Ascii – Art Rendering on CUDA

Seminar: Parallel Computing on Graphics Cards
Author: David Rohr
Tutors: Julian Kunkel, Olga Mordvinova
Date: 16.6.2009

"Once upon a time we've (my friend Kamil and I) bought two old Herculeses as secondary monitors. We didn't know for that time that our Diamond Stealths 64 cards would become obsolete soon. The next day we downloaded the logo of Linux Texas Users Group - nice silly penguin looking like a cowboy! It was so exciting logo ... we decided that we couldn't live without it and we wanted to see it at boot time as a logo on our secondary monitors. There was a small problem - Hercules doesn't support color graphics. So we decided to convert the penguin image to ascii art using netpbm tools."

Folie 1

q1 Quote from aalib creator
qon; 26.07.2009

Introduction – Topic

Problem: Video/Image Output with plain console, escpecially for remote access (ssh etc.)

q2

- First ASCII Rendering approaches to display images on textmode only computers.
- Later also to display graphics during Telnet / SSH sessions.
- In fact no really urgent problem.
- BUT: Since we do image manipulation well suited to demonstrate GPU multiprocessing.

qon; 26.07.2009

Introduction – aalib (mplayer)

- Introduction
  - Topic
  - aalib
- Implementation
- Algorithm
- Benchmarking
- Optimizations
- Summary

Solution: Render video to ascii characters, Display Characters on console. (aalib 1.2 10.3.1998)

Folie 3

q3

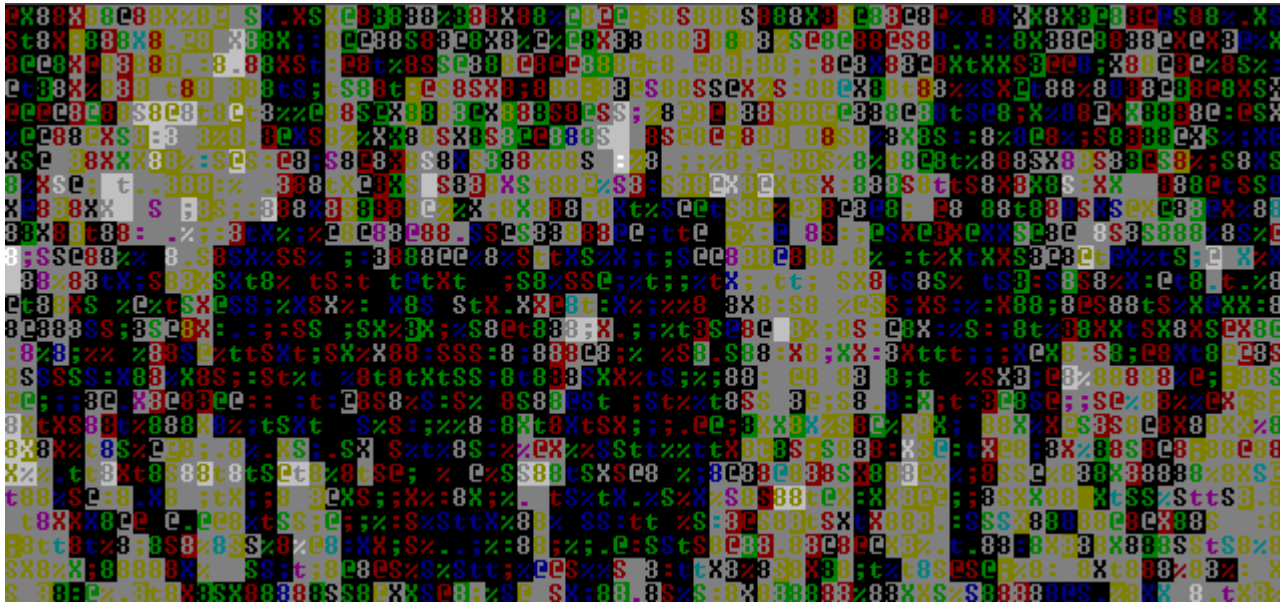
- aalib ASCII Art Library.
- Oldest example for ascii rendering.
- Black and white renderer only.

qon; 26.07.2009

Introduction – libcaca (mplayer)

Alternative: libcaca

(can even handle color)



- Seems buggy using cygwin
- Very bad resolution on cygwin only

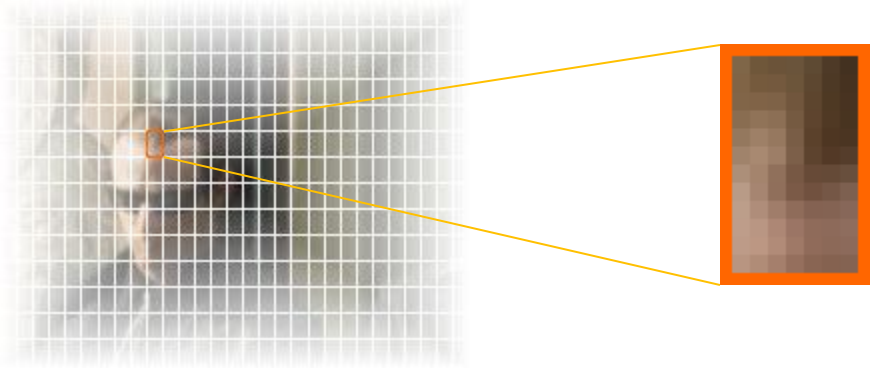
q4

- libcaca.
- More rescent ascii art renderer.
- Can also do colored rendering.
- Image here is bad because of limited cygwin resolution.

qon; 26.07.2009

Implementation – General Approach

Split Image in Fields



Process each Field on its own

Combine Fields

q5

- New approach for ASCII rendering implementation.
- Image is splited in Fields each 7 times 12 pixels.
- ASCII characters have 7 times 12 pixels too. (at least when using raster font)
- Task is to find character, fore- and background-color mathing the field best.
- This can be done for each field on its own.
- Can be massively parallelised.
- Well suired for CUDA.

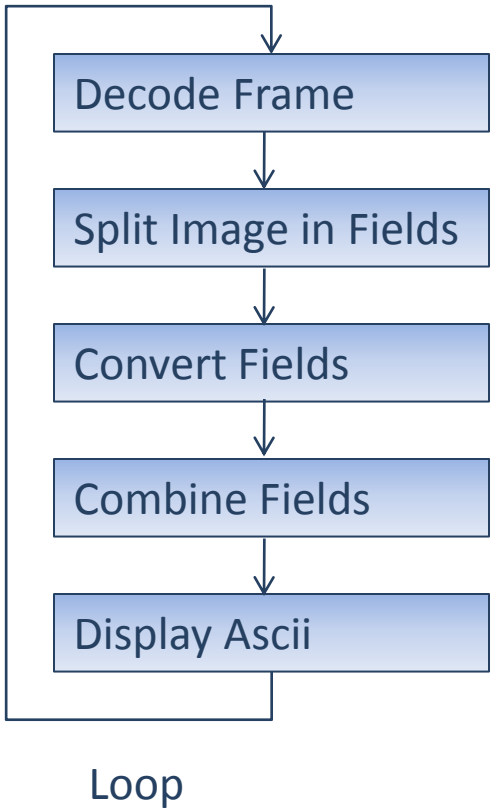
qon; 26.07.2009

Implementation – General Approach

First simple approach:

- Of course splitting and recombining are virtual and included in conversion.

- Fields are totally independant!
  → trivial parallellization



q6

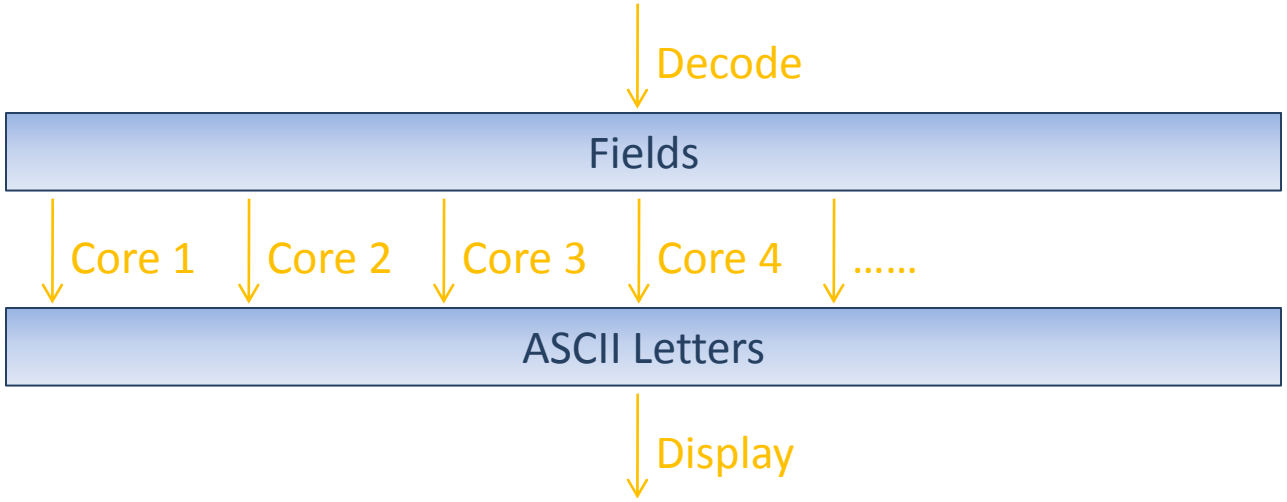
- Main tasks of rederer are: Frame decoding, ASCII rendeing, Display.
- Splitting into fields and recombination are virtual tasks here and do not require CPU time.
- At least the conversion is totally independant and can therefore be parallelized trivially.

qon; 26.07.2009

Implementation – Trivial CPU Multithreading

Multithreaded CPU Approach:

Image: 1024 * 768 Pixels
ASCII Letter: 12 * 7 Pixels
→ 9362 Fields



q7

- First multithreaded approach. (still on CPU)
- Decoder decodes a frame first.
- Renderer starts lots of threads, each one converting some, or even a single field.
- When all threads are finished the result is displayed.
- An image of 1024 times 768 pixels results in approx 10000 Fields.
- More then 1000 threads are at least required to saturate a modern GPU.

Implementation – Trivial CPU Multithreading

Multithreaded CPU Approach:

Distribute among Cores for best Cache Efficiency!

Core 1

Core 2

.
.
.

Consecutive lines reside in consecutive memory space
→ Distribute considering lines rather than cols

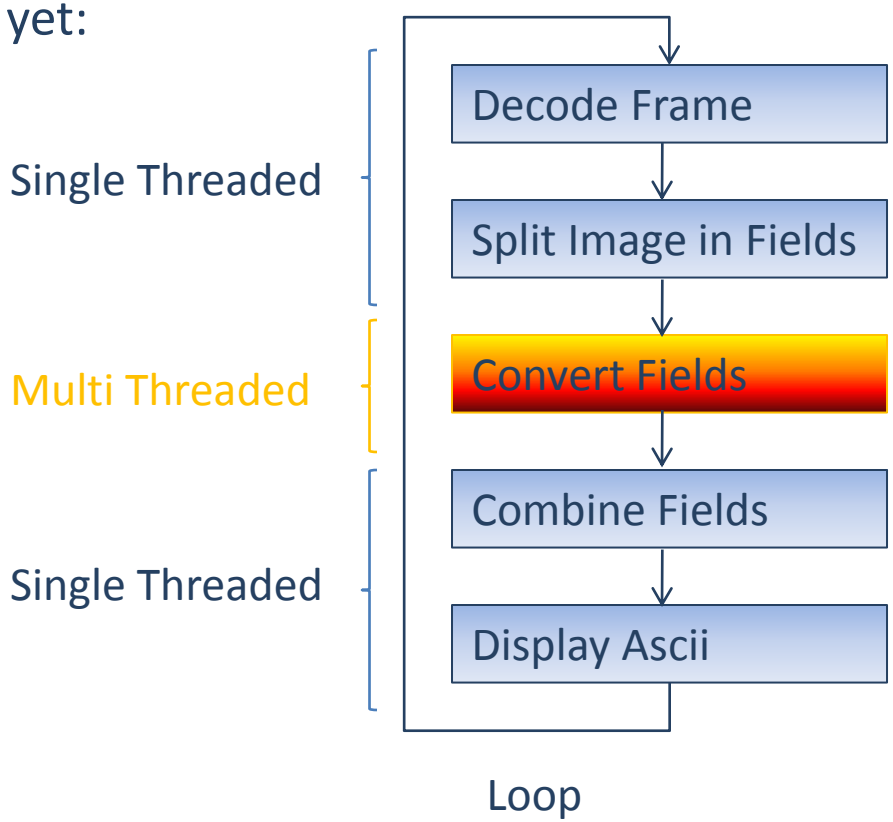
q8

- For the CPU processing each field by a seperate thread is too much overhead.
- Each thread has to produce multiple fields.
- Workload can for example be distributed among lines or cols.
- Lines seem suited best, because then every thread would work on a consecutive memory segment.
- Images are stored in Memory as array of lines.

