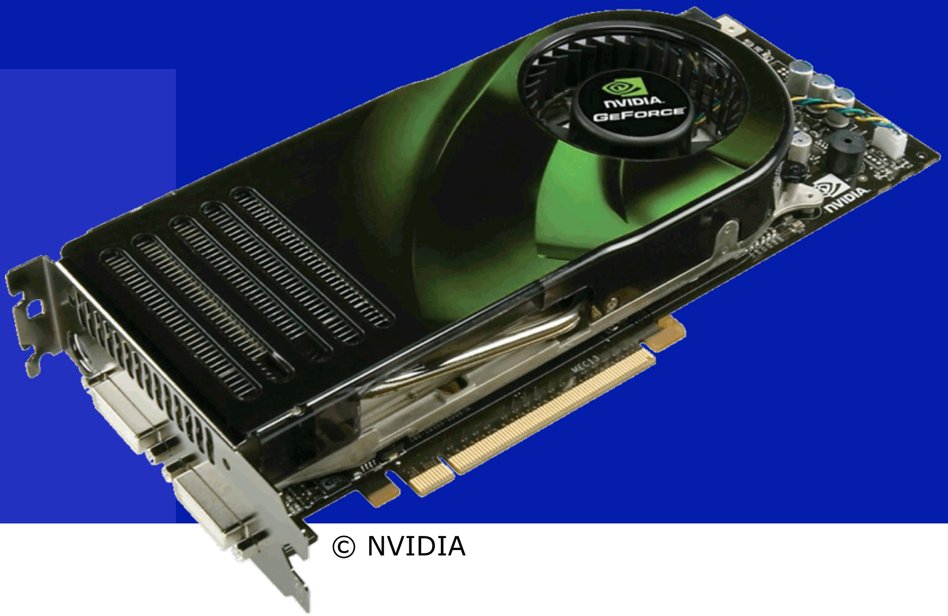
Comparison of High-Speed Ray Casting on GPU using CUDA and OpenGL

November 8, 2008



© NVIDIA

Benjamin Keck[1,2], Andreas Weinlich[1], Holger Scherl[2], Markus Kowarschik[2] and Joachim Hornegger[1]

[1] Chair of Pattern Recognition (Computer Science 5)
Friedrich-Alexander-University Erlangen-Nuremberg

[2] Siemens Healthcare, CV,
Medical Electronics & Imaging Solutions, Erlangen

Outline

- **Motivation: Iterative reconstruction**
- **Methods: Forward projection - ray casting**
- **Implementation**
  - Open Graphics Language (OpenGL)
  - Common Unified Device Architecture (CUDA)
- **Evaluation & Results**
- **Discussion & Conclusion**

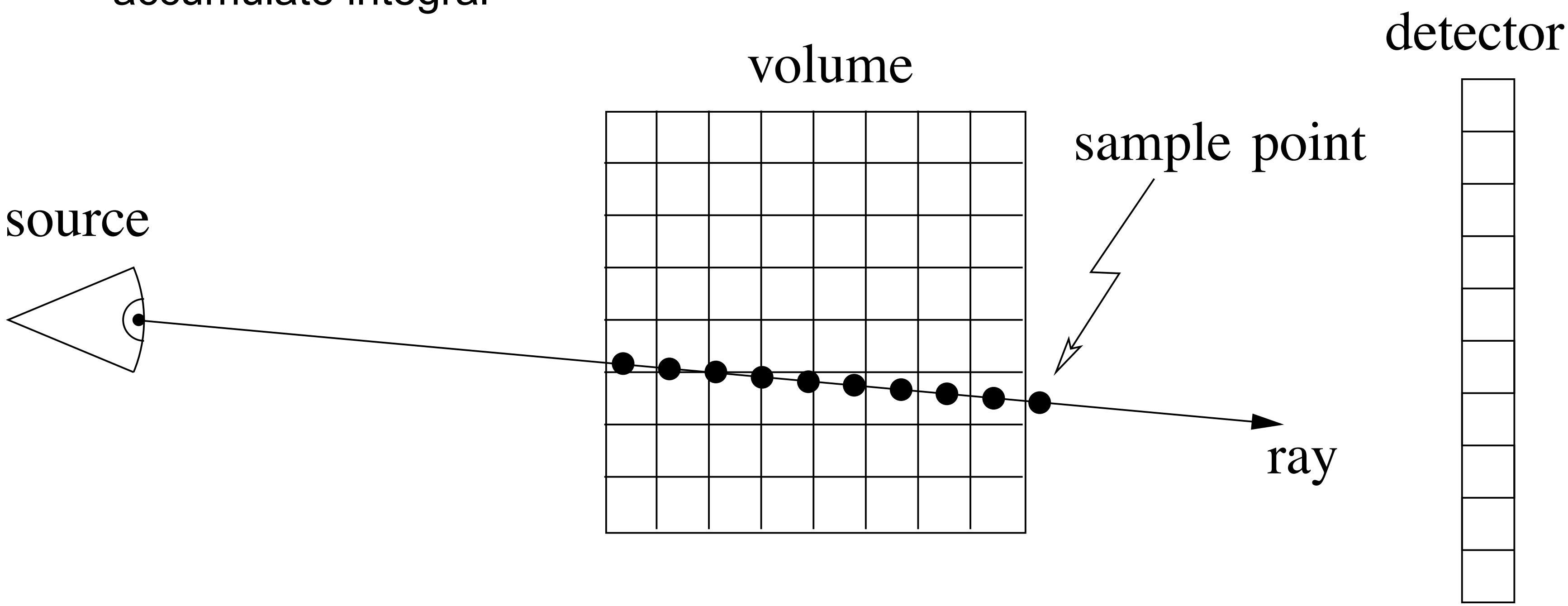
Arnd Dörfler, Neuroradiology, University-Clinic Erlangen

Motivation

- **Iterative 3D volume reconstruction is a computationally demanding and memory intensive application in medical image processing** (forward- / back-projection)
- **Forward-projection:** volumetric ray caster can be used for superior precision
- **Ray casting is easily parallelizable and therefore dedicated for highly parallelized low-cost processing architectures** (like current GPUs)
- **Two recent GPU-programming tools:**
  - Open Graphics Language (OpenGL)
  - NVIDIAs Common Unified Device Architecture (CUDA)

