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.

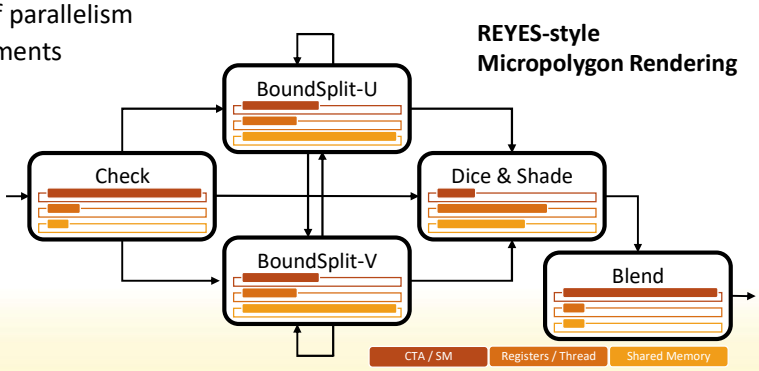
Overview

- 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

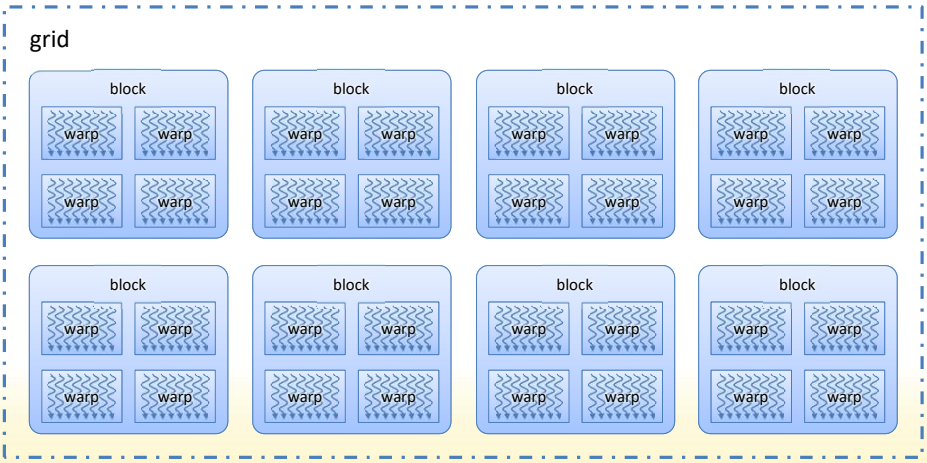
- Parallelize different, independent computation
- Distribute tasks to processors
- e.g., Multitasking, Pipeline Parallelism

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



5

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.

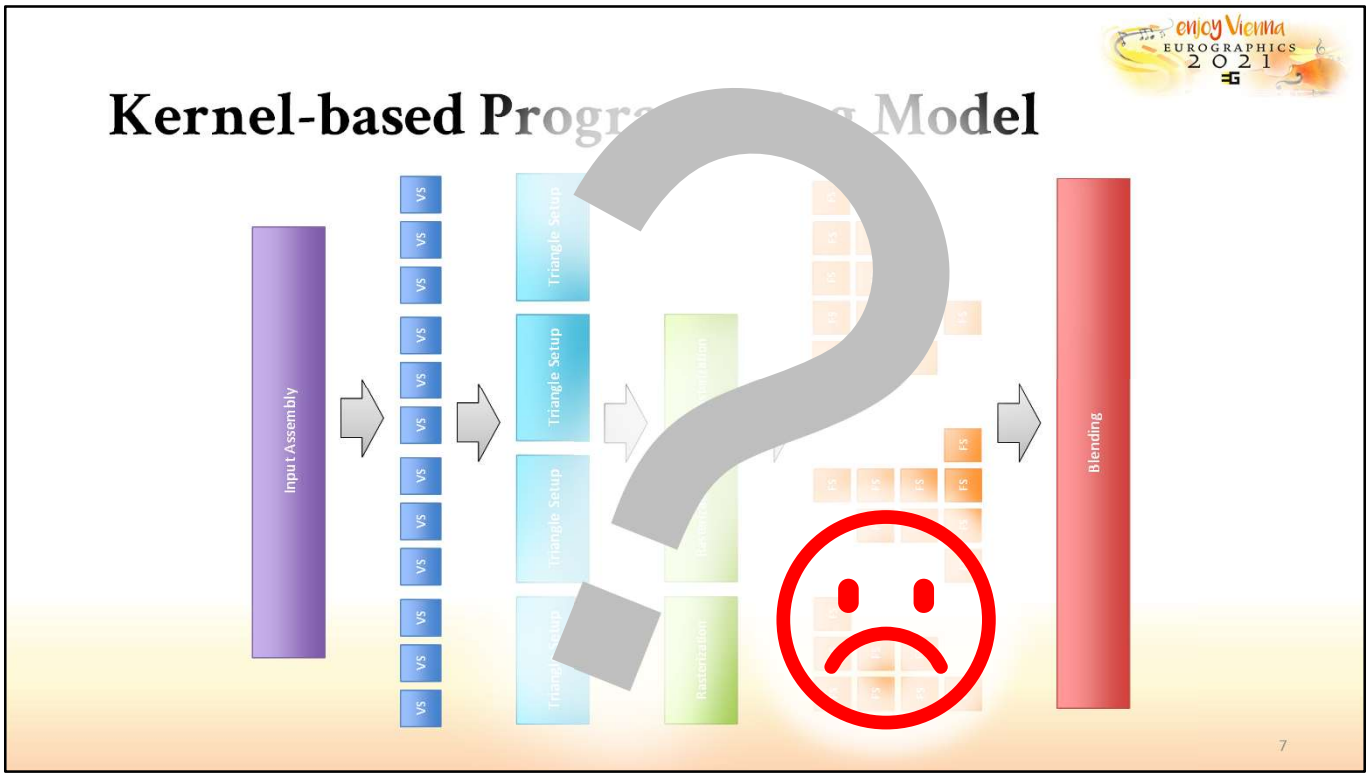
Threads from **different blocks** do not share the same, fast memory in **shared memory**, and also do not have any guarantees if they execute on the same or different SMs or concurrently or one after the other. Hence, threads of different blocks should not rely on cooperation but perform largely independent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

Kernel-based Programming Model

6

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.



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.