qon; 26.07.2009

Implementation – Trivial CPU Multithreading

Problem yet:



Goal:
Make whole Code multithreaded

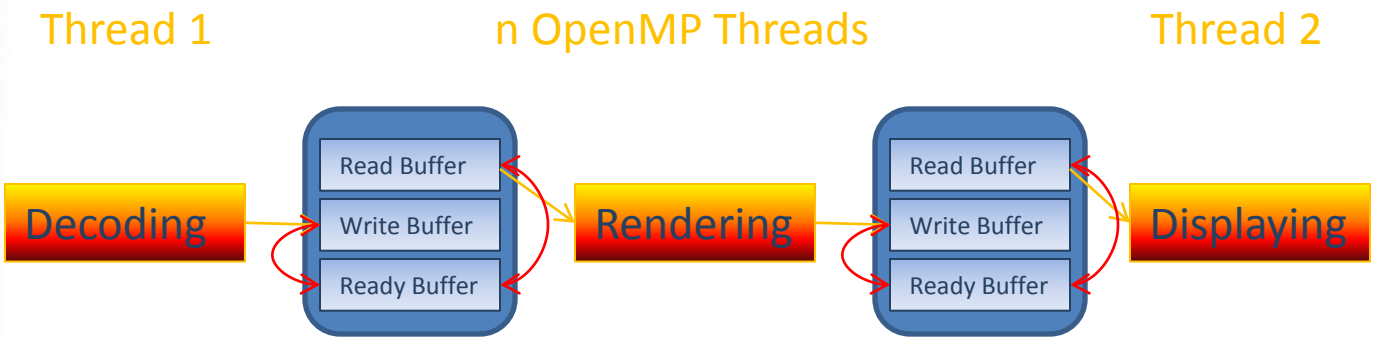
q10

- Until now only the conversion is multithreaded.
- Decoding and display is not.
- Next goal obviously is to multithread the whole code.

qon; 26.07.2009

Implementation – Complex CPU Multithreading

Multithreaded Approach



Decoder / Renderer buffer:
- Decoder writes to write buffer, when frame finished swaps with ready buffer.
- Renderer reads from read buffer, when frame read swaps with ready buffer.
- Only one semaphore used to controll access to ready buffer for pointer swapping.

(Same for Renderer / Display buffer)

Advantages:
- Decoder decodes all frames in time, so audio will still play even if renderer is slow.
- Renderer always renders newest frame.
- Using a motion estimation codec all frames need to be decoded anyway.

q78

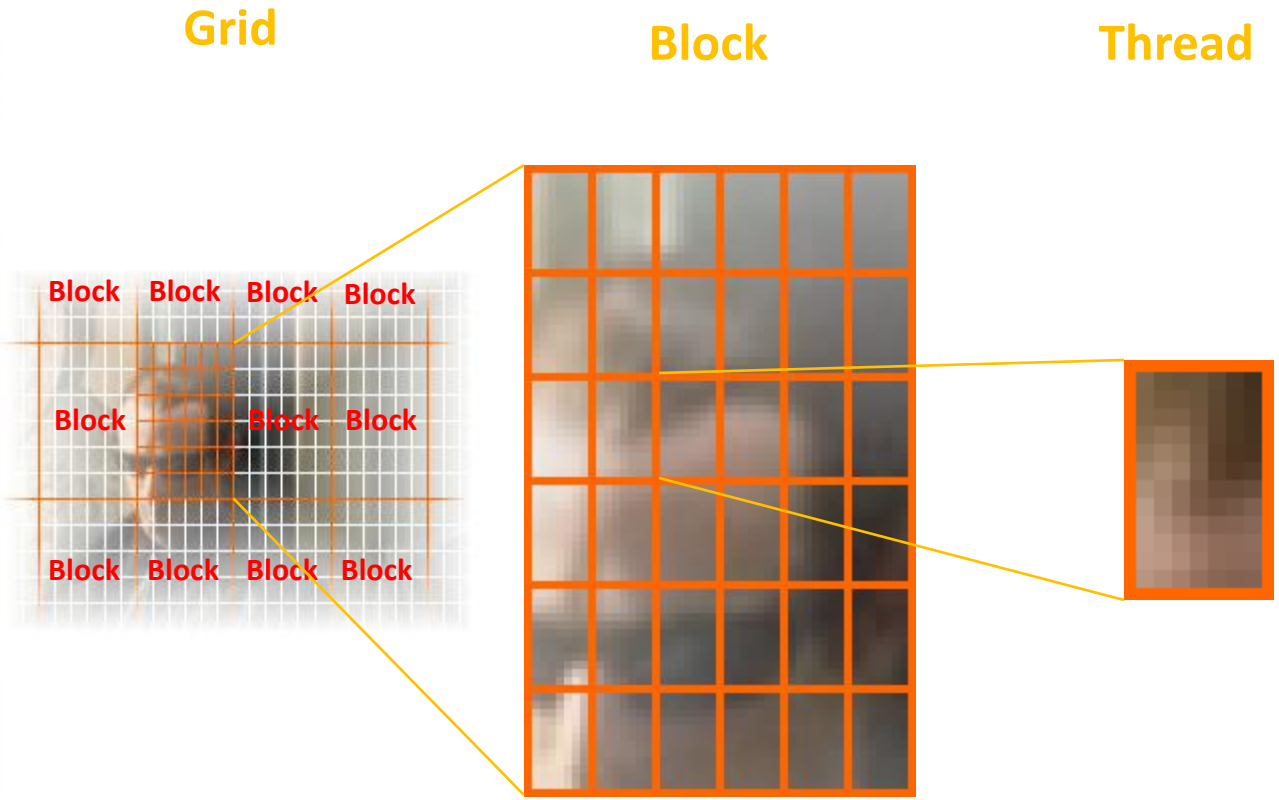
- Problem if decoding cannot be splited in threads very well (Display can in any case be splited :|)
- Make decoder decode next frame while renderer processes last frame with many threads.
- The same for renderer display.
- This way rendering will be real time as long as decoder decodes in real time and total CPU power is sufficiant.
- Implementation uses 3 buffers at any border.
- One read buffer, one write buffer and one ready buffer.
- When read thread or write thread are ready, they lock the ready buffer using a semaphore.
- Then they exchange pointers of their buffer and the ready buffer.
- Another advantage: When CPU cannot handle decoding in real time frames should be skipped.
- But Frames cannot be skipped when using motion estimation codecs.
- With the new approach all frames are always decoded and skipped by the renderer automaticly if the decoder is 2 frames ahead.
- Since decoder usually is responsible for audio playback and audio buffers are usually small, decoder must not pause.

qon; 26.07.2009

Implementation – GPU

Most simple GPU Approach: „ONE THREAD PER FIELD“

aprox. 10000 Fields for 1024*768 should saturate the GPU



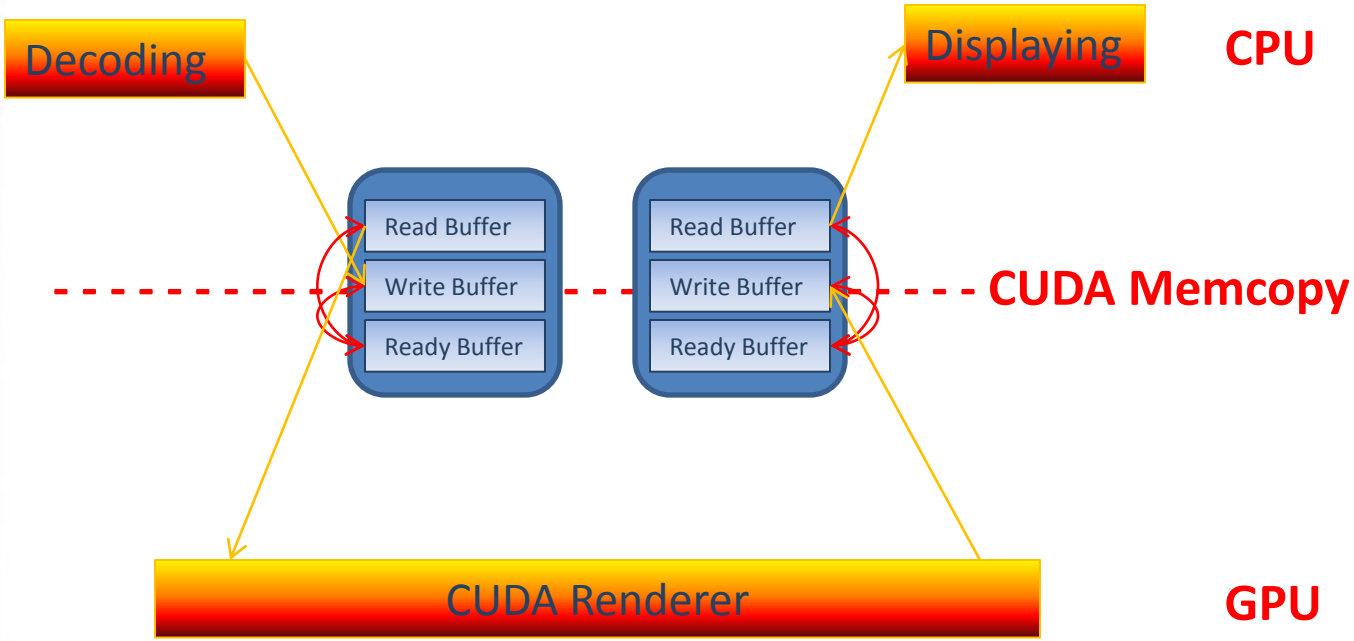
q11

- For the CPU a 1:1 correspondence between threads and fields would not work.
- For the GPU it definitely will.
- For CUDA we need blocks and threads.
- Blocks will be distributed among the multiprocessors of the GPU.
- So we split the image in blocks of 16 times 16 threads.
- (Example here shows 6 times 6 for obvious reasons)
- Blocks should not be too big because of overhead at image borders.
- Blocks of 256 times 1 thread will require 2 blocks per line of fields if image width was for example 257 fields.
- Blocks should also not be too small to saturate GPU.
- 16 times 16 seems well suited.

qon; 26.07.2009

Implementation – GPU & CPU

CUDA Threading and Transfer



q12

- Best implementation would be to make CPU decode and display, both multithreaded.
- GPU does rendering, also multithreaded.
- The Question is where to locate the buffers.
- Since PCIe transfer and GPU rendering can be done in parallel fastest way woulg be to locate the buffer in GPU memory.
- Alternative is buffer in Main memory.
- When GPU is ready it fetches the frame and starts rendering. TIme for fetching the image is lost then.
- In the benchmark later the GPU Memory buffers are not implemented.
- Though some measurements of transfer time will be given.

qon; 26.07.2009

Implementation – Programming Overview

CUDA Threading and Transfer

convert.h

```
#ifdef __CUDACC__
#define lpSource lpGPUSource
#define lpAsciiOut lpGPUAsciiOut
#define __use__global__ global
#define CUDA_START_LOOP \
i = blockIdx.x * blockDim.x + threadIdx.x; \
j = blockIdx.y * blockDim.y + threadIdx.y;
#else
#define lpSource lpCPUSource
#define lpAsciiOut lpCPUAsciiOut
#define __use__global
#define CUDA_START_LOOP \
for (i = 0;i < nLines;i++) { \
for (j = 0;j < nCols;j++) {
#endif

__use__global convert(int nLines, ....) {
#pragma omp parallel for private(.....
  CUDA_START_LOOP
              ...............................
              - Converter Code –
              ...............................
```

Convert.cpp

```
#include " convert.h"
Void CPUConvertAscii()
 {
   convert(......
 }
```

Convert.cu

```
#include " convert.h"
Void GPUConvertAscii()
{
   dim3 dimBlock(16, 8);
   dim3 dimGrid((nCols + dimBlock.x - 1) .....
   cudaMemcpy(lpGPUSource, lpCPUSource,
       nSourceSize, cudaMemcpyHostToDevice);
   convert<<<dimGrid, dimBlock>>>(......);
   cudaMemcpy(lpCPUAsciiOut, lpGPUAsciiOut,
       nFieldCount, cudaMemcpyDeviceToHost);
}
```

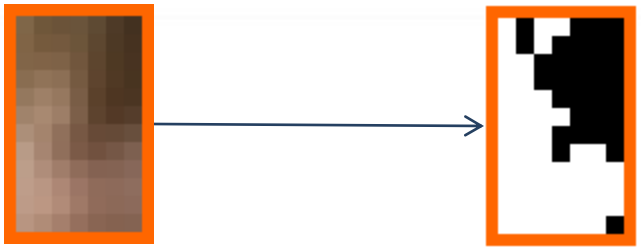
q13

- Idea is to have one converter in convert.h, that does the job on both, GPU and CPU.
- Convert.h is included in c++ converter for CPU and CUDA-converter for GPU.
- Some pointers (lpSource, lpAsciiOut) point to input and output buffers.
- Pointer positions are changed by #define statements.
- Defines are different depending on which compiler is used.
- Convert.cpp will just include convert.h.
- Convert.cu will include convert.h and supply some code for PCIe transfer.