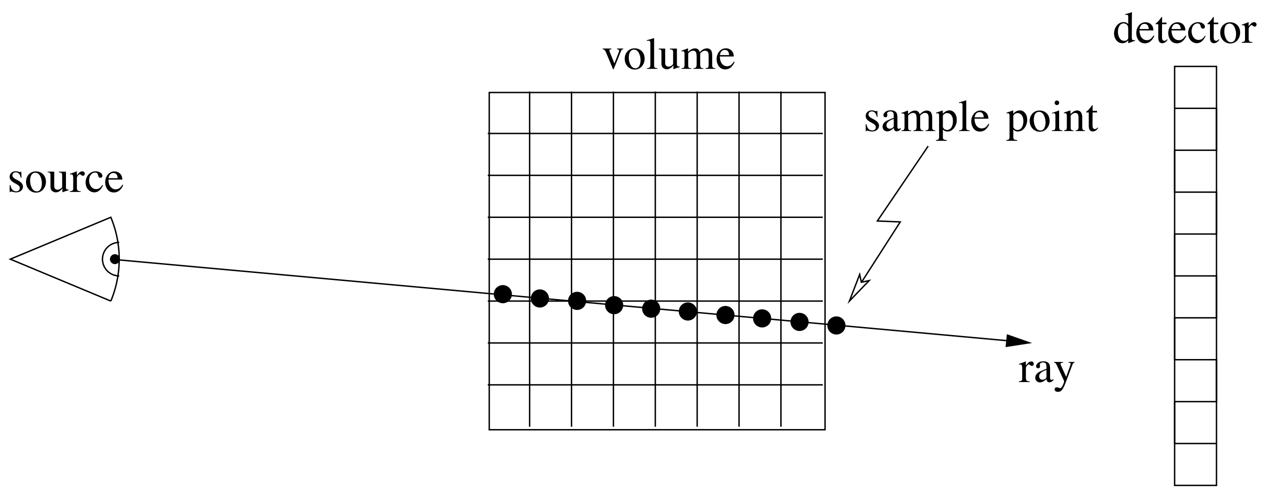
Methods: Ray Casting Principle

- **For each ray**
  - compute coordinate
  - interpolate value
  - accumulate integral



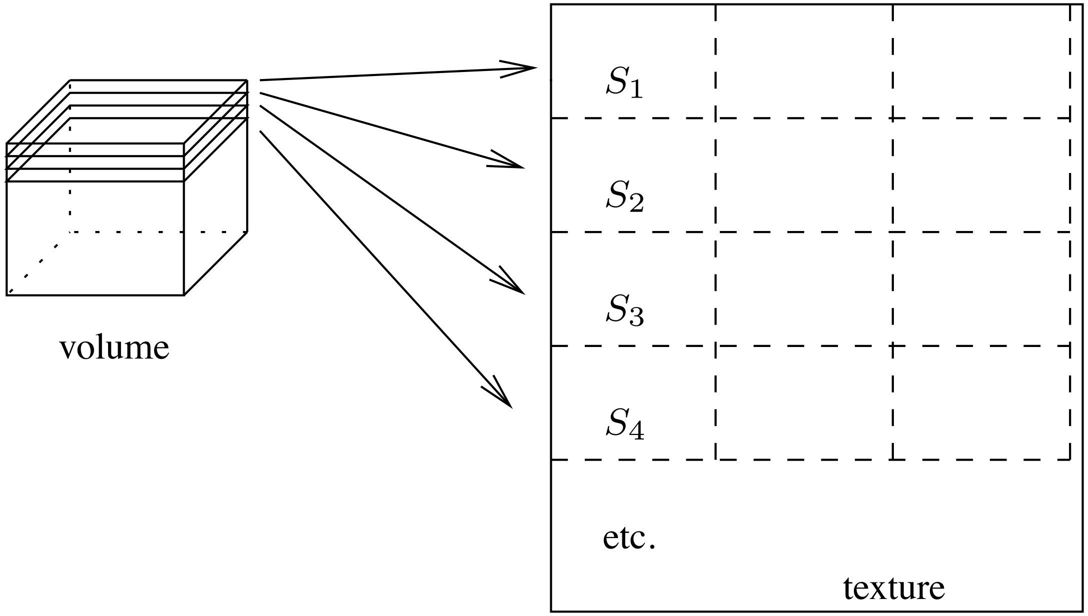
Implementation: CUDA - Pseudo code

- **For each thread (ray)**
  - compute corresponding ray direction
  - compute volume entrance and exit point for this ray
  - while (ray is inside the volume)
    - interpolate value at current position and accumulate intregal value
    - increment position along ray direction for defined stepsize
  - normalize integral value with step size



Implementation: CUDA - value interpolation

- **Recent graphics cards hardware support texture interpolation (1D, 2D, 3D)**
- **CUDA 1.1: only 1D, 2D textures, no 3D texture (December 2007)**
- **CUDA 2.0: also 3D texture support (August 2008)**
- **CUDA 1.1 work around:**
  - distribute volume slices into 2D texture
  - software interpolation (linear) between two bilinear interpolated texture values results into trilinear interpolated value



Implementation: OpenGL

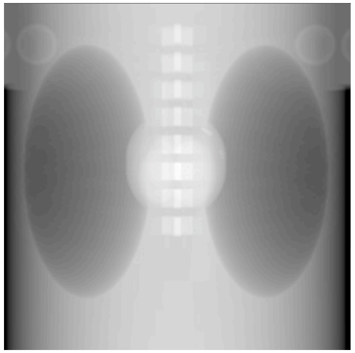
- GLUT (OpenGL Utility Toolkit) and GLSL (OpenGL Shading Language) based implementation

- Implementation differences compared to CUDA:
  - Setup equivalent geometry (cuboid) with vertices such that each resulting viewing pixel correnspondsto a ray
  - Cuboid texturing is replaced by ray casting
  - For each pixel the fragment shader program computes the ray cast analogous to CUDA using 3D textures

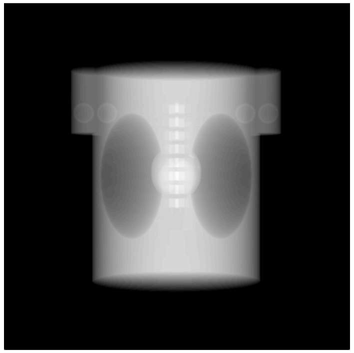
- Parallelization done by API

Evaluation

- **Comparison of CUDA 1.1, CUDA 2.0 and OpenGL for ray casting**
- **512^3 volume with float values (maximal texture size)**
- **Two different view configurations:**
  - near: all rays hit the cuboid
  - far: several rays on the outside do not cross the volume
- **Focus on:**
  - CUDA block size configuration
  - varying projection size (number of rays)
  - varying number of projections (different directions)
  - varying step size