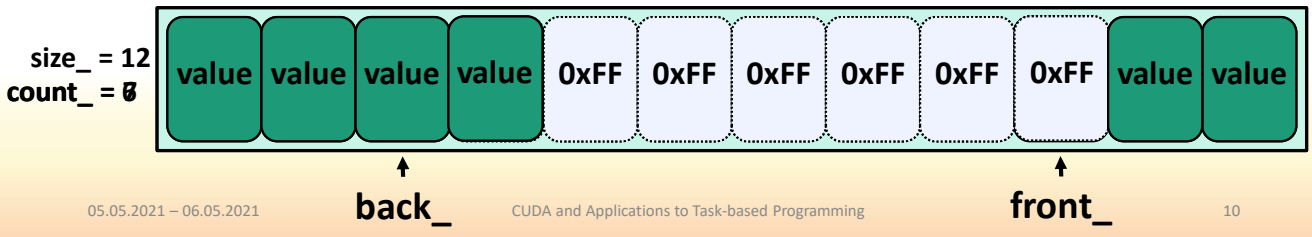
Queues

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;
}
```



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 increment 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

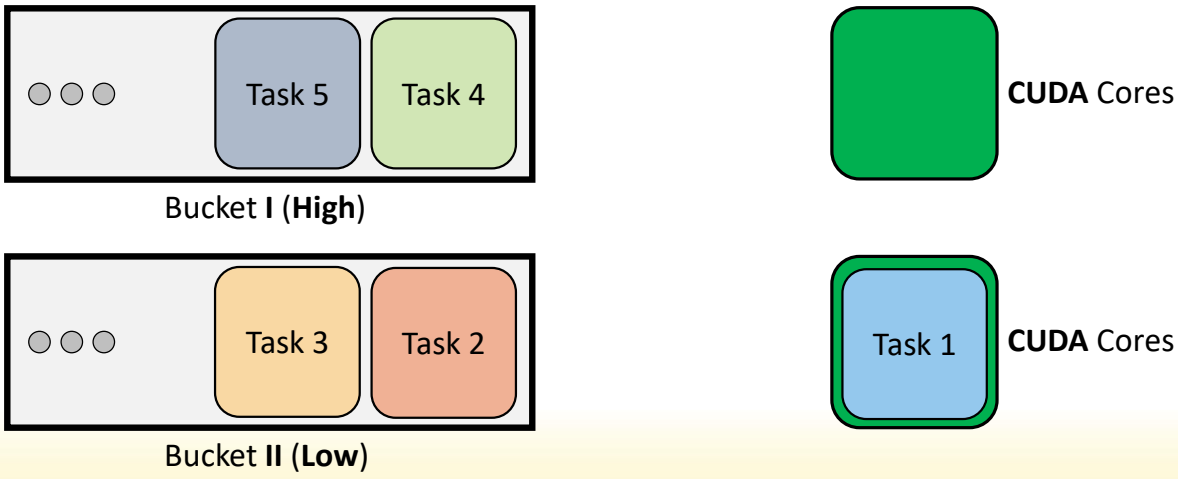
- Applications
  - Prioritization
  - Task Aggregation

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.

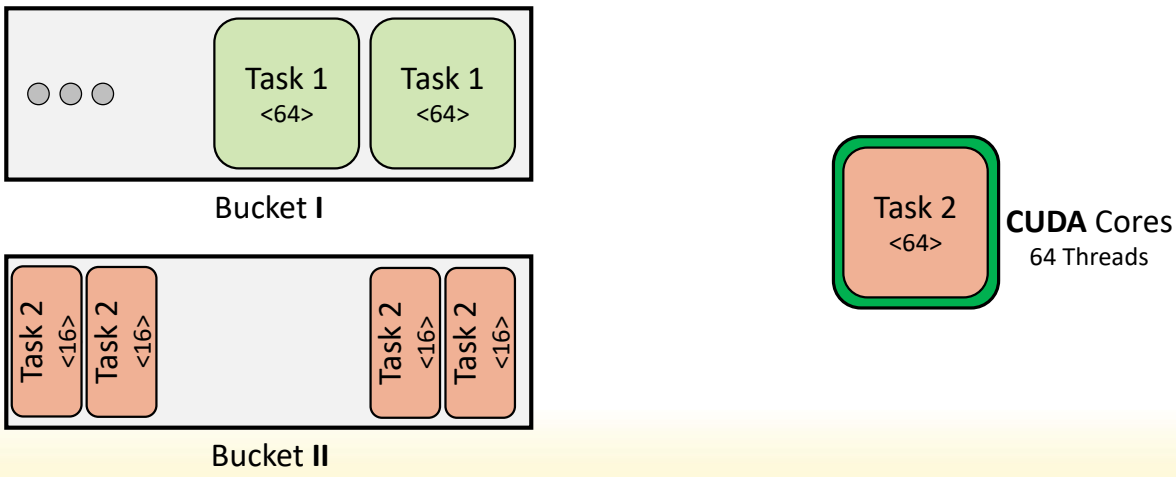
Based on such a design, one can realize new applications, like **prioritization of tasks** as well as **task aggregation**. The underlying queue implementation can follow a similar design to the queue discussed before, but the combination of multiple queues allows for new concepts. This queueing approach 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



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.

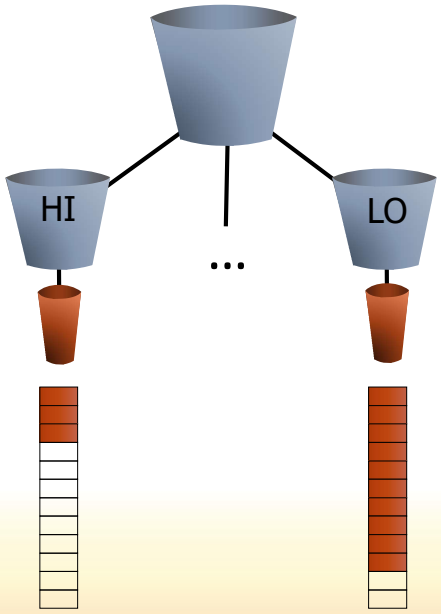
HBQ: Task Aggregation



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

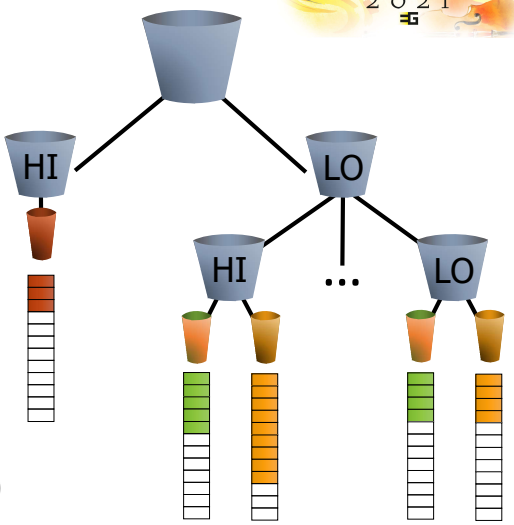


One concrete example for the application of **task prioritization** would be **ray prioritization** in **path tracing**. Here it may make sense to prioritize regions with a high variance, where it can make sense to build up coarse priority intervals, and using the variance as a measure of priority, use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



Render (Dice)
Split horizontally
Split vertically

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

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

- **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

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.