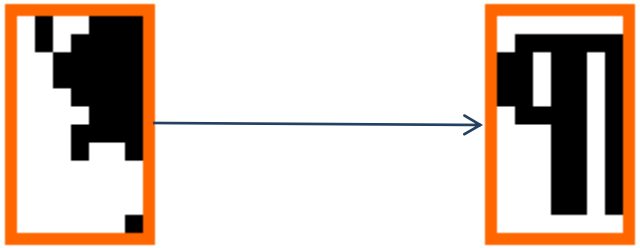
qon; 26.07.2009

Algorithm – Contrast based

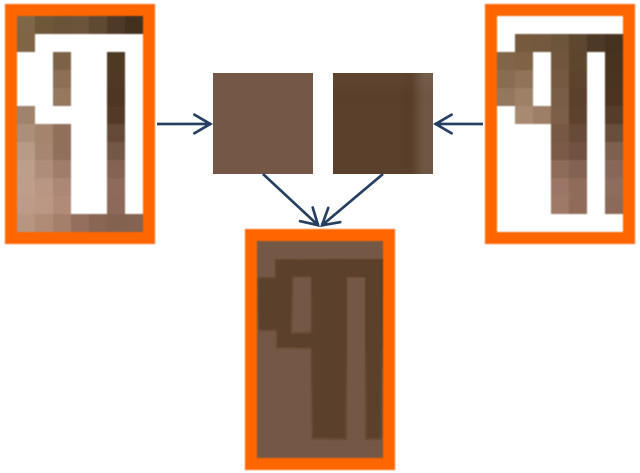
1. Convert Field to B&W



2. Find ASCII Letter that mathes best



3. Get appropriate color by using average values of Field



q14

- Now to the algorithm itself.
- First discuss several ideas.
- Contrast based idea is to reduce the field to a B&W boolean map.
- Boolean values are compared agains Character (which is a boolean map too) and mismatches are counted.
- Character with less mismatches wins.
- Fore- and background color are calculated by averaing color values of the corresponding pixels of the field.

qon; 26.07.2009

Algorithm – Contrast based

Analysis:

- Continous color distribution leads to large pixels each the size of a character.

- Itaration over characters is needed.
  1 memory access per character per pixel. (84 * 256 per Field)
  Not tuned for CUDA.

- Averaging of colors gives RGB values instead of 8 bit ASCII colors.

q15

- Since the field itself is small compared to size of the image, color values usually only slightly differ within a field.
- Except the field is located at a hard border in the image, i. e. a horizon with blue sky and some dark parts below it.
- This makes fore- and background color in most cases almost identical.
- Characters therefore appear as huge pixels, moreeven characters are not even recognized as ascii characters.
- Further when searching for best character the algorith iterates over the characters and compares the boolean maps.
- This results in quite a lot of memory access. Even more each value from memory is read and used exactly one time.
- Not well suired for CUDA, better would be to somehow reduce the information needed to characterize a character.
- Averaging process gives RGB values for the color (16 mio), while ASCII is restricted to 8 bit / 16 colors.

qon; 26.07.2009

Algorithm – Number Reduction

1. Assign each field a 64 bit Number!

0x38A94F3B92387C8D

0x1D9C348A9FF942A5

(This has to be done in a way, that „similar“ Fields are assigned similar Numbers.)

2. Build a list of all numbers assigned to each ASCII character / color combination!
3. Do a binary search for the field‘s number in the letters‘ numbers and take the nearest one! This should be similar to the field itself.

q16

- Another idead is to assign each field / character a 64 bit number, that should represent the shape of the character.
- Similar characters should result in similar numbers.
- This way we proceed our idea of reducing the information required to characterize a character to the very end.
- For the conversion build a list with the numbers for all combinations of characters / ascii colors together with a pointer to this character / color.
- This can be precalculated.
- Whole conversion process now consists of field to number conversion and binary search in the list.
- The character with the closest number to the field should then match the field at best.

qon; 26.07.2009

Algorithm – Number Reduction

Analysis:

- No iteration over characters.
- Algorithm has „intrinsic“ noise resulting in less block building.
- When restricting list to ASCII colors only output will also be ASCII only.

- Random memory access for binary search.

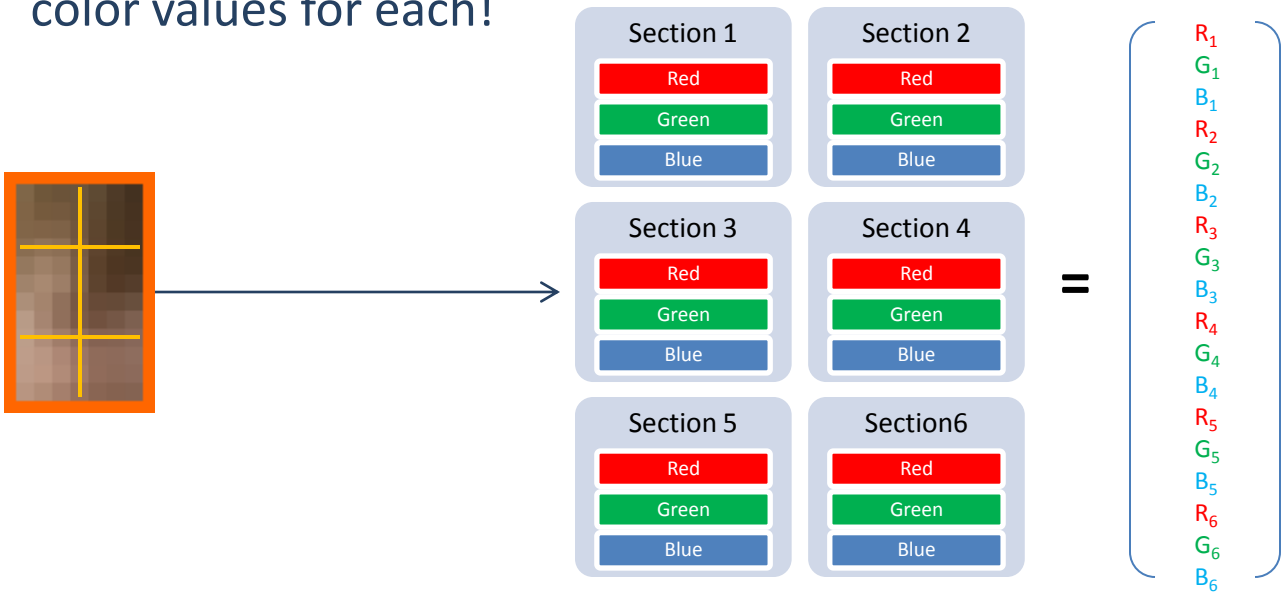
q17

- This avoids the character itaration and reduces memory access.
- Algorith is also exspected not to produce "large pixels" as first one, and in fact does not.
- New problem arising is the random memory access inherent in the binary search over the large list. (several megabytes)

qon; 26.07.2009

Algorithm – Vector based

1. Split each field further in 6 sections and calculate average color values for each!



2. This way each field can be considered as a point in a 18 – dimensional vector space. The same can be done for all ASCII character / color combination. Then take the character / color with the smallest euclidean distance!

q18

- Another way to reduce the characterizing information required is to split the fields further into 6 sections and build average colors within each section.
- Each field not is a 18 dimensonal vector. (18 = 6 * 3 for 3 colors)
- The same can again be done for each character/color combination. Character with smallest euclidean distance is supposed to suite best.

qon; 26.07.2009

Algorithm – Vector based

Analysis:

- Building sections allows variations inside section → less blocks.
- Field represented by average section colors only → less memory access.
- ASCII colors only.

q19

- Results: less block building, ascii colors only (not RGB), less memory access.
- Lots of ways to do a fast search for the best vector using restriction to hyperplanes or building a grid etc.
- Anyway this is about GPUs :|.

qon; 26.07.2009

Algorithm – Differential

1. Again split fields in sections!
2. Iterate over all ASCII Characters!
3. Differentially calculate best colors by minimizing distance!
   (See later for details)
4. Take the best Character

Analysis:

- Building sections allows variations inside section → less blocks.
- Field represented by average section colors only → less memory access.
- Calculating colors multiple times faster than brute forcing.

- Differential calculation results in RGB colors.

q20

- So let's discuss one Calculation intensive way to find the best character / color.
- We itarate over the ascii characters.
- For each character we calculate the best color differentially using as input only some constants for the character.

qon; 26.07.2009

Algorithm – Comparison

Summary Table

Algorithm	Result	Speed	Memory Access	RGB / Ascii colors
Contrast based	Blocks	Medium	Much	RGB
Number reduction	Good	Very fast	Random	Ascii
Vector based	Good	Slow	Less	Ascii
Differential	Very good	Fast	Least	RGB

Analysis:

Differential algorithm: Least memory access
Requires processing power

→ High arithmetic density
→ Suited for CUDA

q21

- Before further desciption of differential algorithm some overview.
- Differential seems suited best for our CUDA approach.

qon; 26.07.2009

Algorithm – Some Maths

Distance Calculation

(minimize distance $\longleftrightarrow$ minimize distance squared)

$$d(v, v')^2 = ||v - v'||^2 = \left(\begin{pmatrix} R_1 \\ G_1 \\ B_1 \\ R_2 \\ G_2 \\ B_2 \\ R_3 \\ G_3 \\ B_3 \\ R_4 \\ G_4 \\ B_4 \\ R_5 \\ G_5 \\ B_5 \\ R_6 \\ G_6 \\ B_6 \end{pmatrix} - \begin{pmatrix} R'_1 \\ G'_1 \\ B'_1 \\ R'_2 \\ G'_2 \\ B'_2 \\ R'_3 \\ G'_3 \\ B'_3 \\ R'_4 \\ G'_4 \\ B'_4 \\ R'_5 \\ G'_5 \\ B'_5 \\ R'_6 \\ G'_6 \\ B'_6 \end{pmatrix} \right)^2$$

v constant, calculated from field.
v‘ depends on character specific constants, foreground f and background b.

Minimize distance:
Jacobi Matrice = 0, Hesse Matrice positive definit
$Jac(d(v, v'(f, b))^2, (f, b)) = 0$

q22

- Distance is calculated by the square root of the scalar product square of the vector difference.
- Minimizing square root of a function is the same as minimizing the function, so forget about the root.
- Distance is then a polynomial of second degree in the colors of the two points.
- Colors of the field are fixed, colors for the character have 6 components: fore- and background in each RGB value.
- Fortunately the RGB values are completely independant so we will restrict to only one.
- Observe the distance as function of fore- and background value.
- For the minimum distance the Jacobi Matrice has to be 0.
- Since there are only squares in the distance polynomial, the derivatives only contain linear terms.
- The equations for the Jacobi matrice are therefore two linear equations in two variables and can be solved.
- Next step would be to check if Hesse matrice is positive definit.

qon; 26.07.2009

Algorithm – Some Maths

Distance Calculation

(For one color (red, green, blue))

Due to the shape of the distance function there is one well definid Minimum.

Hesse matrice automatically positive definit.

