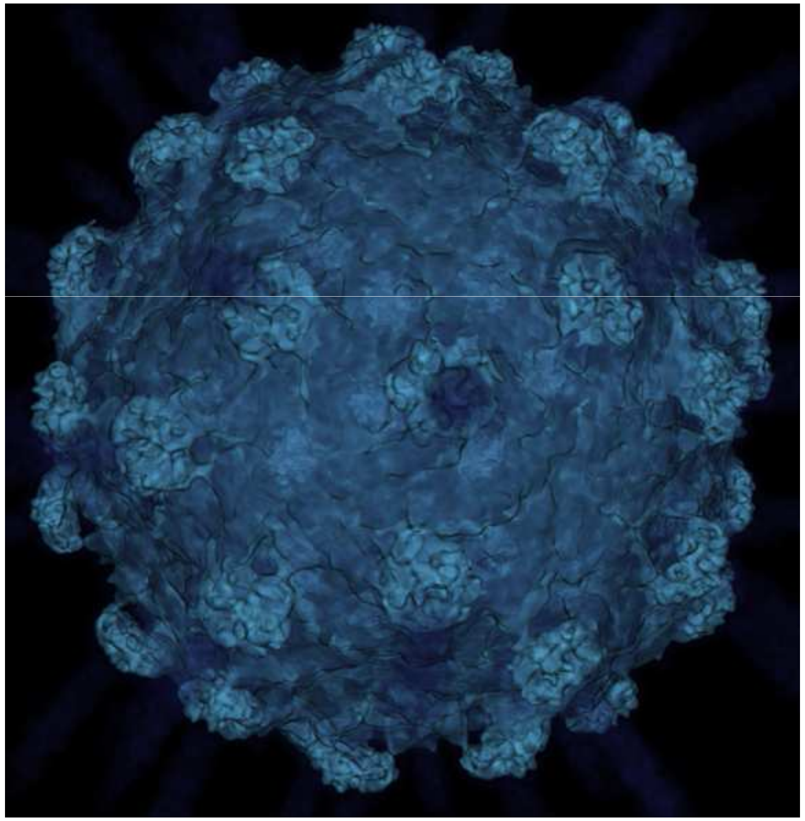
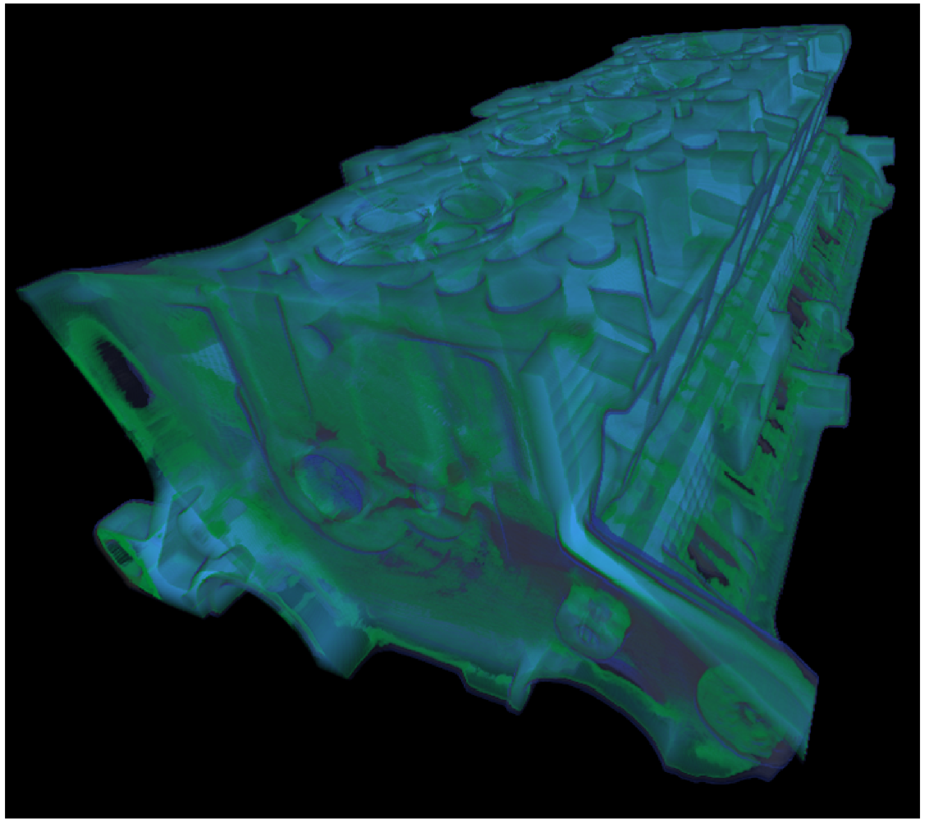
# Interactive Volume Rendering





Lukáš Maršálek, May 2009, CGG MFF UK, CGUdS, Lukas.Marsalek@mff.cuni.cz

## Contents

- Why speed?
  - Why to try?
- Lossless acceleration techniques for CPUs
  - Accelerating ray casting and slicing
- Mapping to GPUs
  - How to map the algorithm to GPUs
- Summary

