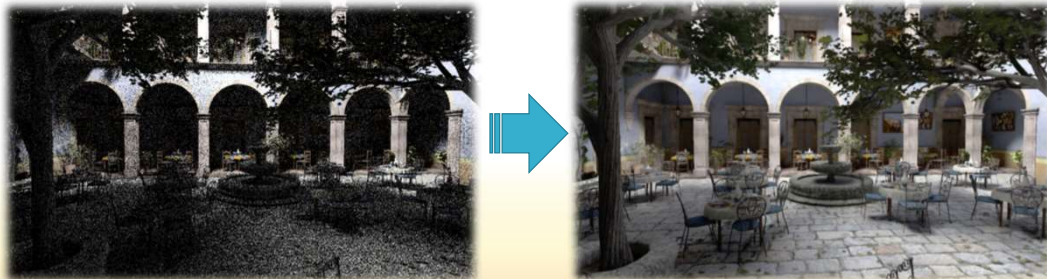
__global__ void wmma_example(half* a, half* b, float* c)
{
    // Declare the fragments
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::col_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> 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);
}
```



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.

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]



05.05.2021 – 06.05.2021

CUDA and Applications to Task-based Programming

32

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



`__match_sync_any(0b111111, val) = {0,3,5} {1} {2,4} {0,3,5} {2,4} {0,3,5}`

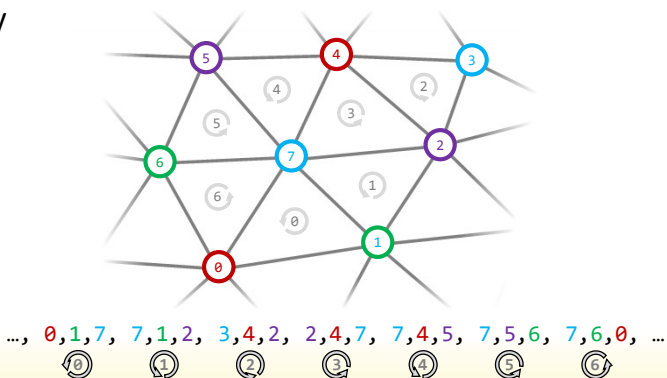
- `__reduce_sync` (new since compute capability 8.0, Ampere)
 - Perform warp-wide reduction (addition, OR, XOR, MIN, ...)

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!



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: Aggregate 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

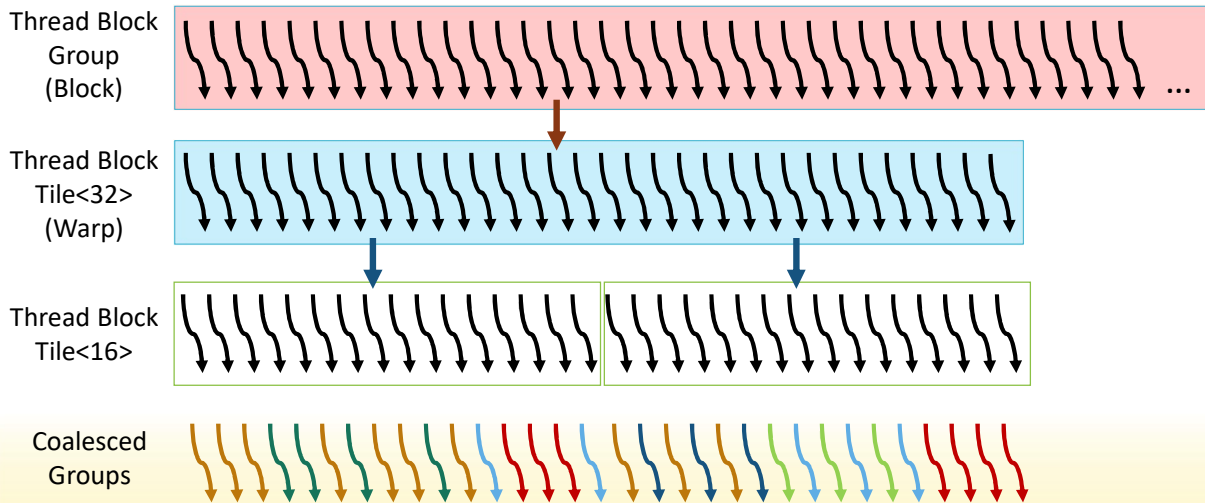
This is exactly the topic that we will be dealing with in the next section of this tutorial.

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.

Cooperative Groups



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...
- `libc++` 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 `libc++`, 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 namespace `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
- Further support (`std::iostream`, `std::vector`) in progress

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 container 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...

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

- But the PTX ISA defines common coherency model

```
__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 `libc++`
 - 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 `libc++` 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); RED.E.ADD.STRONG.GPU [R2.64], R5
}
// Now
__device__ void increment_new(cuda::atomic<int, cuda::thread_scope_device>* val)
{
    val->fetch_add(1, cuda::memory_order_relaxed); 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.

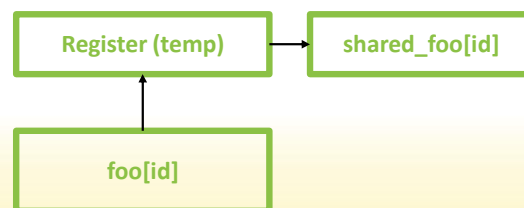
Use Case: Asynchronous Data Copies

- 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[id];
    __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.

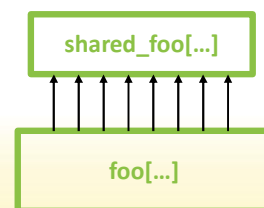
Use Case: Asynchronous Data Copies

- 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.

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 caveatas, 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 are 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 release 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, libc++...)
 - 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
- Forget as much as you can about `threadfence` and `volatile`
 - Replace as many instances as you can with new `cuda::std`

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.

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>



CUDA and Applications to Task-based Programming

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

These are the course notes for the final portion of the tutorial on “CUDA and Applications to Task-based Programming”, as presented at the Eurographics conference 2021, wherein we discuss relevant results from dedicated efforts in the scientific community, as well as the established and state-of-the-art use cases for applications of task-based programming with CUDA.

Overview

- Different levels of the GPU hierarchy and GPU queues
- Task-based Scheduling
 - Host Controller Architecture
 - Persistent Threads & Megakernels
 - Dynamic Parallelism
- Mixed Parallelism usage scenarios

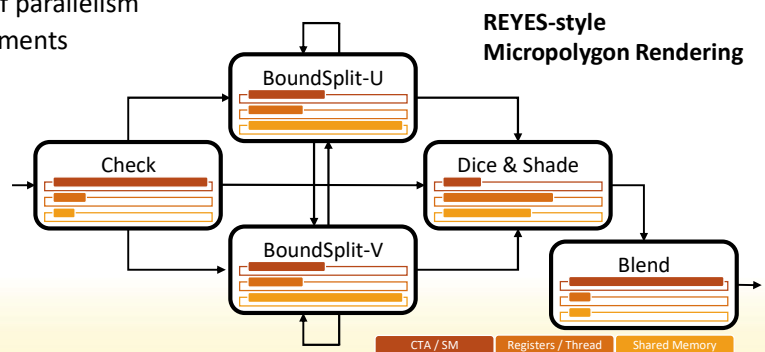
Let us now turn to Task-based scheduling on the GPU. In this part of the tutorial, we will cover the different levels of the GPU hierarchy and how they can be exploited for different programming patterns. We then turn to Task Scheduling, first detailing queues on GPUs, a core component of most task scheduling approaches. Based on such queues, we then build different schemes for task scheduling on the GPUs, controlled from the CPU or entirely from the GPU. Lastly, we will hear about some examples, which greatly benefit from task parallelism and typically exhibit mixed parallelism during execution.

Motivation

- Heterogeneous parallelism in many applications

- Different stages
 - May have different levels of parallelism
 - May have different requirements
 - Shared Memory
 - Registers
 - May generate new work

- Hard to fit into existing programming model



When considering many applications one might like to parallelize, we notice that many of those exhibit **heterogeneous parallelism** throughout. This can manifest differently depending on the application

- Some might simply experience different levels of parallelism throughout the stages of an application, where, to give a hypothetical example, a work item might best be handled by a single thread for the first stage but by a block in the last stage. Choosing one or the other overall will result in poor performance
- Different stages might also have different requirements, i.e., need more or less shared memory or registers, etc.
- Lastly, stages might also generate new work and dynamic resource management is really challenging on the GPU

Overall it is quite clear that fitting all of that into the existing programming model can be quite challenging and requires a lot of manual effort and performance tuning to get right.

Types of Parallelism

Task Parallelism

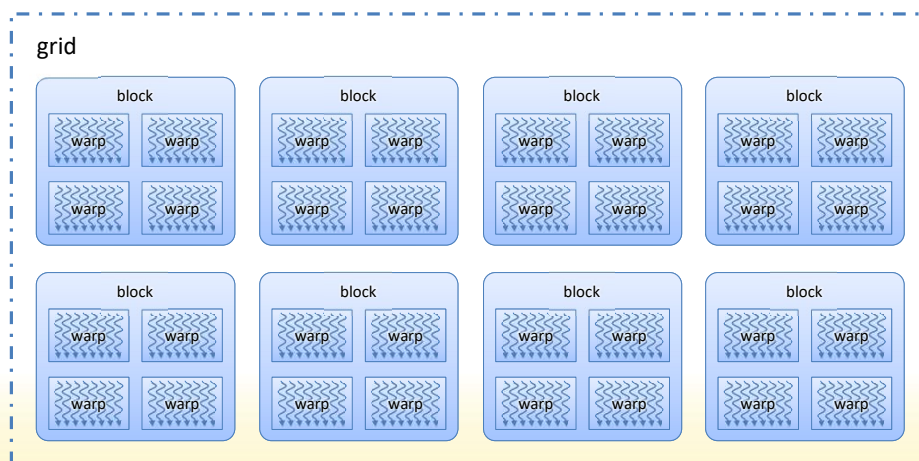
- Parallelize different, independent computation
- Distribute tasks to processors
- e.g., Multitasking, Pipeline Parallelism

Data Parallelism

- Parallelize same computation on different, independent data
- Distribute data to processors
- e.g., Image Processing, Loop-level Parallelism, Tiling, Divide and Conquer

When we talk about parallelism in general, there are typically two types that come to mind, task parallelism as well as data parallelism. In general computing environments, we typically experience **task parallelism**. This means, we have different and independent computations and we want to parallelize these computations by distributing the tasks to the available processors. **Multitasking** and **Pipeline Parallelism** are typical examples of **task parallelism**. On the GPU, we generally work with **data parallelism**, which means that we perform the same computation on many different, independent data items. Here, the data is distributed to the processors. The classical example would be any form of **image processing** (performing some operation per pixel), but also **loop-level parallelism** falls into that category as well as **tiling** and **divide-and-conquer** approaches. As our focus in today's tutorial is on **task scheduling**, we will try to see how this data-parallel architecture on the GPU can be appropriated for task-parallel operations.

Kernel-based Programming Model



To shortly recap the overall terms and hierarchy on the GPU, here is a short overview.

Starting at the lowest level, we have **threads**, whereas 32 threads are executed together as a **warp**, scheduled by the **warp scheduler**. Multiple **warps** are combined into so-called **blocks**. All threads within a **block** are furthermore guaranteed to reside on **one** multiprocessor (SM) and share a faster cache (L1) and have access to fast, shared memory, useful for communication between threads in a block.

