

# GPU Programming in Computer Vision

Thomas Möllenhoff, Robert Maier,  
Lingni Ma, Caner Hazırbaş

Optimization

Technical University Munich, Computer Vision Group  
Winter Semester 2015/2016, March 15 – April 15

# Outline

- **Branch Divergence**
  - **Pitch Allocation for 2D Images**
  - **Host-Device Memory Transfer**
  - **Occupancy**
- 
- **See the Programming Guide for more details**

**BRANCH DIVERGENCE**

# Branch Divergence

- All 32 threads in a warp execute **the same** instruction
  - always, no matter what

```
__global__ void kernel (float *result, float *input)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    if (input[i]>0)
        result[i] = 1.f;
    else
        result[i] = 0.f;
}
```

What if different paths  
are taken within a warp?

# Branch Divergence: Serialization

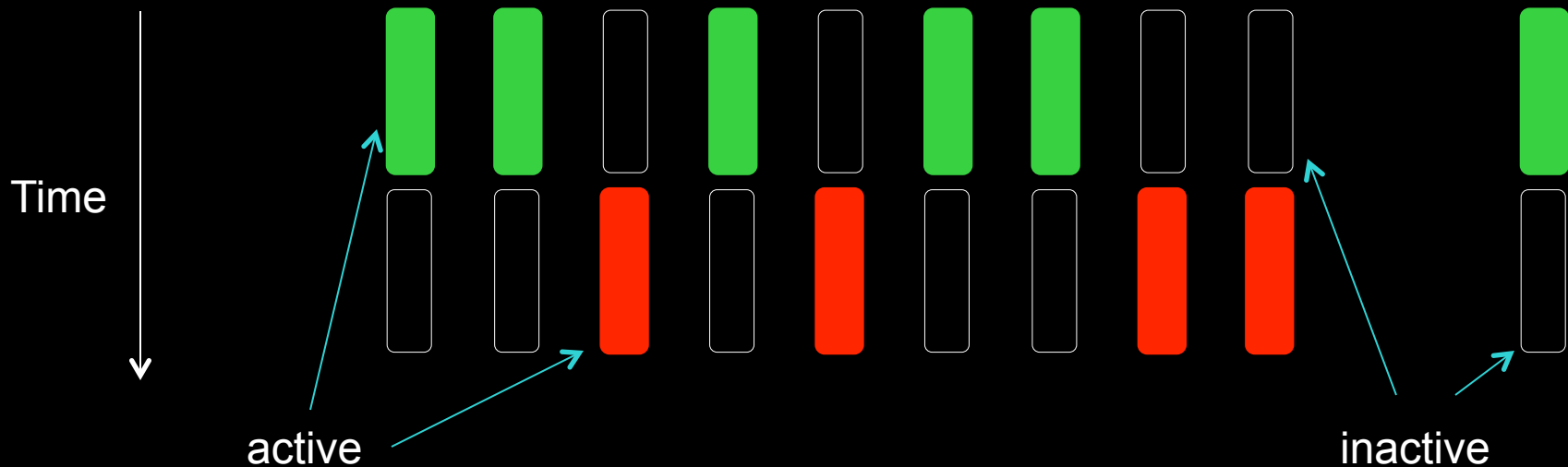
```
if (input[i]>0) result[i] = 1.f; else result[i] = 0.f;
```

- If threads diverge **within a warp** execution is **serialized**
  - all 32 threads must execute the same instruction
- **Each path is taken** by each of the 32 threads
- Threads which do not correspond to this path are marked as **inactive** during execution

# Branch Divergence: Serialization

```
if (input[i]>0) result[i] = 1.f; else result[i] = 0.f;
```

threadIdx.x:	0	1	2	3	4	5	6	7	8	...	31
input[i]:	7	23	-2	5	-1	66	24	-41	-3	...	18
input[i]>0:	T	T	F	T	F	T	T	F	F	...	T