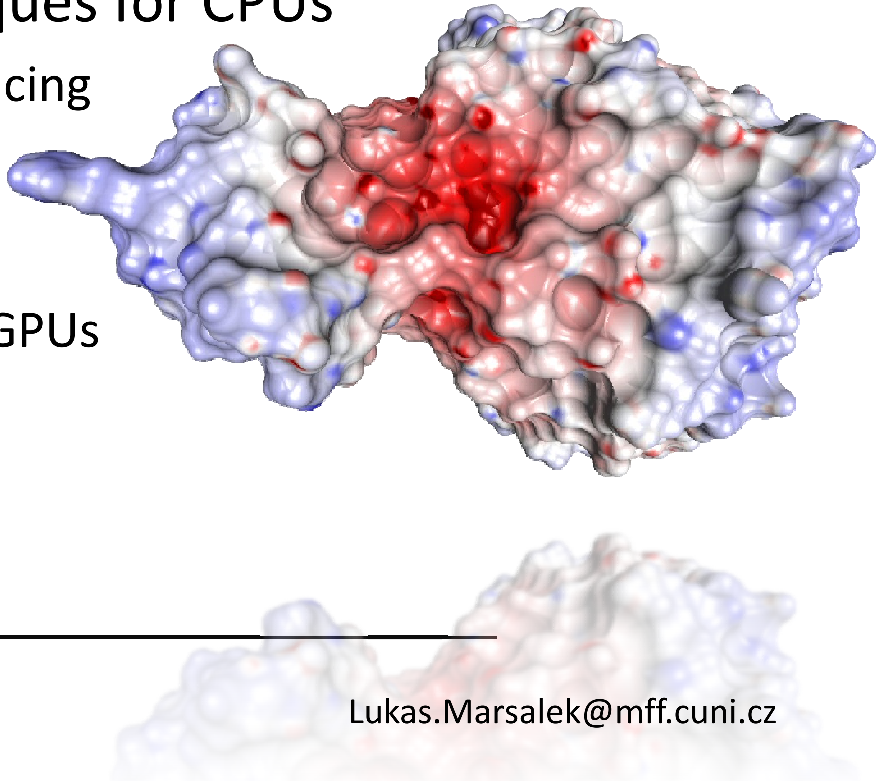


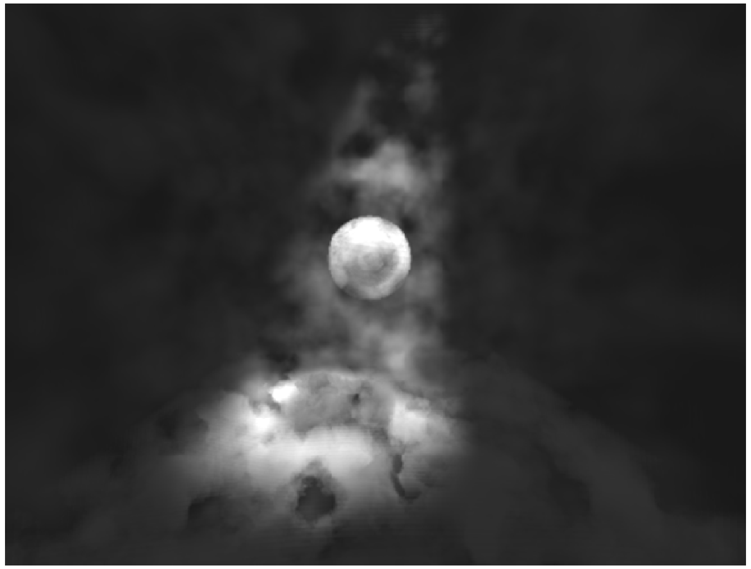
Image courtesy of [BallView 09]

## Importance of speed

- So far …
  - Standard technique overview
  - Non-interactive
- Interactivity
  - Enhances spatial perception
  - Faster inspection in day-to-day work
  - Essential for volume rendering
- Quality demands
  - High-resolution for image & data

## Limiting the scope

- Most common approaches
  - Ray casting
  - Volume slicing

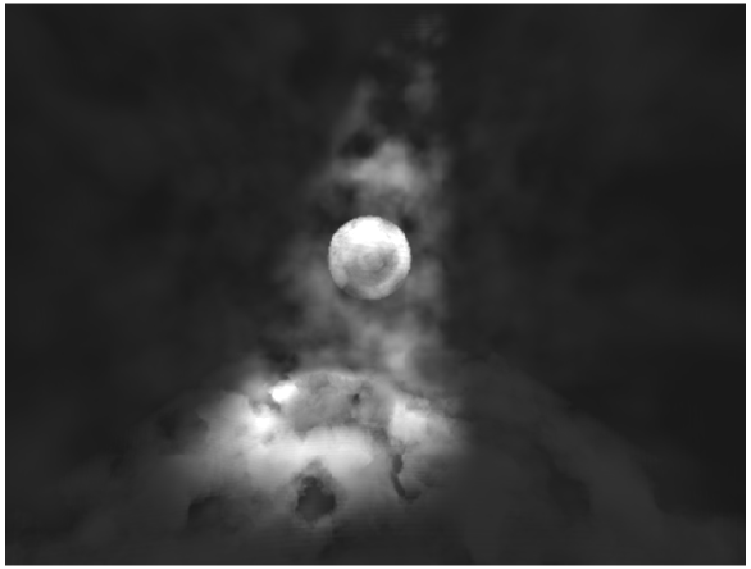


- High-speed
- High-quality
- Flexibility
- Ease of implementation

- Slicing is restricted GPU-friendly ray casting

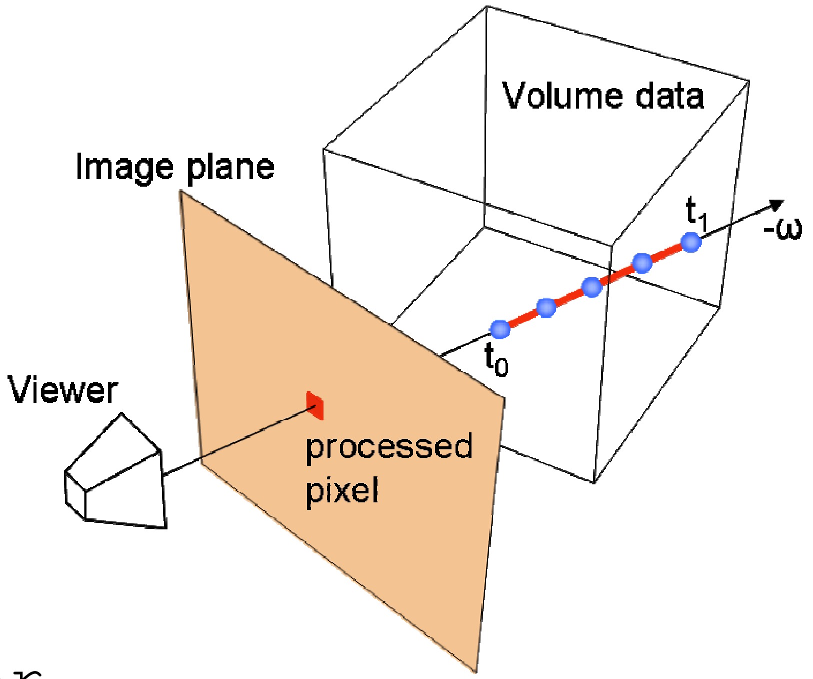
## How to speed-up?