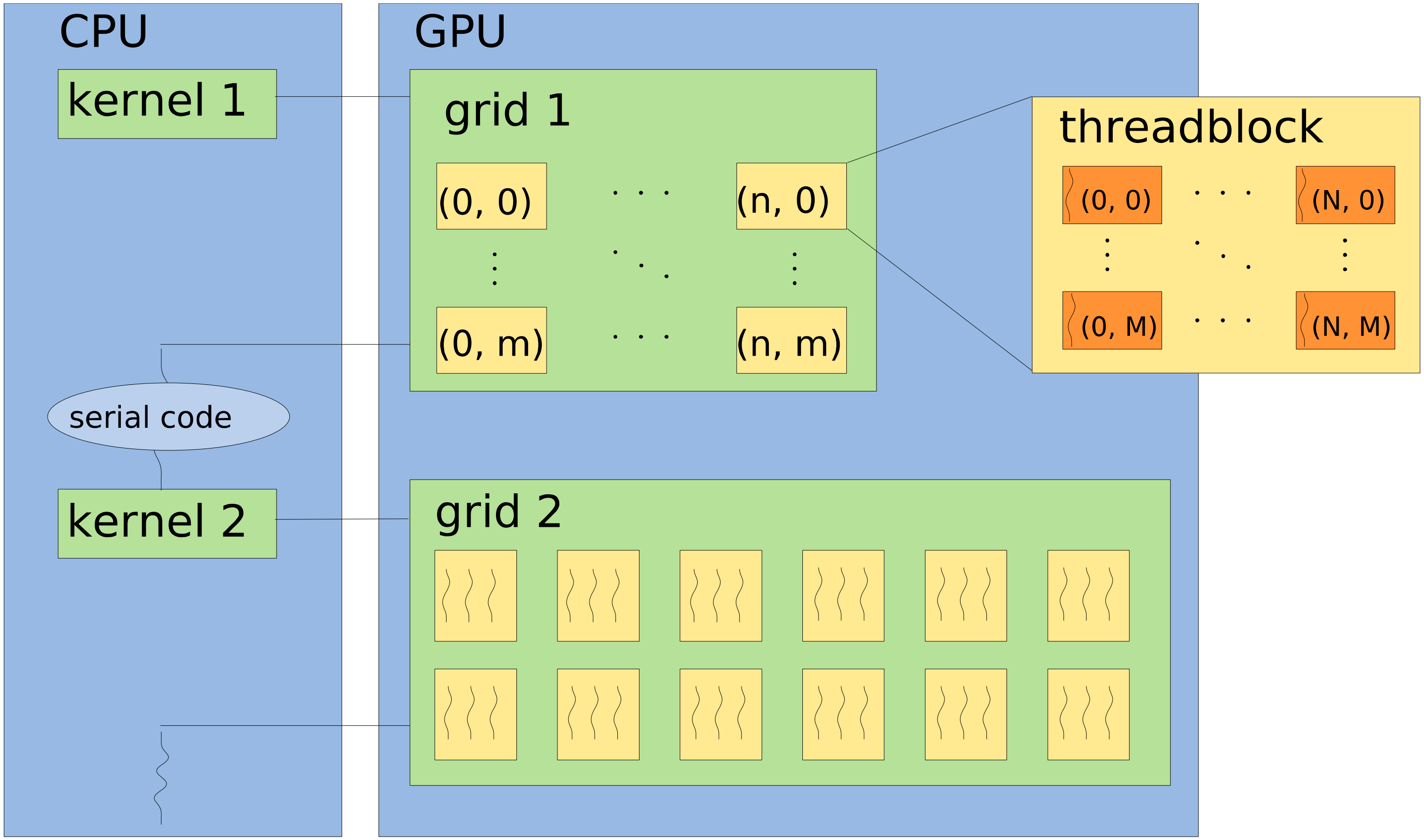
near



far

Implementation: CUDA - Parallelization Into Threads

- **High parallelization necessary for optimal performance**
- **Scalability due to dual abstraction level (grid / block)**



Benjamin Keck

Evaluation: CUDA blocksize setup

- **Performance in seconds**
- **GeForce 8800 GTX**
- **CUDA 2.0**
- **400 projections**
- **step size: 0.25 * voxel size**
- **varying block size**

	512^2 pixels		1024^2 pixels		2048^2 pixels	
Blocksz.	near	far	near	far	near	far
16×16	48.2	87.7	106	107	409	301
32×8	50.5	101	109	111	412	315
32×16	46.4	113	107	116	411	308
64×4	59.8	127	109	138	424	340
64×8	54.4	129	111	127	415	330
128×2	74.0	132	121	222	425	397
128×4	57.8	124	115	185	431	372
256×1	98.2	140	169	302	449	597
256×2	68.9	124	122	218	448	467
512×1	100	141	167	253	441	593

Evaluation: CUDA blocksize setup

- **Performance in seconds**
- **GeForce 8800 GTX**
- **CUDA 2.0**
- **400 projections**
- **step size: 0.25 * voxel size**
- **varying block size**

Blocksz.	512^2 pixels near	512^2 pixels far	1024^2 pixels near	1024^2 pixels far	2048^2 pixels near	2048^2 pixels far
16×16	48.2	87.7	106	107	409	301
32×8	50.5	101	109	111	412	315
32×16	46.4	113	107	116	411	308
64×4	59.8	127	109	138	424	340
64×8	54.4	129	111	127	415	330
128×2	74.0	132	121	222	425	397
128×4	57.8	124	115	185	431	372
256×1	98.2	140	169	302	449	597
256×2	68.9	124	122	218	448	467
512×1	100	141	167	253	441	593

Evaluation: graphics cards

	NVIDIA GeForce 8800GTX	NVIDIA QuadroFX 5600
Core clock	575 MHz	600 MHz
Shader clock	1350 MHz	1400 MHz
Memory amount	768 MB	1500 MB
Memory interface	384-bit	384-bit
Memory clock speed	900 MHz	800 MHz
Memory bandwidth	86.4 GB/s	76.8 GB/s

Results: 1024^2 projections

- **Performance in seconds**
- **Step size: 0.25 * voxel size**

		GeForce 8800 GTX			QuadroFX 5600		
		1024^2 pixels			1024^2 pixels		
# Proj.	FoV	CUDA 1.1	CUDA 2.0	OpenGL	CUDA 1.1	CUDA 2.0	OpenGL
1	near	9.4	**3.8**	12.1	6.38	**1.60**	3.22
	far	9.4	**3.8**	11.9	6.71	**1.59**	3.25
16	near	20.6	**7.5**	15.5	16.2	**5.16**	6.94
	far	27.3	**8.2**	15.5	21.4	**5.02**	7.09
100	near	86.4	**28.4**	36.4	70.5	**25.1**	27.4
	far	126	**30.2**	37.3	114	**24.6**	29.5
400	near	299	**107**	115	245	**99.8**	103
	far	527	**108**	116	515	**90.9**	109