Gives 2 linear equations:

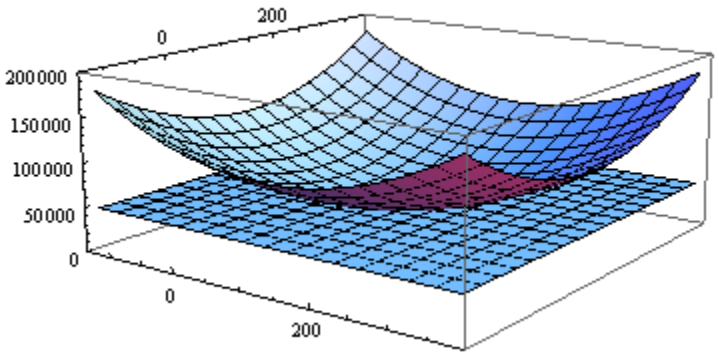
$$\frac{\partial}{\partial f}\, d(v, v'(f,b)) = 0$$

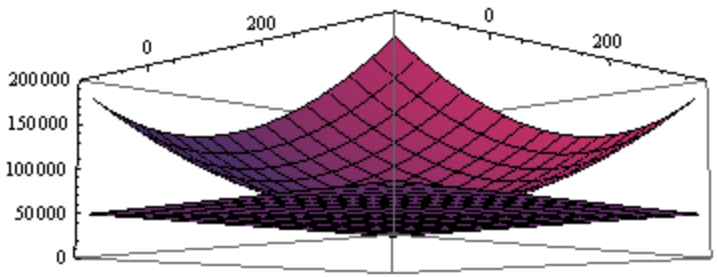
$$\frac{\partial}{\partial b}\, d(v, v'(f,b)) = 0$$

→ Problem can be solved

Distance for red, green, blue unrelated

→ Can be solved independantly for red, green, blue





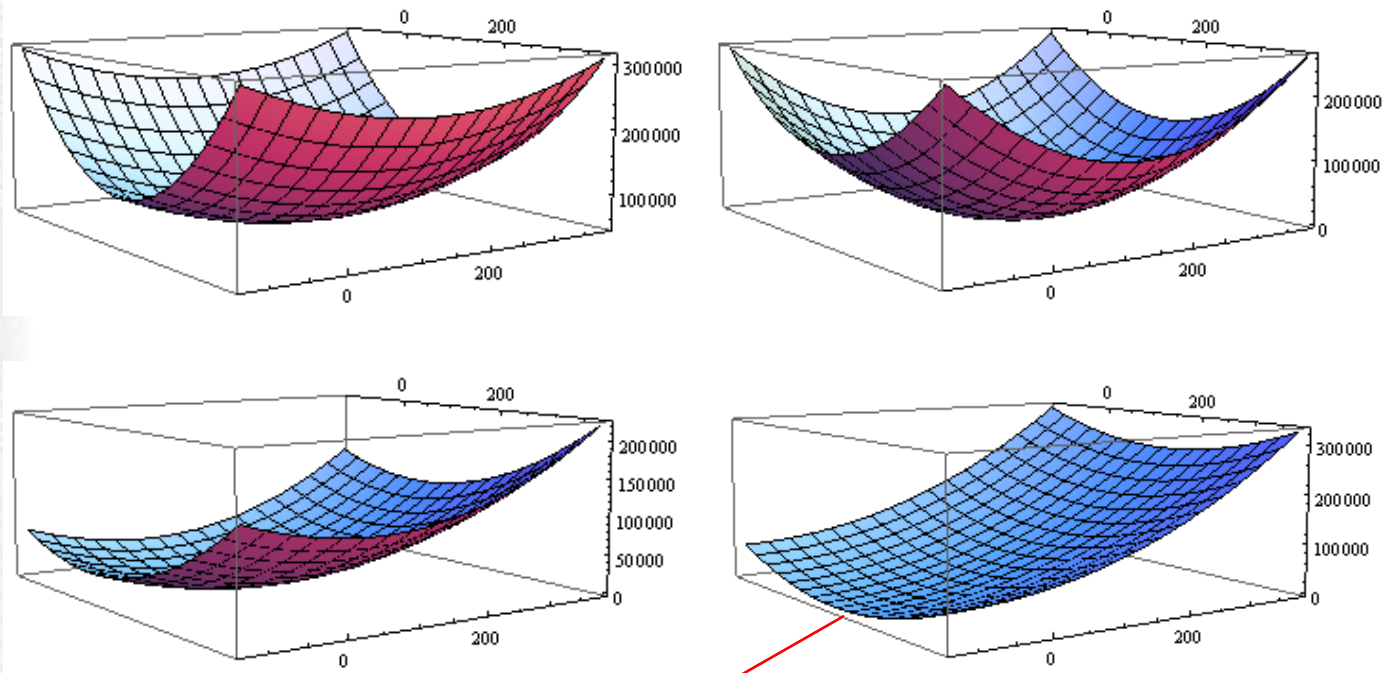
q23

- Plot of the distance function.
- It's obvious this has a well defined minimum and hesse matrice is automatically positiv definit.
- Mathematically distance must go to infinity for infinite color inputs in any way, so we can restrict to a compact subset.
- There the continous distance function must have a minimum.
- Since the Jacobi equation has one well defined solution, this must already be the minimum.

qon; 26.07.2009

Algorithm – Some Maths

Distance Calculation, some more Plots



Problem: Minimum not in (0, 255) x (0, 255)

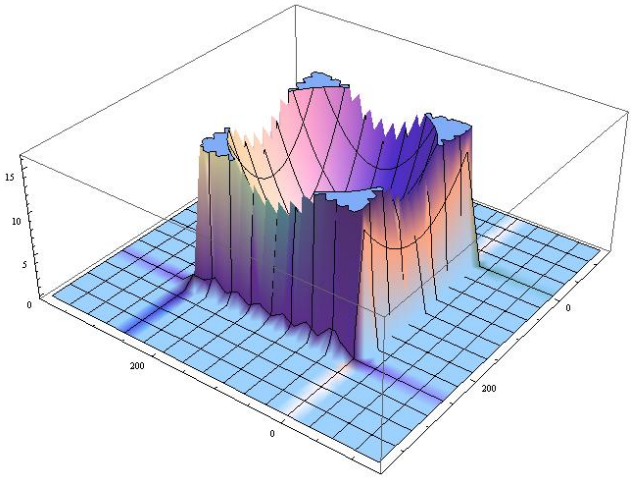
q24

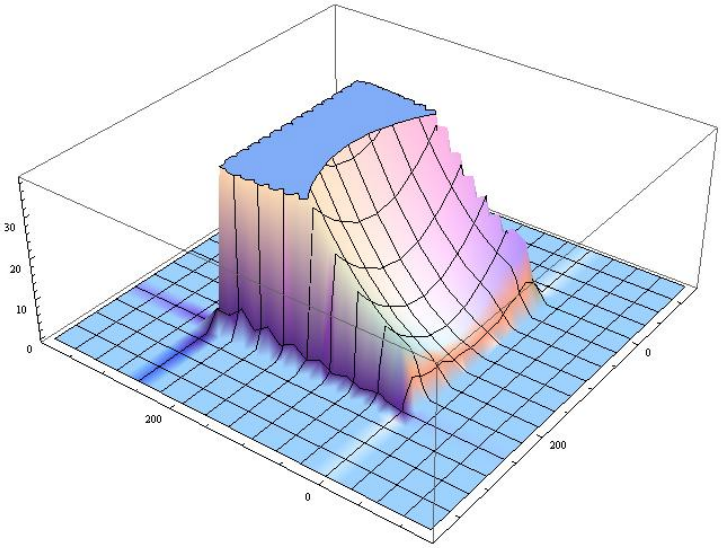
- Some more plots of distance functions.
- Upper left: stretched.
- Upper right: rotated.
- Lower left: shifted.
- Lower right: shifted even further.
- Lower right will produce an error.
- Distance function always posesses a minimum, but this is not necessarily in the range of 0 to 255 for fore- and background color.
- In this case the minimum is out of the (0, 255) x (0, 255) box.

qon; 26.07.2009

Algorithm – Some Maths

Distance Calculation





Minimum at (128, 128)

Minimum at (128, -32)

Solution: Search for minima at the boundaries:

$$f = 0,\ f = 255,\ g = 0,\ g = 255$$

and take the least one!

q25

- Some more plots capped at the 0 and 255 borders.
- In the problematic case (Minimization with boundary conditions) if the minimum is not within the boundaries it must lie at the boundary itself.
- So do the same differential calculation on the borders.

qon; 26.07.2009

Algorithm – Final differential algorithm


```
for all fields do
{
        calculate_v_vector();
        for all characters do
        {
                calculate_v'_constants();
                for red, green, blue do
                {
                        calculate_distance();
                        if (distance > 255 || distance < 0) do
                        {
                                calculate_boundary_distances();
                        }
                }
                if (distance < best_distance)
                {
                        SetBestCharacter();
                }
        }
}
```

q26 - Sample algorithm.

qon; 26.07.2009

Algorithm – First run

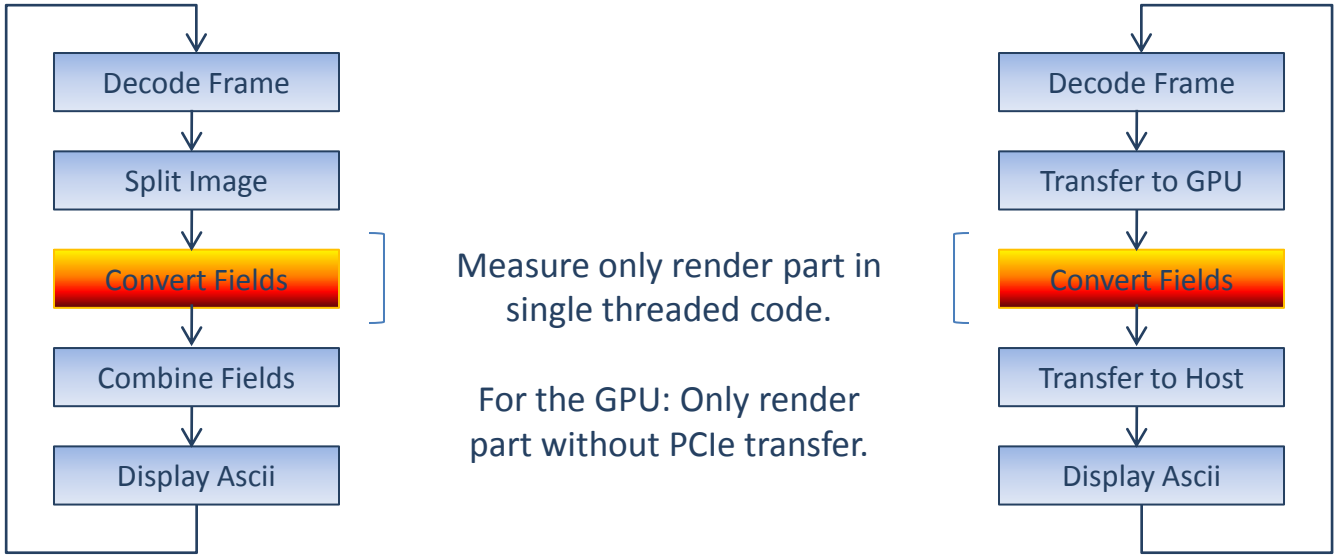
Differential ASCII Art renderer:

q27 - Output of differential ASCII renderer as described here.
qon; 26.07.2009

Benchmarking the algorithm

How to benchmark the algorithm itself:

Threaded code unfair for CPU



Measure only render part in single threaded code.

For the GPU: Only render part without PCIe transfer.

q28

- Benchmarking the well threaded code is unfair for the cpu since it still has other things to do as compared to the GPU.
- So the benchmark should measure the pure rendering time (multithreaded renderer of course) while no other threads are runing.

qon; 26.07.2009

Benchmarking the algorithm

How to benchmark the algorithm itself:

Skipping frames makes the renderer render different frames.
Different frames result in different boundary conditions and might affect speed.
→ Render a video with a constant frame.

Why not rendering a single frame?
For cache reasons result would be inaccurate.

→ Render constant frame and take average over constant time.
(here: 30 seconds, 1600 * 1200 pixel image)

Folie 29

q29 - One has to make sure to benchmark the same frames because the boundary condition depend on the frame content and therefore the computation time differs.

qon; 26.07.2009

Benchmarking the overall performance

Overall performance difficult to measure

Idea: - Use the fully threaded version with GPU.
- Test how many FPS are possible at max by changing video FPS rate.