- Algorithmic improvements
  - What are the opportunities?
- More, more, more hardware
  - Cluster-based rendering
- Better hardware
  - Free speed-up due to Moore's law
  - Specialized hardware



# Ray casting pipeline

```
generate rays for pixels

while(inVolume){

    read data from volume

    map data to color

    accumulate color
}
write color to framebuffer
```

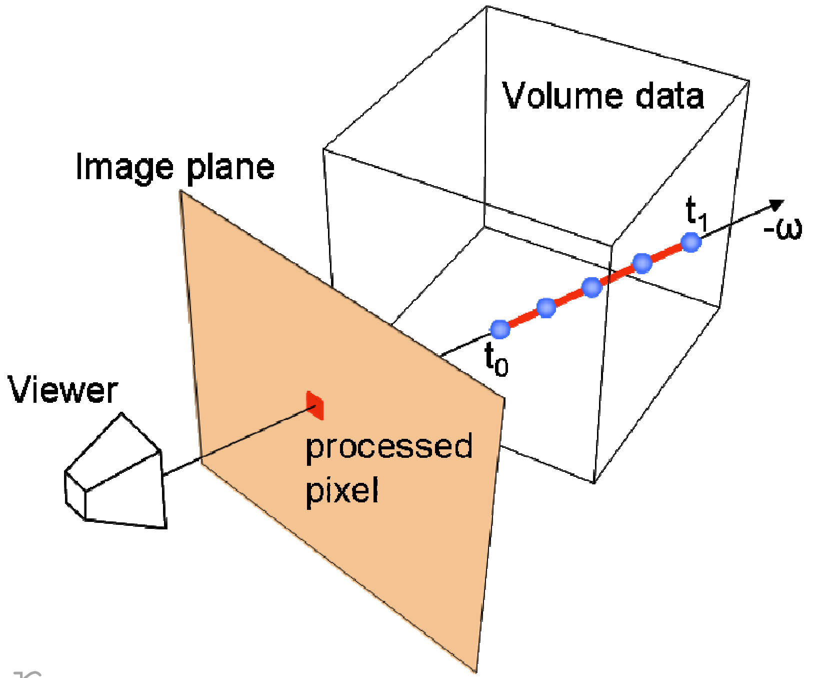


# Optimization opportunities

```
generate rays for pixels
```

Generate less rays

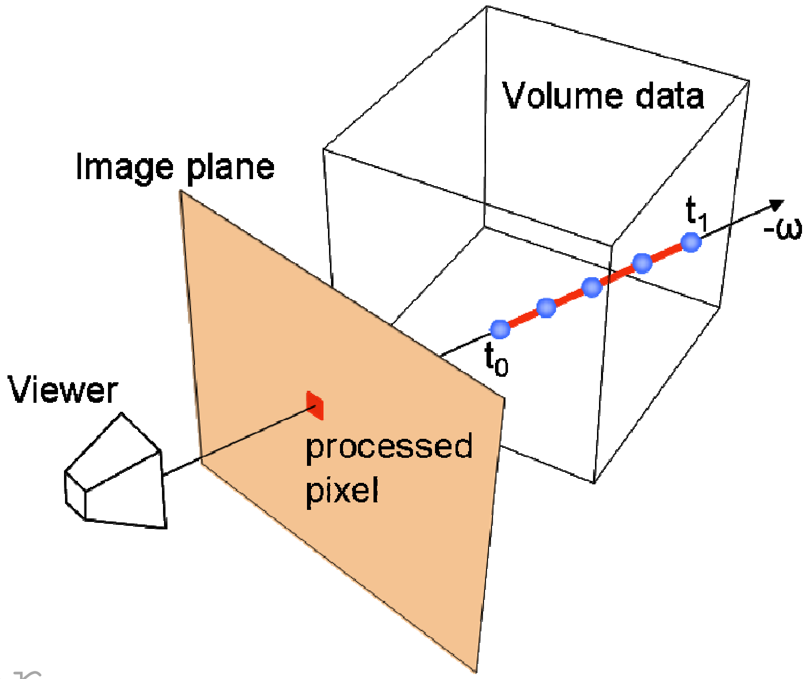
```
while(inVolume){
    read data from volume
    map data to color
    accumulate color
}
write color to framebuffer
```



Lukas.Marsalek@mff.cuni.cz

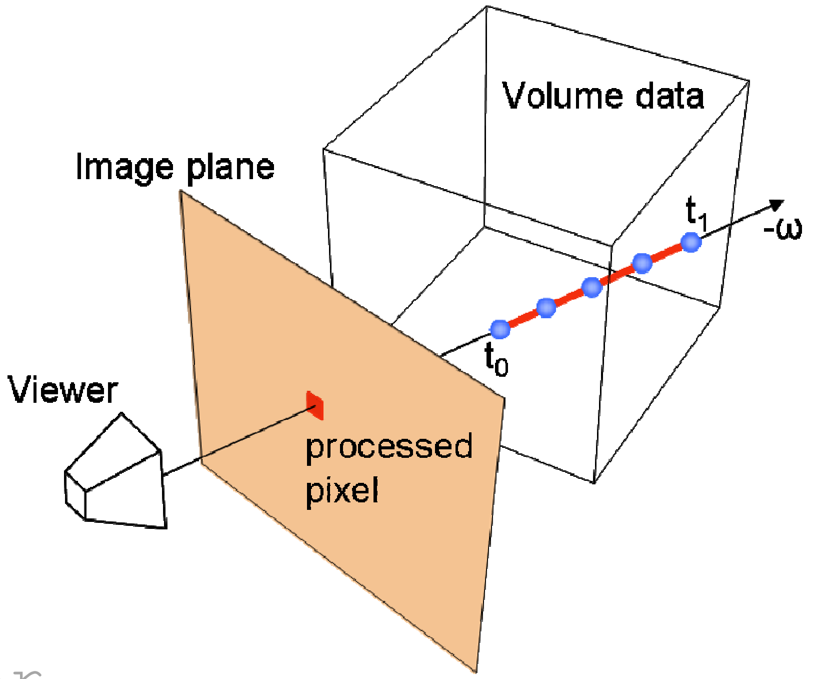
## Optimization opportunities

```
generate rays for pixels

while(inVolume){
    Use fewer samples
    read data from volume

    map data to color

    accumulate color
}
write color to framebuffer
```