# Branch Divergence: Serialization

- Branch serialization occurs whenever the execution path **within a warp** diverges
  - `if / for / while / case`
- Potential divergence:
  - `if (input[x]>0) {...}`
  - `for(int i=0; i<num_iters[x]; i++) {...}`
- Divergence in **different warps**: No serialization
  - `if (threadIdx.x/32==0) {...}`

# PITCHED ALLOCATION FOR 2D IMAGES



# 2D Images: Linear Allocation

- One can allocate 2D images as 1D-arrays and access in a linearized way: `img[x+w*y]`
- This works, but is in general **suboptimal** for CUDA
- For a `6*3 float` image, the addresses `&img[x+6*y]` are

48	52	56	60	64	68
24	28	32	36	40	44
0	4	8	12	16	20

- Read/write accesses are fastest when the **starting address of each row** is a **multiple of a big power of 2**
  - at least **128**, or even **512**
  - reason: requirement for memory coalescing, see later

# 2D Images: Pitched Allocation

- Adding padding bytes at the end of each row resolves this

64	68	72	76	80	84	88	92
32	36	40	44	48	52	56	60
0	4	8	12	16	20	24	28

- The total new width in bytes is called **pitch**
  - here: **pitch = 32 bytes** ( $= 8 * \text{sizeof}(\text{float})$ )
  - in general, **pitch** != multiple of element size
    - example: 10\*10 float3 array
    - $\text{sizeof}(\text{float3}) = 12$ ,  $w * \text{sizeof}(\text{float3}) = 120$ , **pitch = 128**
- **cudaMallocPitch** (**void \*\*pointer, size\_t \*pitch, size\_t widthInBytes, size\_t height**);

# 2D Images: Pitched Allocation

- **On host:**

```
float *d_a;  
size_t pitch;  
cudaMallocPitch(&d_a, &pitch, w*sizeof(float), h);
```

- **In kernel:**

```
float value =  
    *((float*) ( (char*)a + x*sizeof(float) + pitch*y) );
```

- **Copying: `cudaMemcpy2D(...)`**
  - see NVIDIA Programming Guide

- **For 3D-Data: `cudaMalloc3D()`**

# HOST-DEVICE MEMORY TRANSFER

# Host-Device Memory Transfer

- **Memcpy from device to host and vice versa is very slow**
  - orders of magnitude slower than device-to-device
- **Minimize transfers**
  - leave data for as long as possible on GPU for processing
  - only transfer main inputs to GPU, and transfer main outputs back
- **Group transfers**
  - one large transfer much faster than many small ones
- **Overlap transfers with kernel executions**
  - if possible by hardware
  - uses pinned host memory and streams

# Pinned Host Memory

- Enables **highest memcopy performance**
- Enables **asynchronous memcopy** (CC>=1.1)
- Enables **direct access from GPU** (CC>=1.1)
- `cudaMallocHost(void **pHost, size_t size, unsigned int flags);`
- `cudaFreeHost(void *ptr);`
- **page-locked**, allocating too much may degrade your system
- `flags = cudaHostAllocMapped`: **direct access from GPU**  
`void *pDev; cudaHostGetDevicePointer(&pDev, pHost, 0);`
- `flags = 0`: **default**

# Asynchronous Memory Copy

- Usual `cudaMemcpy` is **blocking**
  - waits until memcpy is done
- `cudaMemcpyAsync(dst, src, size, dir, 0);`
  - **asynchronous**, non-blocking
  - `cudaMemcpyDeviceToHost`, `cudaMemcpyHostToDevice`
  - 0 is the default stream (more later)
- Requirement: **"pinned"** host memory
  - allocated using `cudaMallocHost`