Results: 1024^2 projections

- **Performance in seconds**
- **Step size: 0.25 * voxel size**

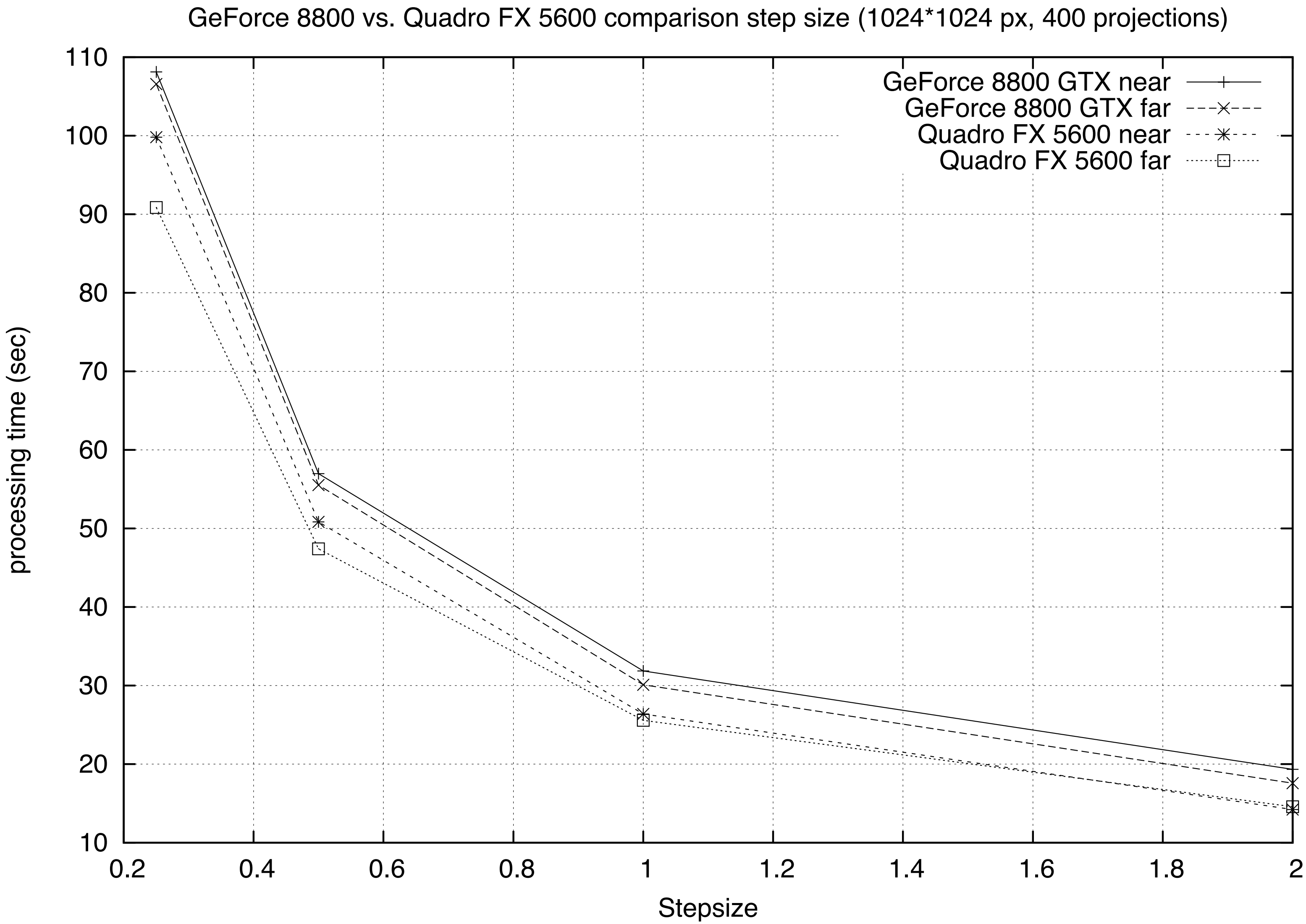
		GeForce 8800 GTX			QuadroFX 5600		
		1024^2 pixels			1024^2 pixels		
# Proj.	FoV	CUDA 1.1	CUDA 2.0	OpenGL	CUDA 1.1	CUDA 2.0	OpenGL
1	near	9.4	**3.8**	12.1	6.38	**1.60**	3.22
	far	9.4	**3.8**	11.9	6.71	**1.59**	3.25
16	near	20.6	**7.5**	15.5	16.2	**5.16**	6.94
	far	27.3	**8.2**	15.5	21.4	**5.02**	7.09
100	near	86.4	**28.4**	36.4	70.5	**25.1**	27.4
	far	126	**30.2**	37.3	114	**24.6**	29.5
400	near	299	**107**	115	245	**99.8**	103
	far	527	**108**	116	515	**90.9**	109

Results: 1024^2 projections

- Performance in seconds
- Step size: 0.25 * voxel size

		GeForce 8800 GTX			QuadroFX 5600		
		1024^2 pixels			1024^2 pixels		
# Proj.	FoV	CUDA 1.1	CUDA 2.0	OpenGL	CUDA 1.1	CUDA 2.0	OpenGL
1	near	9.4	**3.8**	12.1	6.38	**1.60**	3.22
	far	9.4	**3.8**	11.9	6.71	**1.59**	3.25
16	near	20.6	**7.5**	15.5	16.2	**5.16**	6.94
	far	27.3	**8.2**	15.5	21.4	**5.02**	7.09
100	near	86.4	**28.4**	36.4	70.5	**25.1**	27.4
	far	126	**30.2**	37.3	114	**24.6**	29.5
400	near	299	**107**	115	245	**99.8**	103
	far	527	**108**	116	515	**90.9**	109