Threads from **different blocks** do not share the same, fast memory in **shared memory**, and also do not have any guarantees if they execute on the same or different SMs or concurrently or one after the other. Hence, threads of different blocks should not rely on cooperation but perform largely independent computations. The whole configuration running on the GPU is called a **grid**.

Kernel-based Programming Model



Here we have a classical example which fits a rigid grid configuration quite well with image processing.

Here, one can start one entity (can be a thread, a sub-group of a warp, a full warp or block) for each pixel and perform any kind of operation per pixel. As long as these operations are uniform over the whole image, we expect no differences in run-time between pixels and overall a well-optimized execution pattern.

Kernel-based Program Model



76

On the other hand, let's think about the graphics pipeline in general. We have various stages with very different levels of parallelism, levels of utilization of the GPU, requirements for sorting at certain points, etc.

This is a prime example of mixed parallelism that is hard/impossible to capture with one single, rigid grid configuration and requires more effort to efficiently execute. One core problem is inherent in the dynamic nature of the problem, given a certain input to the input assembly stage, the number of shader invocations in the following stages is scene-dependent and requires support for dynamic work generation.

How to organize work?

- What we want
 - Keep track of work items
 - Allow simultaneous access by all cores for best utilization of cores
 - Allow for work generation
- Organized **work** as **tasks** and store it in **queues**
 - Allow “software scheduler” to fetch/append work
 - Linearizable
 - Low resource footprint

Based on this problem of dynamic work generation, we first have to think about the organization of the work at hand. In a general environment, we want to keep track of a number of work items, allow access to these simultaneously by all cores and also allow the cores to dynamically generate new work.

One possibility in this case would entail organizing work as tasks and storing these tasks or references to these tasks in **queues**. These allow a software scheduler to fetch new work to execute but also enqueue new work to be executed by a different core. Furthermore, it would be great if the **queue** is also linearizable and has a low resource footprint, since especially memory resources can be quite scarce on the GPU.

Queues

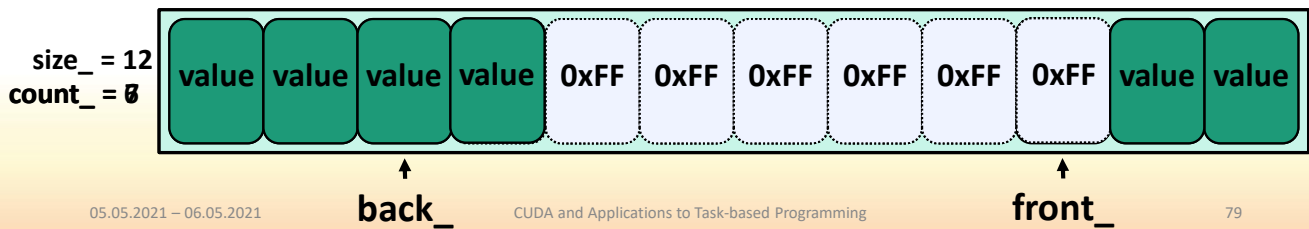
In the following, we'd like to present to you three different variants of queues that we have used in a number of our own publications for various purposes. Hence this is not an exhaustive list of different queue types on GPUs, but a selection based on our own research directions.

Index Queue

- Queue with support for integral values
 - Fixed size
 - Supports concurrent enqueues and dequeues

```

__device__ bool IndexQueue::dequeue(index_t& element)
{
    if (atomicSub(&count_, 1) <= 0)
    {
        atomicAdd(&count_, 1);
        return false;
    }
    unsigned int pos = atomicAdd(&front_, 1) % size_;
    while ((element = atomicExch(queue_ + pos, FREE)) == FREE)
        sleep();
    return true;
}
    
```



05.05.2021 – 06.05.2021

CUDA and Applications to Task-based Programming

79

Let's start with a simple queue that can be used for integral values. These values can be used for multiple purposes but typically they form a reference to a task or resource. This queue has a fixed size as well as a front and a back pointer, acts as a ringbuffer and supports concurrent enqueues and dequeues, which is a very important requirement for task scheduling with dynamic work generation.

During an enqueue operation, first the count (counting the number of elements currently in the queue) is incremented and a check against the size protects against overwriting existing data. Most current queue implementations do not explicitly handle "out-of-queue-storage", hence choosing a sensible size from the beginning is important.

After that, the **back** pointer is incremented atomically, resulting in a position in the queue modulo the queue size.

To enable concurrent enqueues/dequeues, elements are not just taken from the queue as the assigned slot might have been reported as free by another thread in a concurrent dequeue operation, but the data might not have been read yet. To protect against **write-before-read**, writing to the queue is done using an **atomic Compare-And-Swap** operation, which will not alter the queue state until the position is marked as free.

The sleep operation is done using `__nanosleep()` on post-Volta architectures and done using a `threadfence()` on older architectures, which we have found to also work heuristically, resulting in re-scheduling.

Dequeue operations are expected to fail quite often, as multiple threads might query for new work to become available. Hence if decrementing the count fails, it is just incremented again atomically and control is returned to the user. Otherwise, the **front** pointer is moved back, once again resulting in a position in the queue modulo the queue size. And as with **enqueue**, an element is not just taken from the queue but this is done using an **atomic Exchange**, as a queue position might have already been advertised as containing a value but the write to this position has not happened yet. This protects against **read-before-write** problems, whose frequency typically depends on the number of concurrent threads potentially accessing the queue and the size of the queue.

Queues like this found use in multiple of our projects, ranging from dynamic graph management, where a queue could track dynamic vertices or edges, to dynamic memory management, tracking free pages of memory within the system.

Hierarchical Bucket Queue

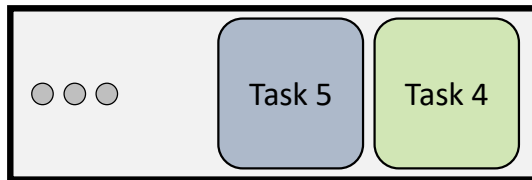
- If memory is abundant
 - Multiple Queues (**buckets**)
 - Access policy determined by user
- Applications
 - Prioritization
 - Task Aggregation

Hierarchical Bucket Queuing for Fine-Grained
Priority Scheduling on the GPU
Bernhard Kerbl, Michael Kenzel, Dieter Schmalstieg, Hans-Peter Seidel,
Markus Steinberger
EG'17

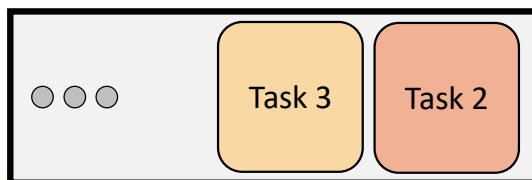
Another type of queue could be an approach called “Hierarchical Bucket Queue”, which relies on the abundance of memory and allows for new applications by instantiating multiple queues, so-called **buckets** with a user-determined access policy.

Based on such a design, one can realize new applications, like **prioritization of tasks** as well as **task aggregation**. The underlying queue implementation can follow a similar design to the queue discussed before, but the combination of multiple queues allows for new concepts. This queueing approach was introduced by Bernhard Kerbl and colleagues as a paper at Eurographics 2017, called “Hierarchical Bucket Queuing for Fine-Grained Priority Scheduling on the GPU”, if one wants to read up on the details.

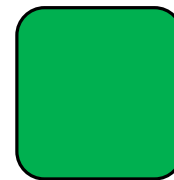
HBQ: Prioritization



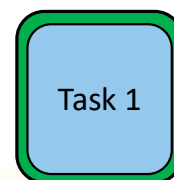
Bucket I (High)



Bucket II (Low)



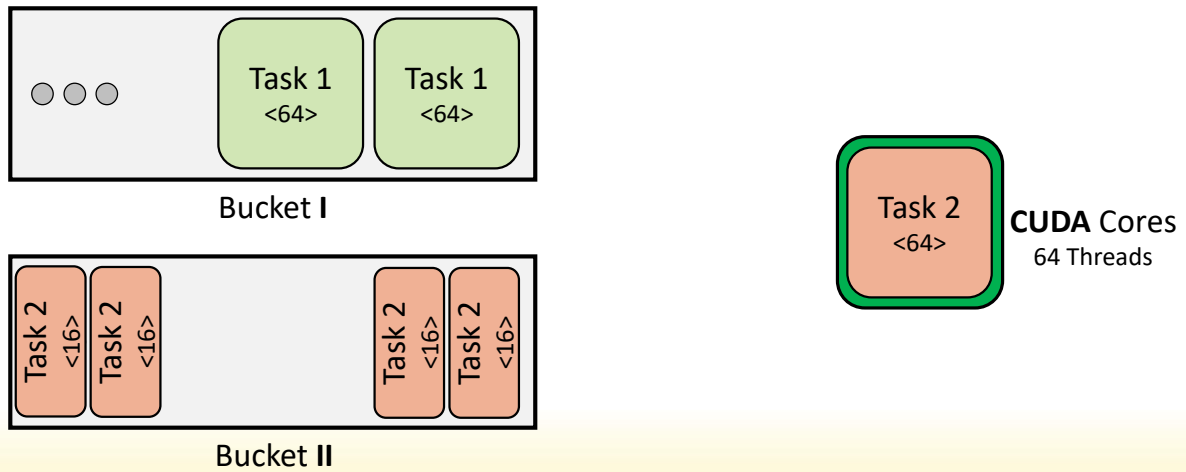
CUDA Cores



Task 1
CUDA Cores

One new concept would be **prioritization** of tasks. One simple way of achieving prioritization would be to instantiate multiple **queues** with varying priorities. This way, executing threads would query high priority queues preferentially first before taking work from lower priority **queues**. This system can also be extended hierarchically, where more than two **queues** would be instantiated into multiple levels of a priority hierarchy. We will show an example of something like that later on.

HBQ: Task Aggregation



Another new concept would be **task aggregation**, whereas one queue could hold simple task items that are executed one by one, while another might hold smaller tasks, that are then executed as an aggregate for more efficient execution. In this example here, Bucket I has larger tasks that have 64 work items in them, efficiently handled by 64 threads and generates a number of smaller tasks with 16 items each. The second Bucket hence acts as an aggregation queue, where the executing cores always withdraw 4 tasks with 16 work items each, hence once again 64 work items for 64 threads to execute the work efficiently.