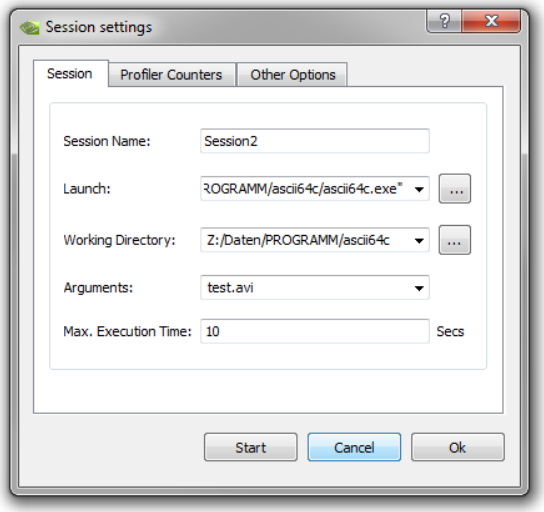
q30

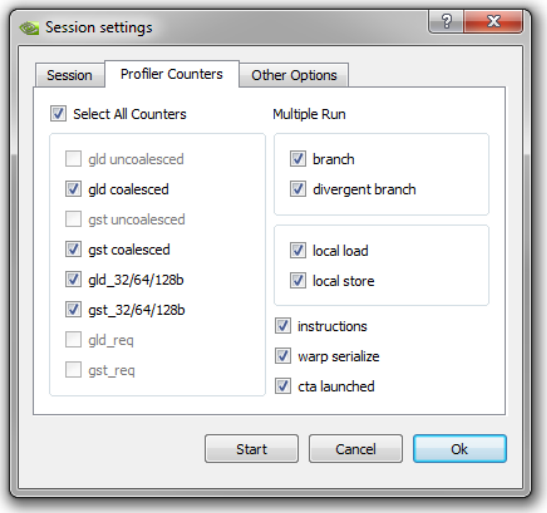
- Overall performance more difficult to measure. Here we definitely want the threaded version.
- Solution here: Check how many FPS the renderer can process in realtime at max.

qon; 26.07.2009

Benchmarking using the Nvidia CUDA Profiler

Easy to set up:





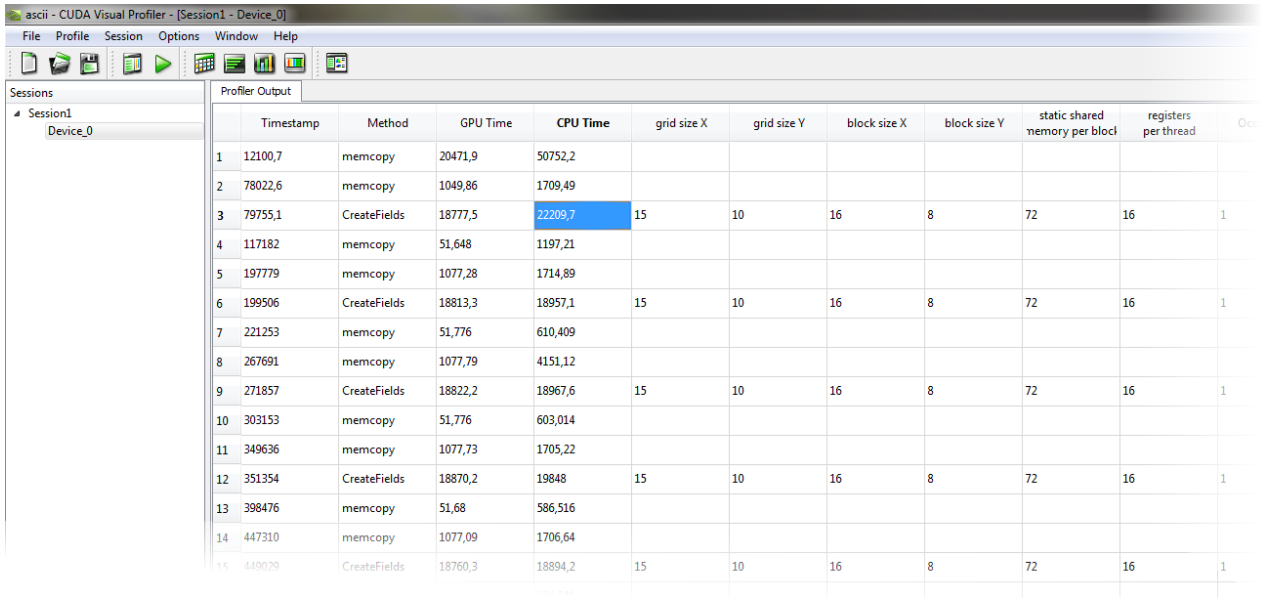
q31

- NVIDIA CUDA Profiler.
- Usefull and easy to set up profiling tool.
- Needs only binary and arguments as input.
- And some settings what to profile.
- Evtl. needs multiple runs to profile everything.
- Can profile things as memory access, divergent branches, instruction throughput, etc.

qon; 26.07.2009

Benchmarking using the Nvidia CUDA Profiler

Overall statistics (Execution time, size, etc.)



	Timestamp	Method	GPU Time	CPU Time	grid size X	grid size Y	block size X	block size Y	static shared memory per bloc	registers per thread	Oc
1	12100,7	memcopy	20471,9	50752,2							
2	78022,6	memcopy	1049,86	1709,49							
3	79755,1	CreateFields	18777,5	22209,7	15	10	16	8	72	16	1
4	117182	memcopy	51,648	1197,21							
5	197779	memcopy	1077,28	1714,89							
6	199506	CreateFields	18813,3	18957,1	15	10	16	8	72	16	1
7	221253	memcopy	51,776	610,409							
8	267691	memcopy	1077,79	4151,12							
9	271857	CreateFields	18822,2	18967,6	15	10	16	8	72	16	1
10	303153	memcopy	51,776	603,014							
11	349636	memcopy	1077,73	1705,22							
12	351354	CreateFields	18870,2	19848	15	10	16	8	72	16	1
13	398476	memcopy	51,68	586,516							
14	447310	memcopy	1077,09	1706,64							
15	449029	CreateFields	18760,3	18894,2	15	10	16	8	72	16	1

q32 - Output is a list of all cuda kernel calls together with kernel statistics.

qon; 26.07.2009

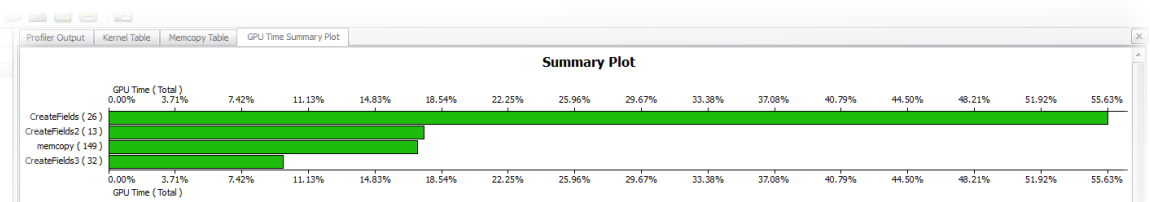
Benchmarking using the Nvidia CUDA Profiler

Kernel overview

Profiler Output | Kernel Table

	Method	#Calls	grid size X	grid size Y	block size X	block size Y	static shared memory per block	registers per thread
1	CreateFields	26	15	10	16	8	72	16
2	CreateFields2	13	15	10	16	8	1136	50
3	CreateFields3	32	15	10	16	8	76	33

Kernel statistics



Which kernel is the most time critical?

Profiler Output | Summary Table | Kernel Table | Memcopy Table | GPU Time Height Plot | GPU Time Summary Plot | GPU Time Width Plot

	Method	#Calls	GPU usec	%GPU time	mem read throug (GB/s)	glob mem write throughput (GB/s)	glob mem overall throughput (GB/s)	instruction throughput
1	CreateFields2	157	1,96208e+06	90,8	3,60458	0,0942281	3,69881	0,503151
2	memcopy	316	198664	9,19				

How well is the device utilized?

q33

- Then every kernel can be analysed further be averaging all calls of this particular kernel.
- Second plot gives a good idea which kernels are the hot spots and need improvement.
- Last plots give 2 numbers: instruction throughput and memory bandwidth, giving a good idea how well the device is saturated.
- Maximum instruction throughput is more than 1 because of "Dual Issue" capability of NVIDIA cards.

qon; 26.07.2009

Benchmarking platform used

Processor:	Intel Nehalem 3,8 GHz	
Mainboard:	Asus P6T6 WS Revolution	
Memory:	12 GB DD3-1600	
Harddisk:	4 * WD Raptor 74GB Raid 0	
GPU1:	Nvidia GeForce 285,	1 GB GDDR3
GPU2:	Nvidia GeForce 8400 GS,	524 MB RAM

0 : GeForce GTX 285 - Properties

Property	Value
Compute capability	1.3
Multiprocessor count	30
Maximum threads per block	512
Maximum x-dimension of a block	512
Maximum y-dimension of a block	512
Maximum z-dimension of a block	64
Maximum x-dimension of a grid	65535
Maximum y-dimension of a grid	65535
Warp size	32
Maximum registers per block	16384
Maximum shared memory per block	16384
Total constant memory	65536
Maximum pitch allowed by memcopy	262144
Clock frequency (in kilohertz)	1476000
Texture alignment	256
GPU overlap	1

OK

1 : GeForce 8400 GS - Properties

Property	Value
Compute capability	1.1
Multiprocessor count	2
Maximum threads per block	512
Maximum x-dimension of a block	512
Maximum y-dimension of a block	512
Maximum z-dimension of a block	64
Maximum x-dimension of a grid	65535
Maximum y-dimension of a grid	65535
Warp size	32
Maximum registers per block	8192
Maximum shared memory per block	16384
Total constant memory	65536
Maximum pitch allowed by memcopy	262144
Clock frequency (in kilohertz)	918000
Texture alignment	256
GPU overlap	1

OK

q34 - Benchmark system used, Two NVIDIA cards for comparison, one very high end, one low end.

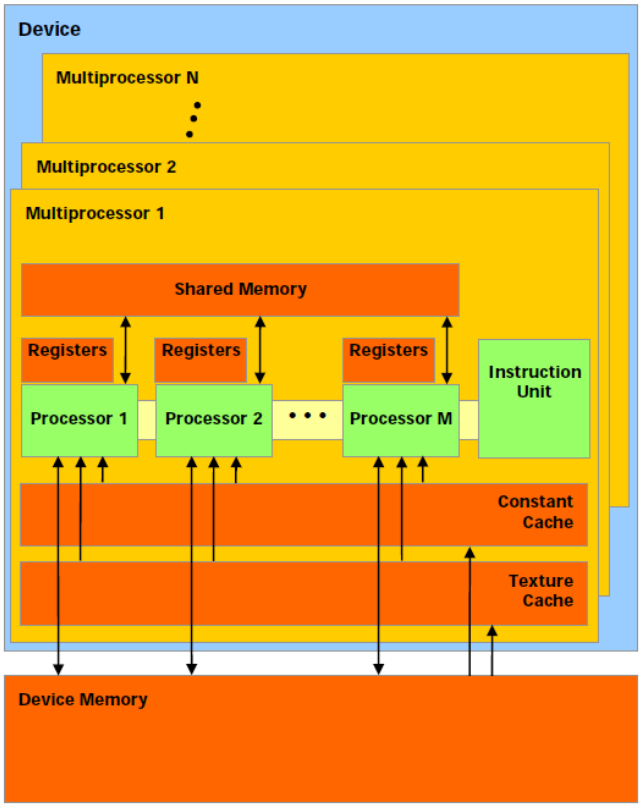
qon; 26.07.2009

Optimizations – Memory Cache

3 Types of cached memory:

- Constant memory } read only
- Texture memory } read only
- Shared memory } read / write

→ Use as much of them as possible



Memory required by algorithm:

1. Character constants
2. Source image
3. Output array
4. Parameters
5. Local variables

q35 qon; 26.07.2009

Optimizations – Constant memory

1. Character constants:

As name implies: put into constant memory!

But: Size of constant memory: 64k

- Exchange constants depending on algorithm to make them fit.
- Use small variables such as short or even char where possible.

q36

- Size of constant memory is restricted to 64k.
- For different kernels requiring different constants and for long time kernels, constants should be exchanged depending on kernel.
- Also reduce variable size where possible.

qon; 26.07.2009

Optimizations – Constant memory benchmark

Total converion time in nanoseconds

Renderer	Time	Speedup
GPU (global memory)	16.911.568	1,000
GPU (constant memory)	15.882.250	1,064

q37 qon; 26.07.2009

Optimizations – Texture memory

2. Source image:

Source image is also constant, but:
- Full HD image: 1920 * 1080 pixels give 7,91 MB.
- Constant memory size is limited to 64 KB.