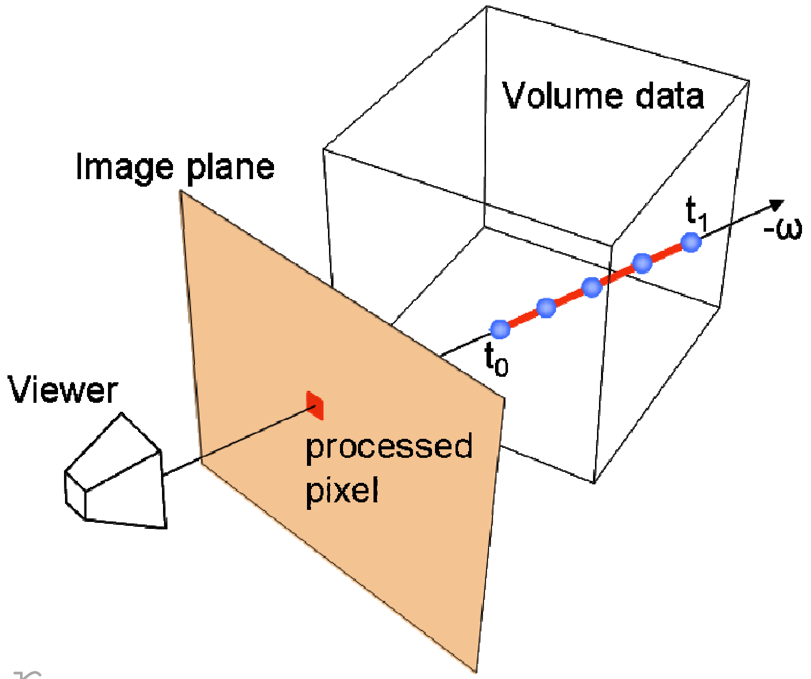
# Optimization opportunities

```
generate rays for pixels

while(inVolume){

    read data from volume
    Read data faster
    map data to color

    accumulate color
}
write color to framebuffer
```



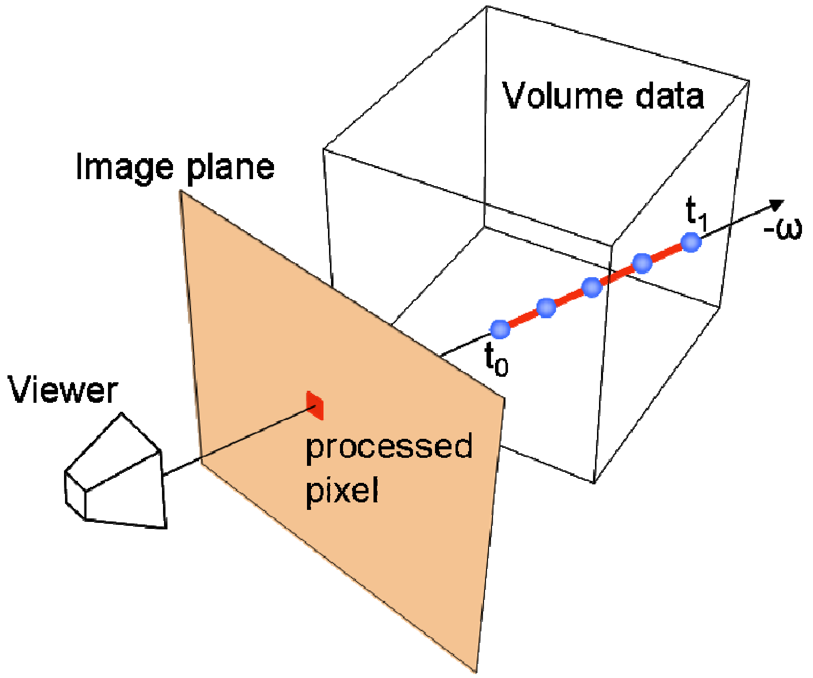
## Optimization opportunities

```
generate rays for pixels

while(inVolume){

    read data from volume

    map data to color
    Map less often
    accumulate color
}
write color to framebuffer
```



## Optimization opportunities

```
generate rays for pixels

while(inVolume){

    read data from volume

    map data to color

    accumulate color
```
Accumulate only if needed
```
}
write color to framebuffer
```

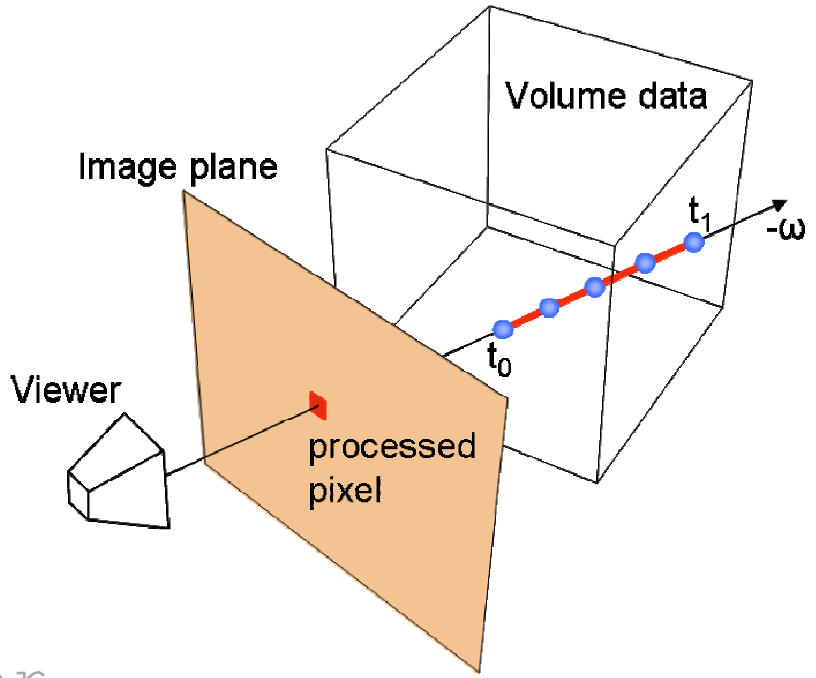


# Optimization opportunities

```
generate rays for pixels
```

Generate less rays

```
while(inVolume){
    read data from volume
    map data to color
    accumulate color
}
write color to framebuffer
```