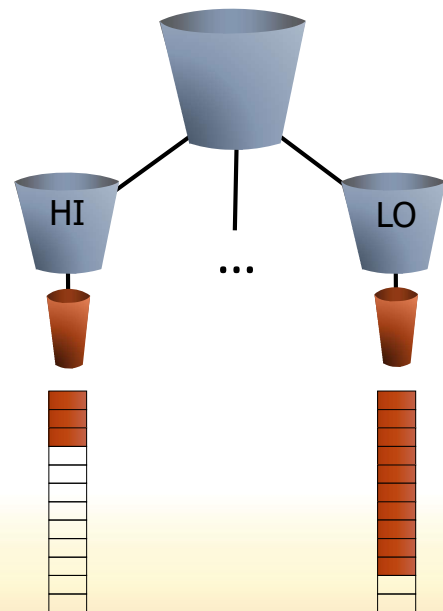
HBQ: Examples

- Ray-Prioritization in Path Tracing

- Regions with high variance need more samples
 - Use coarse priority intervals
 - High-to-Low Prioritization
 - Use variance as Priority

- **N** bucket queues of fixed size

- Choose bucket based on current observed variance
- Linear sorting of work according to image error
- More threads scheduled to work on noisy regions
- Achieve uniform quality with non-uniform sampling

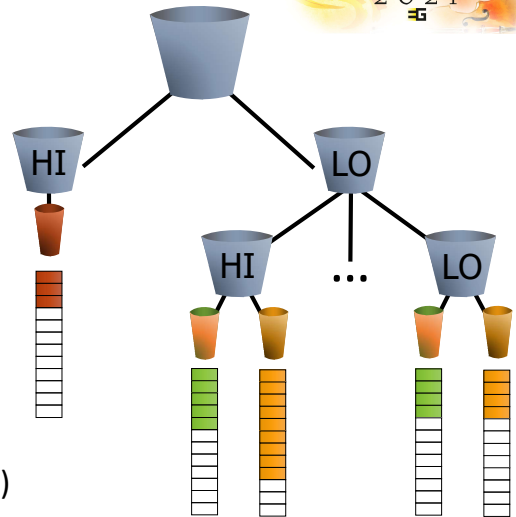
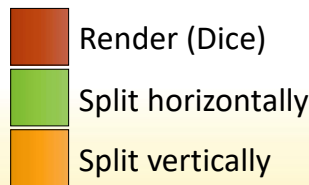


One concrete example for the application of **task prioritization** would be **ray prioritization** in **path tracing**. Here it may make sense to prioritize regions with a high variance, where it can make sense to build up coarse priority intervals, and using the variance as a measure of priority, use a **high-to-low prioritization**.

In this concrete example, one could instantiate a number of bucket queues with a fixed size per queue, whereas a bucket is chosen depending on the currently observed variance.

HBQ: Examples

- Reyes-style Micropolygon Rendering
 - Prefer render jobs over split jobs
 - Two buckets for different routines
 - Prioritize splits based on focus distance
 - **High:** Near
 - **Low:** Far
- **Dual-level scheduling**



Another example would be classical Reyes-style Micropolygon Rendering, an application consisting of multiple stages that are executed, as shown in the graphic on the right. Since visual output is most important, it would be favorable to prefer render jobs over splitting jobs to guarantee smoother playback. Furthermore, one can prioritize geometry splits based on the distance to the camera, once again favoring geometry close to the camera compared to further away.

That way, Rendering is prioritized over splitting geometry, whereas splitting near geometry is prioritized over splitting geometry further away or maybe not in focus in an Augmented Reality scenario.

Broker Queue | Design

- **Static** ring buffer of size **N**
 - **Head/Tail** pointers (packed into 64-bit integer)
 - Can contain elements or pointers
 - **Head** and **Tail** can wrap around buffer
- **Ticketing System**
 - Enqueue/Dequeue associated with **ticket number**
 - Operations only execute if their ticket has been issued
 - Position in buffer can have multiple tickets
 - Results in fair ordering
 - Operations with earlier ticket is guaranteed to finish first

```
void waitForTicket(I Pos, I ExpectedTicket)
{
    auto Ticket = Tickets[Pos];
    while (Ticket != ExpectedTicket) do
    {
        backoff();
        Ticket = Tickets[Pos];
    }
}
```

The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU
 Bernhard Kerbl, Michael Kenzel, Joerg H. Mueller, Dieter Schmalstieg and Markus Steinberger
 ICS'18

Finally, let's look at another design for a queue, called the **Broker Queue**. The basic queue is once again very similar to the basic **index queue** discussed before, build on a **static ring buffer** of a certain size with **head** and **tail** pointers (in this case packed into one 64bit integer). It can also contain just references to tasks but also complete tasks as well.

The main change compared to the previous approach is the introduction of a **ticketing system**. Each operation on the queue, each enqueue/dequeue operation, is associated with a ticket number. An operation only executes once its ticket has been issued, resulting in fair ordering overall. Operations that have an earlier ticket are guaranteed to finish first. Furthermore, each queue position can have multiple tickets concurrently. This queue design is based on a paper, once again by Bernhard Kerbl and colleagues, at ICS'18 called: "The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU".

Broker Queue | Access Data

- Write/Read Data
 - Increment head/tail to get ticket
 - Wait for ticket → perform operation
 - If successful → issue next ticket for slot

```
void putData(T Element)
{
    auto Pos = atomicAdd(&Tail, 1);
    auto P = Pos % N;
    waitForTicket(P, 2 * (Pos/N));
    RingBuffer[P] = Element;
    Tickets[P] = 2 * (Pos/N) + 1;
}
```

```
T readData()
{
    auto Pos = atomicAdd(&Head, 1);
    auto P = Pos % N;
    waitForTicket(P, 2 * (Pos/N) + 1);
    Element = RingBuffer[P];
    Tickets[P] = 2 * ((Pos + N) / N);
    return Element;
}
```

Accessing the queue now utilized the ticketing system to grant or temporarily deny access to a queue element. A position is found by increment the head or tail pointer as before, resulting in a position modulo the queue size.

But before an access can occur, each executing thread has to wait for its ticket to be issued. Only once this has happened, the operation, enqueue or dequeuing from the queue, can occur and after completion, the next ticket for the current slot will be issued.

Broker Queue | Broker

- **Broker**

- Acts as safeguard
 - Many overlapping operations
 - Won't let just any trying thread pass
- Keeps tally of **promised** operations
 - Ensures **balanced ratio** between **enqueue/dequeue**



- **Count**

- Reflects fill state after promised operations
- Modified via **atomicAdd/Sub**
- Contended atomics less of an issue on GPU
 - Would be a problem on CPU

Additional to the **ticketing system**, there exists also a so-called **Broker**, which acts as a safeguard in-between the incoming enqueue and dequeue operations, as there can be many overlapping operations, while the actual write/read accesses only occur much later and also in unpredictable order. It keeps a tally of the number of **promised operations** and overall tries to keep a **balanced ratio** between the enqueue and dequeue operations.

This tally is tracked via an atomic **count** variable, which reflects the fill state after a promised operation has been performed.

As is the case with all queue designs discussed up until now, all rely heavily on atomics, but since atomics are very well optimized on the GPU, contended access is much less of an issue compared to the CPU, where such a design might be problematic.

Broker Queue | Enqueue

• Enqueue/Dequeue

- Wait for Broker
 - Ensure operations is balanced
- Always check **full/empty** state
 - Broker and queue parameters loosely connected
 - Both have to reflect same state
- Check in **loop**
 - Non-blocking behavior
 - Makes queue a linearizable FIFO queue

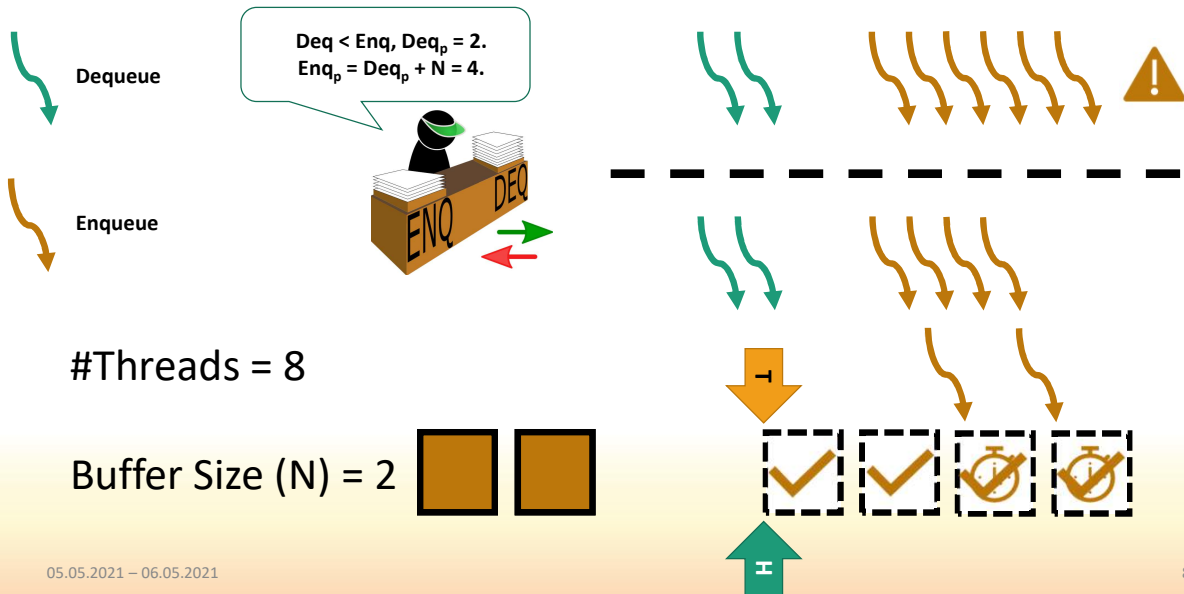
```

STATUS enqueue(I element)
{
    while(not ensureEnqueue()) do
    {
        auto s = queue_state; // Read head/tail
        if(N <= s.tail - s.head < (N + MaxThreads/2))
            return FULL;
    }
    putData(element);
    return SUCCESS;
}

I dequeue()
{
    while(not ensureDequeue()) do
    {
        auto s = queue_state; // Read head/tail
        if((N + MaxThreads/2) <= s.tail - s.head - 1)
            return EMPTY;
    }
    return readData();
}
    
```

Before the ticketing system is now accessed, each executing thread first has to get by the **Broker**, which ensures that the operations are balanced. While waiting for the **Broker**, the state is always queried, as the individual parameters of the **Broker** and the **queue** itself are only loosely connected and may return differing state information. Hence, in a loop the state is checked using non-blocking access, which makes this queue design a **linearizable FIFO queue**.

Broker Queue | Example



05.05.2021 – 06.05.2021

89

Here we have a concrete example, with a **Broker** with a certain policy, in this case enqueue operations should be prioritized over dequeue operations and a certain number of operations, in this case six, might access the queue at one point in time. In this example, we have a buffer size of two and eight threads trying to access the queue, two trying to dequeue and the others waiting on an enqueue operation.

Given this policy, the **Broker** will let six threads through to the actual ticketing system and the enqueueing threads will start their work. As there are more threads present than there are physical queue spaces, the other threads are waiting on tickets to be fulfilled, while two enqueueing threads can start their work immediately, the other two enqueue threads move the head pointer, but wait on their tickets. The remaining two threads waiting for the enqueue operation are currently held back by the **Broker**. As soon as the enqueue operations are done, the two dequeuing threads can take this work from the queue and signal the tickets of the remaining enqueueing threads.

