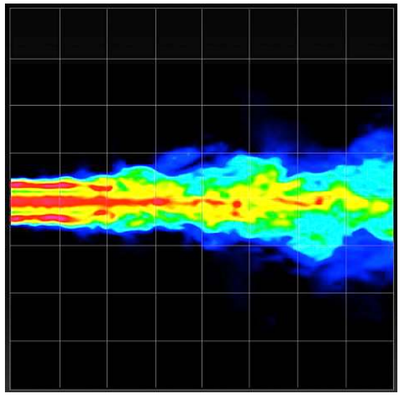
Massively Parallel Algorithms
Dynamic Parallelism

G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

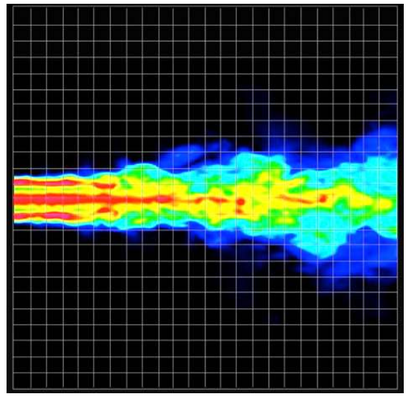
Features

- Kernels may launch other kernels, i.e., a *thread* may instantiate a new and separate *grid of threads*
- Potential benefits:
  - Flow control and kernel scheduling can be performed on the GPU
    - I.e., task parallelism on the GPU (a grid = execution of a task)
  - Allows recursion and subdivision of the problem domain, i.e., dynamic load balancing

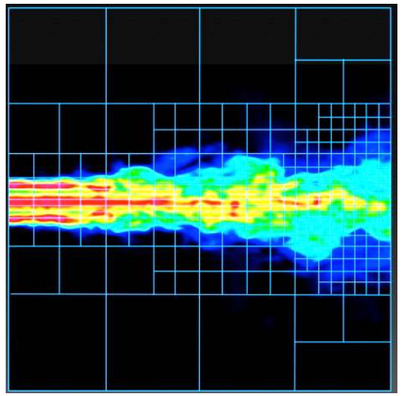
Too coarse in some places



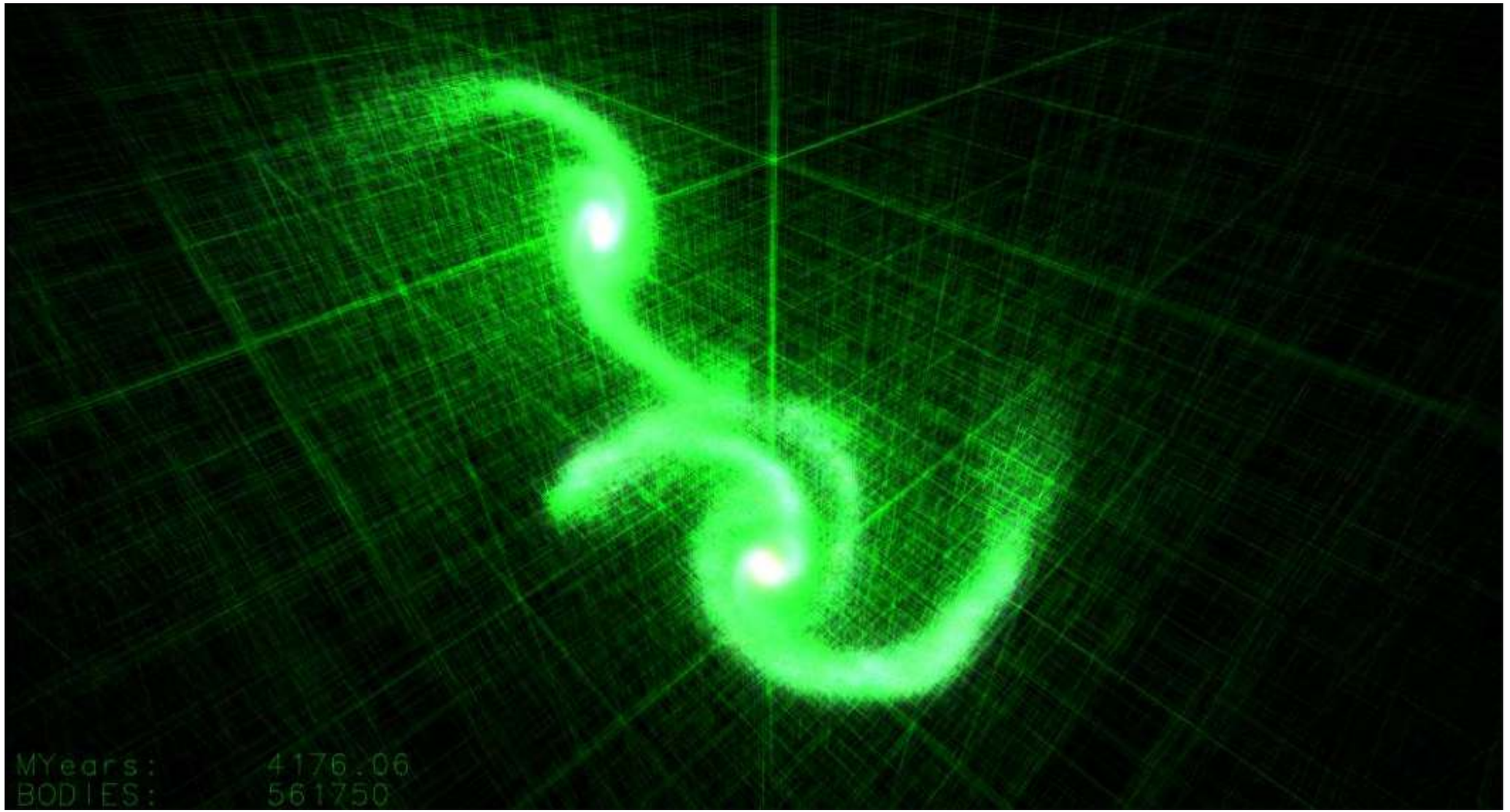
Too fine in some places



Adaptively balanced



Adaptive Domain Partitioning in 3D



General Usage

- Syntax is exactly the same as on host side:

Kernels called from host

Other kernels called from device

```
void main() {
   cudaMalloc( &data, .. );
   initialize data
   A <<< ... >>> ( data );
   B <<< ... >>> ( data );
   C <<< ... >>> ( data );
   cudaDeviceSynchronize();
   retrieve data
}

__global__
void B( float * data ) {
   do_stuff_with data
   X <<< ... >>> ( data );
   Y <<< ... >>> ( data );
   Z <<< ... >>> ( data );
   cudaDeviceSynchronize();
   do_more_stuff data
}
```

Compilation

- Need to add flags & libraries to enable dynamic parallelism
  - Might be the default anyway
- One invocation (compile + link):

```
% nvcc -arch=sm_35 -rdc=true myprog.cu -lcudadevrt -o myprog
```

- Separate compile/link invocations:

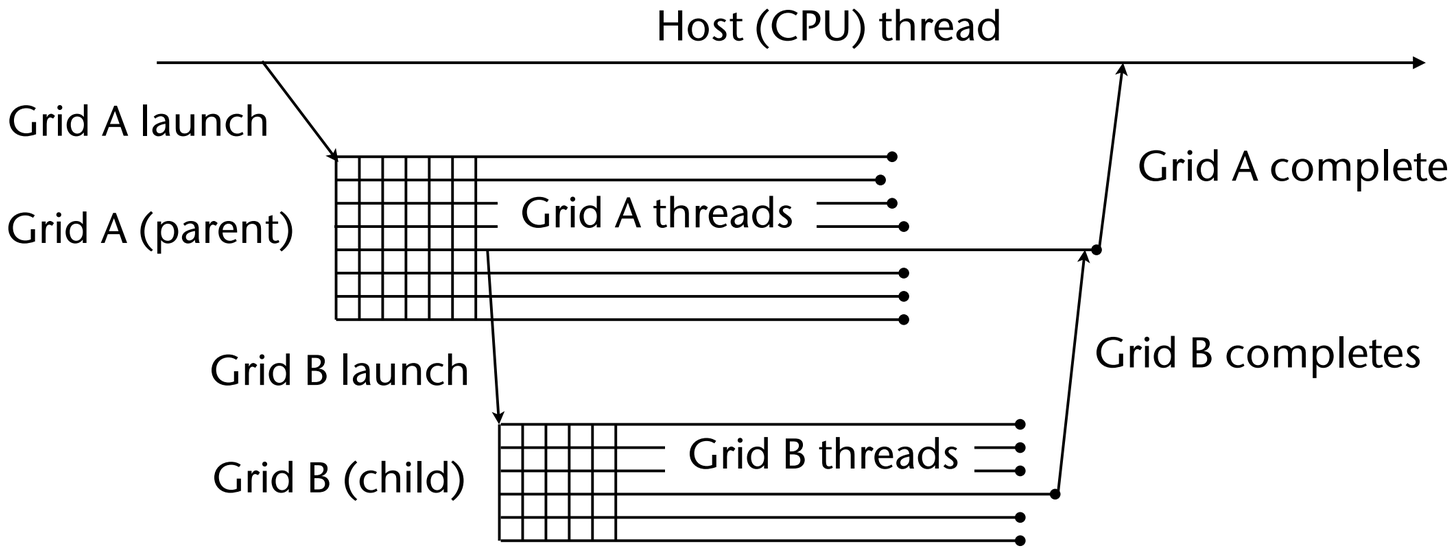
```
% nvcc -arch=sm_35 -rdc=true myprog.cu -o myprog.o

% nvcc -arch=sm_35 myprog.cu -lcudadevrt -o myprog
```

  - **`rdc`** = relocatable device code
  - **`cudadevrt`** = CUDA 5+ runtime library on the device

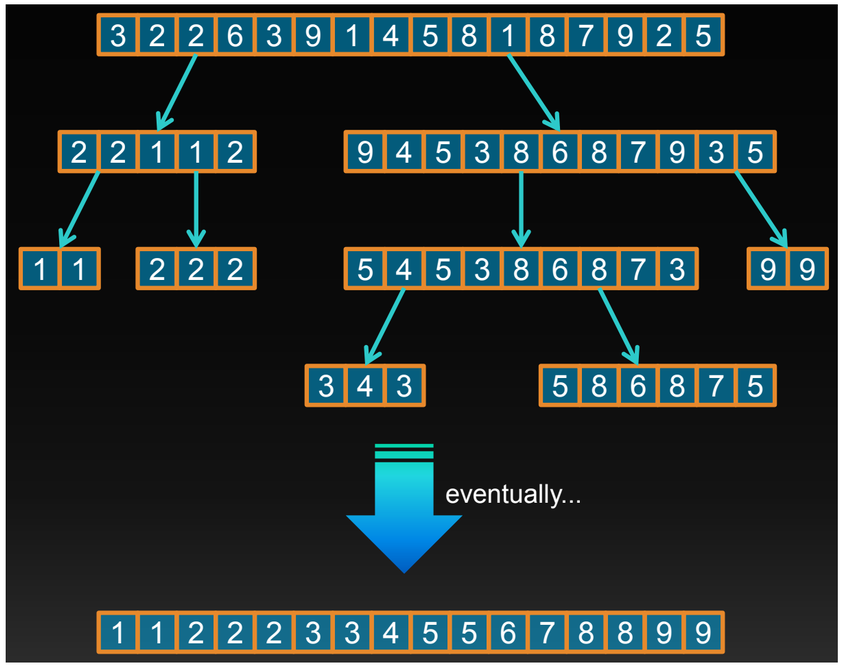
Execution Model

- A kernel launch within a kernel is asynchronous but nested:
  - Parent kernel/grid continues directly after the child kernel launch
    - Just like host code continues after kernel launch
  - Child kernel/grid is guaranteed to be finished before the parent kernel/grid returns
  - Child kernels inherit the shared memory *configuration* of the parent



Simple Example: Recursive Quicksort

- Typical divide-and-conquer algorithm
- Recursively partition and sort data
- Entirely data-dependent execution
- Therefore, notoriously hard to do efficiently with data-*independent* algorithms (e.g., for older Fermi architecture)



Example implementation

```
__global__ void qsort( float * data, int l, int r )
{
   // Partition data around pivot value
   float pivot = data[l];
   int cut_index;           // index where to "cut" data
   partition( data, l, r, pivot, &cut_index );

   // Achieve concurrent kernels by launching left- and
   // right-hand recursive sorts in separate command queues
   cudaStream_t s1, s2;
   cudaStreamCreateWithFlags( &s1, cudaStreamNonBlocking );
   cudaStreamCreateWithFlags( &s2, cudaStreamNonBlocking );

   if ( l <= cut_index )
      qsort<<< ..., 0, s1 >>>( data, l, cut_index );
   if ( cut_index < r )
      qsort<<< ..., 0, s2 >>>( data, cut_index+1, r );
}
```

Things to Watch Out For

- *Memory consistency* is maintained only from parent thread to child grid (only for global memory)
  - OK: if parent thread writes to global memory, *then* launches child grid, the child grid will see the value of the parent in global memory
  - Bad: if child thread writes to global memory, parent thread might not see that value – only with *proper synchronization*
  - This is because of nesting / overlapping of child grid with parent grid
- Example:

```
__device__ int v = 0;
__global__ void childKernel( void )  {
   printf( "v = %d\n", v );
}

__global__ void parentKernel( void )  {
   v = 1;
   childKernel<<<1,1>>>();
   v = 2;                          // race condition!
```

- Caveat 1: because of asynchronous kernel launches, explicitly synchronize by **`cudaDeviceSynchronize()`**, if parent thread needs results of child grid

```
__device__ volatile int v = 0;

__global__
void parentKernel( void )
{
    printf( "Primary\n" );
    childKernel<<<32,32>>>( &v );
    cudaDeviceSynchronize();
    ... do something with v
}
```

Is marked as deprecated since CUDA 11.6! Will be replaced by a "replacement programming model" in later CUDA versions!

- Caveat 2: remember, all threads execute the same code!
- How many threads running **`childKernel`** does this program launch?
- $2^{20} \approx 1$ Million!

```
__global__
void childKernel( void )  {
    printf( "Secondary\n" );
}
__global__
void parentKernel( void )  {
    printf( "Primary\n" );
    childKernel<<<32,32>>>();
}
int main( void )
{
    printf( "Hello World!\n" );
    parentKernel<<<32,32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

- Solution (here in case you want only one child grid per parent thread block):

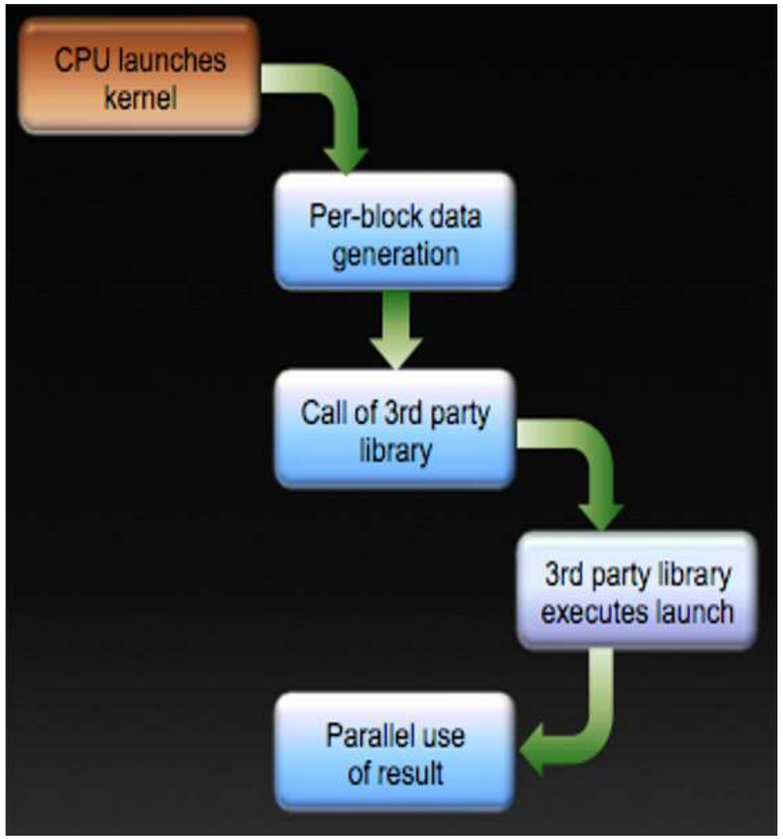
```
if ( threadIdx.x == 0 )
    childKernel<<< num_blocks, n_threads_per_block>>>( );
```

- Caveat 3: **`cudaDeviceSynchronize()`** only knows about the child grid(s) launched by "its own" (parent) block
  - It has no idea about the status / execution progress of any child grids launched by other threads in other (parent) blocks!
  - So, usually, the idiom to wait for the completion of the child grid is:

```
void parentKernel( void )  {
   ...
   __syncthreads();              // common pattern here
   if ( threadIdx.x == 0 )
   {
      childKernel<<<n,m>>>( );
      cudaDeviceSynchronize();
   }
   __syncthreads();              // all threads in block wait
   all threads in this block consume the child grids' data
```

Simple Example: cuBLAS Calls

```
__global__
void libraryCall( float *a, float *b, float *c )
{
  // All threads generate data
  createData( a, b );
  __syncthreads();
  if ( threadIdx.x == 0 )
  {
    // Only one thread calls library
    cublasDgemm( a, b, c );
    cudaDeviceSynchronize();
  }
  // All threads wait
  __syncthreads();
  // Now continue
  consumeData( c );
}
```



Things to Watch Out For

- Shared memory and local memory is private to each block of threads, *cannot* be seen by child threads
- So, how to return a value from a child kernel?
  - Remember, kernel declarations need to be **`void`**

```
__global__ void
child_kernel( void * p )
{   ...   }

__global__ void
parent_kernel( void )
{
   ...
   int r = 0;
   child_kernel<<<1, 256>>>( &r );
   ...
}
```

```
__global__ void
child_kernel( void * p )
{   ...   }

__device__ int r = 0;    // global mem
__global__ void
parent_kernel( void )
{
   ...
   child_kernel<<<1, 256>>>( &r );
   ...
}
```

- Caveat 4: beware of out-of-memory
  - Each child kernel launch requires ~150 MB of GPU memory
    - For storing the state of *all* threads (potentially maximal number)

Example: the Mariani-Silver Algorithm for the Mandelbrot Set

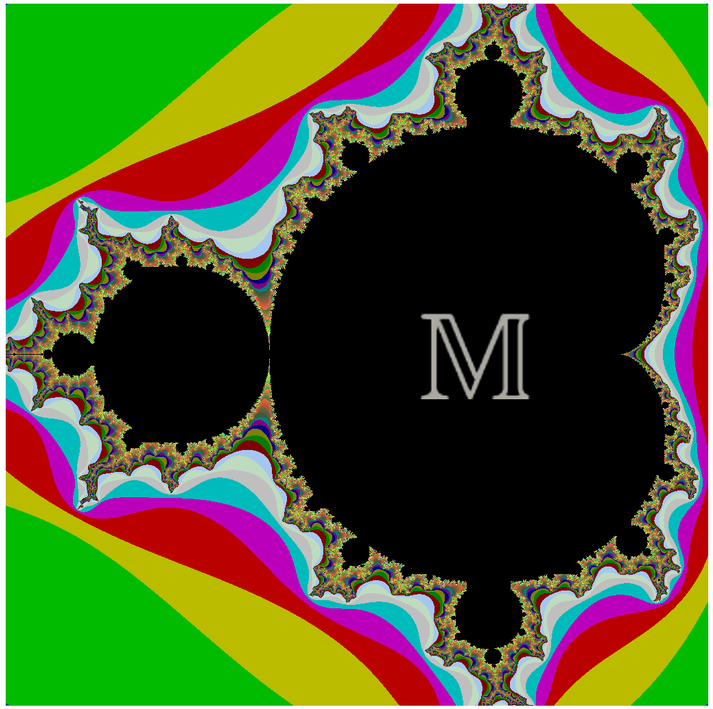
- Reminder: we need to calculate this sequence for all "pixels" $c \in \mathbb{M}$

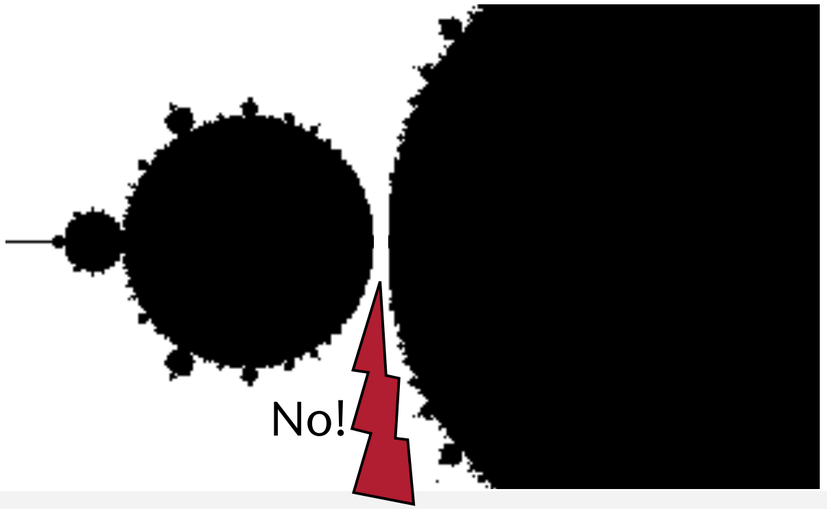
$$z_{i+1} = z_i^2 + c\ , \quad z_0 = 0$$

- Denote with escape count the first index i where $|z_i| > 2$
- Recap: the "escape" algorithm to compute the Mandelbrot image
  - Input: c_{min} and c_{max} = lower left & upper right corner of window in $\mathbb{C}$, resp.