## Adaptive image sampling

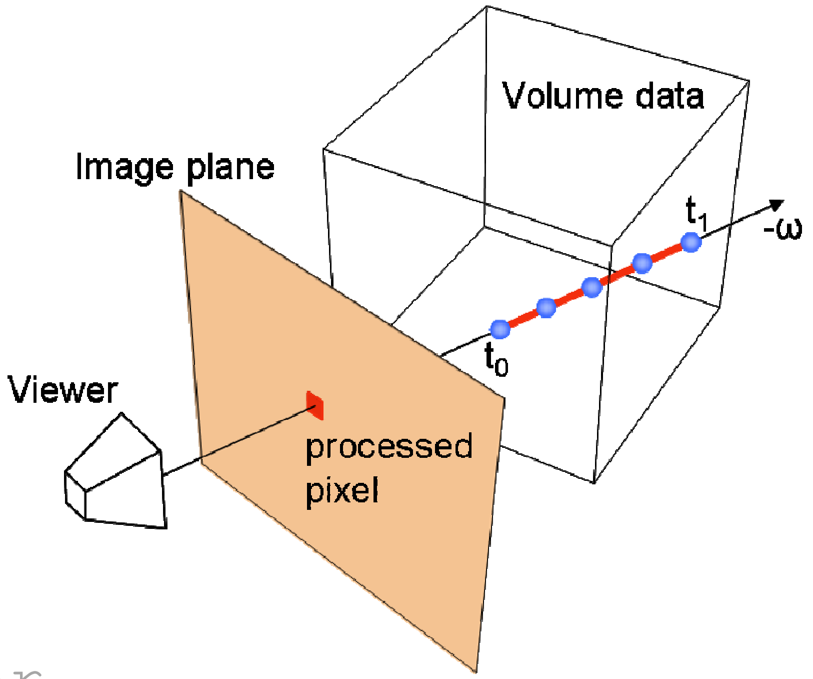
- Generate less rays
- Do we need ray per pixel?
- NO!
- Shoot rays only in interesting regions
  - How to identify them?

## Tile-based sampling by Ljung

- Split the image to fixed-size tiles
  - Balance overhead vs. gains
- Shoot probe rays to evaluate tile importance
  - Pre-classified volume blocks importance
- Assign tile sampling rate
  - Higher importance => more samples in tile
- Not restricted to pixel-aligned samples
  - Arbitrary sampling within the tile

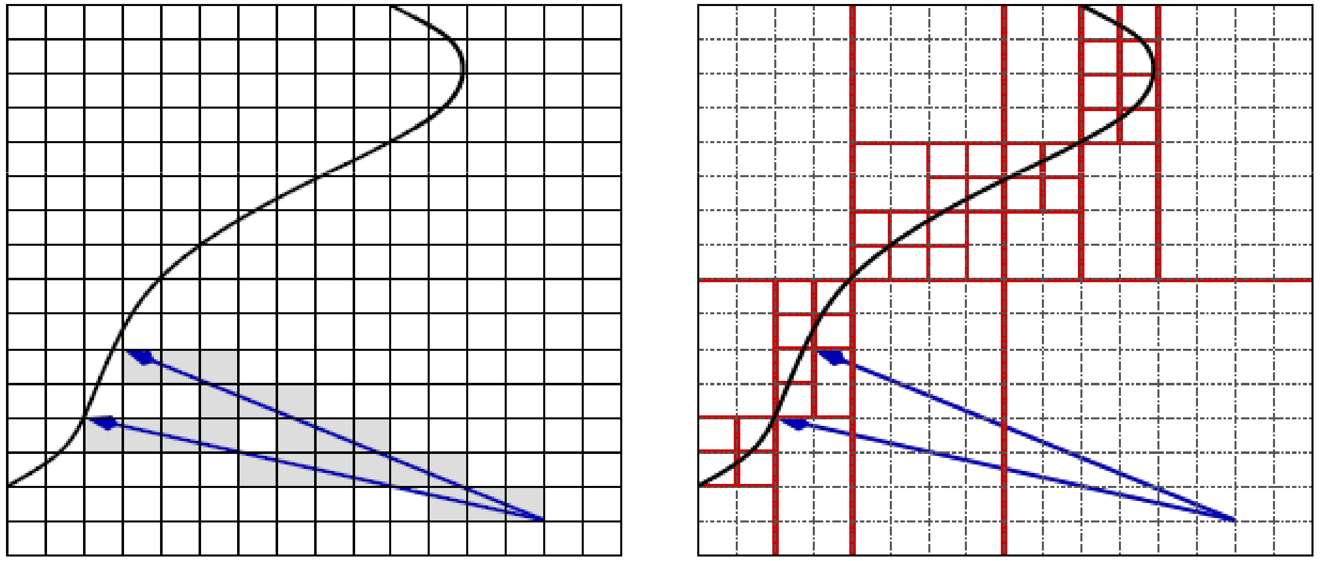
## Optimization opportunities

```
generate rays for pixels

while(inVolume){
    Use fewer samples
    read data from volume

    map data to color

    accumulate color
}
write color to framebuffer
```



# Empty space leaping

- Only about 5-10% of voxels contribute to image
- Why to traverse all?



Images by [Marmitt 06]

- Depends on Transfer Function!!!