Broker Queue | Non-linearizable variants

- **Broker Work Distribution**

- Ignores loop
 - May report erroneous state
- Benefits
 - Simpler!
 - Potentially faster

- **Broker Stealing Queue**

- Multiple Broker Queues
- Steals work if available
- Ensures looping
 - Locally consistent
 - Not globally linearizable

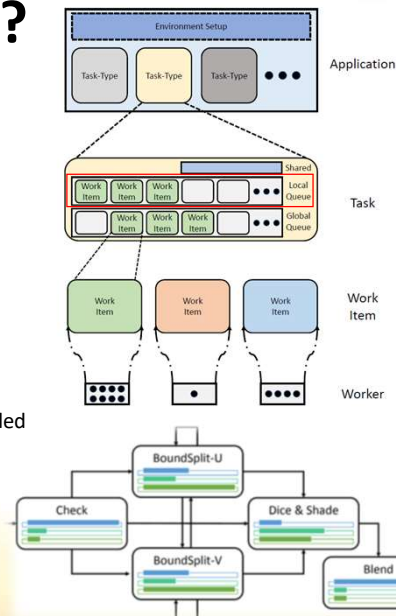
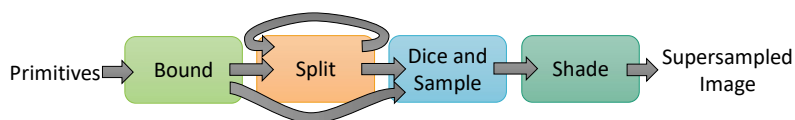
This base design can also be utilized in different, non-linearizable variants, two of which are noted here. By ignoring the loop, one can build a simpler and potentially faster **work distribution** at the cost of potentially erroneous state information intermittently. Another option would include a so-called **Broker Stealing Queue**, which consists of multiple **Broker Queues** which still remain locally consistent and can steal work from another, but are not globally linearizable.

Task-based Scheduling

After this introduction to some queue types, let's now focus on **task-based scheduling** itself.

What do we need to solve?

- We want
 - Ability to handle heterogeneous workloads
 - Dynamic work generation
 - Efficient scheduling
 - Exploit shared memory



05.05.2021 – 06.05.2021

CUDA and Applications to Task-based Programming

92

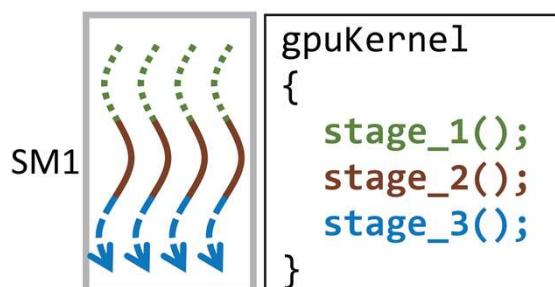
What do we need to solve? What are the properties of applications that our task scheduling system should be able to handle?

- First of all, the individual tasks might have very different requirements and levels of parallelism. The two plots on the right show different representations of such an application setup. On top, we can visualize an application consisting of multiple tasks, each of these tasks can have a queue in global memory associated with it which can contain work items. It may also have a local queue, exploiting shared memory and each work item might be handled by a different number of threads, starting from just one thread, sub-groups within a warp, a warp or even a full block handling one item. On the bottom, we see a visual representation for a Reyes-style renderer, with different stages and the bars for each stage visualize the number of threads required per item, the shared memory requirements, as well as the register requirements for each stage -> overall the requirements are very heterogeneous in this scenario.

- The system should also be able to handle dynamic work generation, once again considering Reyes-style rendering, the number of splits depends on the geometry currently in view and hence results in a dynamic number of samples to shade
- All these different requirements can make efficient scheduling quite challenging
- Lastly, if possible, we should try to exploit shared memory to increase performance even further

Run to Completion

- Simplest execution model
- All stages in a single kernel
 - Does NOT support
 - Global synchronization
 - Dynamic work generation
 - Requirements of largest stage



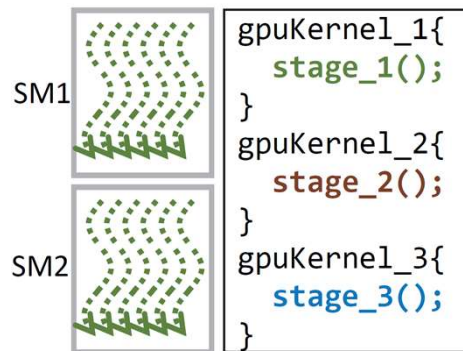
Let's start by investigating very simple models for such task-based applications. One of the simplest, although likely not the one typically chosen, would be the **Run to Completion** model, which puts all stages of our application into one, single kernel.

Since we cannot guarantee that all blocks fit on the device at once, we cannot guarantee support for **dynamic work generation** (also in this simple model, we typically also don't have a queue for work items), also no global synchronization between stages is possible. Furthermore, the requirements of the largest stage (i.e. register requirements, shared memory, etc.) count towards the possible occupancy achieved.

On the positive side, this model does not require synchronization with the CPU and may hold data in shared memory from one stage to the next, but the drawbacks largely outweigh these benefits.

Kernel by Kernel

- Most commonly used
- Split application into series of kernel launches
 - Each kernel tailored to task
 - Requirements per kernel
 - CPU Synchronization
 - Requires controller on **CPU** for dynamic work generation



Next, we have the most well-known approach, so-called **Kernel by Kernel**, where the application is simply split into a series of kernel launches for each stage in the application. The obvious benefit is that each kernel is specifically tailored to the task, hence we can reach optimal occupancy for each of the stages.

On the downside, we now require CPU synchronization, which means additional overhead and removes the possibility of using shared memory to keep memory local from one stage to the next. And in general, it would require some form of a controller on the CPU to allow for dynamic work generation, as otherwise the stages would just run once for the given work and then are done.

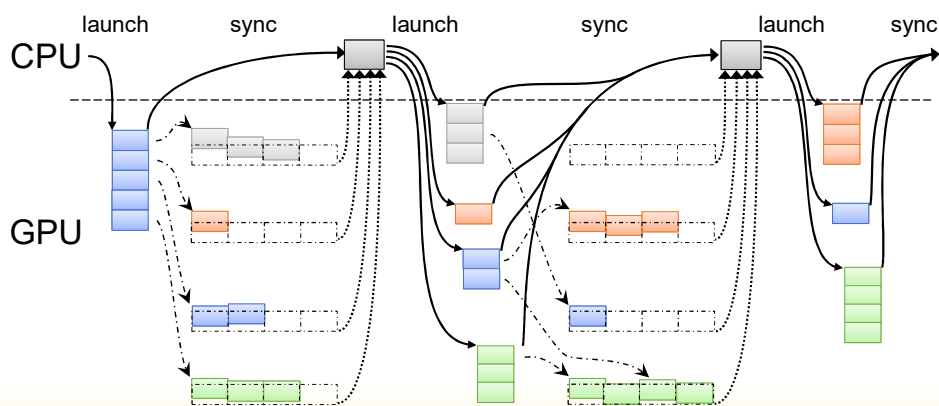
Time-Sliced Kernels

- Variant of **KBK** that supports dynamic work generation
- **CPU** checks amount of work per task
 - Launches kernels with work
 - Into separate streams for concurrent execution
 - Wait for kernels to finish
 - Check work again and start launching again

A variant of the **Kernel by Kernel** approach is typically called **Time-sliced Kernels**. This augments the basic approach by a controller on the CPU side to allow for dynamic work generation. This also means that work queues have to be used.

The controller then can read back the current queue fill levels from the GPU and then launch new kernels with work, possibly also in separate streams for potential concurrent execution. This checking is done in a loop, where the controller waits for the kernels to finish, checks the amount of work and potentially launches new kernels.

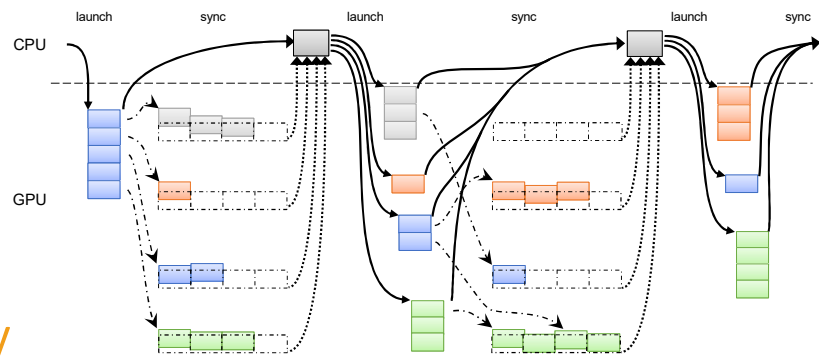
Time-Sliced Kernels



e.g. Laine et al. [2013]

Here we can see a visualization of this approach. The CPU controller is in charge of monitoring the current amount of work, and after fixed synchronization points it can start new work. This means copying the fill levels of the GPU queues back to the CPU at each synchronization point, so that the host controller can decide how much new work to launch on the device.

Time-Sliced Kernels



+no divergence
+optimal occupancy

-CPU synchronization

97

The benefits of this approach are

- There is no (added) divergence within a kernel
- This also means that we should observe optimal occupancy for each kernel

The drawbacks are

- There is need for CPU synchronization, which adds some overhead to the execution
- We cannot easily use shared memory to keep data local from one stage to the other (only within one stage, consider a stage that could generate new input for itself)
- Load imbalance might be a problem
 - If one kernel runs longer than the others due to longer processing, parts of the device might be unused until the next CPU sync as no new work can be launched until the synchronization point with the CPU comes up

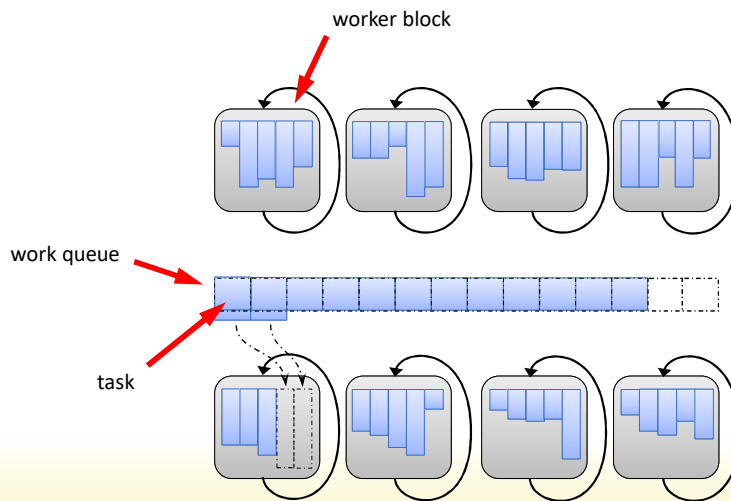
Persistent Threads

- Threads execute in a loop
- Global work queue
 - Draw in new work from queue
 - Execute work
 - Enqueue new work (depends on the queue implementation)
 - Continue until no work left
- Implicit load balancing

One of the first ideas that shouldered the responsibility of scheduling directly on the GPU was called **Persistent Threads**. With this approach, threads execute in a loop and draw in new work from a global work queue. This queue, at least as first mentioned, supports only one task type.

Each thread (or work unit) can draw in new work from the queue, execute it, enqueue new work (if the queue supports concurrent enqueues/dequeues) and simply continues until no work is left. Since each thread can immediately draw new work as soon as it is finished, this results in **implicit load balancing**.

Persistent Threads



e.g., Aila and Laine [2009]

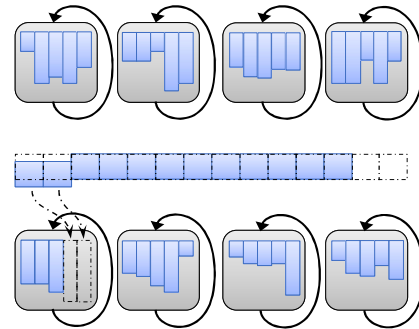
In its original form, it mainly dealt with the issue of load balancing, but the queue as used by Aila and Laine does not support dynamic work generation. Each block keeps executing as long as work is available in the work queue, hence load balancing is done implicitly.

As no new work can be generated, at least with this basic design, blocks simply return if the queue is empty.