```
for each pixel c inside (cmin,cmax) do in parallel:
   esc = compute_escape_count( c, max_iter )
   if ( esc == max_iter )
      set pixel c's color = black
   else
      convert esc to color, e.g., hue value in HSV
      set pixel c's color
```

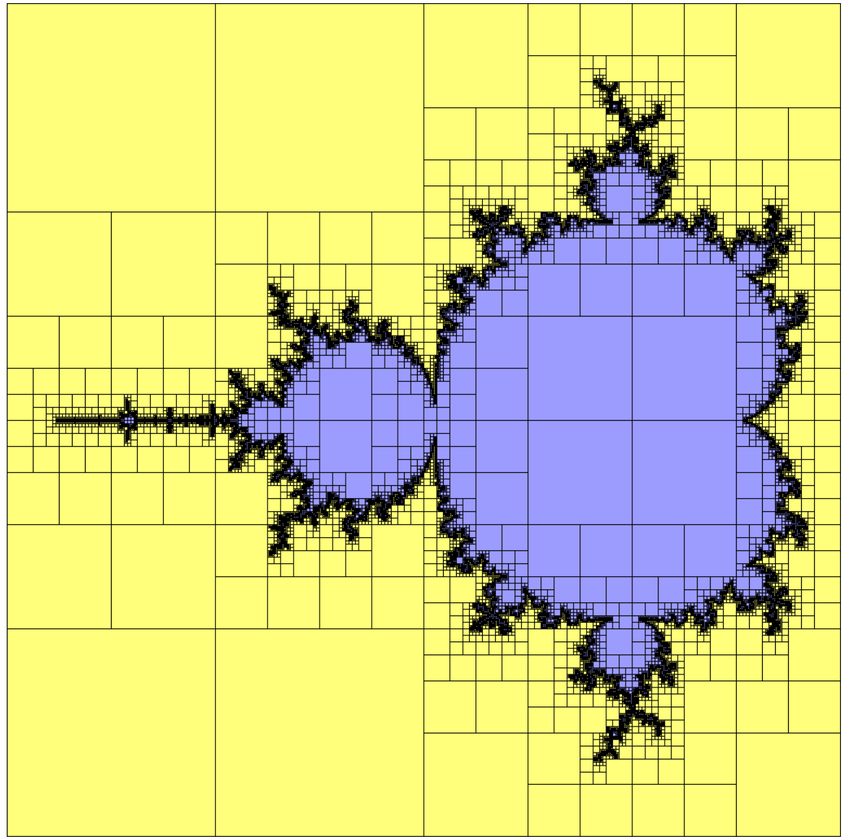
- Observation: far away from $\mathbb{M}$, large regions have the same esc value
- Theorem (w/o proof):
  $\mathbb{M}$ is connected
- Consequence: if the border of a region *R* is (completely) inside $\mathbb{M}$, then all points of *R* are inside $\mathbb{M}$
- Goal: exploit these two facts to create an *adaptive algorithm*
  - Algorithm will spend more time on pixels close to the border of $\mathbb{M}$





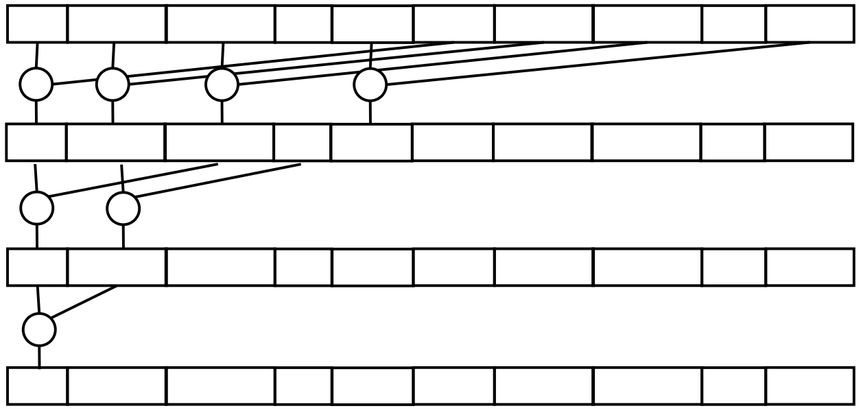
Approach

- Calculate escape value for all pixels on the border of a sub-rectangle inside the current window
- Check whether or not all escape values on this border are equal
- If yes, color all pixels in this sub-rectangle with same color
- If no, partition rectangle into smaller sub-rectangles and recurse
- Stop recursion, if size of rectangle is "small enough", continue with vanilla algo



A Few Details

- Escape values for all pixels on the border:
  - Don't necessarily need to sample the border by pixel distance; we could sample border more densely or more coarsely
  - Error "probability" introduced by sampling will be larger or smaller, resp.
  - Store escape values in shared memory (for second step)
- Check whether all escape values are equal: use parallel reduction with '==' as binary operator

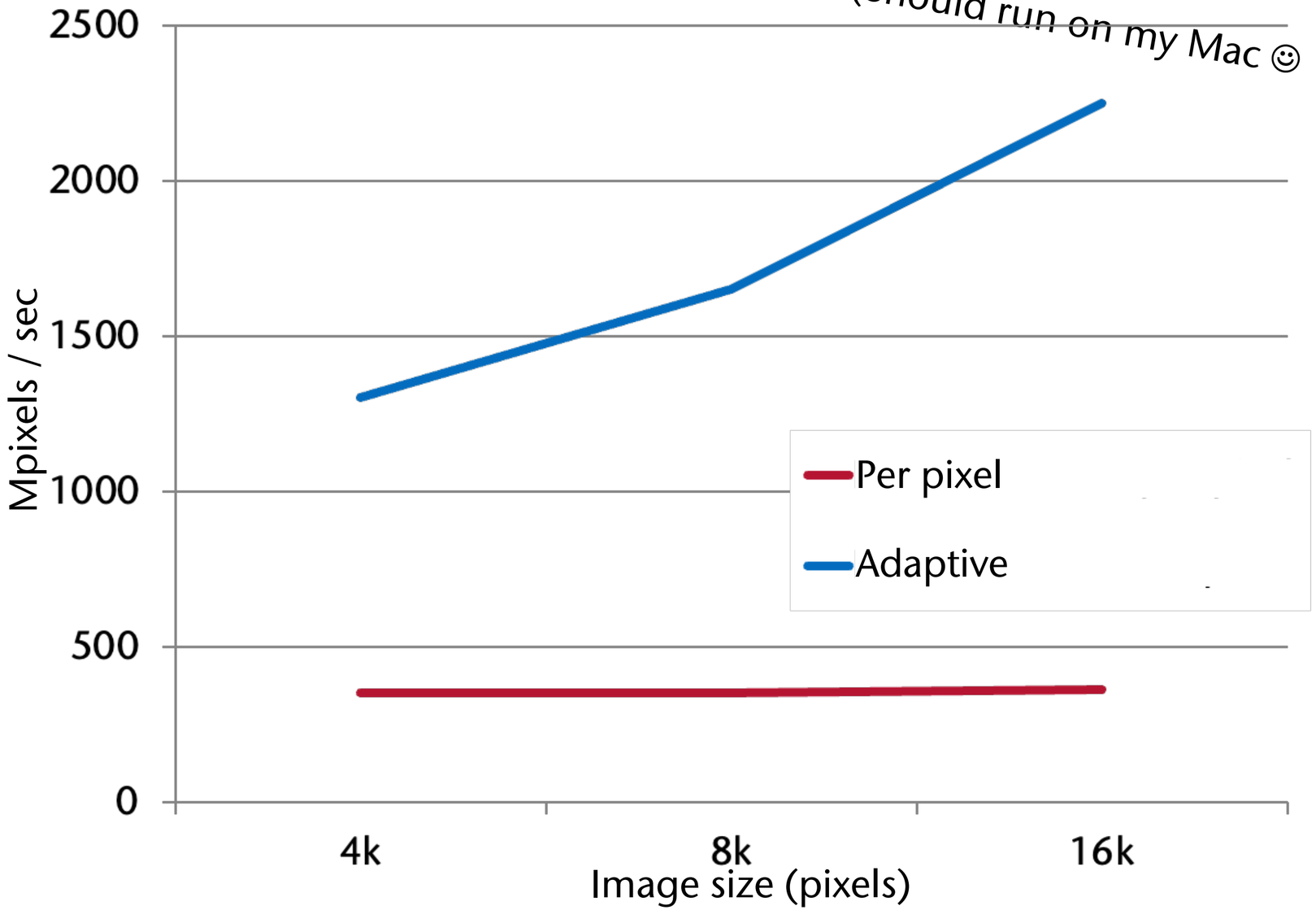


Implementation