Results: Graphics cards comparison



Benjamin Keck

Results: 512^2 and 2048^2 projections

- **Performance in seconds**
- **Step size: 0.25 * voxel size**

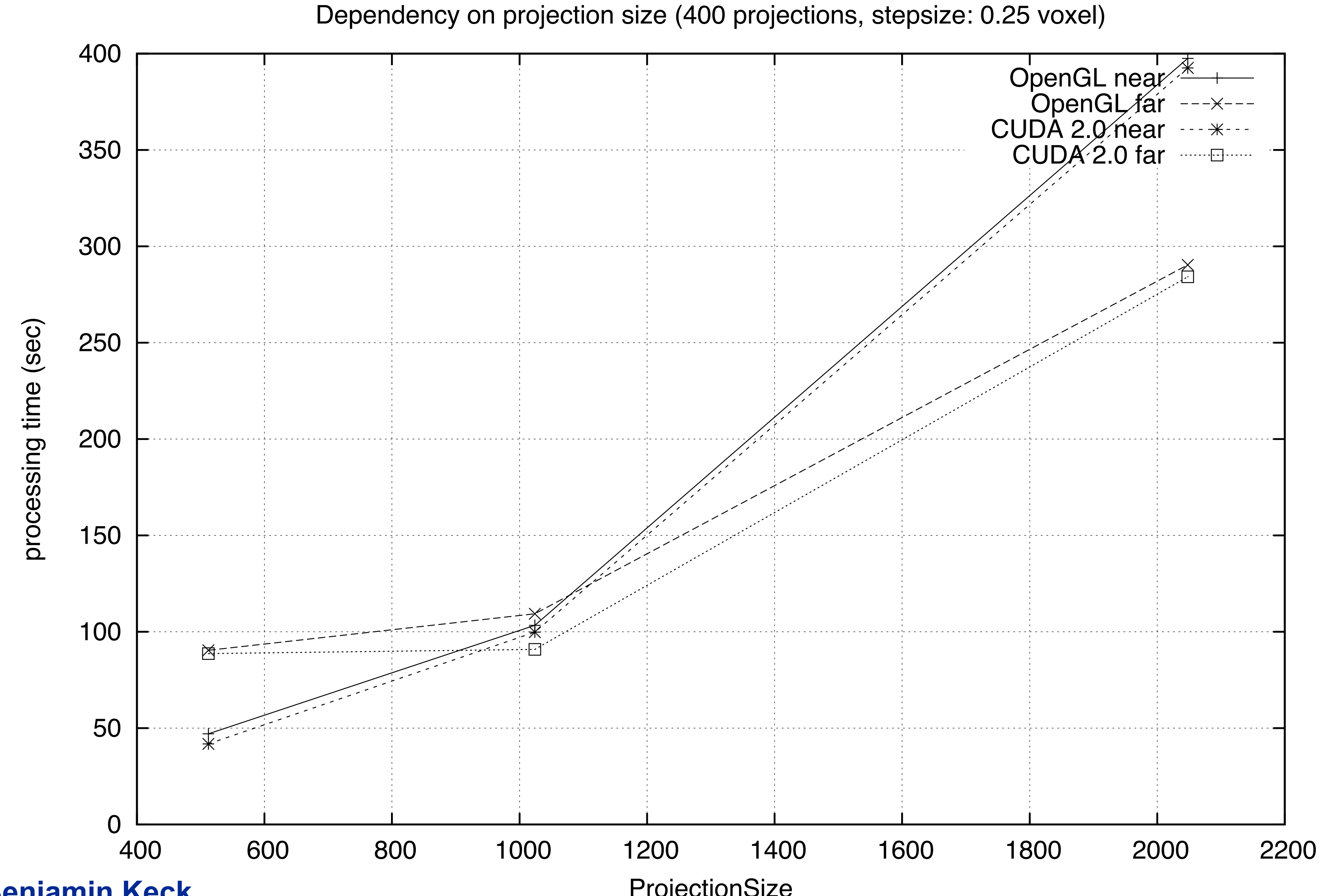
		QuadroFX 5600			QuadroFX 5600		
		512^2 pixels			2048^2 pixels		
# Proj.	FoV	CUDA 1.1	CUDA 2.0	OpenGL	CUDA 1.1	CUDA 2.0	OpenGL
1	near	6.22	**1.60**	3.25	7.70	**1.59**	3.27
	far	6.47	**1.60**	3.24	7.26	**1.58**	3.26
16	near	14.2	**3.30**	5.32	37.7	**15.8**	17.7
	far	18.2	**4.97**	6.45	30.6	**11.4**	13.3
100	near	55.5	**13.1**	21.7	208	**95.4**	98
	far	92.5	**24.4**	25.3	173	**67.4**	70.4
400	near	145	**41.8**	47.0	841	**392**	397
	far	386	**88.7**	90.3	864	**284**	290

Results: 512^2 and 2048^2 projections

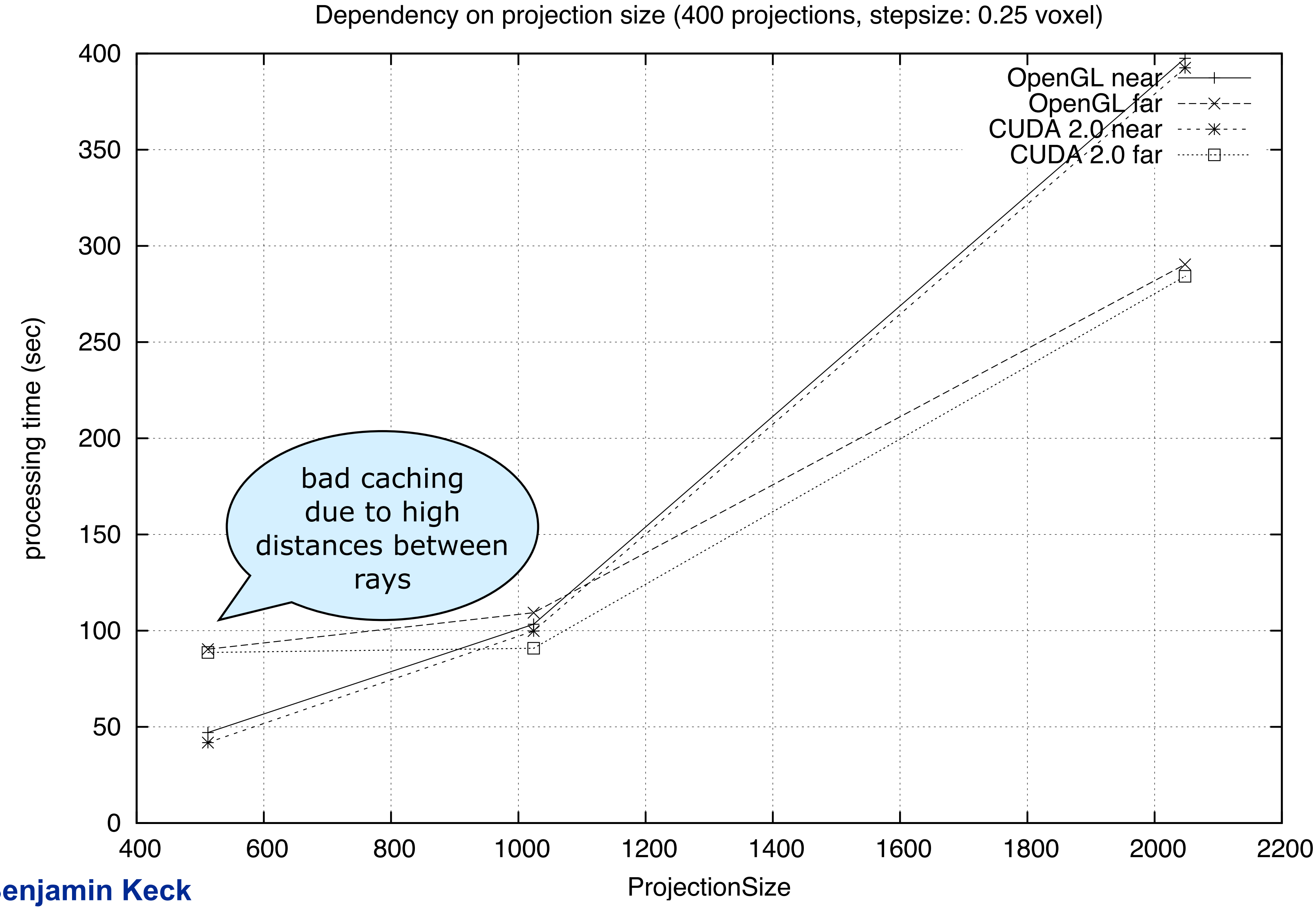
- **Performance in seconds**
- **Step size: 0.25 * voxel size**