Persistent Threads

+load balancing
+(dynamic work generation)

–only one type of task

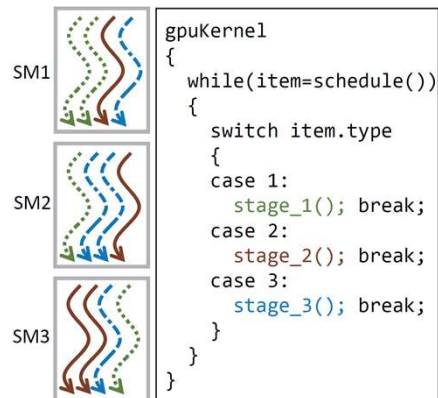


Persistent threads improve upon the load balancing issues of the **time-sliced kernels** approach and may in theory also support dynamic work generation, depending on the queue implementation. But in this basic version, only one task type is possible.

The generalized form of persistent threads is called **MegaKernel** and is discussed next.

Megakernel

- Generalized version of **persistent threads**
 - Can handle different task types
 - Depending on queue also dynamic work generation
- May suffer from divergence
- Occupancy still bound by largest procedure

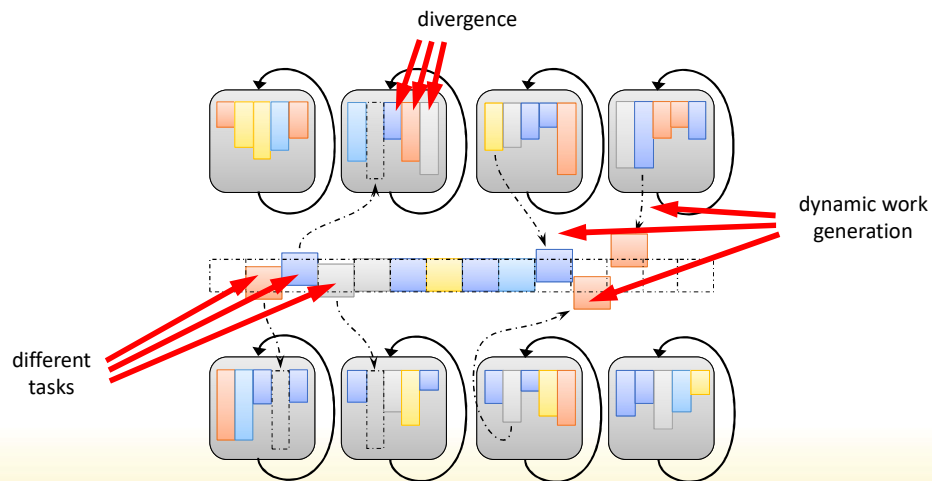


Taking the basic concept from **persistent threads**, i.e. having the blocks execute in a loop on the GPU and drawing in new work from work queues, we can get to so-called **MegaKernels** by allowing for different task types. This requires additional scheduling between the different work queues and depending on the queue implementation, this also supports dynamic work generation.

While we now can offer the same functionality as with **Time-sliced kernels**, just with implicit load balancing directly on the GPU and with no explicit CPU synchronization required, there are still some drawbacks:

- The occupancy is still tied to the largest procedure, as every block has to be able to execute each task
- Furthermore, as each block might execute multiple, different tasks at the same time, there is also potential for divergence negatively affecting overall performance within blocks

Persistent Megakernel



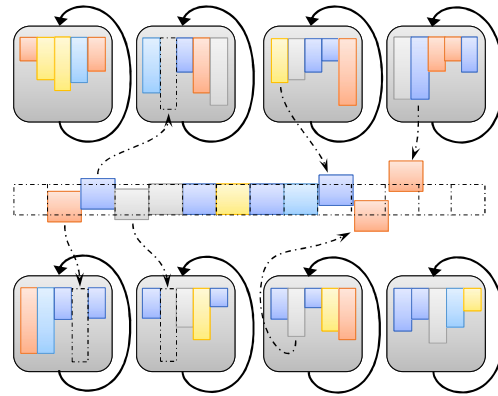
e.g. Steinberger et al. [2012]

Here we can see one visualization of a **MegaKernel**, based on our own work called **Softshell**. The queue supports multiple task types (typically with an abstraction around multiple queues for one task type) and also dynamic work generation. Each block still draws in new work after all work has been finished per block, hence load balancing is quite well handled but still divergence may occur within a block.

Persistent Megakernel

+load balancing
+dynamic work generation
+multitasking

–divergence
–suboptimal occupancy
–bottleneck: work queue



To sum up, the benefits of a **MegaKernel** are:

- Implicit load balancing over the blocks, as each can immediately start new work upon finishing execution of “old” work
- The queues support **dynamic work generation**
- And multiple tasks types are support as well

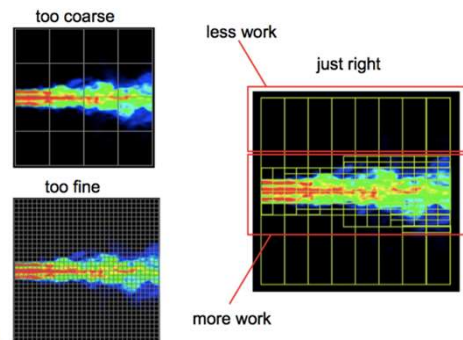
The drawbacks include

- Divergence within a block can reduce overall performance, especially if there are large discrepancies between run-times of different tasks
- Occupancy is tied to the largest stage, hence large discrepancies between stages once again reduce performance overall
- The work queue has to be efficient, as many blocks keep polling for new work

Dynamic Parallelism

- Nested parallelism occurs in many applications

- Since CUDA 5.0
 - Kernels can launch other kernels
 - Dynamically adapt to amount of work
- Link with *cuda-devrt*
 - Compile with *-rdc*

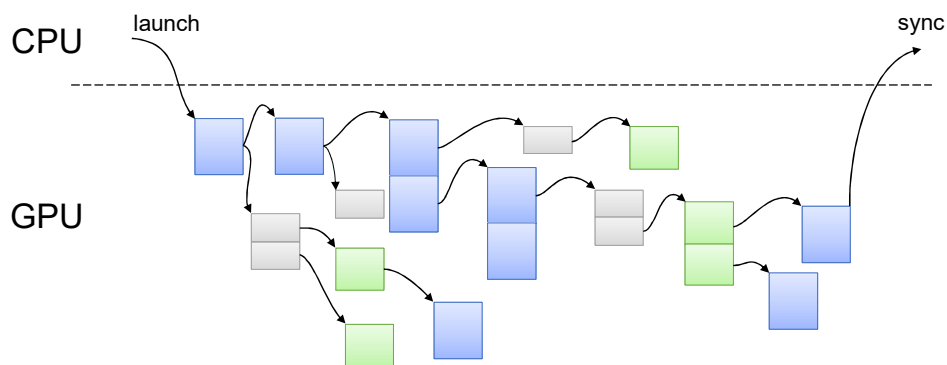


NVIDIA Developer Blog, Adaptive Parallel Computation with CUDA Dynamic Parallelism, 2014
<https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism>

Before diving into the last set of techniques, let's first introduce **dynamic parallelism**. Starting with CUDA 5.0, NVIDIA reacted to the problem of nested parallelism being common in many applications by allowing for kernels to launch other kernels. This way one can dynamically adapt to the amount of work. On the right you can see a typical problem, where it can be quite hard to find a good grid size selection for some simulation problem, as it can be too coarse or too fine overall. Being able to react to the coarseness of the problem directly on the GPU can be a great benefit.

To use dynamic parallelism, **device linking** has to be enabled and one has to link against the **CUDA Device Runtime**.

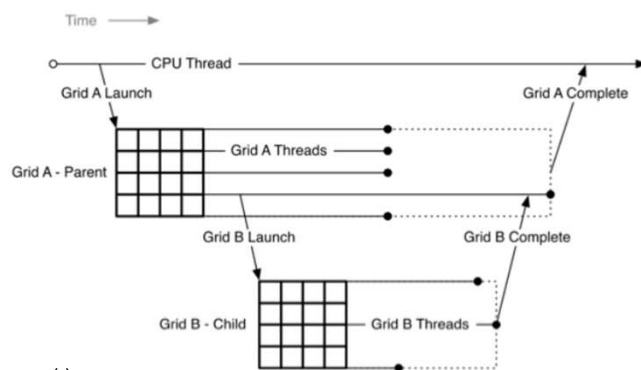
Dynamic Parallelism



Here we see a visualization of how a task scheduling could work using dynamic parallelism. The CPU would launch an initial block, which then could launch new work in new kernels, specifically tailored to the amount of work as well as the type of work. Hence, occupancy should be quite optimal.

Dynamic Parallelism

- Group of thread blocks is called a *grid*
Parent grid launches *child grids*
- **Child grid** inherits attributes
 - L1 cache
 - Shared memory configuration
 - Stack Size
- **Child grids** are *fully nested*
Parent grid can `cudaDeviceSynchronize()`
 Only thread which launches is aware of actual kernel launch



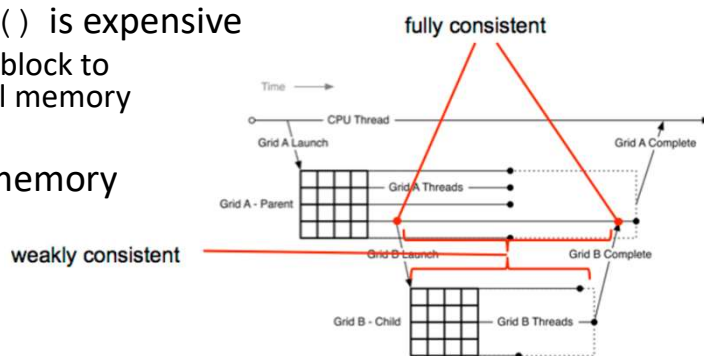
NVIDIA Developer Blog, CUDA Dynamic Parallelism API and Principles, 2014
<https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles>

Back to the basics on **Dynamic Parallelism**: A group of blocks (each consisting of a certain number of warps, each consisting of 32 threads), is called a grid. In the context of **DP**, we speak of a **parent grid** launching a **child grid**.

The **child grid** inherits some attributes from the parent, this includes the configuration of Unified (L1) cache and shared memory as well as the stack size. Child grids are always fully nested within the parent launch as one can see in the graphic on the right. The parent grid implicitly waits for the child grid to finish, but can also explicitly synchronize with the child grid by calling `cudaDeviceSynchronize()`. One **important note**, only the thread that actually performed the launch is aware of the child grid and can synchronize.

Dynamic Parallelism

- `cudaDeviceSynchronize()` is expensive
 - May cause the currently running block to be paused and swapped to global memory
- *Fully-consistent view* of global memory
 - Both directions with sync
 - Weakly consistent in-between
- Passing pointers to *child grid*
 - Global, zero-copy host and constant
 - Shared and local memory



NVIDIA Developer Blog, CUDA Dynamic Parallelism API and Principles, 2014
<https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles>

Unfortunately but expectedly, a full `cudaDeviceSynchronize` can be quite expensive as it might cause the currently running block to be paused and swapped to global memory. This means that all the current state of a block (registers, shared memory etc.) has to be copied to and from global memory. But, at least for global memory, there exists a **fully-consistent** view between child and parent, so a parent writing to memory and then launching a child grid is guaranteed that the child sees the value. Furthermore, if a child writes something and the parent synchronizes on the child, it is also guaranteed to observe the value. Inbetween the model is weakly consistent and there is no guarantee. One further limitation is given by what can be passed to the child grid regarding memory:

- Global memory, managed (or zero-copy host) memory as well as constant memory can be passed between parent and child
- Shared memory as well as local memory **cannot** be passed to the child grid

Dynamic Parallelism

- *Child grids* launched sequentially
 - Happens even if launched by different threads
 - **Use streams**

```
cudaStream_t s;
cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
```

- Streams on device are *non-blocking*
 - *Kernels* in different streams **can** execute concurrently
 - **Do not rely on that!**
 - Streams in different blocks are *different*
 - Streams in same block can be used by all threads in block
 - `cudaStreamDestroy()` returns kernels immediately

Identical to the host, **child grids** are launched sequentially, even if launched by different threads by default. To allow for concurrent execution, one has to use **streams**.