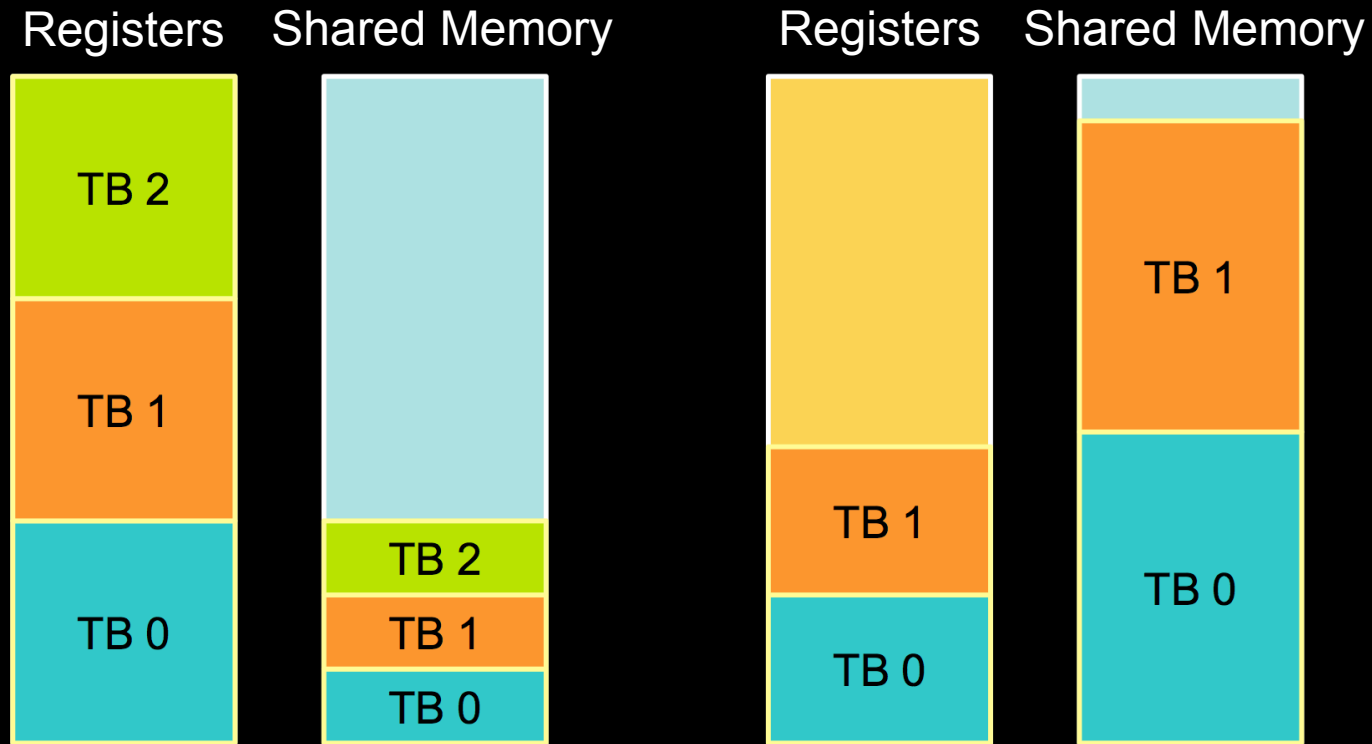
**OCCUPANCY**



# Occupancy

- Multiprocessors (SMs) can have many more **active threads** than there are CUDA Cores
- High occupancy is important
  - if some threads stall, the SM can switch to others
- Pool of limited resources per SM
- Occupancy determined by
  - **Register usage** per thread
  - **Shared memory** per block

# Resource Limits



- Each block grabs registers and shared memory
- If one or the other is fully utilized:
  - no more blocks per SM possible

# Find Out Resource Usage

- Compile with `nvcc` option `-ptxas-options=-v`
- Per kernel **registers** and (static) **shared memory**:

```
ptxas info    : Compiling entry function '_Z10add_kernelPfPKfS1_i' for 'sm_10'  
ptxas info    : Used 4 registers, 44 bytes smem
```

- Amount of resources per multiprocessor:
  - run `deviceQuery`

# Optimize Algorithms for the GPU

- Maximize **independent parallelism**
- Maximize **arithmetic density** (math/bandwidth)
- Sometimes it's **better to recompute** than to cache
  - GPU spends transistors on computation, not memory
- Do more computation **on the GPU** to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to/from host