		QuadroFX 5600			QuadroFX 5600		
		512^2 pixels			2048^2 pixels		
# Proj.	FoV	CUDA 1.1	CUDA 2.0	OpenGL	CUDA 1.1	CUDA 2.0	OpenGL
1	near	6.22	**1.60**	3.25	7.70	**1.59**	3.27
	far	6.47	**1.60**	3.24	7.26	**1.58**	3.26
16	near	14.2	**3.30**	5.32	37.7	**15.8**	17.7
	far	18.2	**4.97**	6.45	30.6	**11.4**	13.3
100	near	55.5	**13.1**	21.7	208	**95.4**	98
	far	92.5	**24.4**	25.3	173	**67.4**	70.4
400	near	145	**41.8**	47.0	841	**392**	397
	far	386	**88.7**	90.3	864	**284**	290

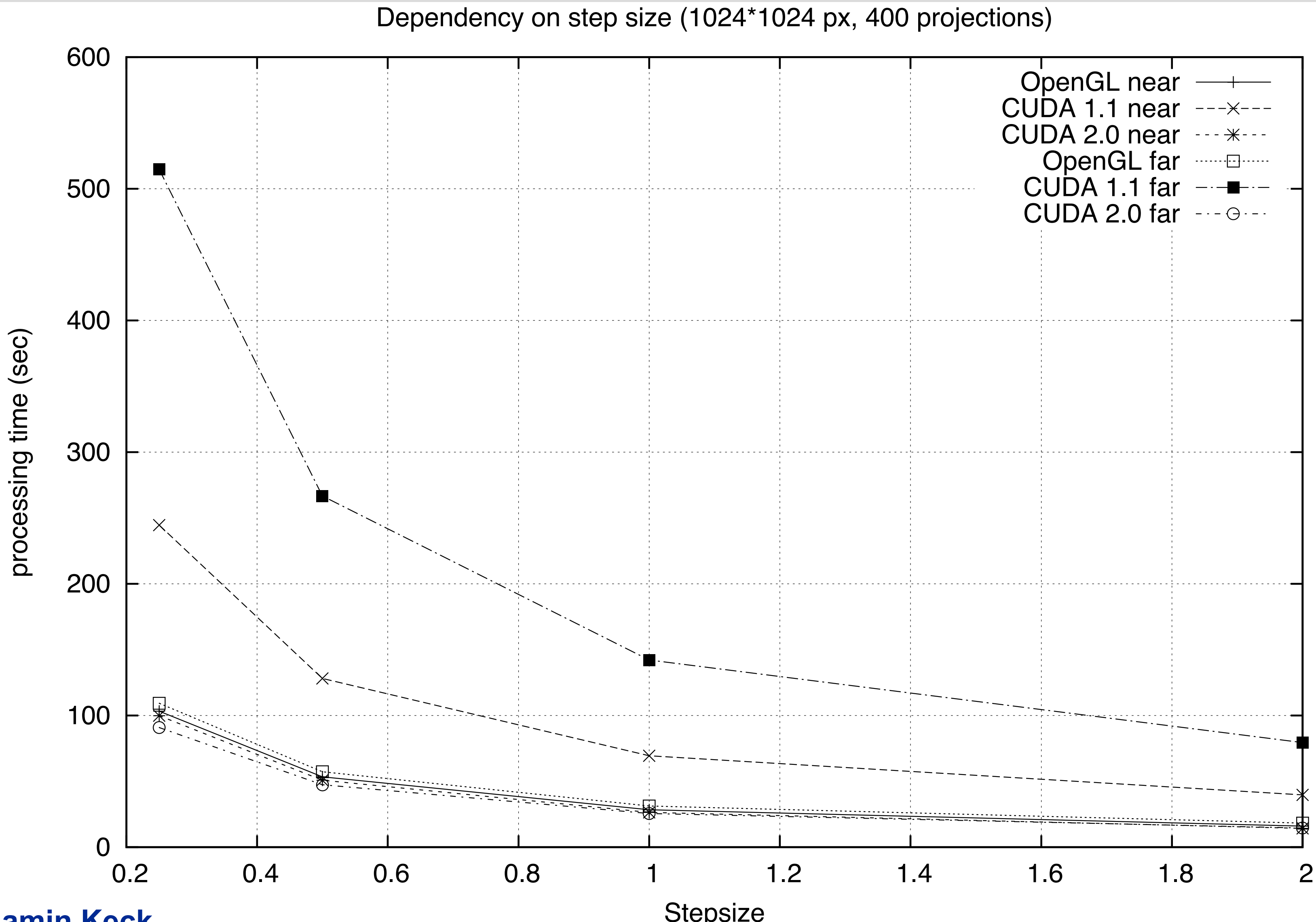
Results: Projection size dependency



Results: Projection size dependency



Results: Step size dependency



Benjamin Keck

Discussion & Conclusion

- Different initialization time between CUDA and OpenGL
- Parallel computation dispatching is more efficient for increasing image size (OpenGL and CUDA)