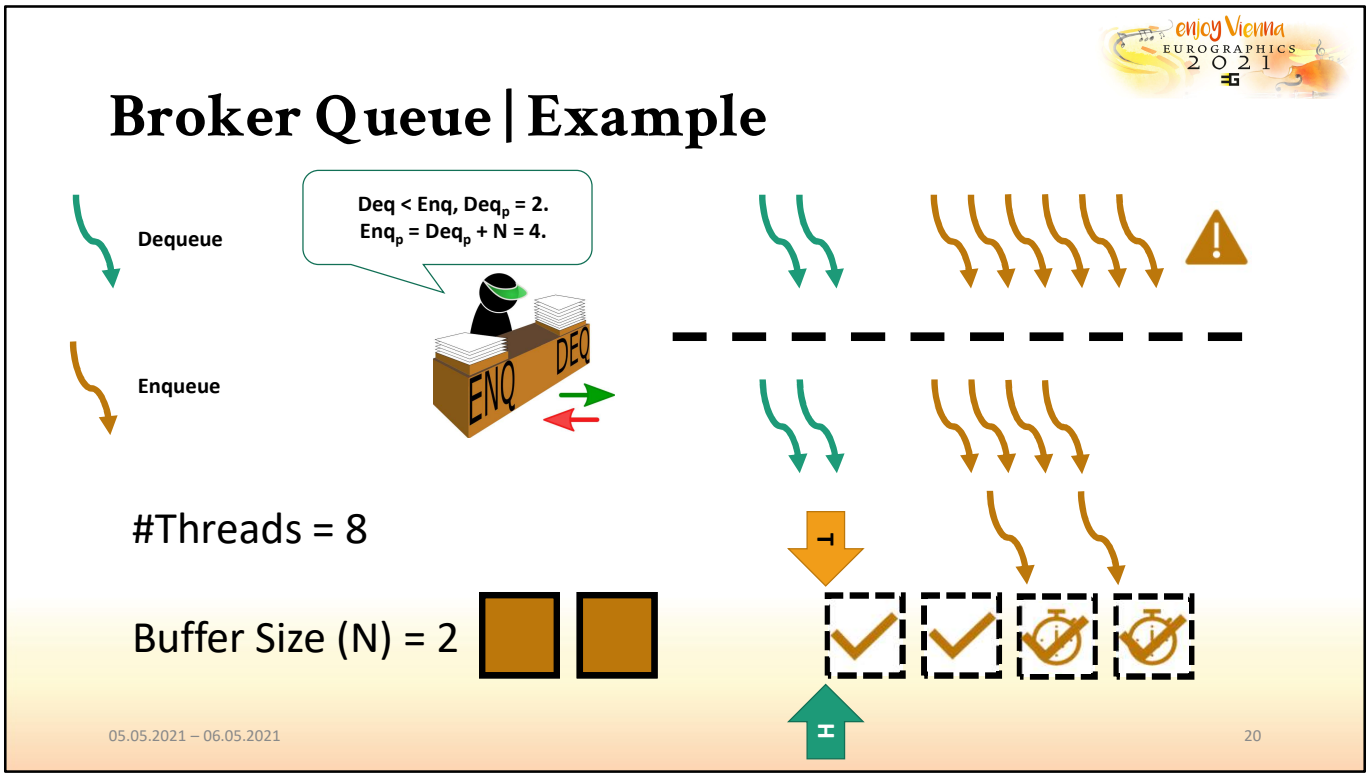
Broker Queue | Enqueue

- **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(T 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;
}
```

```
T 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.



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 enqueuing threads will start their work. As there are more threads present then there are physical queue spaces, the other threads are waiting on tickets to be fulfilled, while two enqueuing 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 dequeing threads can take this work from the queue and signal the tickets of the remaining enqueuing 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

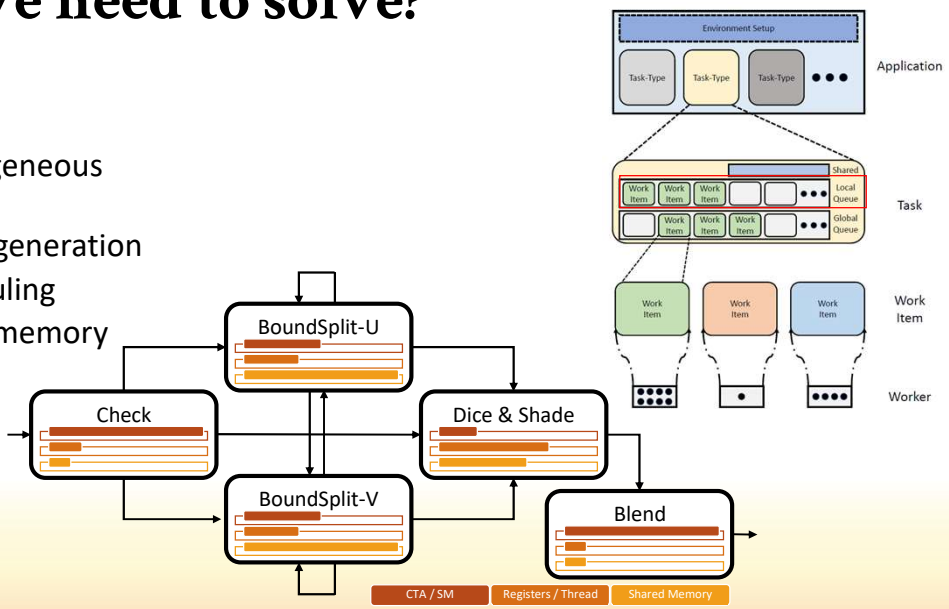
This base design can also be utilized in different, non-linearizable variants, two of which are noted here. By ignoring the loop, one can build a simpler and potentially faster work distribution at the cost of potentially erroneous state information intermittently. Another option would include a so-called 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
  - Handle heterogeneous workloads
  - Dynamic work generation
  - Efficient scheduling
  - Exploit shared memory



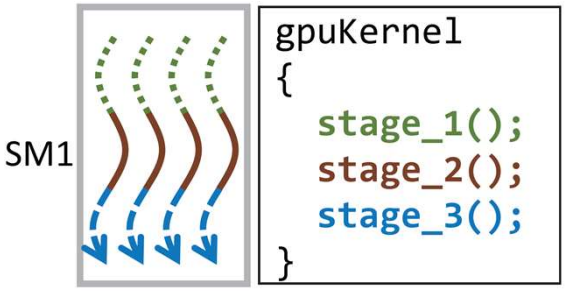
What do we need to solve? What are the properties of applications that our task scheduling system should be able to handle?

- 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



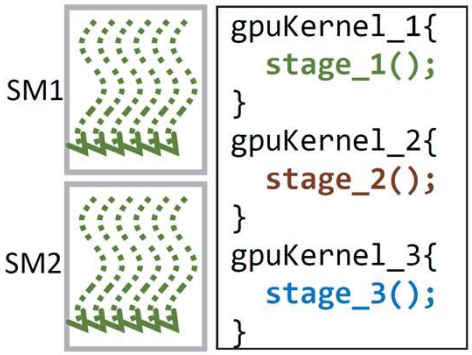
Let's start by investigating very simple models for such task-based applications. One of the simplest, although likely not the one typically chosen, would be the **Run to Completion** model, which puts all stages of our application into one, single kernel.