```
int main( void )
{
   // set up initial grid of blocks
   dim3 threads_per_block( ... );
   dim3 blocks( ... );
   // call kernel with user-defined rectangle of C
   mandelbrot<<<...>>>( image, cmin, cmax );
}
```

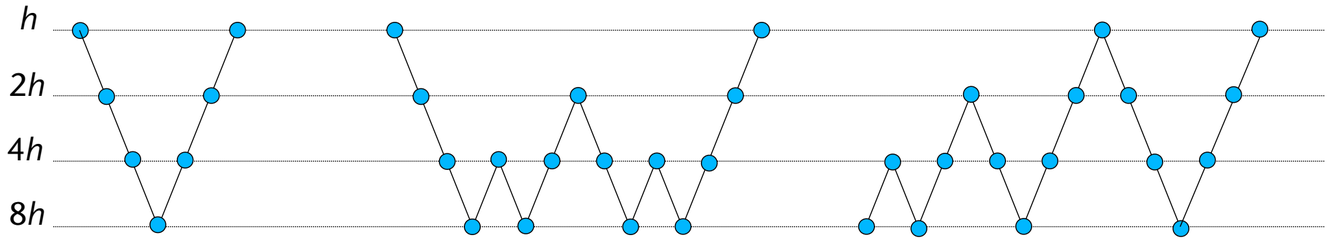
```
__global__ mandelbrot( Color image[..][..], cmin, cmax )
{
  // each block corresponds to a rectangle in [cmin,cmax]
  if ( size(rectangle(blockIdx)) ) < blockDim.x * blockDim.y )
      every thread within block computes its "own" pixel color
   else
   {
      // every thread calculates esc for a point on the border
      __shared__ int esc[blockDim.x][blockDim.y];         // overkill
      calcBorderEscValues( blockIdx, blockDim, cmin, cmax, esc );
      reduce esc, afterwards esc[0][0] contains answer
      if ( esc[0][0] says "equal" )
         parallel fill this block's rectangle with color(esc)
      else
      {
         if ( threadIdx.x != 0 || threadIdx.y != 0 )
            return;                  // only thread 0 will launch sub-grid
         cmin_of_block = cmin + (cmax-cmin)/blockDim * blockIdx;
         cmax_of_block = ...
         mandelbrot<<<...>>>( image, cmin_of_block, cmax_of_block );
      }
   }
}
```

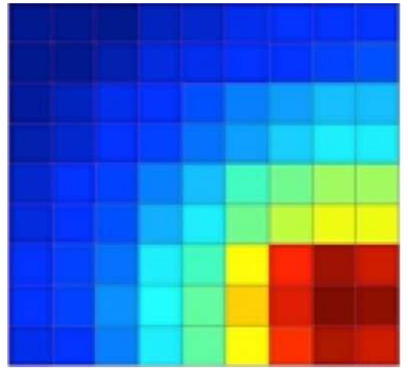
Bachelor thesis: anyone interested to implement an interactive Mandelbrot viewer using this algorithm?
(Should run on my Mac ☺ and have a few sliders)

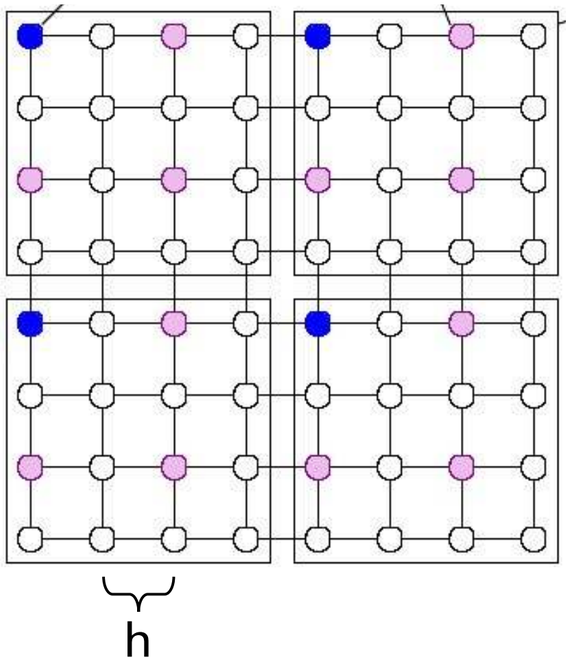


A General Pattern for Solvers on Grids

- Whenever you solve differential equations on a grid, consider to use the multigrid method
  - E.g., heat transfer
  - Propagate values on coarse grid (distribute over wide range) and on fine grids (for accuracy)