Streams on the device are non-blocking (launches in the same stream occur still sequentially), hence kernels in different streams can execute concurrently. One **important note**: Do not rely on this, as there is no guarantee that two kernels will actually run concurrently, so a producer-consumer system between two kernels is not guaranteed to work. Furthermore, beware that streams in different blocks are different, while streams in the same block can be used by all the threads. Lastly, one can use `cudaStreamDestroy()` to immediately return a kernel.

Dynamic Parallelism

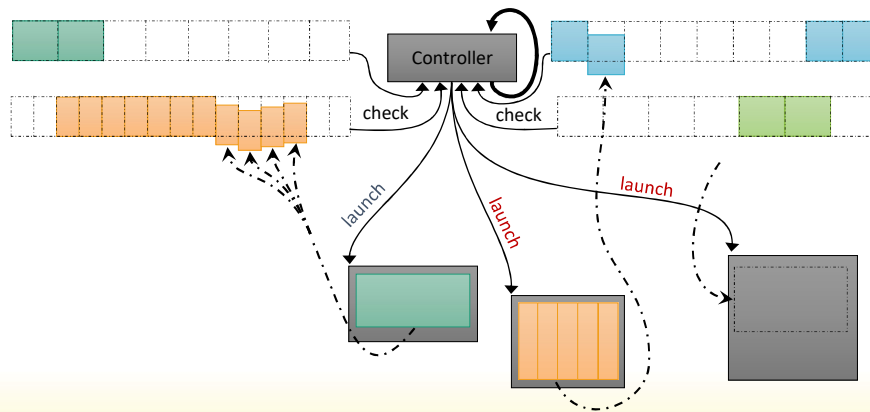
- Recursion depth
 - Nesting depth
 - Kernels launched from host (Depth = 0)
 - **Hardware Limit = 24**
 - Synchronization depth
 - Deepest level to sync (Default = 2)
 - `cudaLimitDevRuntimeSyncDepth()`
- Pending launches (Default = 2048)
 - `cudaDeviceSetLimit(cudaLimitDevRuntimePendingLaunchCount, 123456)`
 - *Virtualized pool* (more flexible, but additional launches more costly)

If all of that sounds great, here are now a few caveats:

- There exist some hardware limits:
 - There is a maximum nesting depth of **24**, limited by the hardware. Kernels are launched at depth 0 from the host -> recursive launches only work up to the given hardware limit
 - Furthermore, there is a limit how far the synchronization is possible.
- The number of pending launches is also limited
 - Once can increase this from the default of **2048**, but this can be quite costly

All of these limits exist as there are physical limitations, as states have to be stored in memory etc. Overall, performance is limited quite a lot as soon as one approaches any of these limits.

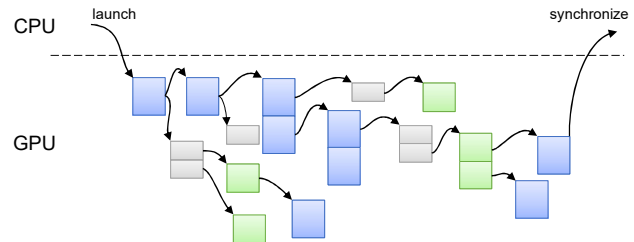
Hybrid Dynamic Parallelism (HDP)



One possible solution would look something like that, with a controller on the GPU, checking the individual queues, launching new work into separate, tailored kernels. This design mimics the **TSK** design from earlier, with one central controller unit (possibly a single thread, or warp), that routinely checks the work queues for new work and launches corresponding new kernels.

Dynamic Parallelism

- +dynamic work generation
- +GPU autonomy
- +optimal occupancy



- no fine grained work generation though
- cannot use local memory to pass on data
- limited launch depth

Overall, to summarize the benefits of DP:

- It automatically supports dynamic work generation
- It is GPU autonomous, same as the MegaKernel, foregoing the synchronization with the host
- In contrast to the MegaKernel, it can tailor each launch to the specific task, resulting in optimal occupancy

But there are some severe limitations:

- Due to the limit (and performance penalty) of launching many small kernels, one cannot successfully allow for fine-grained work generation
- One cannot pass local memory directly to a kernel, only through global memory
- The limited launch depth limits the approach of each kernel launching new work (which would render the controller obsolete)

Feature Comparison

	TSK	HDP	WMK
collect tasks per procedure	•	•	•
GPU autonomy		•	•
optimal occupancy	•	•	
adaptive scheduling			•
fast local queuing			•

112

Lastly, let's compare all three approaches (**Time-sliced kernels TSK**, **Hybrid Dynamic Parallelism HDP** as well as **MegaKernel** (in this case an already advanced version called Whippletree, based on our own work, called **WMK**):

- All of them support dynamic work generation and collecting tasks from some form of queue for each stage/procedure of an application
- Only **HDP** and **WMK** are GPU autonomous
- Only **TSK** and **HDP** can reach optimal occupancy as each kernel handles just one task type, while **WMK** is limited by the resource requirements of the largest kernel
- But only **WMK** supports both adaptive scheduling as well as fast local queuing in shared memory

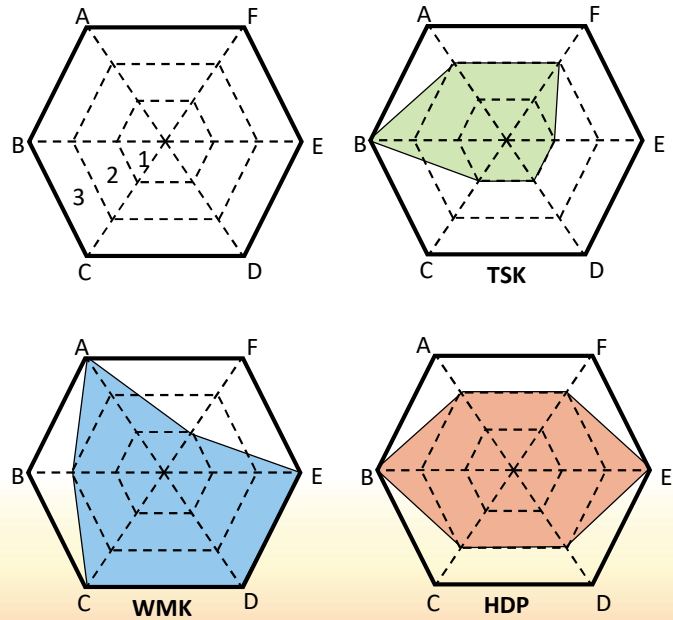
Feature Comparison

Characteristics

- A – Adaptive Scheduling
- B – Optimal Occupancy
- C – Local Queueing
- D – Launch Overhead
- E – GPU autonomy
- F – Mixed Requirements

Level

1 – negative 2 – neutral 3 – positive



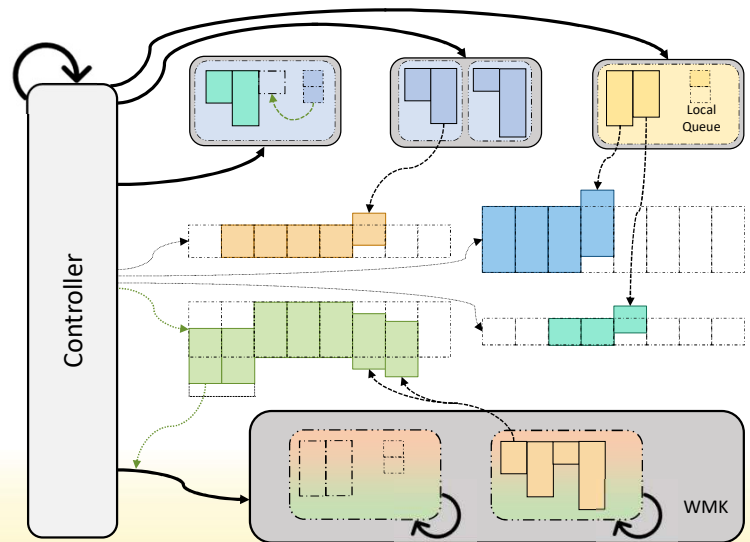
113

Here we have a different visualization of six characteristics

- **Adaptive Scheduling:** This is a great benefit of the **MegaKernel**, which can only be approximated with the other approaches
- **Optimal Occupancy:** HDP and TSK can tailor their kernels to the requirements, contrary to the **MegaKernel**
- **Local Queueing:** The **Megakernel** can support that for different tasks, HDP only for recursion
- **Launch Overhead:** CPU synchronization is worst, followed by GPU synchronization and then no launches at all for the **MegaKernel**
- **GPU Autonomy:** **WMK & HDP** are autonomous, **TSK** requires synchronization
- **Mixed Requirements:** Neither approach can fully utilize mixed requirements, as homogeneous stages fit **Megakernel** best and heterogeneous stages fit **TSK & HDP** best

Future Ideas

- Allow combination of **MegaKernel** and **HDP**
 - Controller can launch individual procedures or smaller Megakernels
- Benefits
 - Combine homogeneous workloads in MegaKernel
 - Split apart heterogeneous workloads



Following on this last idea, one possible evolution of these concepts would be a combination of the benefits of the **MegaKernel** and **HDP**. The controller in this instance can not only launch individual kernels for tasks but also smaller Megakernels.

This way, one can combine homogeneous workloads into a MegaKernel and split apart heterogeneous workloads into different kernels, in theory combining the benefits of both approaches.

Examples

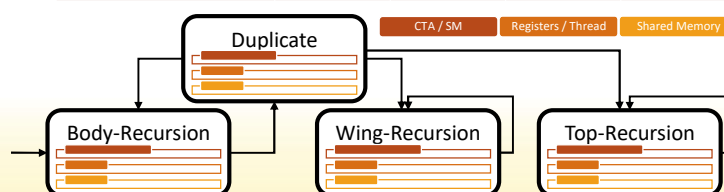
Lastly, let's look at some examples, starting with a few applications that require a **task-scheduling** framework on the GPU and then we finish on a software implementation of a rendering pipeline.

Procedural Geometry Generation

- Spaceships generated randomly
 - Similar to approach by **Ritchie et al.** on the CPU
 - Input
 - Number of Cubes
 - Random Parameter Table

Type	Registers	Worker Size	Shared Memory	Occupancy
Body-Recursion	56	1	2064	50%
Wing-Recursion	56	1	2064	50%
Top-Recursion	56	1	2064	50%
Duplicate	61	1	2064	50%

- Recursive Tasks
 - Responsible for different parts of Spaceship
- Very homogenous overall
 - MegaKernel performs best
 - Local Queues help with recursion



First of we can look at **Procedural Geometry Generation**. Here we set up an example which generates random spaceships, similar to an approach by Ritchie and colleagues on the CPU. One can input the number of cubes that should make up the spaceship and a parameter table that steers the random generation of the wings and top structure of this spaceship.