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

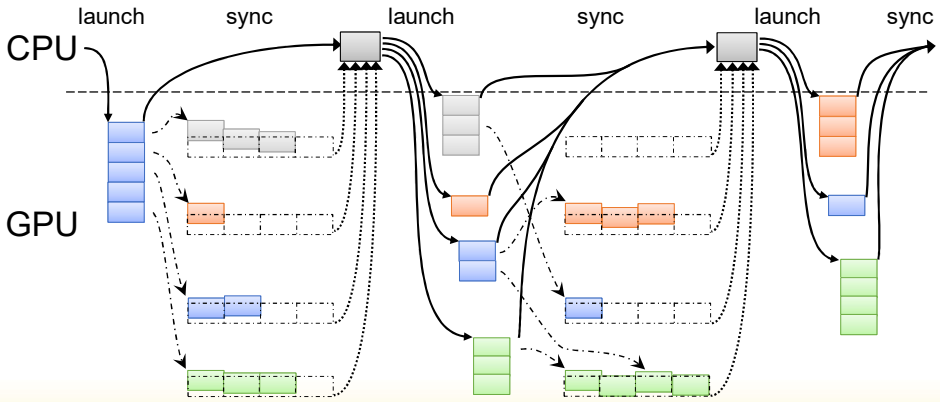
- Variant of **KBK** that supports dynamic work generation

- **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

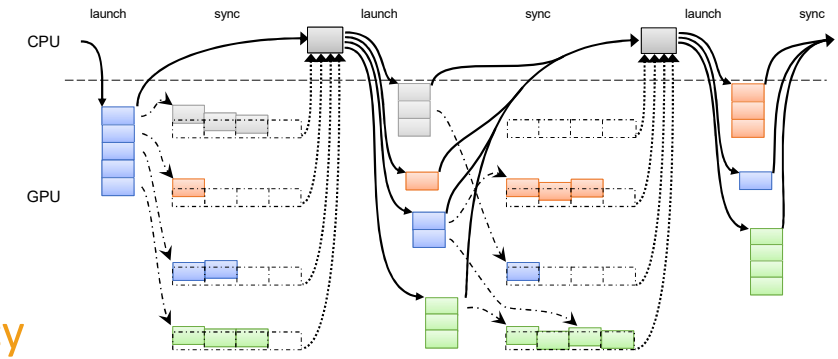


e.g. Laine et al. [2013]

27

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

28

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

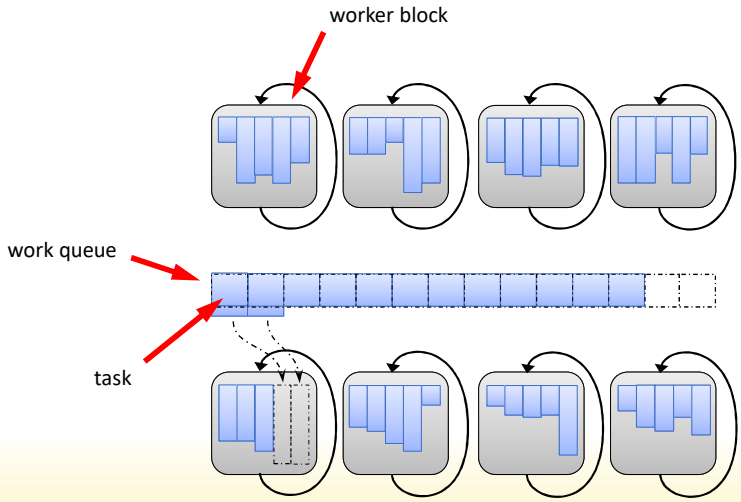
Persistent Threads

- 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



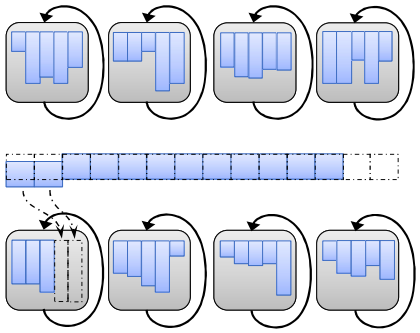
e.g., Aila and Laine [2009]

30

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

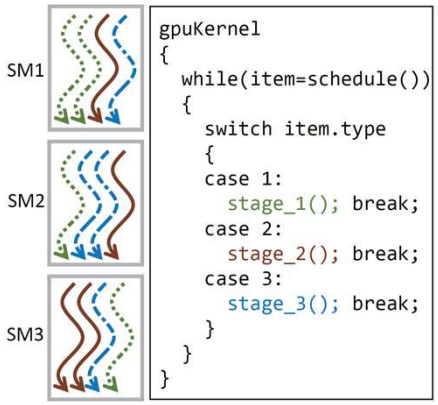
31

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

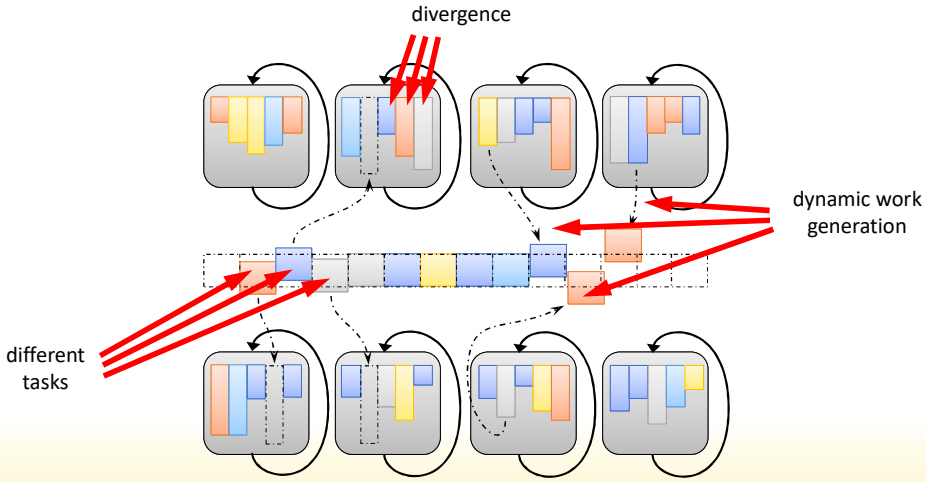


Taking the basic concept from **persistent threads**, i.e. having the blocks execute in a loop on the GPU and drawing in new work from work queues, we can get to so-called **MegaKernels** by allowing for different task types. This requires additional scheduling between the different work queues and depending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

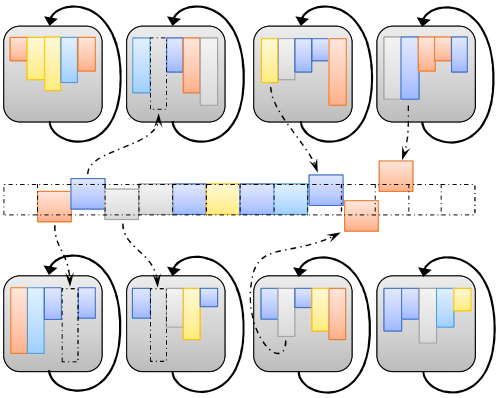
33

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



34

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

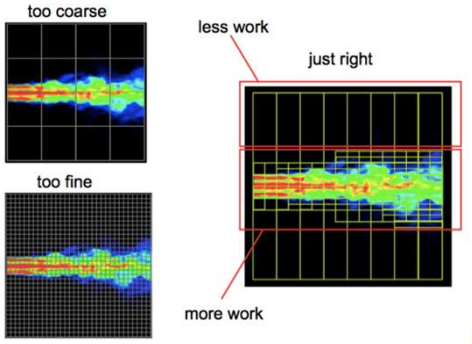
- Implicit load balancing over the blocks, as each can immediately start new work upon finishing execution of "old" work
- The queues support **dynamic work generation**
- And multiple tasks types are support as well

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 *cudadevrt*
  - Compile with *-rdc*