## Block-based skipping

- Encode empty space in additional structure

- Octrees & kD-trees
  - Implicit tree with min/max values in each node
  - Summed-area table for transfer function
  - Both semi-transparent and isosurfaces
  - Size may be problem

- Flat blocks
  - Fixed-size block with min/max values
  - Limited effectivity

## PARC & Warping

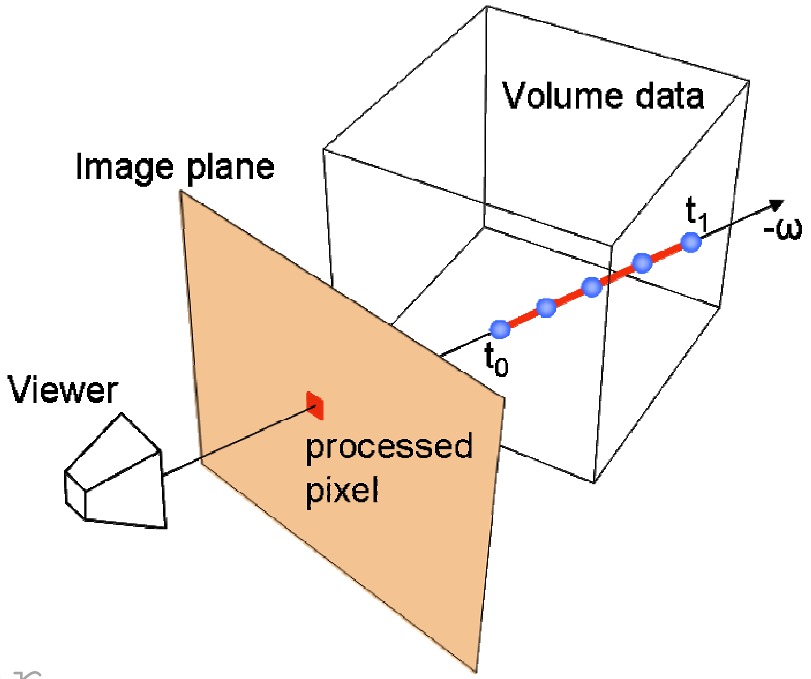
- Helps to determine ray entry point
- Do not skip all empty space

- Polygon-assisted ray casting (PARC)
  - Render polygons with outlying value
  - Start rays from there

- Warping
  - Encode position of first non-empty voxel
  - Warp and reuse next frame

## Optimization opportunities

```
generate rays for pixels

while(inVolume){

    read data from volume
    Read data faster
    map data to color

    accumulate color
}
write color to framebuffer
```

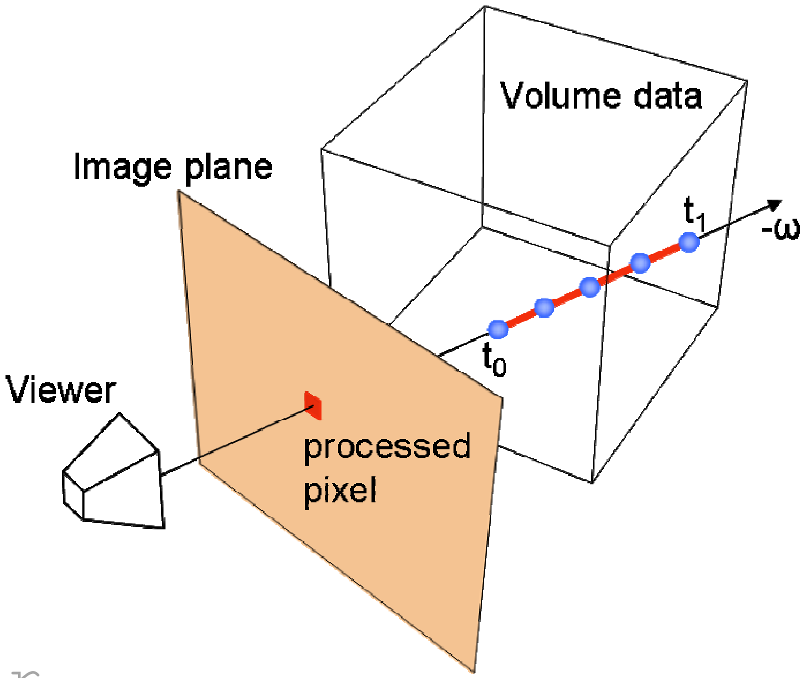


## Volume swizzling

- Heavy memory bandwidth requirements
- Ray casting generates scattered memory access
  - Cache unfriendly
- Maps neighboring pixels to be close in memory
- Increases cache efficiency
  - Noticeable speed-up (2-3x)

## Optimization opportunities

```
generate rays for pixels

while(inVolume){

    read data from volume

    map data to color
    Map less often
    accumulate color
}
write color to framebuffer
```



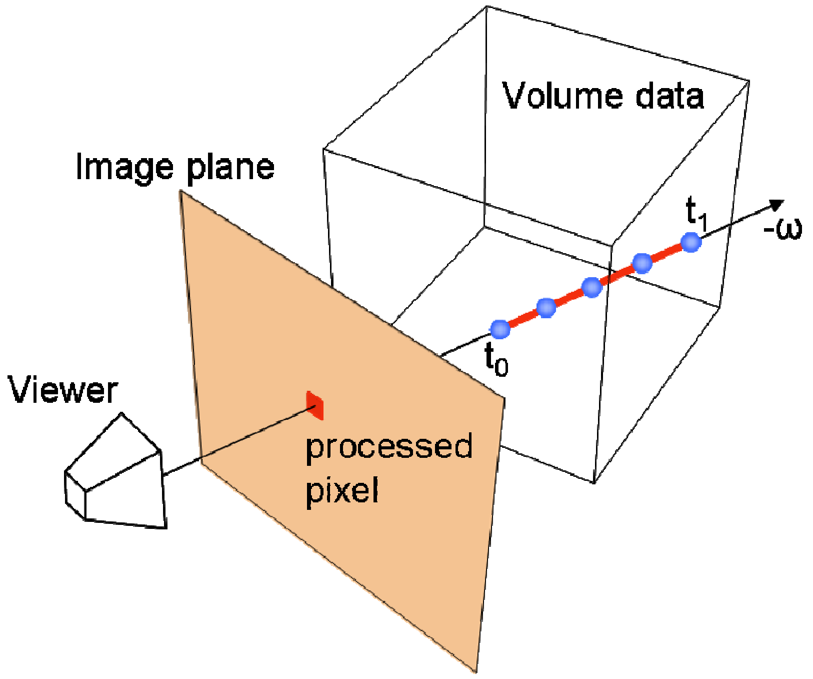
Lukas.Marsalek@mff.cuni.cz

## Deferred shading

- For expensive shading
  - Complex local lighting models
  - Shadows, Fresnel approximation, ambient occlusion
- Shade only if required
- Skip highly transparent samples
  - Contribution is insignificant
- Skip low gradient magnitude samples
  - Improves more quality than speed

# Optimization opportunities

```
generate rays for pixels

while(inVolume){

    read data from volume

    map data to color

    accumulate color
    Accumulate only if needed
}
write color to framebuffer
```