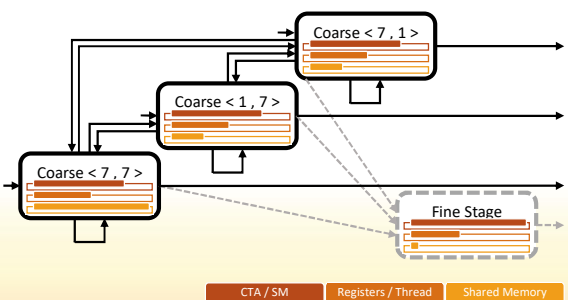
This pipeline is very homogeneous overall with loads of recursive tasks, benefiting from local queueing. Overall, a **MegaKernel** approach performs best here.

SVG Rendering

- Implements hierarchical rasterization approach
 - Some coarse rasterization tasks
 - Determine potential coverage, depending on hierarchy different size
 - Fine rasterization stage
- Heterogeneous requirements
 - Especially worker size and shared memory
 - Lots of recursion

HDP & TSK on-par with WMK

Type	Registers	Worker Size	Shared Memory	Occupancy
Coarse <7,1>	70	16	3600	38%
Coarse <1,7>	70	16	3600	38%
Coarse <7,7>	71	8	21008	38%
FineStage	60	128	32	50%



Next, we can look at a hierarchical SVG rasterization approach, consisting of some coarse stages, which determine first the potential coverage and then are executed, depending on the current hierarchy level and there is also a fine rasterization stage. Overall, the requirements are quite heterogeneous, especially considering worker size and shared memory. But there is also significant recursion and local queueing helps, so overall all approaches are on a similar level regarding performance.

Catmull-Clark Subdivision

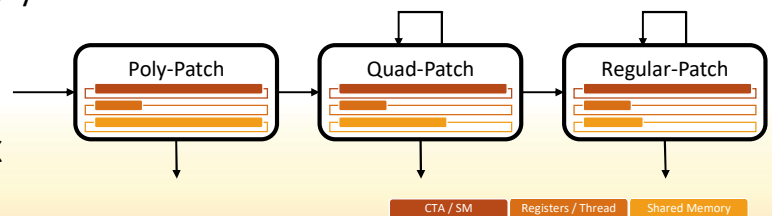
- Simple input mesh → detailed geometry

- Recursive subdivision
- Split mesh into patches

Type	Registers	Worker Size	Shared Memory	Occupancy
Poly-Patch	60	16	22736	50%
Quad-Patch	64	16	15952	50%
Regular-Patch	64	16	7056	50%

- Execution

- Heterogeneous shared memory requirements
- Large input data
 - up to 500B
- **TSK & HDP** outperform **WMK**



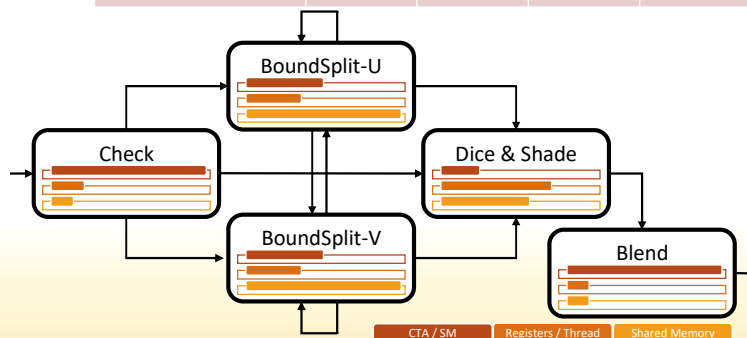
Next, we can look at **Catmull-Clark** Subdivision, which takes a simple input mesh and, using recursive subdivision by splitting the mesh into patches, generates highly detailed output geometry.

We observe quite heterogeneous shared memory requirements overall and have to load quite a bit of data for each input patch. Overall, **TSK & HDP** outperform **WMK**, but not by a huge margin.

Reyes Rendering

- Split scene recursively into micropolygons
 - Recursively split and render
- Heterogeneous workload
 - Different worker size, shared memory, registers
 - **TSK & HDP** outperform **WMK**

Type	Registers	Worker Size	Shared Memory	Occupancy
Check	24	16	2192	100%
Bound/Split U	63	4	14864	50%
Bound/Split V	62	4	14864	50%
Dice & Shade	104	256	6168	25%
Blend	14	1	2072	100%

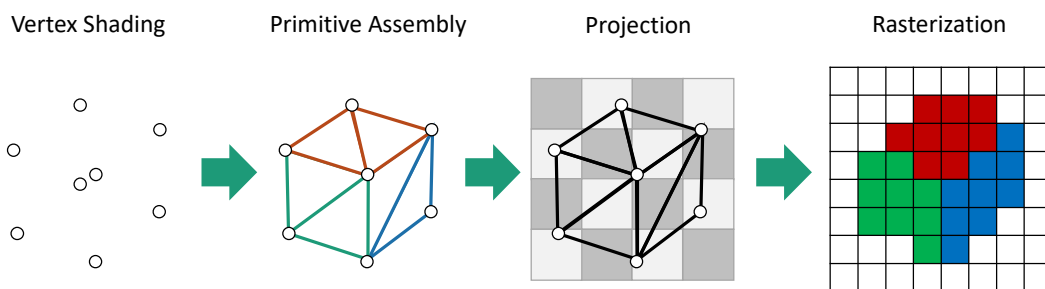


Lastly, the previously mentioned Reyes Rendering, where the scene is recursively split into micropolygons, which are further split up to a certain level and then rendered in the end. Here we have a prime example of a heterogeneous workload, with different numbers of workers per item, different register requirements as well as shared memory requirements. Here, **TSK & HDP** clearly outperform the **MegaKernel** approach.

CURE

Lastly, we can look at one project of ours which dealt with implementing a software rendering pipeline.

Basic Graphics Pipeline

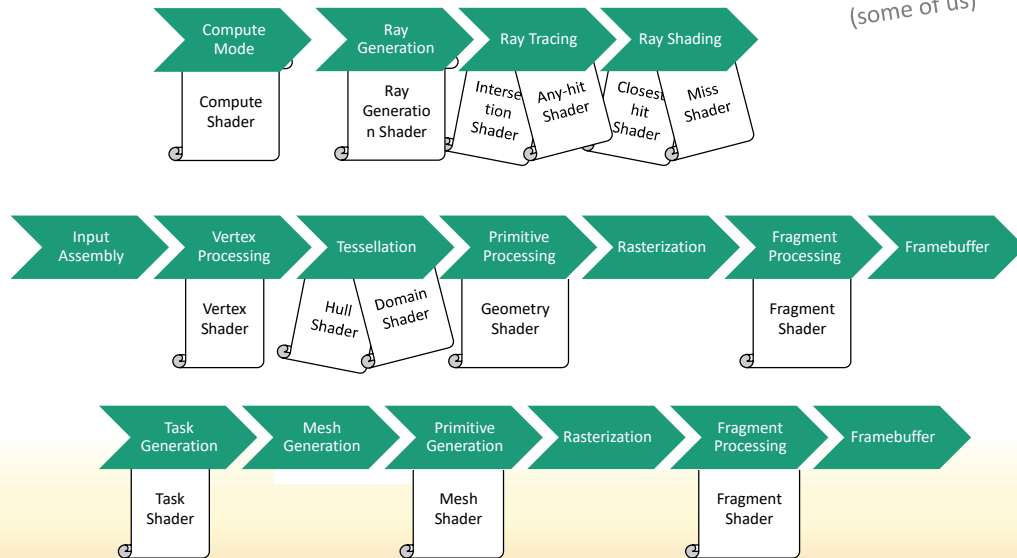


Here we have the basic graphics pipeline as it existed some years ago, consisting of:

1. Vertex Shading
2. Primitive Assembly
3. Projection
4. Rasterization

Back then, everything was fixed-function and was purpose-built for the task of rendering simple meshes.

today!
(some of us)



Today, we have access to different types of pipelines, depending on the GPU in your system. The classical pipeline, now augmented by Tessellation and also further means of geometry processing, still exists and still is mostly used today for most rendering applications. But also new pipeline models have been introduced in the recent years on modern GPUs.

This includes a pipeline based on Mesh Shaders (introduced with Turing GPUs) and can replace the traditional pipeline. It adds two new shader stages, the **task shader** (operates in work groups and can emit **mesh shader** workgroups) as well as the **mesh shader** (generates primitives), both similar to compute shaders and having greater flexibility and scalability at possibly a reduced bandwidth.

Furthermore, we also got **Ray Tracing** support (also introduced with Turing GPUs) as well.

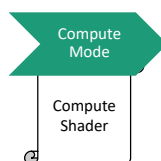
Programmable Hardware Pipeline



Hardware-accelerated Software Pipeline?

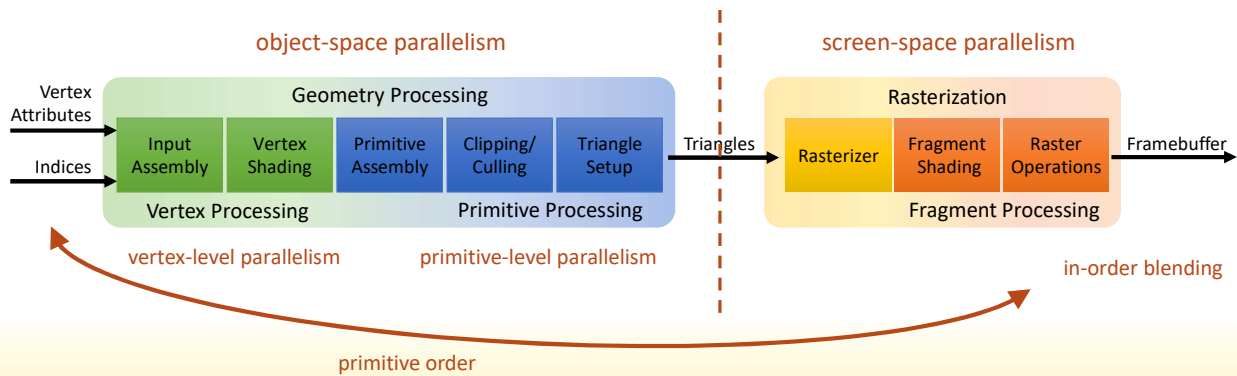
Depending on the actual use case, different pipelines might work best. But still, those pipelines have a rigid structure, which might not fit all scenarios equally well. Hence we thought about the possibility of moving from a programmable hardware pipeline to a hardware-accelerated software pipeline to be able to adapt to specific use cases and test the benefits of new pipeline designs.

tomorrow?



So instead of using fixed-function units, the question is if we can just do everything in compute mode, is that feasible?

