JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Day3

- Introduction to the CUDA memory architecture and efficient memory usages
- Implementation the reduction algorithm on GPU (cont.)

Dr. Tuan-Tu Tran
trant@uni-mainz.de

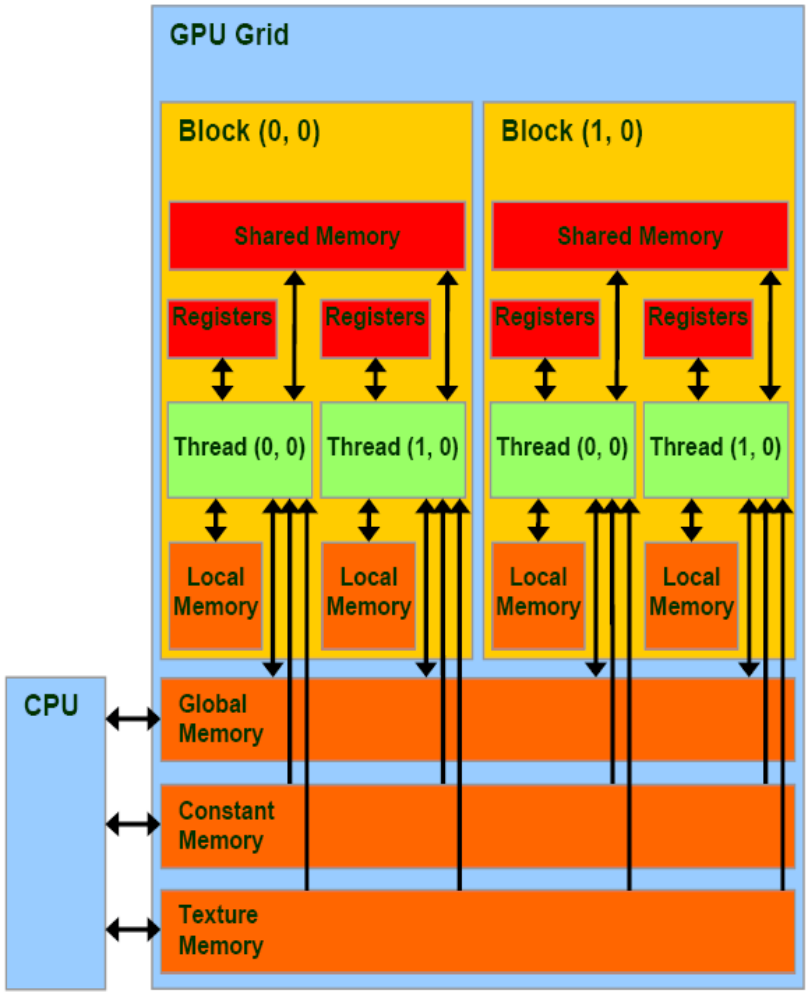
JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Part 1

Introduction to the CUDA memory architecture and efficient memory usages

The CUDA Memory architecture

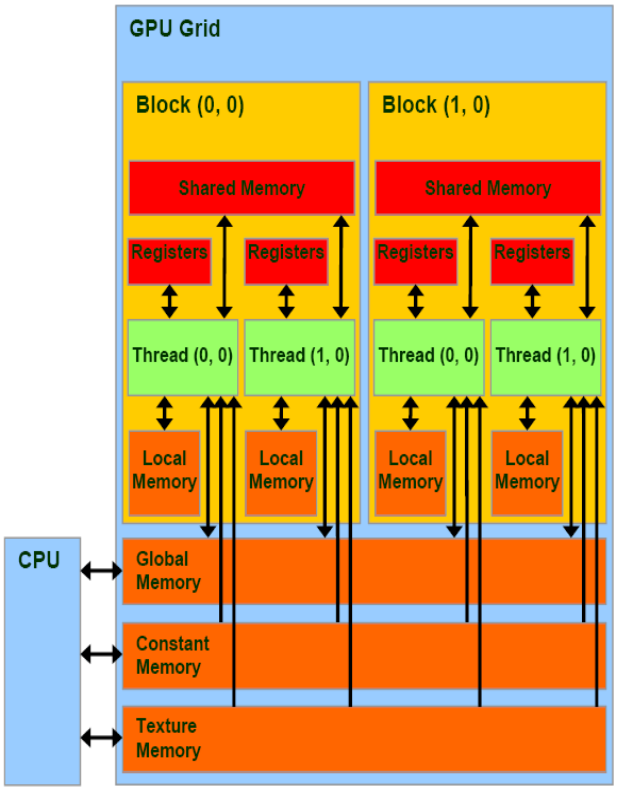
JOHANNES GUTENBERG UNIVERSITÄT MAINZ

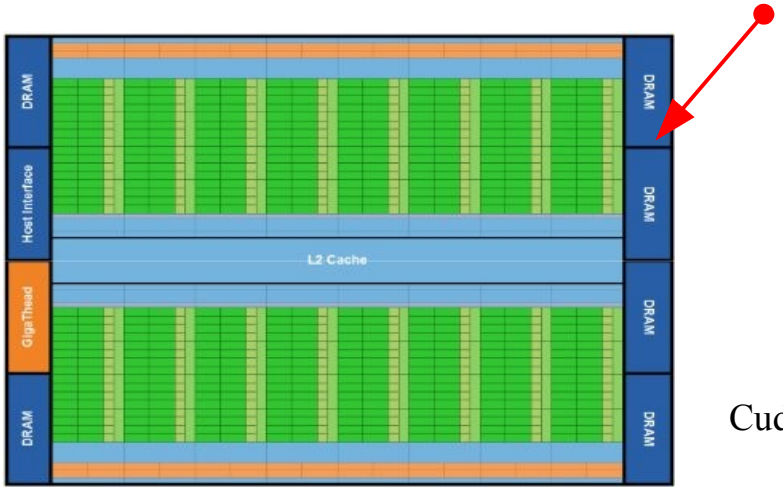


- Global Memory
- Registers
- Local Memory
- Constant Memory
- Texture Memory
- Shared Memory