## Early Ray Termination

- Most samples are hidden behind opaque parts
- Stop the computation if the ray is saturated
- Empirical threshold
  - Usually 0.95
- Trivial implementation

## How to speed-up?

- Algorithmic improvements
  - What are the opportunities?
- More, more, more hardware
  - Cluster-based rendering
- Better hardware
  - Free speed-up due to Moore’s law
  - Specialized hardware

## How to speed-up?

- Algorithmic improvements
  - What are the opportunities?
- More, more, more hardware
  - Cluster-based rendering
- Better hardware
  - Free speed-up due to Moore's law
  - Specialized hardware

# Better hardware

- So far only about CPUs
  - What about something faster?
  - What means “faster”?
- Parallelism, Parallelism, Parallelism, Parallelism, Parallelism
  - Our pixels are completely independent !!!
- Special purpose hardware
  - Trivial tasks can be hardwired
- Meet … Graphics Processing Units (GPUs)

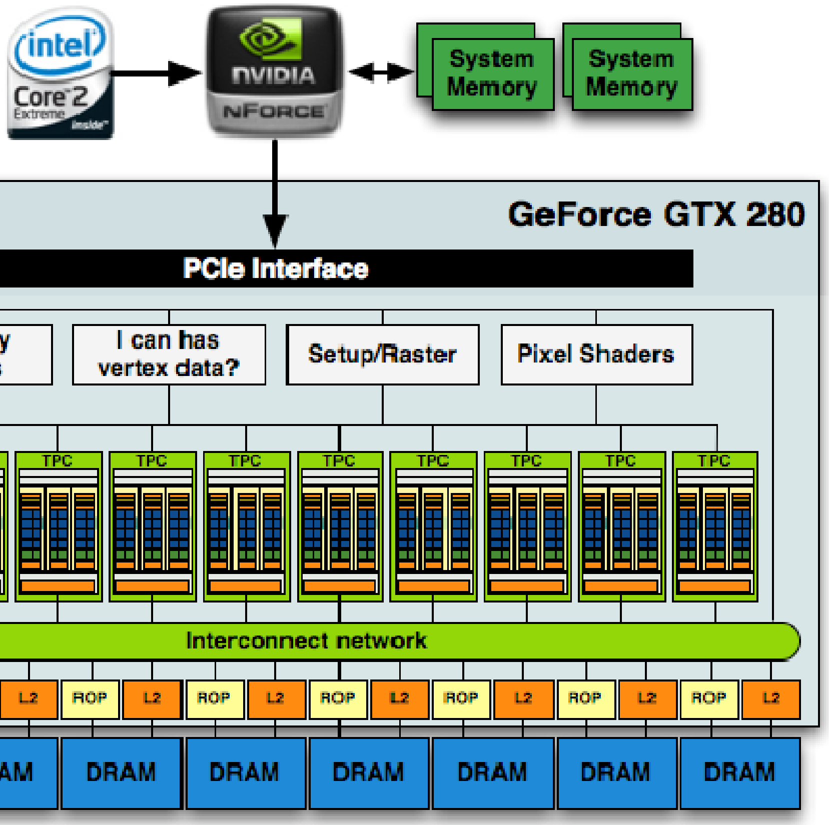
## Analyzing parallelism

- Data-level parallelism
  - SIMD architectures
- Thread-level parallelism
  - Multithreading, coarse or fine-grained
- Task-level parallelism
  - Traditional approach

- Moore’s law re-defined
  - Not with frequency, but with parallel cores

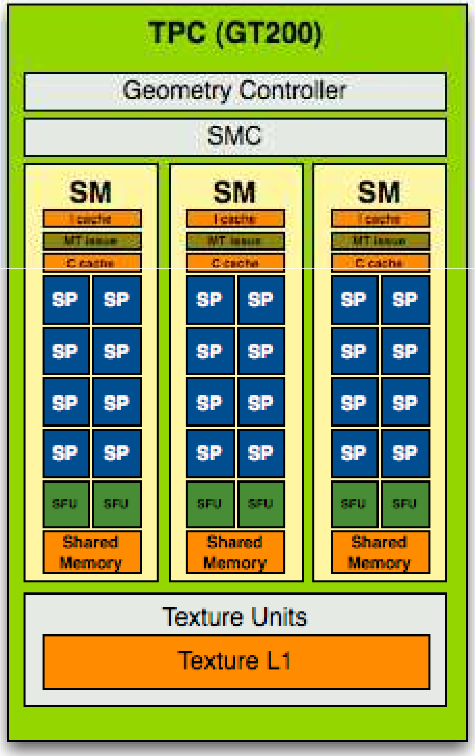
# Tesla architecture

- Massively parallel
  - 30 SIMD SMs
  - 240 scalar cores



## CUDA programming model

- New programming paradigm
  - SIMT – Single Instruction Multiple Threads
- Implicit SIMD management
  - Thread divergence
  - Memory coherence
- User-controlled caches
  - Hardware-managed texture cache
  - User-managed shared memory



# Interpolation and blending

- First hardwired subsystem
  - First Graphics Accelerators
  - Bilinear
- Fixed, trivial algorithm
  - “Only MADD instructions”
- Performed EXTREMELY often
  - ~ billion times per second
- Accelerated blending

## Rasterization & Memory

- Suitable for hardware implementation
  - Well-known algorithm
  - Hierarchical approach, simple operations
  - Very parallel on triangle level

- Hundreds of millions of triangles per second

- Wide, high-bandwidth memory bus
  - 512 bit bus
  - Theoretically dozens of gigabytes per second

## Texture-based slicing

- Designed to take advantage of new HW
  - Bi-linear or tri-linear interpolation  blending
  - High rasterization throughput
  - High bandwidth memory

- Until recently the fastest and most used algorithm

- Problems
  - Rasterization is not fast enough for large volumes
  - Artifacts
  - Inflexible

## GPU-based Ray Casting

- Most flexible algorithm
  - Can incorporate speed-up techniques easily
- Balances the requirements on the GPU
  - Can use texture interpolation hardware
  - Can use rasterizers, but does not depend on them
- Shader based vs. CUDA-based implementations
  - On par ☺

## Shader implementation

- Rasterize bounding box of the volume
  - Ray generation phase
- Volume rendering loops runs in fragment shader
  - For SM >= 3.0
  - Need for dynamic branching
- Transfer function implemented as texture

# High-speed volume ray casting with CUDA

Lukáš Maršálek
Saarland University

Armin Hauber
Saarland University

Philipp Slusallek
DFKI Saarbrücken & Saarland University

## Introduction

Volume ray casting receives a renewed interest as graphics hardware enables its realtime implementations.
Currently [2, 1, 3], special shader languages and graphics APIs are used, making implementation uncomfortable and difficult. It also hinders performance, as it bends the programming model for something it was not designed to.
Recently, new generation of GPUs has been introduced together with CUDA, a C-language API. CUDA exposes the hardware not as a streaming graphics pipeline but as a general highly parallel co-processor.
We aim at evaluating this new increased flexibility versus any performance losses. For the study we have chosen ray casting with front-to-back traversal, pre-integration, and early ray termination.