- 3D texture support is essential for ray casting due to the hardware-accelerated interpolation (drawback on CUDA 1.1)

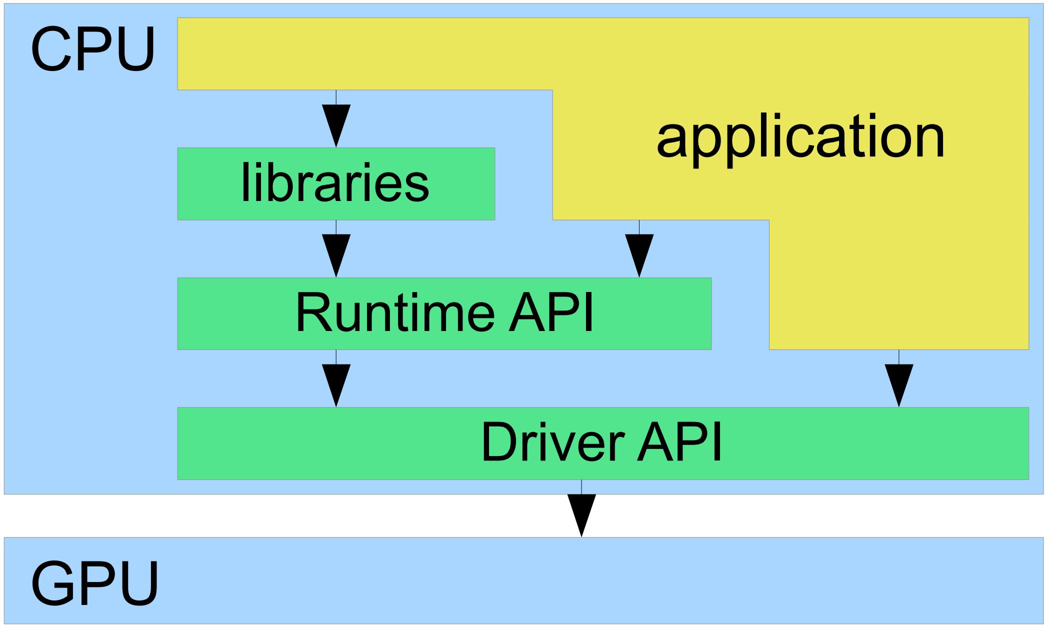
- OpenGL requires more implementation efforts for non-experts
- Ray casting can be easily ported to the GPU using CUDA

- In comparison to a single-threaded non-optimized straight-forward CPU implementation we achieve a speedup factor of ~150.

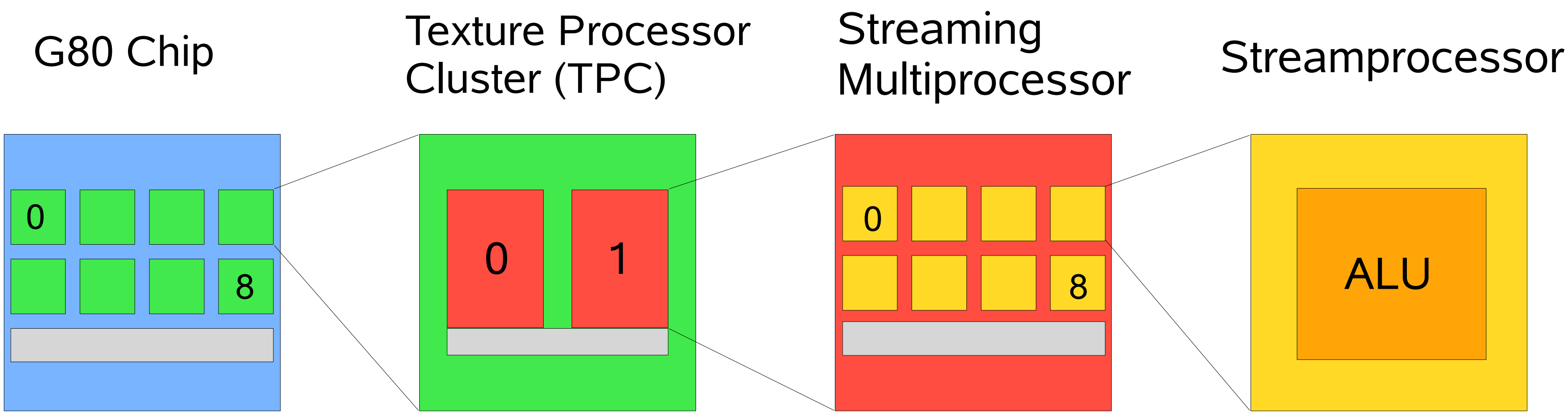
Thank you!

Implementation: CUDA - Software

- **CUDA-capable devices: started with GeForce-8 series**
- **CUDA libraries: mathematical function with high abstraction level**
- **CUDA Runtime API: simplified memory-, device- and texture-management**
- **CUDA Driver API: low-level API, no emulation-mode**