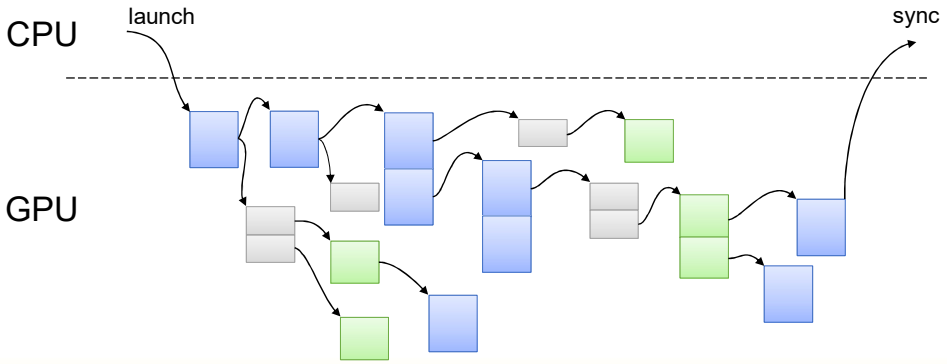


NVIDIA Developer Blog, Adaptive Parallel Computation with CUDA Dynamic Parallelism, 2014
https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism

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

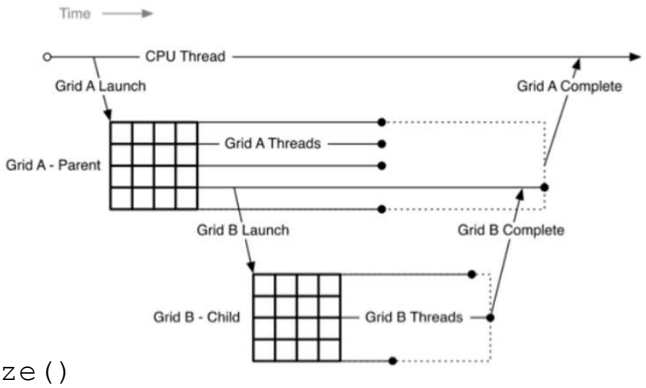


36

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



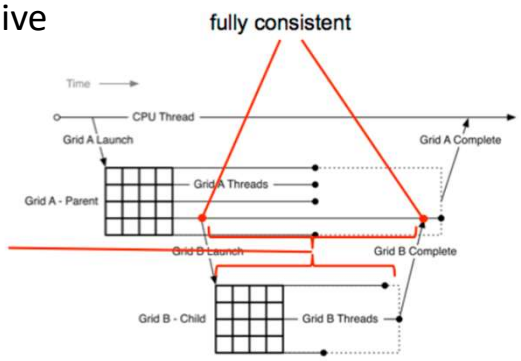
NVIDIA Developer Blog, CUDA Dynamic Parallelism API and Principles, 2014
https://developer.nvidia.com/blog/cuda-dynamic-parallelism-api-principles

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)

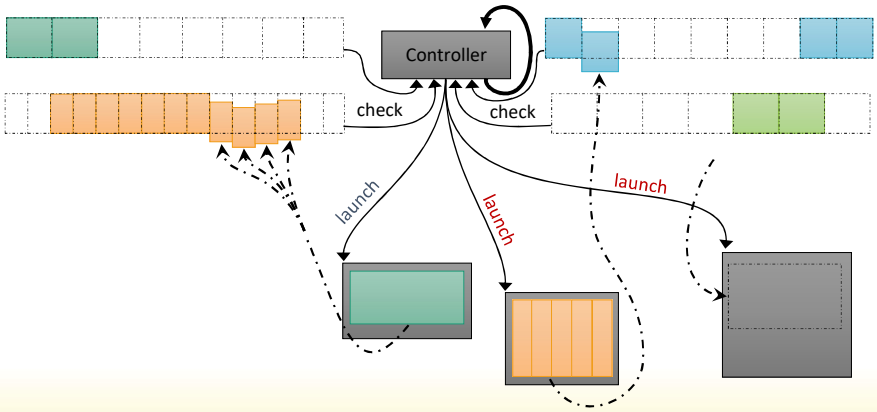
      `cudaLimitDevRuntimeSyncDepth()`
- Pending launches (Default = 2048)
  - `cudaDeviceSetLimit(cudaLimitDevRuntimePendingLaunchCount, 123456)`
  - *Virtualized pool* (more flexible, but additional launches more costly)

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.

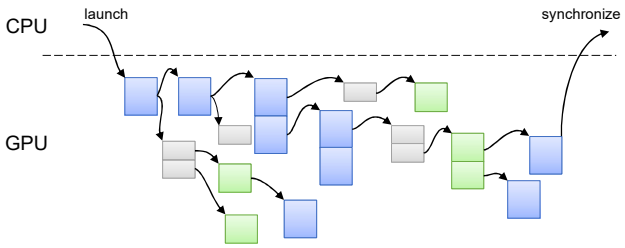
Hybrid Dynamic Parallelism (HDP)



41

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

42

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

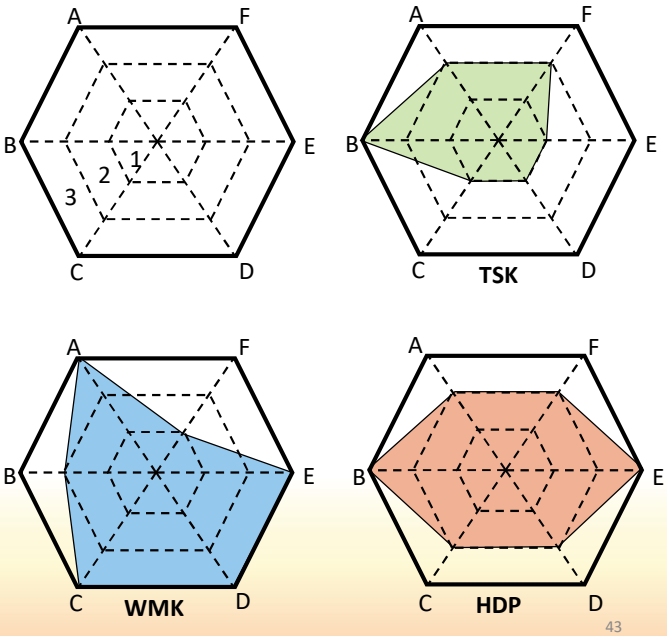
But there are some severe limitations:

- 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

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



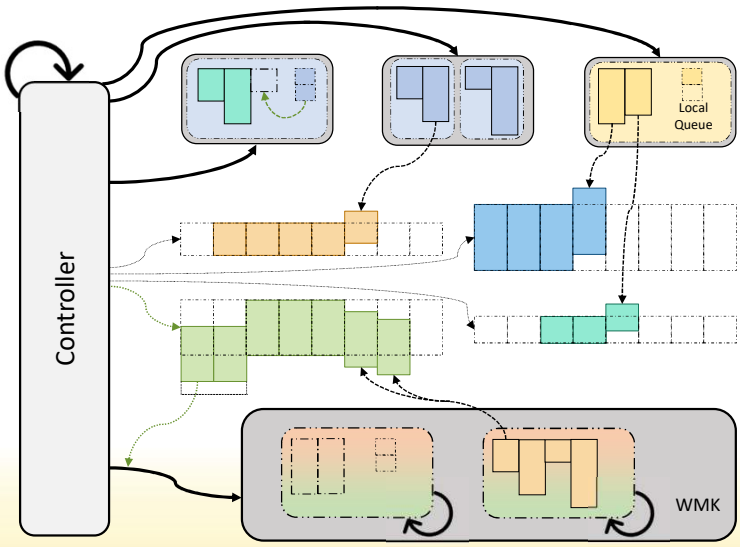
43

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 Queuing: 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

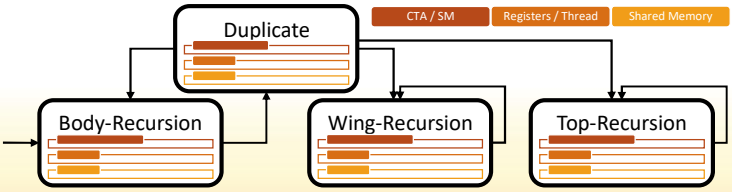
Lastly, let's look at some examples, starting with a few applications that require a task-scheduling framework on the GPU and then we finish on a software implementation of a rendering pipeline.