Challenges



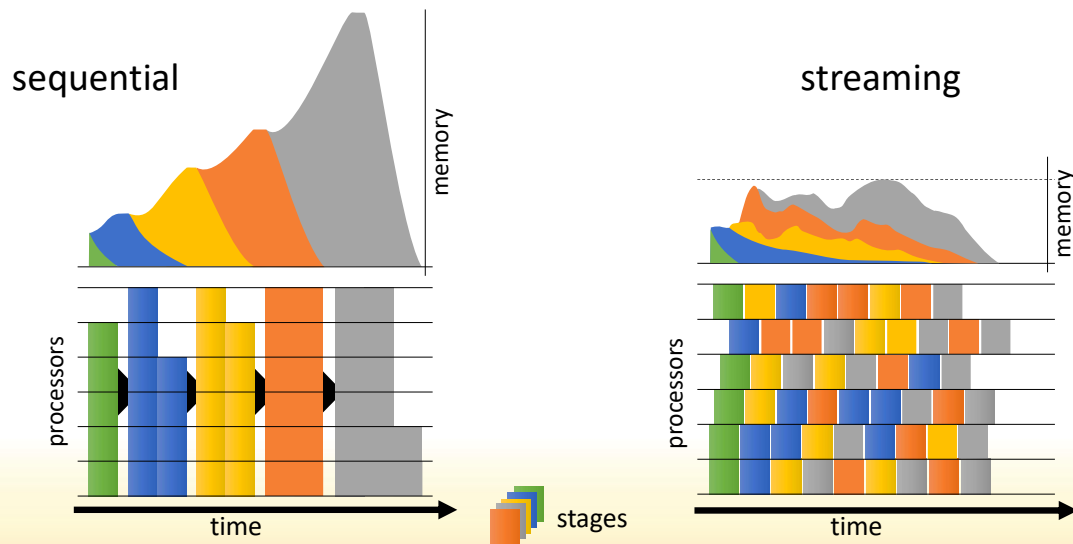
During a classical rendering pipeline, we not only have multiple different stages, but also have to think about different levels of parallelism and maybe have to obey primitive order.

The first part of the pipeline deals with **object-space** parallelism, while the second part deals with **screen-space** parallelism.

When we look more closely at the first part, we can further distinguish between **vertex-level** and **primitive-level** parallelism.

Furthermore, if we require in-order blending, primitive order has to be kept the same throughout the pipeline.

GPU Pipeline Implementation



05.05.2021 – 06.05.2021

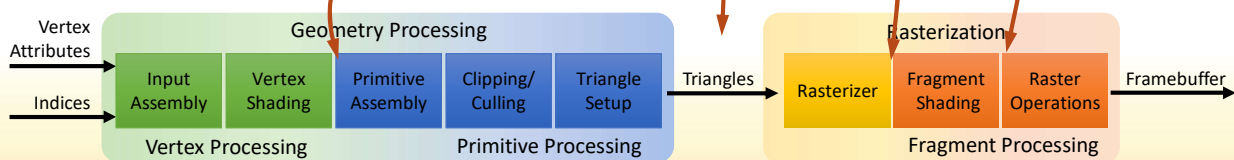
CUDA and Applications to Task-based Programming

126

When we think about execution patterns, we have to be careful about our memory footprint. Using a sequential design (like **KBK**), executing one stage of the pipeline after the other, we quickly run into problems with memory consumption, as is visualized on the left side. Rendering pipelines are usually built on a streaming approach, as can be seen on the right side, here we use much less memory overall.

How do we implement it?

- Design Principle:
 - globally sort middle
 - locally sort everywhere else



05.05.2021 – 06.05.2021

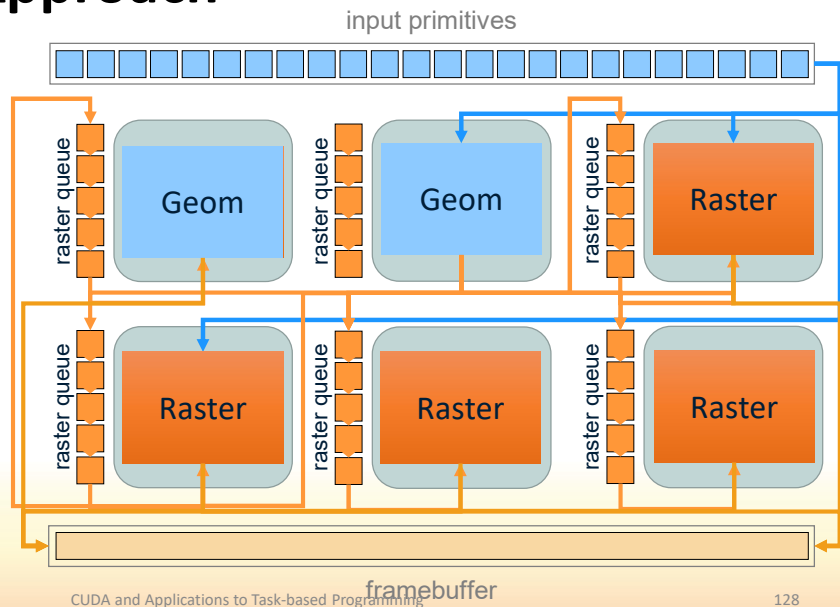
CUDA and Applications to Task-based Programming

127

To keep primitive order, we also have to think about sorting. One sensible solution is to **globally sort middle** and **locally sort everywhere else** during the pipeline.

Megakernel Approach

- fill GPU with worker blocks
- run either
 - Geometry Processing or
 - Rasterization
- global load balancing: raster queues
- local load balancing: on-chip buffers



05.05.2021 – 06.05.2021

CUDA and Applications to Task-based Programming

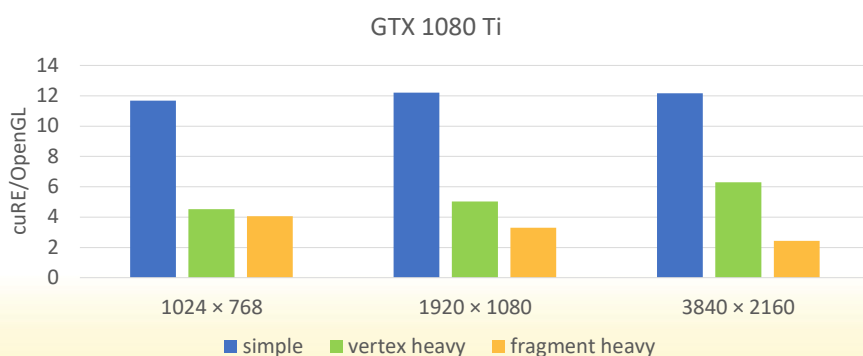
framebuffer

128

In our design, we build on a **MegaKernel** approach and start by filling the GPU with worker blocks. Each block can handle either **Geometry Processing** or **Rasterization** tasks. Global load balancing is handled via the raster queues, but also local load balancing is possible by using shared memory directly on-chip for improved performance, so only in the end one has to write to global memory again. This is based on work by Michael Kenzel and colleagues (“**A high-performance software graphics pipeline architecture for the GPU**“ at Siggraph’18).

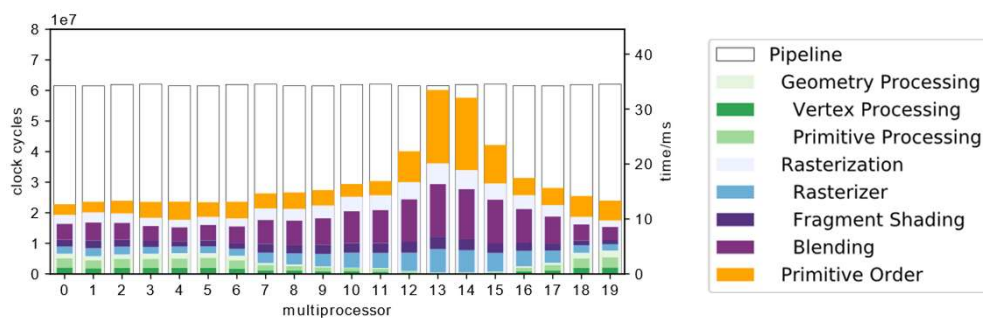
Comparison with Hardware Pipeline

increase shader load \Rightarrow pipeline overhead less significant



We also did some comparisons against the standard hardware pipeline. In this plot, you can see the overhead plotted. As can be seen, there is quite significant overhead compared to the specific hardware units which obviously are faster than a respective software implementation. But we can see that by increasing the shader load, i.e., minimising the overhead accumulated from the software pipeline compared to the hardware pipeline, the performance actually gets quite close to the hardware pipeline overall.

Performance Breakdown

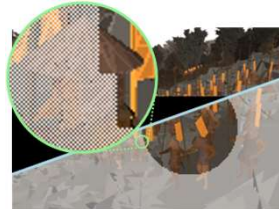
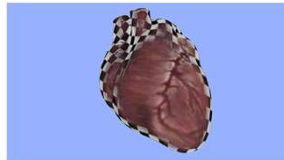


- workload dominated by
 - framebuffer
 - primitive order

We also looked more closely at the performance cost of the individual stages. The workload overall is dominated by the primitive ordering as well as writing to the framebuffer, as ROPs are not directly accessible via software yet. If one could access the ROPs directly and primitive order is not a huge factor, performance would actually be really competitive.

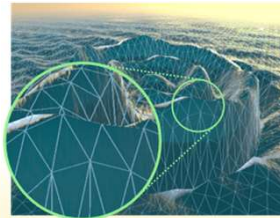
Application examples

mipmapping



foveated rendering with
adaptive sampling rate

programmable primitive
topology



programmable blending

Lastly, what such a modular pipeline design allows are applications which can be quite hard to handle using the traditional, fixed pipeline.

Here we have four examples

- Checkerboard Rendering
- Foveated Rendering with an adaptive sampling rate
- Heightmaps can lead to issues, here the geometry shader could be used but is typically slower
- Programmable blending (different blending that is available with ROPs)

Conclusion

- Task-parallelism vs. Data-parallelism
- Need to organize work
 - **Queues**
- Different scheduling techniques
 - **Time-Sliced Kernels**
 - **MegaKernel**
 - **Dynamic Parallelism**
- Many examples benefit

This concludes our tutorial session, so let's summarize quickly what you should take with you:

- We initially looked at the general CUDA programming model and how it fits to different applications
- We discussed the need to organize work using some data structure and we introduced several variants of a queue
- Then we talked in detail about different techniques for scheduling tasks on the GPU
- Finally, we mentioned a few examples and compared the individual techniques regarding their feasibility on some examples

References

Softshell: Dynamic Scheduling on GPUs

Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel and Dieter Schmalstieg
TOG'12

Hierarchical Bucket Queuing for Fine-Grained Priority Scheduling on the GPU

Bernhard Kerbl, Michael Kenzel, Dieter Schmalstieg, Hans-Peter Seidel and Markus Steinberger
EG'17

A high-performance software graphics pipeline architecture for the GPU

Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg and Markus Steinberger
Siggraph'18

Whippletree: Task-Based Scheduling of Dynamic Workloads on the GPU

Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter and Dieter Schmalstieg
TOG'14

The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU

Bernhard Kerbl, Michael Kenzel, Joerg H. Mueller, Dieter Schmalstieg and Markus Steinberger
ICS'18

Ouroboros: Virtualized Queues for Dynamic Memory Management on GPUs

Martin Winter, Daniel Mlakar, Mathias Parger and Markus Steinberger
ICS'20

Our work in the area of task scheduling, including

- Softshell: Dynamic Scheduling on GPUs
- Whippletree: Task-Based Scheduling of Dynamic Workloads on the GPU
- Hierarchical Bucket Queuing for Fine-Grained Priority Scheduling on the GPU
- The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU
- A high-performance software graphics pipeline architecture for the GPU
- Ouroboros: Virtualized Queues for Dynamic Memory Management on GPUs