- Several variants, how to switch from coarse to fine and back ("V cycle", "W cycle", ...)







Fine-Grained Priority Scheduling on the GPU

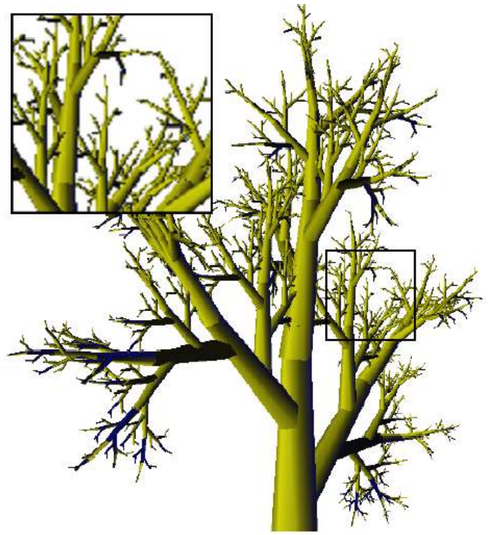
- Goal: fine-grained dynamic load balancing
  - Tasks can have different priorities
- Scenario:
  - Launch one kernel on a number of initial tasks
  - Each thread works on a task and produces more tasks
  - New tasks are assigned a priority (by the producing thread)
- Solution:
  - "Megakernels" (one kernel runs until all tasks are done)
  - Must make sure that threads within block work on tasks of same type

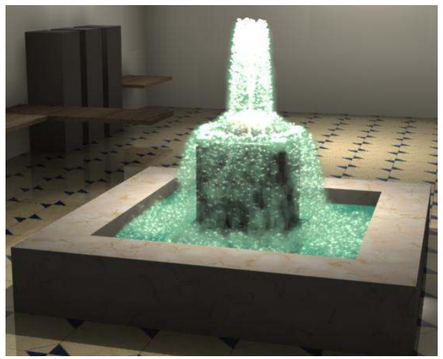
Example Application: REYES Rendering Pipeline

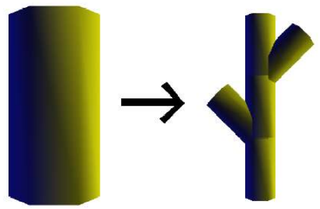
- Invented at Lucasfilm (later Pixar) by Cook et al. 1987
- Goal: no compromise on visual quality (antialiasing, motion blur, transparency, etc.)
  - Real-time is not of concern
  - Compare to OpenGL
- Implemented in Pixar's *Renderman*
  - And many more implementations
  - Pixar has defined a Renderman standard ("Postscript for 3D")

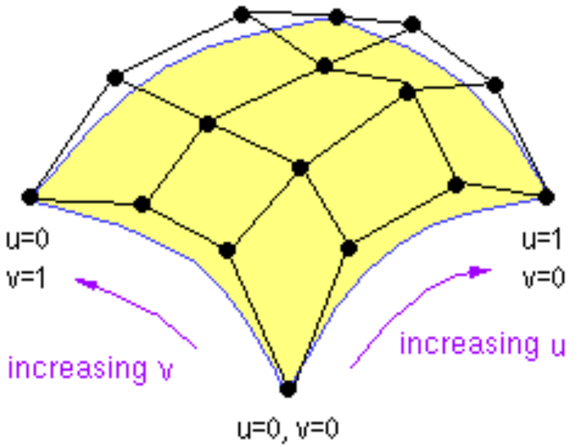
Standard renderer for most animation movies

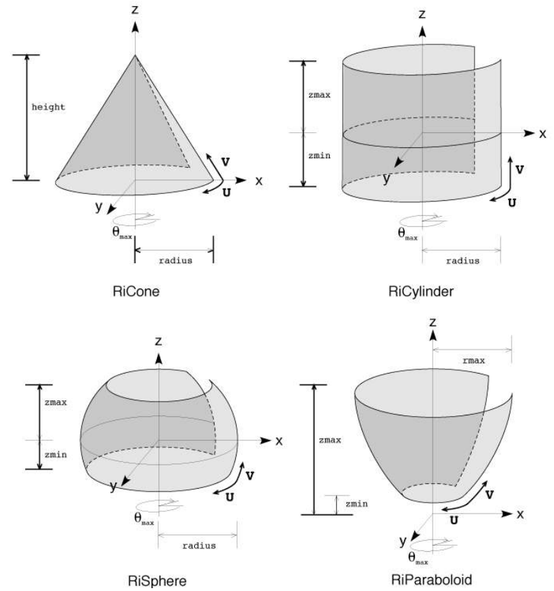
- Available primitives: lots of high-level primitives!
  - Bezier- and B-Spline-Surfaces
  - Quadrics
  - Procedural objects, e.g. L-systems
  - Particle systems
  - ...





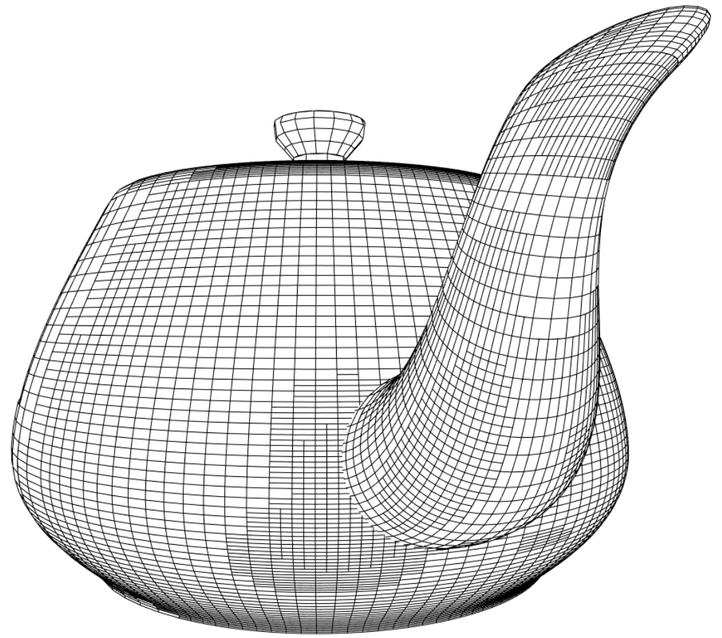






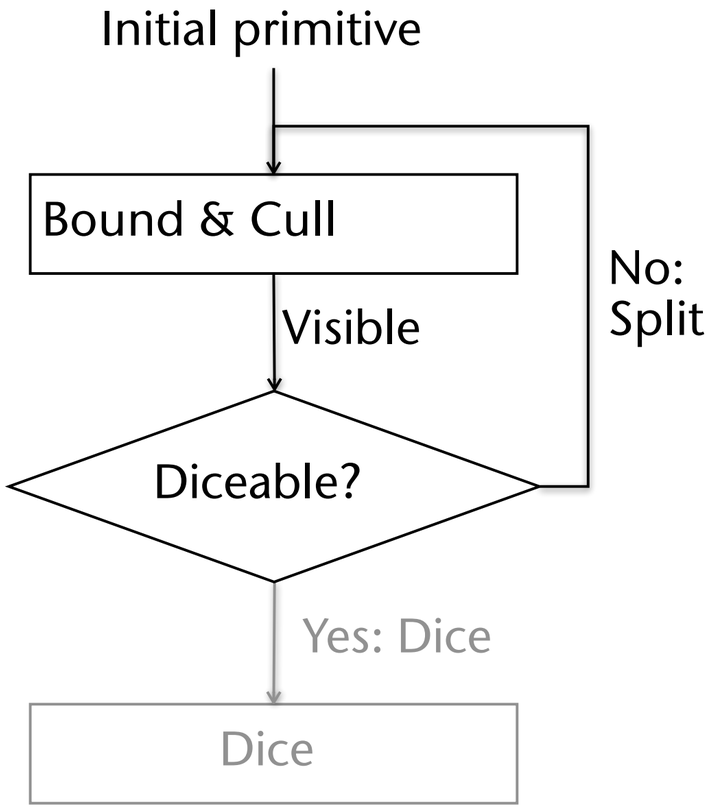
Stages of the Reyes Rendering Pipeline

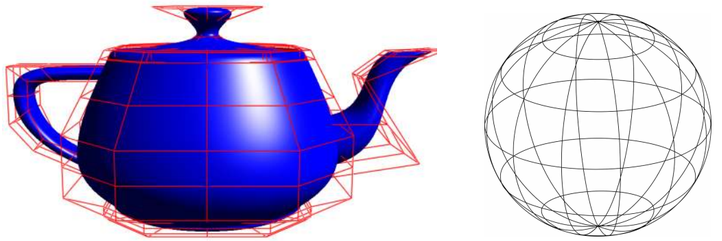
1. Input: higher-order surfaces
2. Generate "micropolygons" from input: "split & dice"
   - Split = recursively subdivide primitive into patches until "small" enough
   - Dice = uniformly tesselate patch into micropolygons (1/2 pixel)