Procedural Geometry Generation

- Spaceships generated randomly
  - Input
    - Number of Cubes
    - Random Parameter Table
- Recursive Tasks
  - Responsible for different parts of Spaceship
- Very homogenous overall
  - MegaKernel performs best
    - Local Queues help with recursion

Parallel Generation of Architecture on the GPU
Markus Steinberger, Michael Kenzel, Bernhard Kainz, Jörg Müller, Peter Wonka and Dieter Schmalstieg
EG'14

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%



05.05.2021 – 06.05.2021 CUDA and Applications to Task-based Programming 46

First of we can look at **Procedural Geometry Generation,** which we also worked on in a paper on "Parallel generation of architecture on the GPU" by Steinberger and collegeaues.
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.

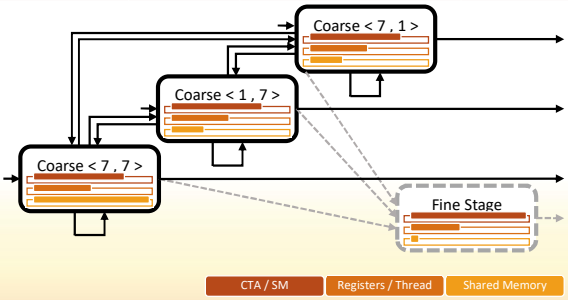
This pipeline is very homogeneous overall with loads of recursive tasks, benefiting from local queueing.
Overall, a **MegaKernel** approach performs best here.

SVG Rendering

Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU
Mark Dokter, Jozef Hladky, Mathias Parger, Dieter Schmalstieg, Hans-Peter Seidel and Markus Steinberger
EG'19

- 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%



05.05.2021 – 06.05.2021 CUDA and Applications to Task-based Programming 47

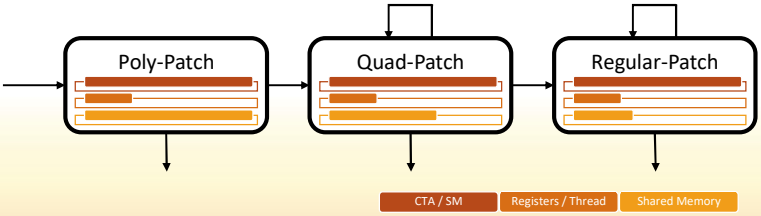
Next we can look at a hierarchical SVG rasterization approach as based on a paper by Mark Dokter and colleagues, called "Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU", consisting of some coarse stages, which determine first the potential coverage and then are 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

Subdivision-Specialized Linear Algebra Kernels for Static and Dynamic Mesh Connectivity on the GPU
Daniel Mlakar, Martin Winter, Pascal Stadlbauer, Hans-Peter Seidel, Markus Steinberger and Rhaleb Zayer
EG'20

- Simple input mesh → detailed geometry
  - Recursive subdivision
  - Split mesh into patches
- Execution
  - Heterogeneous shared memory requirements
  - Large input data
    - up to 500B
  - **TSK** & **HDP** narrowly outperform **WMK**

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%



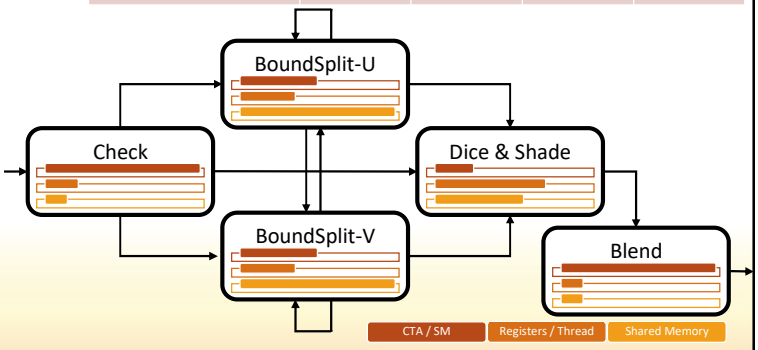
Next we can look at Catmull-Clark Subdivision, where we also did some work in a paper called "Subdivision-Specialized Linear Algebra Kernels for Static and Dynamic Mesh Connectivity on the GPU". Here, we use a simpler implementation, which takes a simple input mesh and, using recursive subdivision by 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%

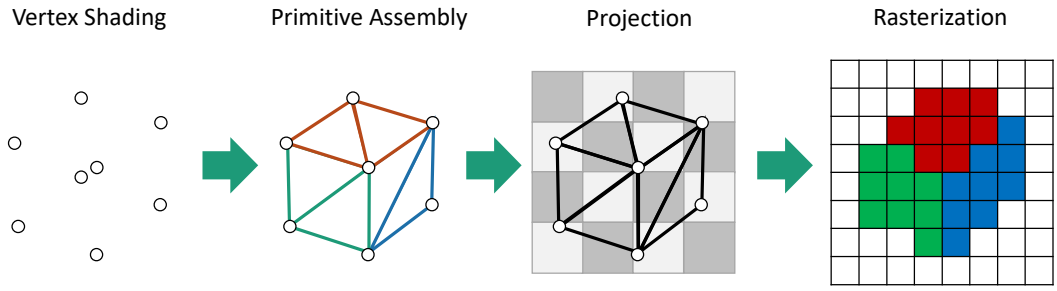


Lastly, the previously mentioned Reyes Rendering, where the scene is recursively split into micropolygons, which are further split up to a certain level and then rendered in the end. Here we have a prime example of a heterogeneous workload, with different numbers of workers per item, different register 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.