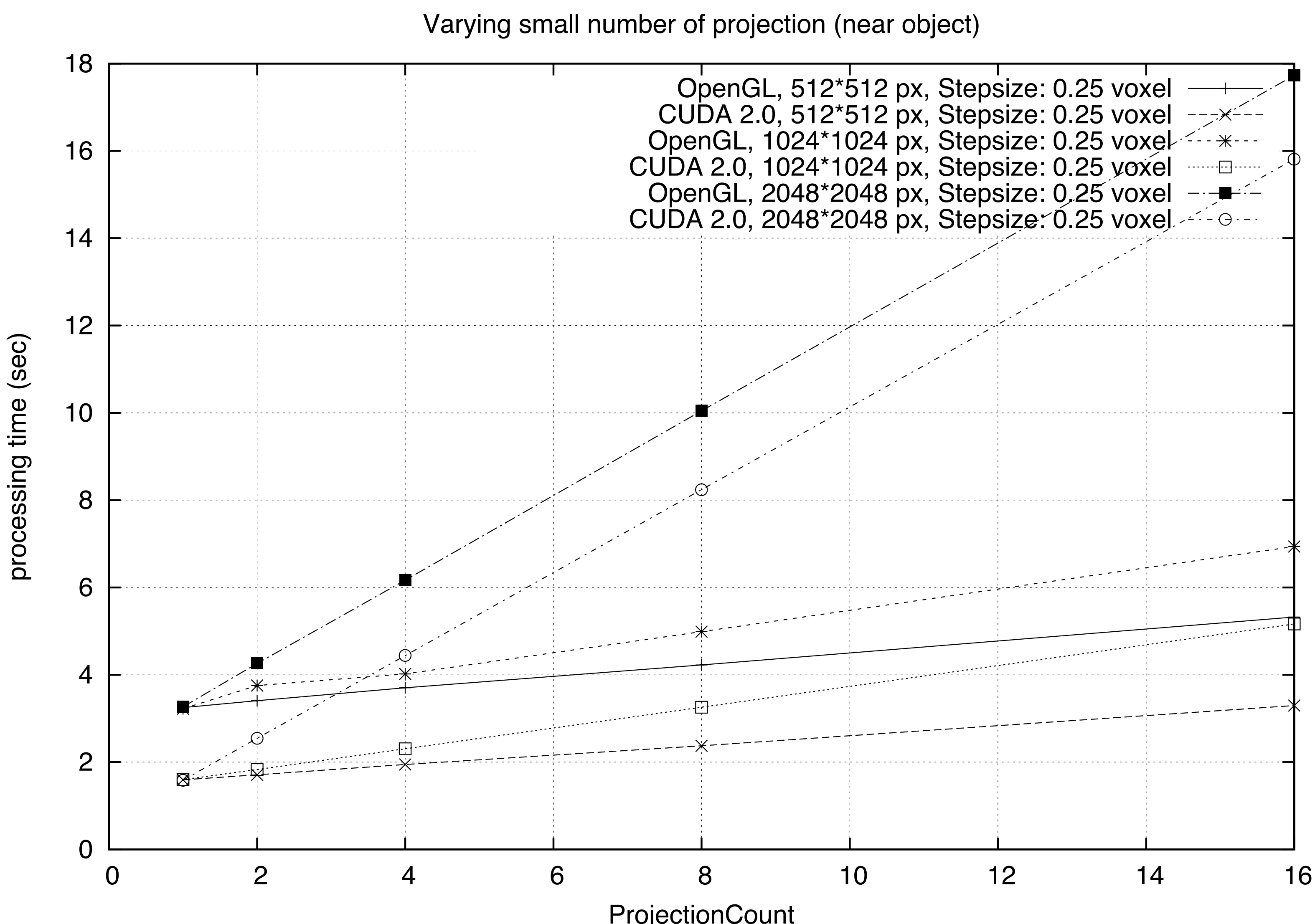


Implementation: CUDA - Hardware



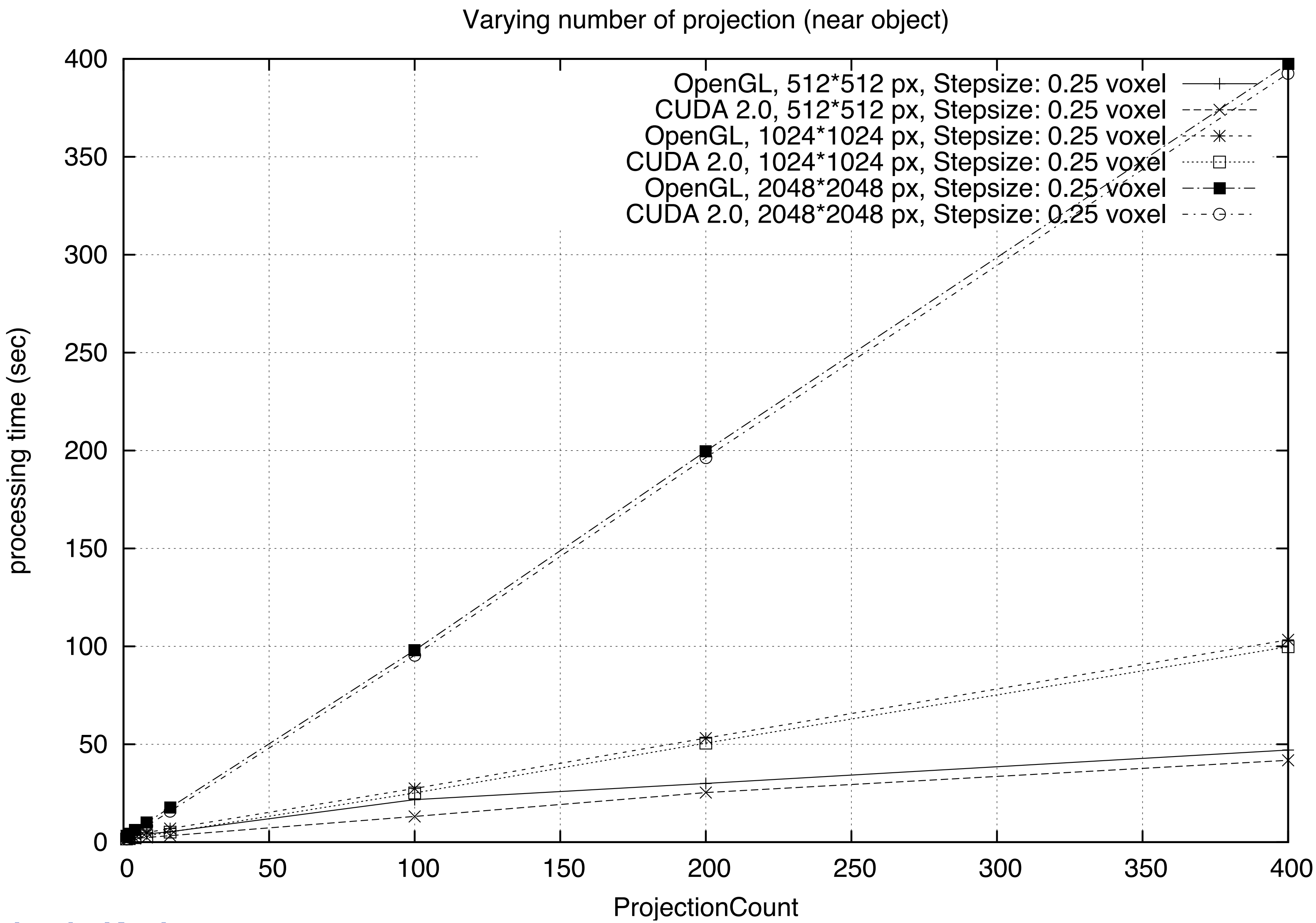
- **8 Texture Processor Cluster**
- **16 Streaming Multiprocessors**
- **128 Streamprocessor (SP) = Shadercores**

Results: Scaleabilty / dependence on # projections



Benjamin Keck

Results: Scalabilty / dependence on projection count



Benjamin Keck