Alternative: Texture memory, but:
- Texture memory can only be read as texture.

q38 - The input frame will not fit in constant memory so handle access using texture cache.

qon; 26.07.2009

Optimizations – Texture memory

Accessing texture memory:

Requires „CUDA-Arrays“.

```
texture<unsigned int, 2, cudaReadModeElementType> texRef;
cudaArray* lpCudaAsciiSourceArray;

void transferSourceToGPU_texture()
{
    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<unsigned int>();
    cudaMallocArray(&lpCudaAsciiSourceArray, &channelDesc, nWidth, nHeight);
    cudaBindTextureToArray(texRef, lpCudaAsciiSourceArray);
    cudaMemcpy2DToArray(lpCudaAsciiSourceArray, 0, 0, lpAsciiSource, dwAsciiSourcePitch, nWidth,
        nHeight, cudaMemcpyHostToDevice);
}

#ifdef CUDA_USE_TEXTURE_MEMORY
#define AVAL(i, j, k, l, o) tex2Da(texRef, (i) * TEXT_WIDTH + (k), (j) * TEXT_HEIGHT + (l), o)
#else
#define AVAL(i, j, k, l, o) lpAsciiSource[CALC_POS(i, j, k, l, o)]
#endif
```

q39

- Texture cache cannot access GPU memory directly but only textures.
- Allocate a CUDA-Array (special data structure) in memory and put data there.
- Then create Texture references to the CUDA-Array.
- Texture references are logically independant from the memory, even multiple references can access different or even the same part of the array.
- Textures can also do bilinear filtering on the array data for free.
- One then hast user defined access function (here tex2Da) to access data from texture memory.

qon; 26.07.2009

Optimizations – Texture memory benchmark

Total converion time in nanoseconds

(texture code is still beta, consistency is not assured)

Renderer	Time	Speedup
GPU (global memory)	15.882.250	1,000
GPU (texture memory)	15.225.080	1,043

Folie 40

q40 qon; 26.07.2009

Optimizations – Memory

3. Output array:

- No cache needed.
- Constant and texture memory unavailable for write access.
- Shared memory too small.

→ Just stay with global memory, try not to write too much data.

q41 qon; 26.07.2009

Optimizations – Memory

4. Parameters:

- Parameters are stored in registers or local memory.
- Local memory is slow so stay with registers.
- Make parameters small to fit there.

Example CUDA Compiler output:

ptxas info : Used 63 registers, 1616+1612 bytes lmem, 1148+124 bytes smem, 60416 bytes cmem[0], 184 bytes cmem[1]

q42

- CUDA compiler gives info of how many registers used.
- Registers are restricted for the multiprocessor.
- When running 16x16 block 64 registers per thread can be used.
- When more registers are needed block size must be decreased.
- Alternatively data can be stored in shared memory to save registers.
- Anyway try not to use too many different parameters in function calls, many can be combined, etc 32 flags can be stored in one integer.

qon; 26.07.2009

Optimizations – Memory

4. Parameters:

Alternative: Move parameters into shared memory.

Automatically done since CUDA 2.1.

ptxas info : Used 63 registers, 1616+1612 bytes lmem, 1148+124 bytes smem, 60416 bytes cmem[0], 184 bytes cmem[1]

(Example code without usage of shared memory)

q43 - In compilation here parameters were explicitely put to shared memory, but still 63 registers used identically to the run before.

qon; 26.07.2009

Optimizations – Registers / Local memory

5. Local variables:

- Fast read/write access only in registers / shared memory.
- Shared memory can be split among threads for more „virtual registers“.

  (For coalesced (see later) access shared memory is as fast as registers)
- Try to use as less memory as possible.

```
for (o = 0;o < 6;o++)
{
   for (p = 0;p < 4;p++)
   {
      AV[p] += aarray[o] * barray[o].bgr[p];
      CV[p] += carray[o] * barray[o].bgr[p];
   }
}
for (p = 0;p < 3;p++)
{
   DIFFERENTIALCOMPAREC(p);
}
```

```
for (p = 0;p < 3;p++)
{
   for (o = 0;o < 6;o++)
   {
      AV[p] += aarray[o] * barray[o].bgr[p];
      CV[p] += carray[o] * barray[o].bgr[p];
   }
   DIFFERENTIALCOMPAREC(p);
}
```

Can be vectorized!

Needs less local memory

q44

- Left side is better suited for CPU since inner loop can be auto vectorized.
- The AV and CV arrays are precalculated and need to be stored in local memory for the DIFFERENTIALCOMPARE script lateron.
- For CUDA create only the arrays needes.

qon; 26.07.2009

Optimizations – Register benchmark

Total converion time in nanoseconds

(texture code is still beta, consistency is not assured)

Renderer	Time	Speedup
CPU („Local memory“)	98.718.904	1,000
CPU („Registers“)	109.476.908	0,902
GPU (Local memory)	50.254.058	1,000
GPU (Registers)	15.882.250	3,164

q45 One sees that "Register" code, right side of last slide, is tremendously faster on GPU, but a bit slower on CPU because code is no longer vectorized.

qon; 26.07.2009

Optimizations – Float data types

Float Data Types:

- 240 single precision ALUs.
- 30 double precision ALUs. (200 Series only, otherwise emulation)

(single and double share hardware)

→ Single 8 times faster than double, so stay with single!!!

q46 - Double precission might be improved with next generation.
qon; 26.07.2009

Optimizations – Integer data types

Integer Data Types:

- CUDA has no ALUs to handle integers!
→ Slow integer emulation.

- 24 Bit integer can be packed into 32 bit single float!
→ Fast 24 bit integer calculation.

Question: Why integer at all?
Required at least for adress calculations.
→ Keep address range in 24 bit!

q47

- Integer is always required for adress calculation.
- 24 bit integers can be packed into 32 bit floats, so 24 bit integer multiplication (as long as result is also 24 bit) can be done in one clock cycle.
- Better use a typedef right now for 24 bit integers since NVIDIA announced to support 32 bit integer calculation, that might then be even faster then 24 bit.

qon; 26.07.2009

Optimizations – 64 bit integer types

Integer Data Types:

No 64 bit integers, inaccurate emulation!

CPU

GPU

q48

- CUDA has no support for 64 bit integer calculation, resulting for example in slight errors in the number reduction algorithm.
- But be carefull, when running in device emulation mode the CPU supports 64 bit, so then emulation will be correct resulting in simulation mismatch.

qon; 26.07.2009

Optimizations – 64 bit integer types

Integer Data Types:

No 64 bit integers, inaccurate emulation!

But be carefull, device emulation mode runs on CPU.
Emulated result is correct and differs from result on GPU.

q49 qon; 26.07.2009

Optimizations – Data type benchmark

Integer / Float comparison

Renderer	Time	Speedup
GPU (Integer)	23,708	1,000
GPU (Float)	20,190	1,174

q50 Changed all integer calculation below 24 bit into floats.
qon; 26.07.2009

Optimizations – PCIe Bandwidth

GPU Characteristics:

Performance	: 1000+ GFlops
Memory bandwidth	: 100+ GB/sec
PCIe bandwidth	: 6 GB/sec

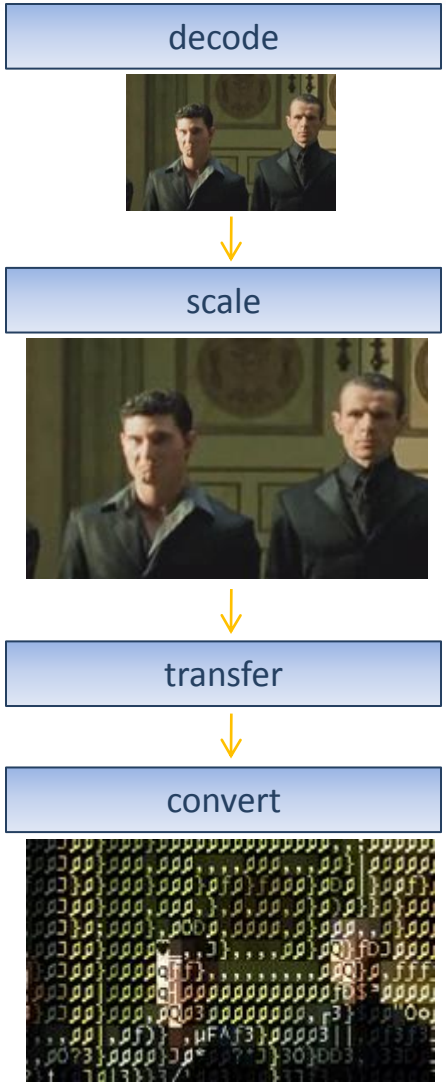
→ Conserve PCIe bandwidth

q51 - PCIe bandwith is the obvious bottleneck.
qon; 26.07.2009

Optimizations – PCIe Bandwidth

Simplest algorithm:

- Scaling done by CPU
- Bigger image transfered through PCIe



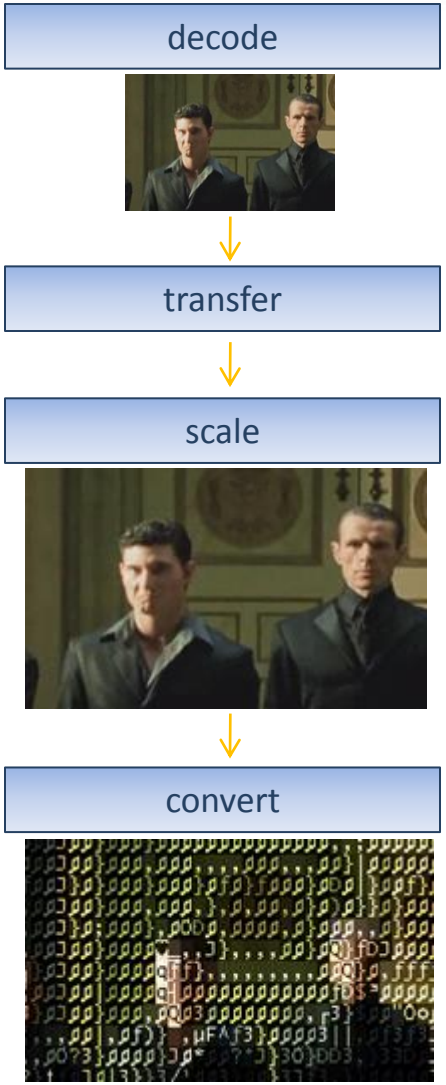
q52 qon; 26.07.2009

Optimizations – PCIe Bandwidth

Better algorithm:

- Scaling by GPU (in hardware)
- Transfer unscaled smaller image through PCIe

(CUDA can scale using textures)



q53

- When scaling the image after the transfer less bandwidth is consumed.
- Even further CUDA can scale for free using the texture cache.

qon; 26.07.2009

Optimizations – PCIe Bandwidth

Comparison:

CPU Scaling:
4130 Microseconds passed during Source Scaling
1069 Microseconds passed during CUDA Transfer to Device
19177 Microseconds passed during CUDA conversion
908 Microseconds passed during CUDA Transfer to Host
9908 Microseconds passed during Display
12954 Microseconds passed during Resize and Overlay Operations

GPU Scaling:
291 Microseconds passed during Source Scaling
677 Microseconds passed during CUDA Transfer to Device
18741 Microseconds passed during CUDA conversion
994 Microseconds passed during CUDA Transfer to Host
9913 Microseconds passed during Display
12819 Microseconds passed during Resize and Overlay Operations

(TEXTURE CODE IS STILL BETA!)

q54

- Speed gain in transfer is not so big since frame is small anyway.
- Algorithm itself got a bit faster due to better use of texture cache.
- Source Scaler got much faster though (In the lower example the source scaler just copies the buffer).

qon; 26.07.2009

Optimizations – PCIe Bandwidth

Comparison:

Conversion:

Renderer	Time	Speedup
GPU (CPU Scaling)	19.177	1,000
GPU (GPU Scaling)	18.741	1,023

Transfer:

Renderer	Time	Speedup
GPU (CPU Scaling)	1.069	1,000
GPU (GPU Scaling)	677	1,579

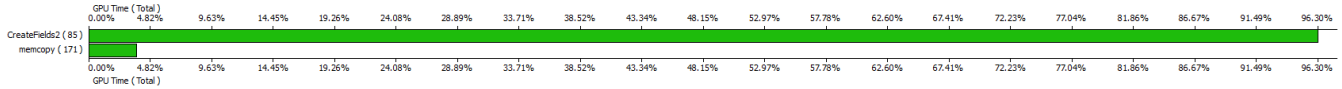
Folie 55

q55 qon; 26.07.2009

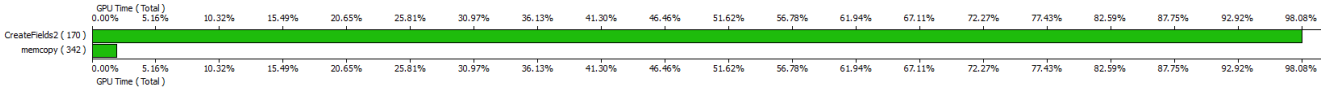
Optimizations – PCIe Bandwidth

Comparison:

CPU Scaler:



GPU Scaler:



q56 - Amount of time required for memcopy is cut down to almost one half with GPU scaler.
qon; 26.07.2009

Optimizations – GPU Usage

GPU Usage statistics:

Memory throughput

Instruction throughput

Profiler Output | Summary Table | GPU Time Summary Plot | GPU Time Height Plot

	Method	#Calls	GPU usec	%GPU time	mem read throug (GB/s)	mem write throug (GB/s)	glob mem overall throughput (GB/s)	instruction throughput
1	CreateFields2	157	1,96208e+06	90,8	3,60459	0,0942282	3,69881	0,503151
2	memcopy	316	198664	9,19				

(Max instruction throughput = 2.0 because of Dual Issue, though 2.0 will never be reached)

→ Renderer seems to be GPU- rather than Memory bound.