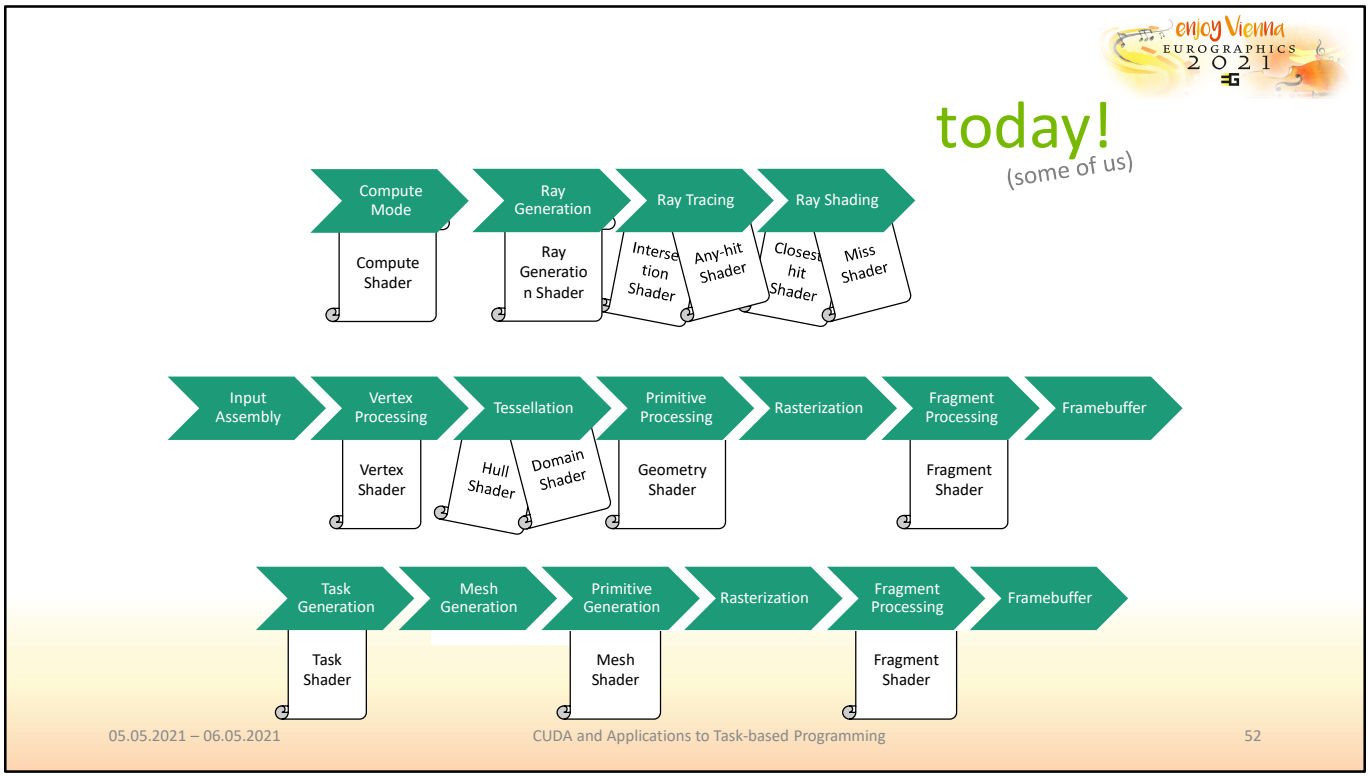
Basic Graphics Pipeline



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, 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.

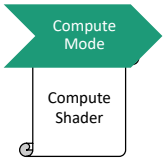
Programmable Hardware Pipeline

⇩

Hardware-accelerated Software Pipeline?

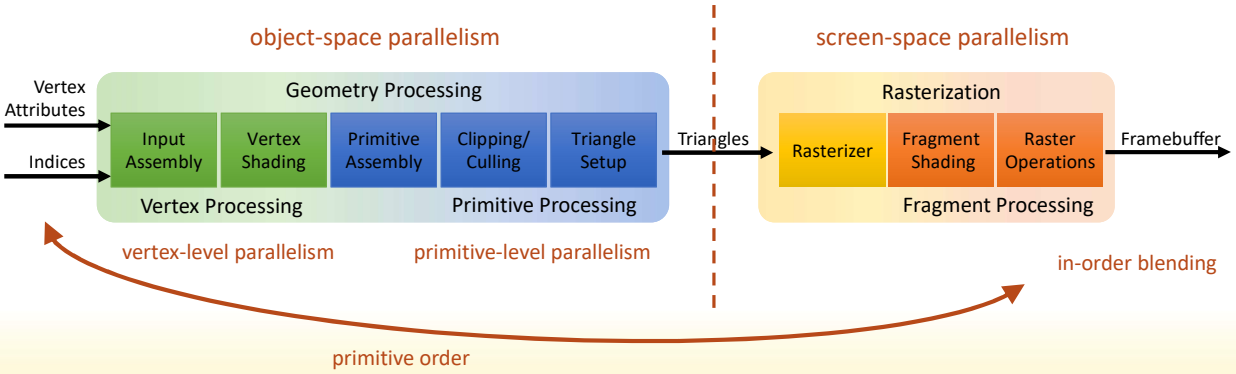
Depending on the actual use case, different pipelines might work best. But still, those pipelines have a rigid structure, which might not fit all scenarios equally well. Hence we thought about the possibility of moving from a programmable hardware pipeline to a hardware-accelerated software pipeline 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?

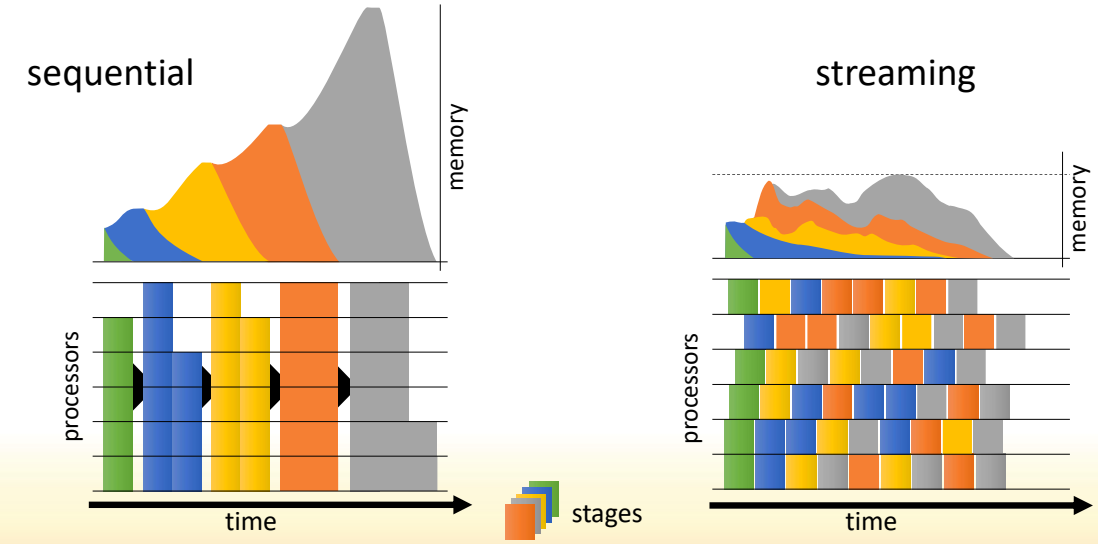
Challenges



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.