3. Shade micropolygons
   - Shoot "shadow feelers" to light sources
   - Evaluate lighting model (e.g., Phong)
4. Perform stochastic sampling
   - Random sampling of screen pixels
   - Store samples (color & depth)
5. Compose using A-buffer
   - For anti-aliasing and transparency

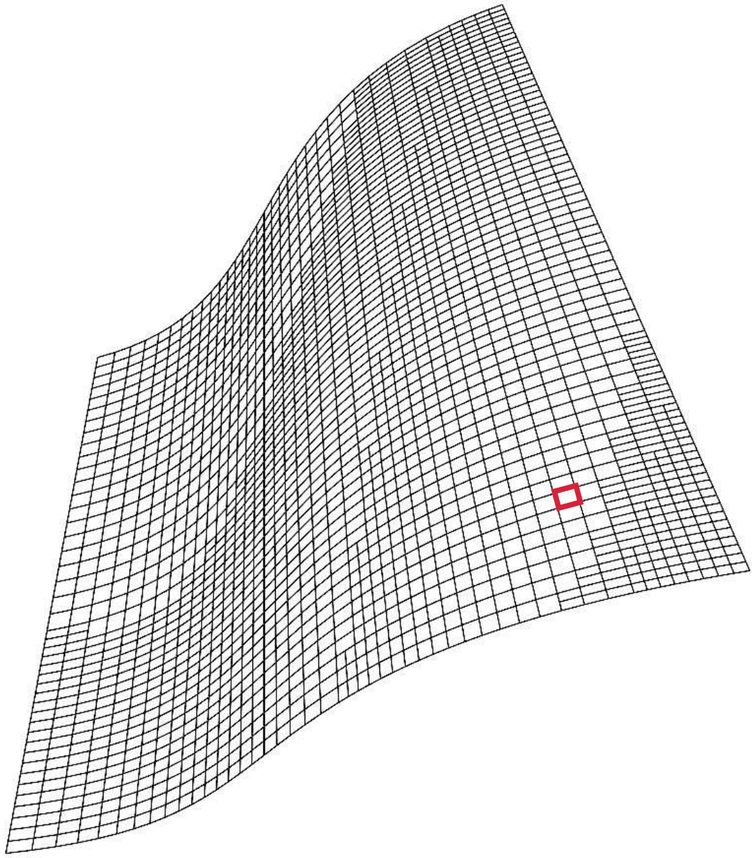


Rough Sketch of the Reyes Pipeline

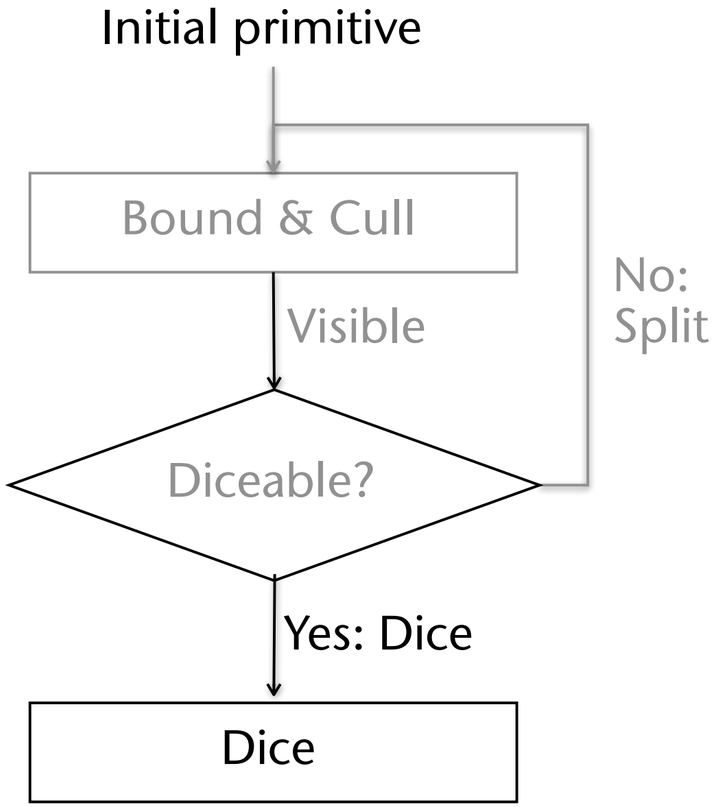
- Split & dice:

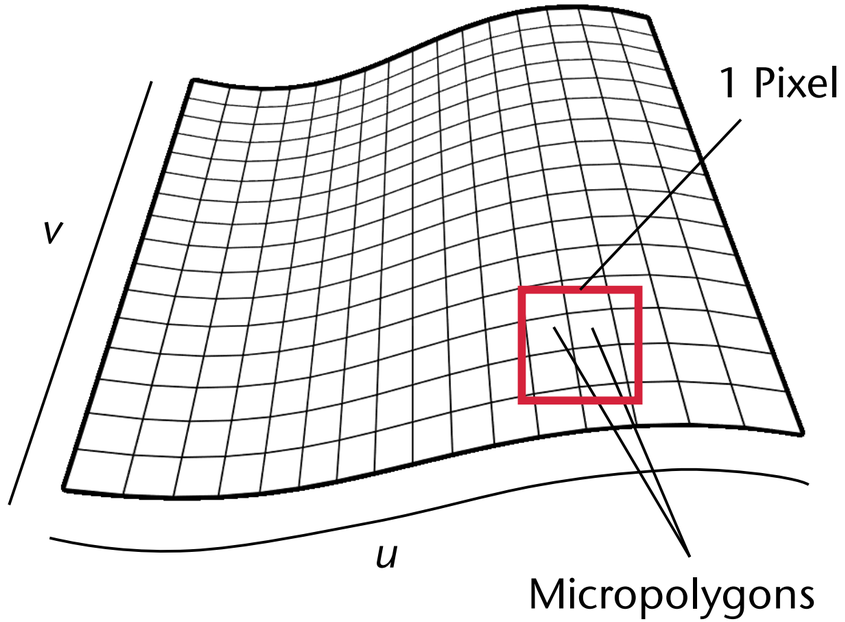






- Dice: tesselate one "small enough" patch along its *u*,*v* coordinates into micropolygons





Remarks on the Advantages of the Reyes Architecture

- Only one shader programming stage (shading of micropolygons)
  - Can do anything you want in the shader (e.g., cast rays, displacement of geometry,
- Supports lots of primitives
  - They just have to bring a procedure for subdivision along *u*,*v*
- Micropolygons can be shaded in parallel trivially
- Simplicity:
  - No attribute interpolation necessary (during rasterization)
  - No perspective correction necessary (like for texture coords)
- Resolution of the final geometry adapts to viewpoint distance automatically
- Allows for expensive effects, e.g., motion blur, depth-of-field, ...
- Render times: approx. 3 hours / frame (2-29h)

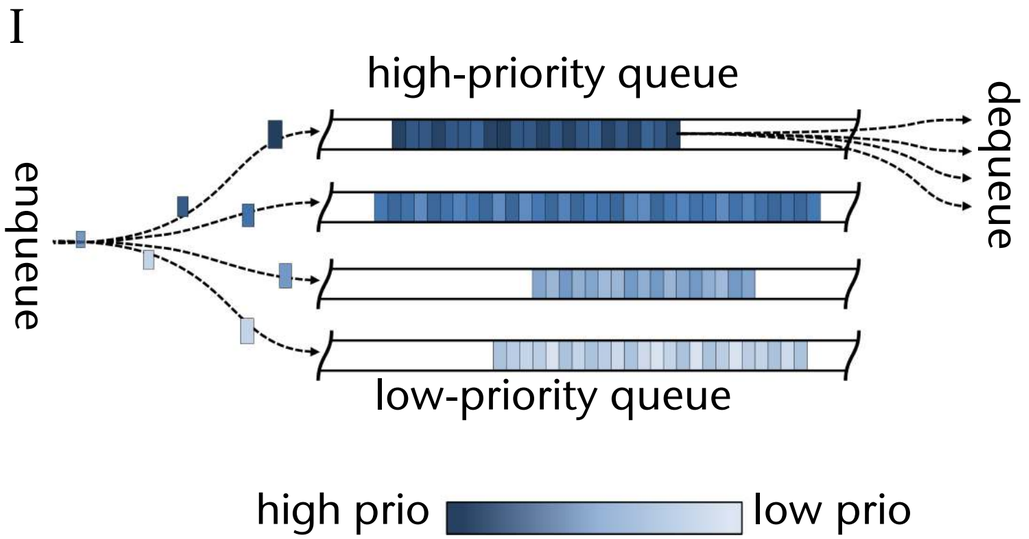
More Adaptive Rendering: Foveated Rendering

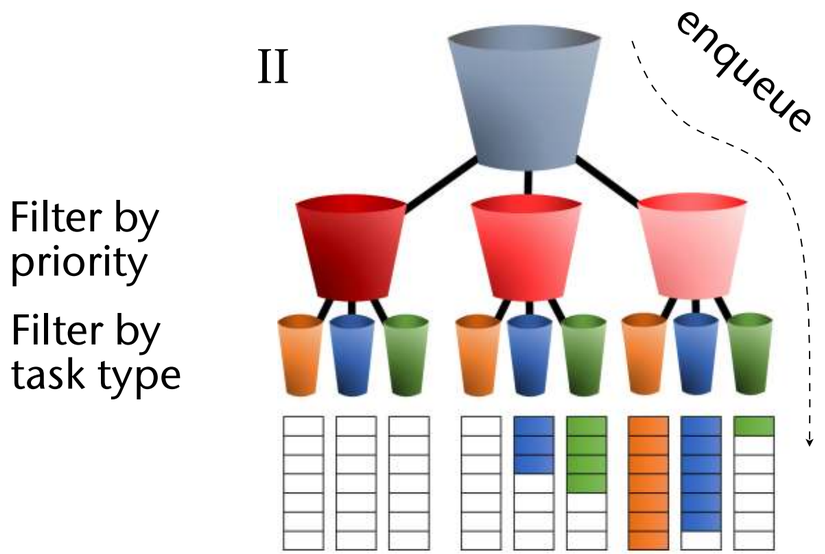
- Idea: spend more time on rendering for image region where user is looking

[Kerbl et al., 2016]

Fine-Grained Task Scheduling

- Requirements:
  1. Maintain multiple work queues for different task types
  2. Lock-free enqueuing and dequeuing of tasks (as frequently as possible)
  3. Reorganization of queues wrt. task priorities must not block worker threads