## Conflict-free SM usage

Storing float3 variables in block-sized array enables completely conflict-free shared memory access. This enables maximum parallelism in thread execution, servicing 16 requests to the shared memory simultaneously.

## Performance results

| FPS performance | Neghip ($64^3$) | Foot ($256^3$) | VisFemSlice ($512^3$) |
|---|---|---|---|
| CRM | 46.7 | 15.2 | 11.6 |
| ST | 41 | 13.5 | N/A |

FPS in 1024x1024 for our (CRM) and Steigmaier [3] (ST) ray caster

## Textures can also be utilized in CUDA

We also use efficient texturing capabilities known from graphics APIs, taking advantage of the automatic interpolation and read-only caching for effective access to volume dataset and pre-integration lookup table.

## Coalesced global memory access

To make the access to the global memory effective, it is imperative to enable the hardware to group individual requests. This is done through coalescing the reads and writes so that they access linear memory in a well-defined stride.

## Implementation

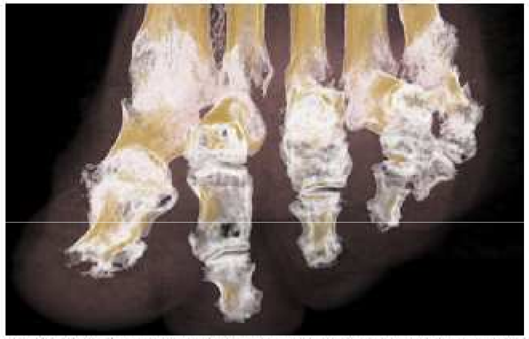
```
__global__ void kernel (unsigned int * pixels) {

__shared__ float3 s_rayDir[320];
__shared__ float3 s_dst[320];


int sIndex = tIdx.y * blockDim.x + tIdx.x;
s_rayDir[sIndex ] = ComputeRayDirection();

IntersectBoundingBox();
if( (tOut - tIn) > 0.0f )
{
   f = c_rayOrigin + tIn * s_rayDir[sIndex];
   old = tex3D(datasetTex, f);

   while(tIn < tOut)
   {
      tIn += c_rayStepSize;
      f = c_rayOrigin + tIn * s_rayDir[sIndex];

      next = tex3D( datasetTex, f );
      f = tex2D(preintTexture2D, old, next);
      old = next;

      AccumulateColor(s_dst[sIndex], f);
      if(RaySaturated()) { break; };
   }
}

pixels[i] = rgbaFloatToInt(s_dst[sIndex]); }
```

## Scalar C++ like code on GPU

Integrated in a C++ framework and itself written in C/C++ mixture with familiar constructs entirely on GPU

## High GPU occupancy

Only 12 live registers for the whole kernel result in 83% GPU occupancy. It is achieved by spilling long-lived variables into the shared memory and recomputing common local expressions.


The *Foot* dataset renderered at 15.2 FPS into 1024x1024 window

## References

[1] J. Krüger and R. Westermann: Acceleration Techniques for GPU-based Volume Rendering. IEEE Visualization 2003.
[2] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser. Smart hardware-accelerated volume rendering, EuroVis 2003.
[3] S. Stegmaier, M. Strenger, T. Klein and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. Volume Graphics, 2005.

## Conclusion

Our optimized CUDA ray caster presents a proof of concept that the flexibility of programming GPUs in C dialect does not come at a performance hit. Rather it enables low-level optimizations, outperforming optimized shader implementations. Our future work will concentrate on advanced acceleration techniques and applications to new branches like anthropology and bioinformatics.

## Acknowledgements

This work was supported by International Max Planck Research School for Computer Science in Saarbrücken, Computer Science Dept. of Saarland University and GAUK grant no. 88008.

# Adaptive Image Sampling

- Directly implementable
- How to render only corner pixels?
- Render small polygons
  - One polygon per tile
- Restrict viewport
  - Using glViewport command

# Empty space leaping

- Flat schemes easy to implement
- Double loop
  - First loop over empty-space structure
- Adaptive Texture Maps
  - Indirect texture lookup into index texture
- Hierarchical are difficult
- Octree-based CUDA traversal

## Deferred shading & ERT

- Both effective through early z-culling
  - Allows to discard fragment before it is processed
- Block-based ERT
  - Double pass algorithm
  - Render bounding boxes of blocks
  - Compare alpha value (ERT test), set z-buffer

## Summary

- What we have covered
  - Basic acceleration techniques for ray casting and slicing
    - Adaptive image-space sampling
    - Memory layout (swizzling)
    - Empty-space leaping
    - Deferred shading
    - Early ray termination
  - Conservative lossless techniques
  - Mapping to GPUs

- What we have NOT covered ?

## Summary

- What we have covered
  - Basic acceleration techniques for ray casting and slicing
    - Adaptive image-space sampling
    - Memory layout (swizzling)
    - Empty-space leaping
    - Deferred shading
    - Early ray termination
  - Conservative lossless techniques
  - Mapping to GPUs

- What we have NOT covered ?
  - Adaptive object-space sampling & multi-resolution rendering

## References

- [BallView 09*] BALLView, molecular modeling and visualization software*, www.ball-project.org, webpage, 2009

- [Marmitt 06] G. Marmitt, R. Brauchle, H. Friedrich, and P. Slusallek : *Accelerated and Extended Building of Implicit kd-Trees for Volume Ray Tracing,* VMV 2006,

## Finish

Thank you for your attention