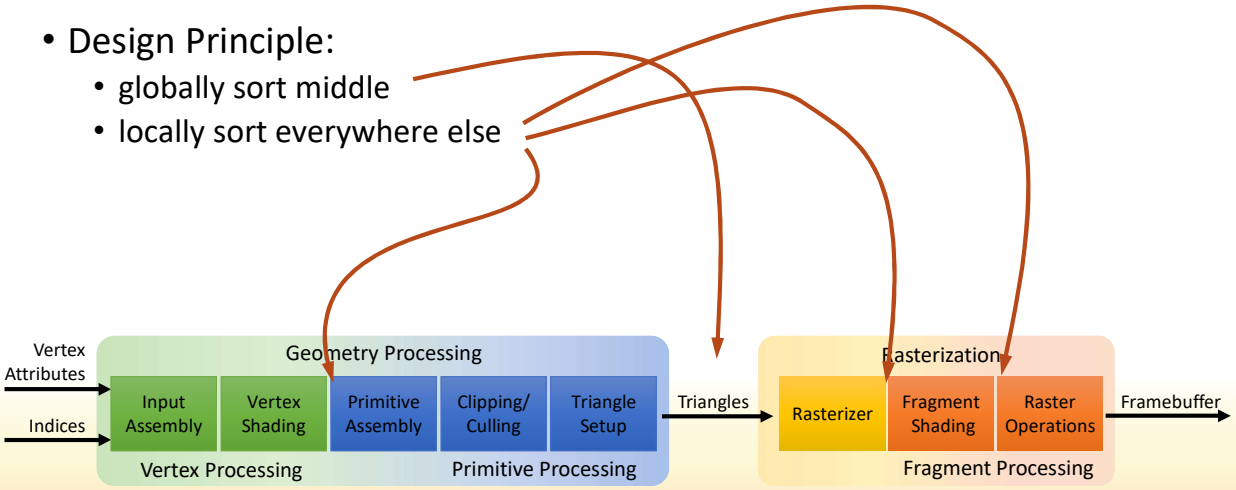
GPU Pipeline Implementation



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?

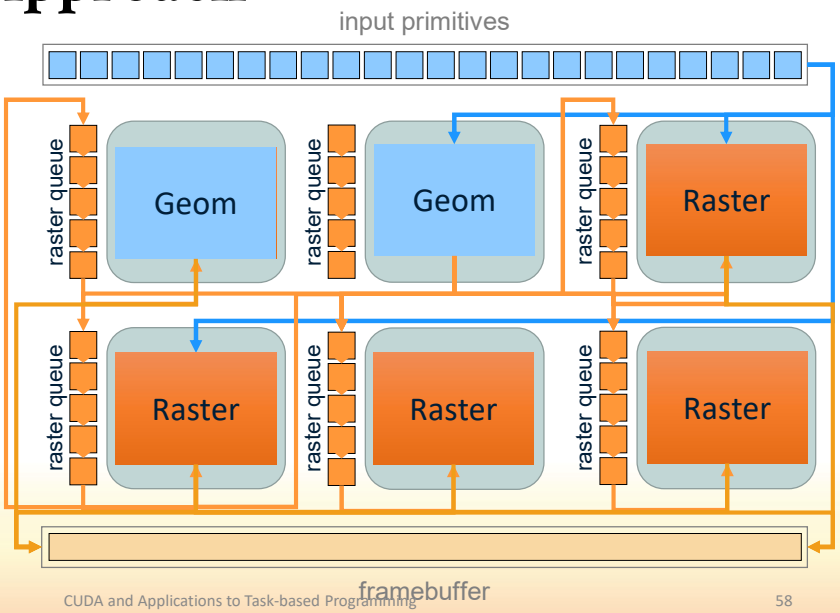
- Design Principle:
  - globally sort middle
  - locally sort everywhere else



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

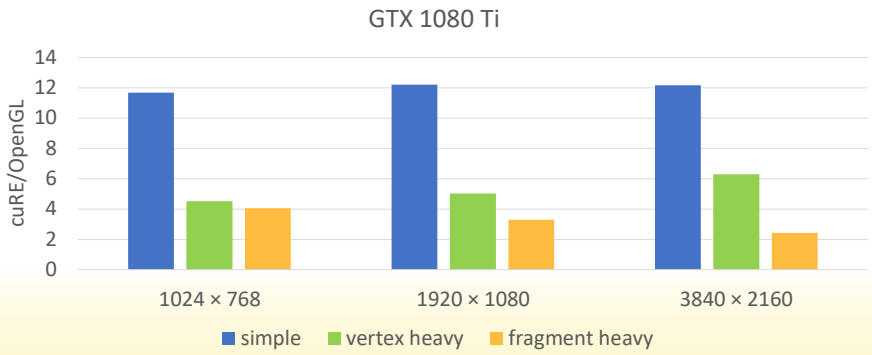
- fill GPU with worker blocks
- run either
  - Geometry Processing or
  - Rasterization
- global load balancing: raster queues
- local load balancing: on-chip buffers



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).

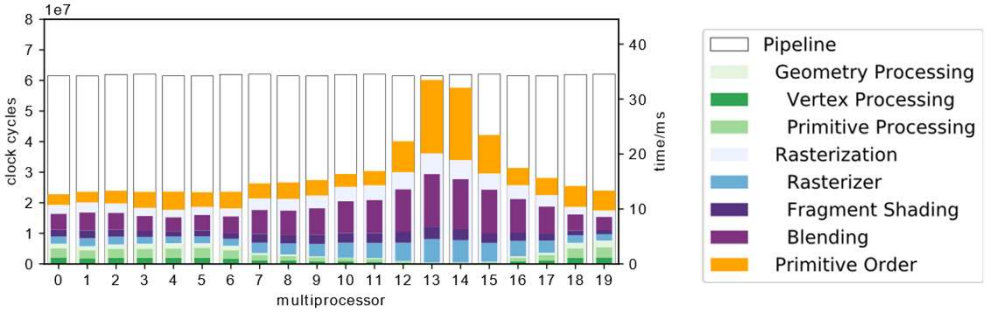
Comparison with Hardware Pipeline

increase shader load ⇨ 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

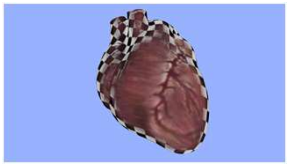


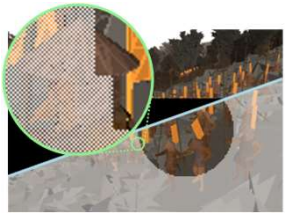
- workload dominated by
  - framebuffer
  - primitive order

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

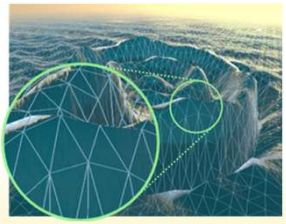
mipmapping





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

Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU
Mark Dokter, Jozef Hladky, Mathias Parger, Dieter Schmalstieg, Hans-Peter Seidel and Markus Steinberger
EG'19

Subdivision-Specialized Linear Algebra Kernels for Static and Dynamic Mesh Connectivity on the GPU
Daniel Mlakar, Martin Winter, Pascal Stadlbauer, Hans-Peter Seidel, Markus Steinberger and Rhaleb Zayer
EG'20

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

Parallel Generation of Architecture on the GPU
Markus Steinberger, Michael Kenzel, Bernhard Kainz, Jörg Müller, Peter Wonka and Dieter Schmalstieg
EG'14

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
- Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU
- Parallel Generation of Architecture on the GPU
- Subdivision-Specialized Linear Algebra Kernels for Static and Dynamic Mesh Connectivity on the GPU