- Methods to organize queues by priority:



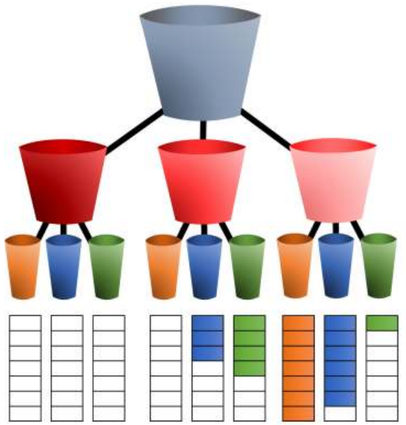


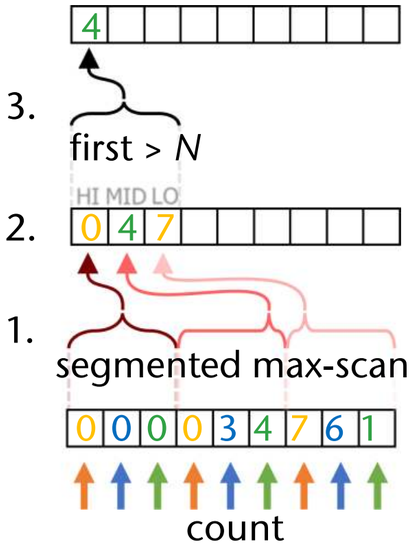
Dequeuing with Method II

- Assume hundreds of queues
- When a block of N threads is finished, it requests N new tasks
  - Use these threads to perform the search for the queue

1. Put queue occupancies in array
2. Perform segmented max-scan
3. Reduction: find first element > N
   - Define binary operator $\diamond$:

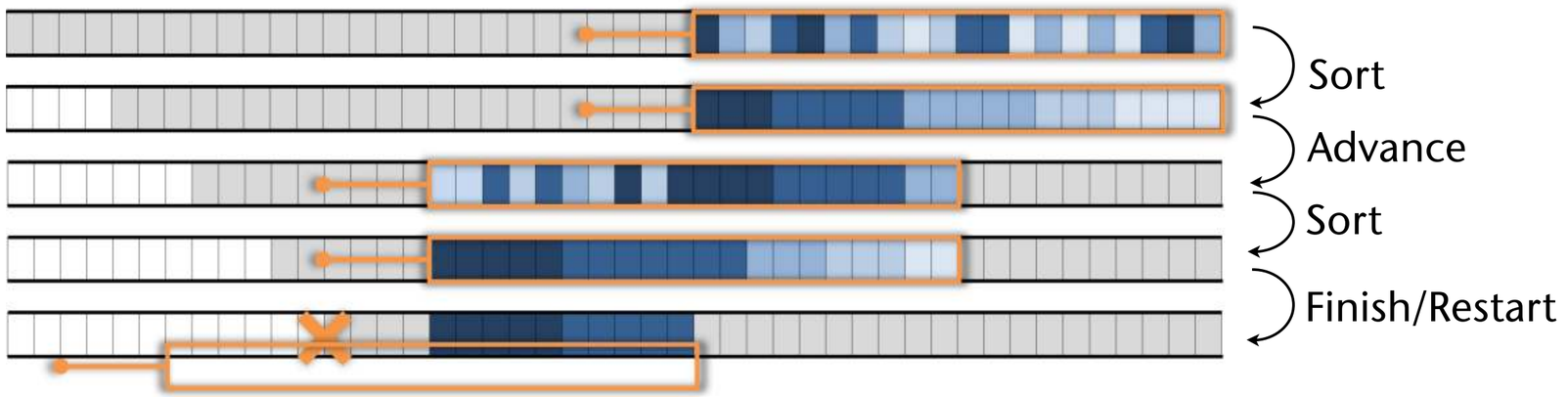
$$a \diamond b = \begin{cases} a & , a \geq N \\ b & , a < N \wedge b \geq N \\ 0 & , \text{else} \end{cases}$$





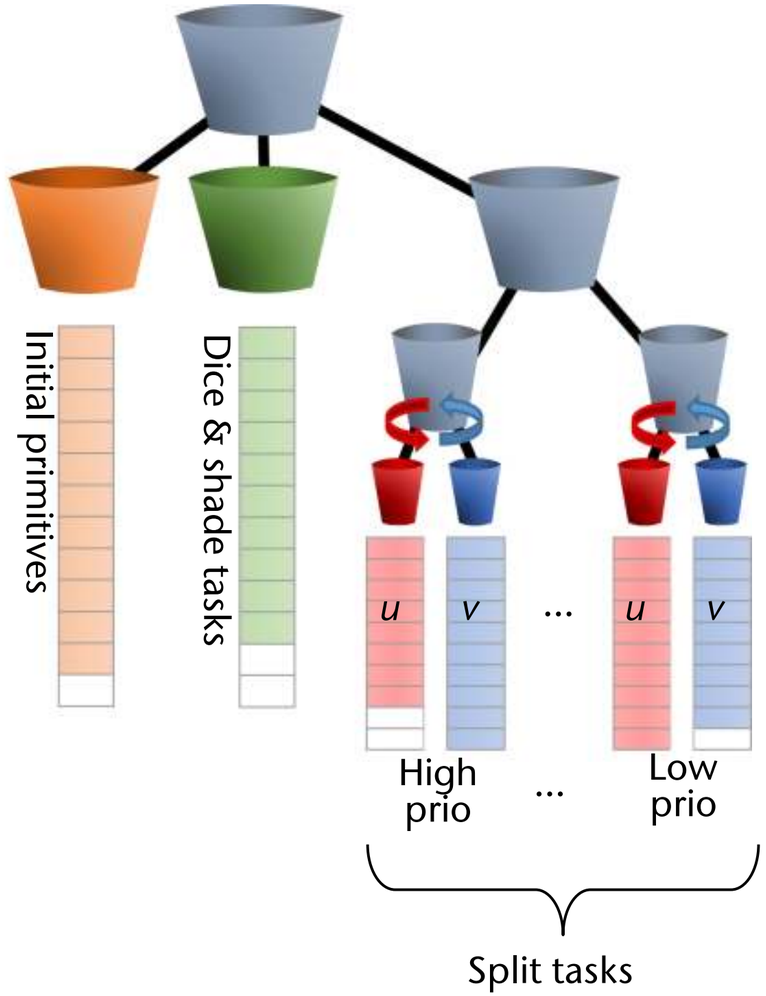
Updating Queues Containing Different Priorities

- Queues containing tasks with different priorities must be reordered, when new tasks are enqueued
- Locking the queue is not an option
- Observation: perfect sorting is not necessary (in most scenarios)
- Idea:
  - Continuously sort inside moving window using a (configurable) number of sorting threads
  - Keep safety margin to front of queue, to allow for task dequeueing



Application to Reyes-Style Rendering

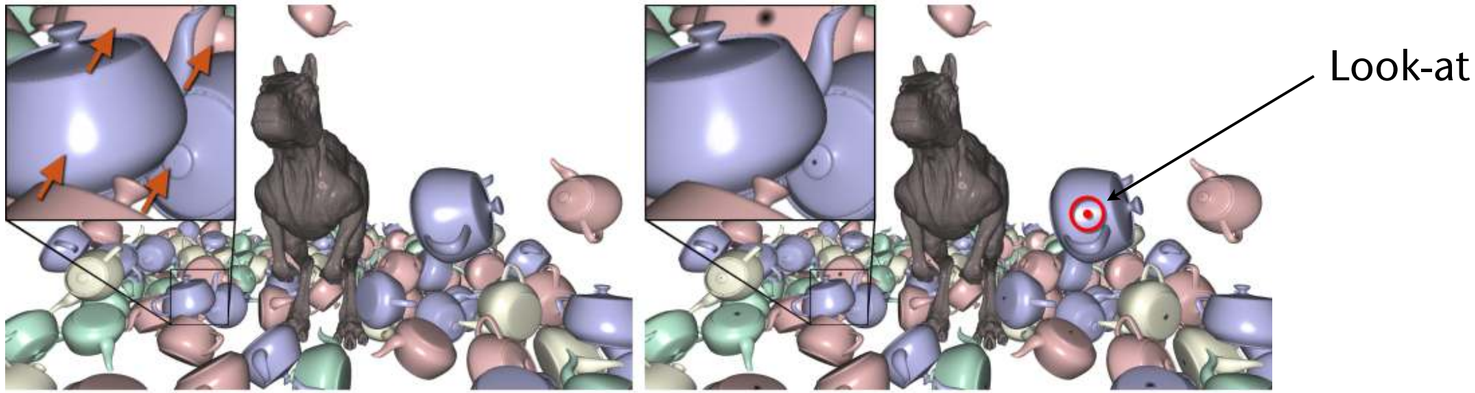
- Fill "initial" (orange) queue with primitives from host
- Priorities: 1. orange, 2. green, 3. split
  - Split queues contain tasks with different prio's
- Priority of patches = function of patch size on screen and distance from look-at point
- Enqueue split tasks into red & blue queues
- Time critical rendering: kernel works on tasks as long as time budget is not exhausted
  - Use cycle counters of the multiprocessors on the GPU



More Results

Full quality: 58 msec

Adaptive quality: 20 msec



Number of splits

Difference image, exaggerated 200x