(utilizes ¼ peak performance)

→ Maximize instruction throughput.

q57

- Instruction throughput is 0,5 so we use the device by almost one half considering computational power.
- (Ignoring dual issue here)
- 3,6 gb/s memory transfer is almost nothing compared to 100 gb/s the device is capable of.
- Renderer seems to be rather GPU bound. This might be a reason our memory optimization were not so successfull.

qon; 26.07.2009

Optimizations – Branching

Branching:

- Block split in warps of 32 threads.
- One instruction decoder per warp.

- All threads in warp musst execute the same code.

→ Avoid branches !

Folie 58

q58 qon; 26.07.2009

Optimizations – Branching

Branching example:

(Blocksize: 32, Only binary integer values used)

Divergent branch

```
int  a[256], b[256], c[256], i;

for (i = threadIdx.x;i < 256;i += BLOCKSIZE)
{
    if (b[i] != c[i])  a[i]++;
}
```

No branch

```
int  a[256], b[256], c[256], i;

for (i = threadIdx.x;i < 256;i += BLOCKSIZE)
{
    a[i] += b[i] ^ c[i];
}
```

Folie 59

q59 qon; 26.07.2009

Optimizations – Memory Coalescing

Memory coalescing:

(Parallel memory access pattern)

Two types:

1. Global memory
2. Shared memory

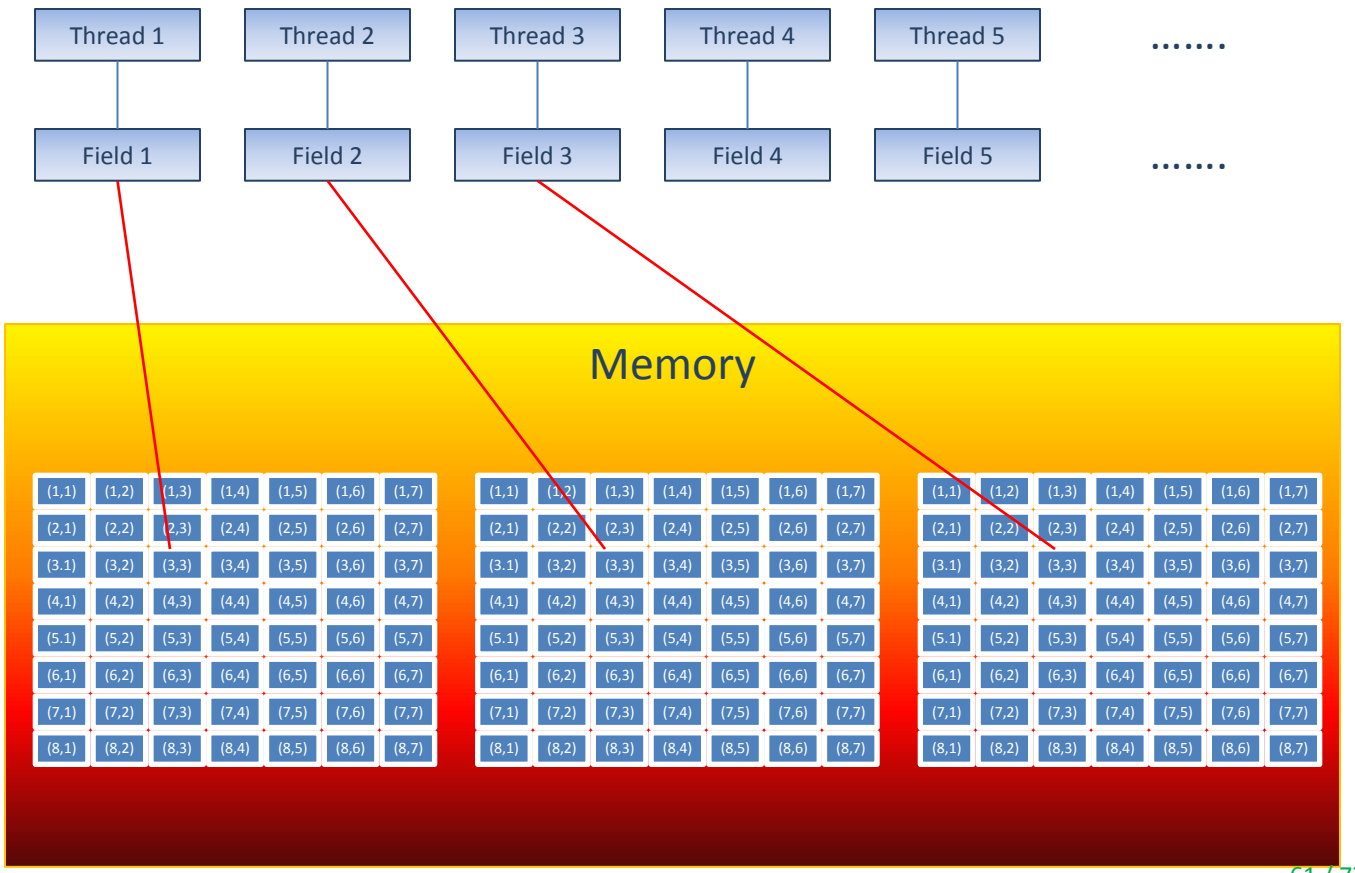
Folie 60

q60 qon; 26.07.2009

Optimizations – Memory Coalescing

Global memory coalescing

Renderer access pattern:



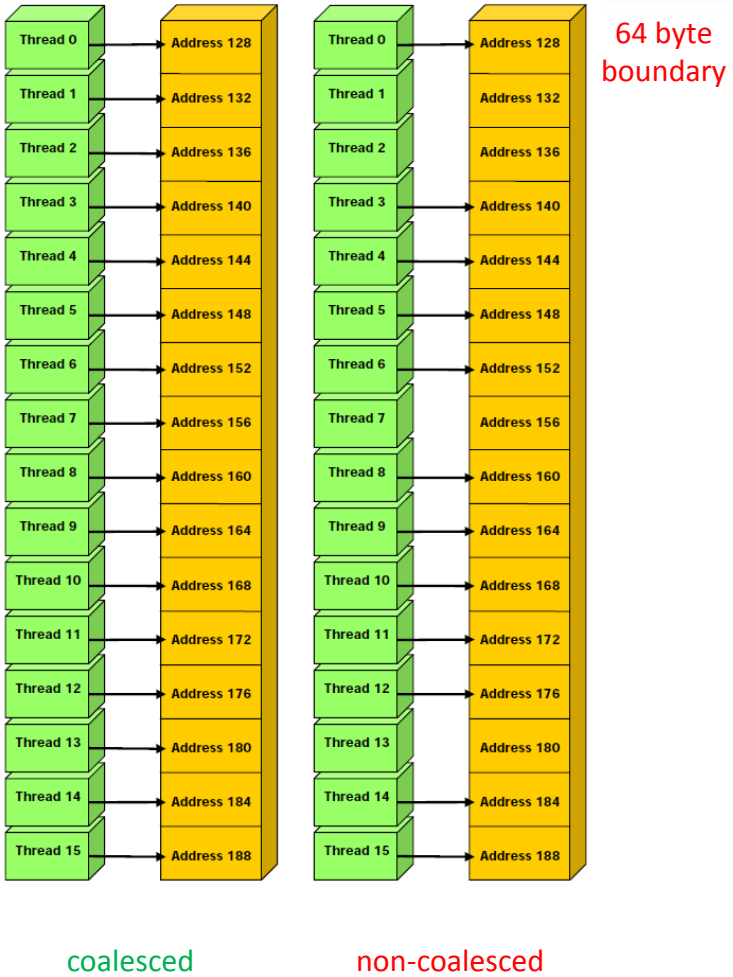
q61 Threads in one warp work on a field. They access memory far away from each other.
qon; 26.07.2009

Optimizations – Memory Coalescing

Coalescing rules:

(Cuda device before GT200)

For non coalesced access every thread issues a seperate memory instruction!!



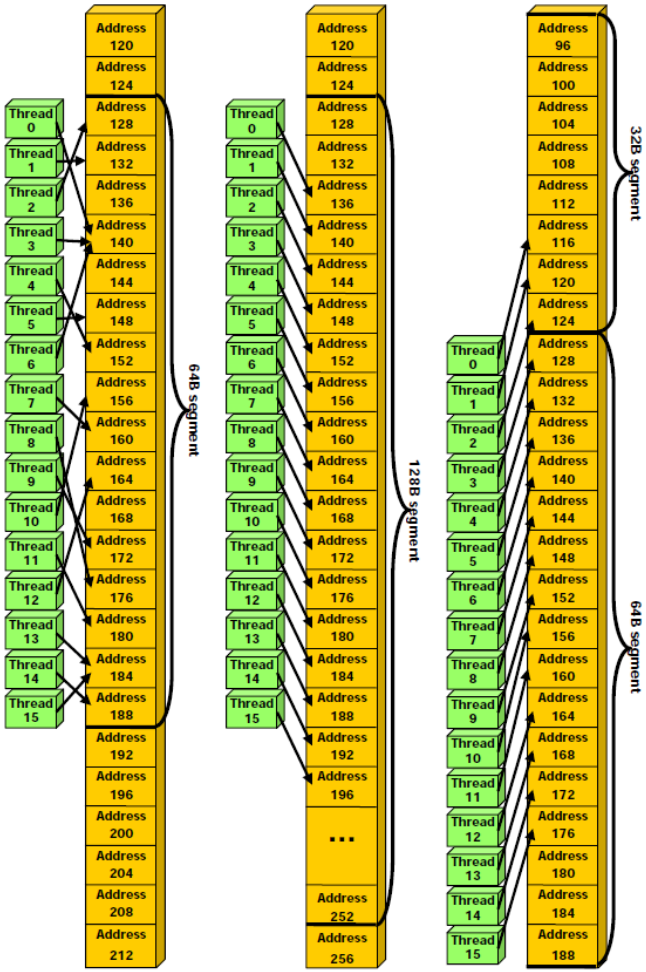
q62 CUDA is optimized for threads in warp accessing consecutive memory aligned to 64 byte boundaries.
qon; 26.07.2009

Optimizations – Memory Coalescing

Coalescing rules:

(GT200 Device)

- Must not cross 128 byte boundary
- No fixed order required

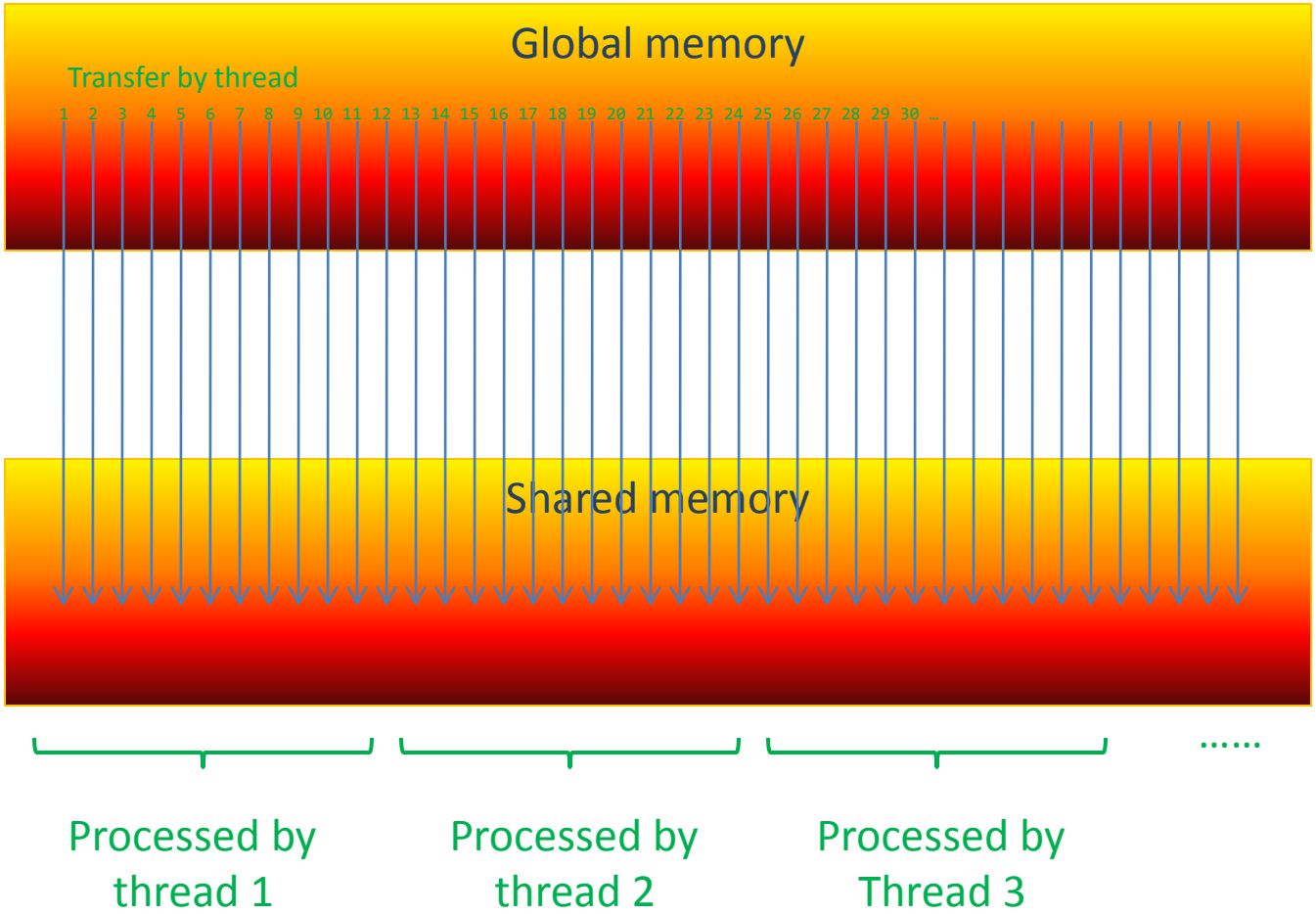


q63 - Since GT200 devices alignment rules are relased but still consecutive access is required, or at least access to one memory segment.

qon; 26.07.2009

Optimizations – Memory Coalescing

Idea: Shared memory cache



q64

- Common solution for this is a shared memory cache.
- At first all threads together transfer data from global to shared memory regarding the coalescing rules.
- Later they process data in shared memory.

qon; 26.07.2009

Optimizations – Memory Coalescing

Shared memory cache, code example

```
#define CALC_POSxy(x, y) ((x) * sizeof(int) + (y) * CUDA_PARAM(dwAsciiSourcePitch))
#define CALC_POS(i, j, k, l, o) (CALC_POSxy((i) * TEXT_WIDTH + (k), (j) * TEXT_HEIGHT + (l)) + (o))

#define AVALshared(i, j, k, l, o) cudaSharedCache[threadIdx.y][l][threadIdx.x * TEXT_WIDTH + k].bgr[o]
#define AVALishared(i, j, k, l) cudaSharedCache[threadIdx.y][l][threadIdx.x * TEXT_WIDTH + k].i

#define AVALreal(i, j, k, l, o) CUDA_PARAM(lpAsciiSource)[CALC_POS(i, j, k, l, o)]
#define AVALireal(i, j, k, l) *((unsigned int*) &CUDA_PARAM(lpAsciiSource)[CALC_POS(i, j, k, l, 0)])


for (j = 0;j < TEXT_HEIGHT;j++)
{
    for (i = threadIdx.x; i + blockIdx.x * blockDim.x * TEXT_WIDTH < nCols * TEXT_WIDTH;i += blockDim.x)
    {
        cudaSharedCache[threadIdx.y][j][i].i = AVALireal(blockIdx.x * CUDA_BLOCKSIZE_X, blockIdx.y *
            blockDim.y + threadIdx.y, i, j);
    }
}
__syncthreads();
```

q65 - Code example for shared cache that has to be put before the actual convert function. __syncthreads() will make sure all threads are finished.

qon; 26.07.2009

Optimizations – Memory Coalescing

Shared memory coalescing:

16 banks that can be accessed in parallel.
Parallel access for 16 threads achieved if stride is no multiple of 2!

q66

- Shared memory consists of 16 banks that can be accessed in parallel.
- So 16 threads can access in parallel if the access different banks.
- This is automatically achieved if the stride size of their memory access position is no multiple of 2.

qon; 26.07.2009

Optimizations – Memory Coalescing

Device saturation – Right block size

Blocksize	Time
8 x 4	22,5
8 x 8	18,5
12 x 12	22,7

- Highest block size not neccessarily the best (remember memory coalescing)
- 16 x 16 threads each 64 registers results in 16384 registers (GT200 limit)

q67

- Benchmark using different blocksizes.
- Obviously 8x8 seems a good idea, and leaves space for more registers as compared to 64 register when running 16 x 16.

qon; 26.07.2009

Optimizations – CPU / GPU Optimization

Primary objective: Keep GPU running
It‘s 1 Tflops as compared to 60 Gflops!!!

Time consumption in stages

4130 Microseconds passed during Source Scaling
1069 Microseconds passed during CUDA Transfer to Device
19177 Microseconds passed during CUDA conversion
908 Microseconds passed during CUDA Transfer to Host
9908 Microseconds passed during Display
12954 Microseconds passed during Resize and Overlay Operations

19177 CUDA conversion time
28969 Host preparation time

In sequential programm CUDA runs only for 39,8% of the runtime!!!

q68

- One step that involves the GPU is CUDA conversion, requiring 19 ms as compared to 29 ms for the CPU time.
- So the multithreading to keep the GPU running seems more urgent than most other optimizations. (In the end they all need to be applied together)

qon; 26.07.2009

Optimizations – CPU / GPU Optimization

Secondary objective: Offload some work to CPU

Nehalem is not so slow anyway….

Renderer	Time	Speedup
CPU	588.524	1,000
GPU	319.199	1,844
GPU + CPU	274.897	2,141

q69

- If the CPU has some resources available while the GPU is saturated one can offload some work to it.
- For differential renderer this is not the case so this is a different algorith involving longer conversion times.
- Obviously distributing work among GPU and CPU can be effective. But only for long calculation times since overhead increases.

qon; 26.07.2009

Optimizations – CPU / GPU Optimization

Realisation:

Simplest Idea: OpenMP

```
OMP parallel for ……
for (i = 0; i < dwFields;i++)
{
        if (ThreadID) == 0 ConvertCUDA();
        else ConvertCPU();
}
```

Folie 70

q70 qon; 26.07.2009

Optimizations – CPU / GPU Optimization

Realisation:

OpenMP does not work.

All CUDA data reside in thread context.
OpenMP might change threads.

Working Solution:
CreateThread(CPUWorkerThread);
ConvertCUDA();
Wait (CPUWorkerThread);

q71

- OpenMP does not work to distribute work between GPU and CPU, since all CUDA internal data are thread based, openmp will change threads though.
- So to work correctly one has to create threads the good old way.

qon; 26.07.2009

Optimizations – Multi GPU

If we run parallel anyway why not use multiple GPU?

Problem: CUDA requires ThreadContexts.

Solution:

- Start one thread for every CUDA device.
- Control CUDA devices using thread communication.

q72 - This problem gets even bigger when using multi GPU.
qon; 26.07.2009

Optimizations – Multi GPU

CUDA GPU Speed:

Problem: CUDA requires ThreadContexts.

GPU	Time	Speedup
Geforce 8400GS	333	1
Geforce 285	19	17

→ Load Balancing required!!

fps: 17.75 (34.960.959 / 36.580.912)
(Direct Draw Display / Differential ASCII Renderer (GPU/CPU) (Color))
[228 Cols / 75 Lines] (Size: 1276 x 720) (228 x 75 Letters)
CUDA Device -1 Time 15.068.721 Lines 4 Speed 265.45
CUDA Device 0 Time 18.412.301 Lines 71 Speed 3856.12
CUDA Device 1 Time 465.910 Lines 0 Speed 0.00

q73

- Comparison of computation time for differential algorithm again between high end and low end GPU.
- When running with multiple different GPUs load balancing is required.
- In the example here:
- Nehalem / Device -1 calculates 4 lines of fields.
- Geforce 285 / Device 0 calculates the rest
- Geforce 8400 / Device 1 does not calculate anything at all (It takes more time to calculate a single line + overhead than the 285 needs for 71).

qon; 26.07.2009

Optimizations – Multi GPU

CUDA GPU Speed:

Problem: CUDA Devices have different capabilities.

Solution: Include multiple cuda codes for differen device generations.

Example:

nvcc –codegen arch=compute_13,code=sm_13 –codegen arch=compute_10,code=sm_10

```
1>tmpxft_00001010_00000000-6_cuda.compute_13.cudafe1.gpu
1>tmpxft_00001010_00000000-10_cuda.compute_13.cudafe2.gpu
1>cuda.cu
1>tmpxft_00001010_00000000-3_cuda.compute_10.cudafe1.gpu
1>tmpxft_00001010_00000000-14_cuda.compute_10.cudafe2.gpu
1>ptxas info    : Compiling entry function '_ZN14namespace_cuda13CreateFields2EjjPhijP12LetterStructjP11FieldStructiiS0_S0_iiiiiiiii'
1>ptxas info    : Used 50 registers, 1616+1612 bytes lmem, 380+124 bytes smem, 60416 bytes cmem[0], 480 bytes cmem[1]
1>ptxas info    : Compiling entry function '_ZN14namespace_cuda13CreateFields3EjjPhiP11FieldStructijiP12NumberStructj'
1>ptxas info    : Used 33 registers, 336+0 bytes lmem, 76+72 bytes smem, 60416 bytes cmem[0], 40 bytes cmem[1]
1>ptxas info    : Compiling entry function '_ZN14namespace_cuda12CreateFieldsEjjjPhijP12LetterStructjP11FieldStruct'
1>ptxas info    : Used 16 registers, 344+336 bytes lmem, 72+64 bytes smem, 60416 bytes cmem[0], 44 bytes cmem[1]
1>ptxas info    : Compiling entry function '_ZN14namespace_cuda13CreateFields2EjjPhijP12LetterStructjP11FieldStructiiS0_S0_iiiiiiiii'
1>ptxas info    : Used 63 registers, 1616+1612 bytes lmem, 380+124 bytes smem, 60416 bytes cmem[0], 184 bytes cmem[1]
1>ptxas info    : Compiling entry function '_ZN14namespace_cuda13CreateFields3EjjPhiP11FieldStructijiP12NumberStructj'
1>ptxas info    : Used 33 registers, 336+0 bytes lmem, 76+72 bytes smem, 60416 bytes cmem[0], 40 bytes cmem[1]
1>ptxas info    : Compiling entry function '_ZN14namespace_cuda12CreateFieldsEjjjPhijP12LetterStructjP11FieldStruct'
1>ptxas info    : Used 16 registers, 344+336 bytes lmem, 72+64 bytes smem, 60416 bytes cmem[0], 44 bytes cmem[1]
```

q74

- When combining GPUs of different generations devices might have different capabilities.
- CUDA has options to produce two device code binaries and include both in a program.
- For different GPUs always the one suiting best is needed.
- In the output one can observe that the CreateFields2 function is once compiled with 50 registers and once with 63.

qon; 26.07.2009

Optimization Summary

Summary speed table:

Renderer	FPS	Speedup
Unoptimized	1.5	1
OpenMP	8	5.3
OpenMP + Threads	7-9	6
CUDA	19	12,6
CUDA + Threads	37	24,6

q75

- Overall speed up.
- This is not conversion time but max FPS the renderer was capable of processing.

qon; 26.07.2009

Sources

Ascii Art Renderer

SVN: https://qon.zapto.org/var/svn/ascii64c/ascii64c/

Binaries: https://ascii.jwdt.org/

aalib (Ascii Art library) http://aa-project.sourceforge.net/aalib/

libcaca http://caca.zoy.org/

mplayer (www.mplayerhq.hu)

CUDA 2.2 Reference Manual

Contact: drohr@jwdt.org

q76

- Binaries are 32 and 64 bit versions.
- 32 Bit version is some very old code.
- 64 Bit is just some beta code that is everything but stable (Version from 27.7.09).
- To get a stable 64 bit try to compile some older revision from SVN.
- To play Videos DirectShow codecs are needed.
- Since they are not easy to find and 64 bit xvid is not easy to compile a provide a binary there too.
- I included the openmp library so everything should run as it is (i hope).
- You might evtl. need the cuda libraries/sdk or at least the cuda driver.
- If the program starts enter name of video file (avi or mpeg) and it should play.
- Press h to get some help for all options!
- Have a lot of fun.
- If you experience problems feel free to contact me at drohr@jwdt.org.
- But as I said, current code is beta since I added some optimizations that messed something up.

qon; 27.07.2009

Thanks for attention!

q77 qon; 26.07.2009