(image credit: NVIDIA)

Global Memory





(image credit: NVIDIA)

- Location: Off-Chip Memory (*K20X: GDDR5, 6 GB*)
  - Large
  - High Bandwidth
  - High Latency
- Advantage:
  - Frequency of read/write transaction is small
  - Data amount of each transaction is large
  - Data in each transaction is in contiguous region
- Disadvantage:
  - High frequency of read/write transaction with light weight data, to the random memory.

Global Memory: Coalescing Access

More details can be found in (*in the relation with the cache and the capability of the device*):

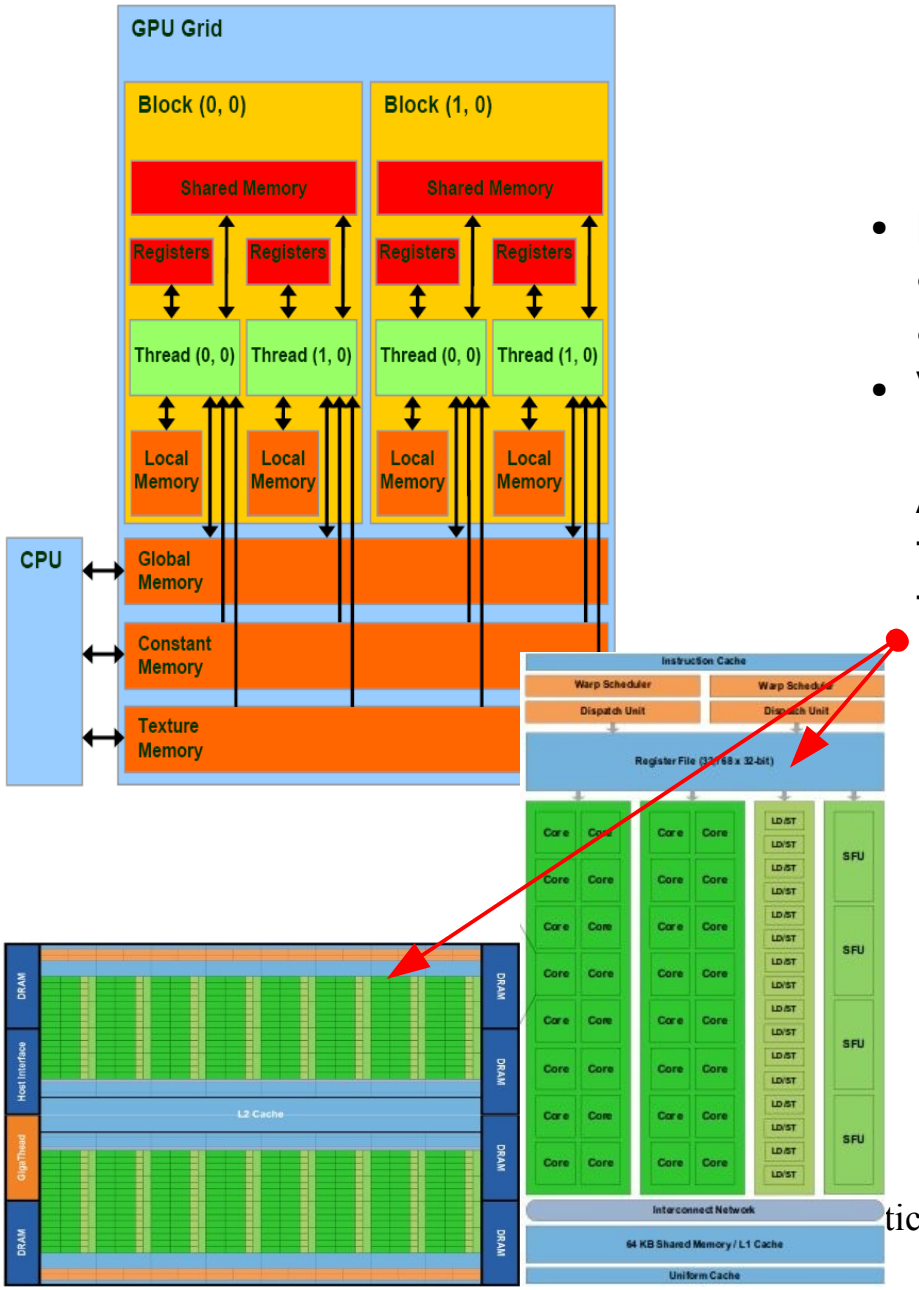
NVIDIA, CUDA C Best Practice Guide,

http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#coalesced-access-global-memory

- One of the basic optimization for memory usage, mentioned in a lot of CUDA documents
- Depended on the compute capability of NVIDIA GPUs

as NVIDIA GPUs still being developed rapidly
=> will be discussed more detaily in Day4

Register

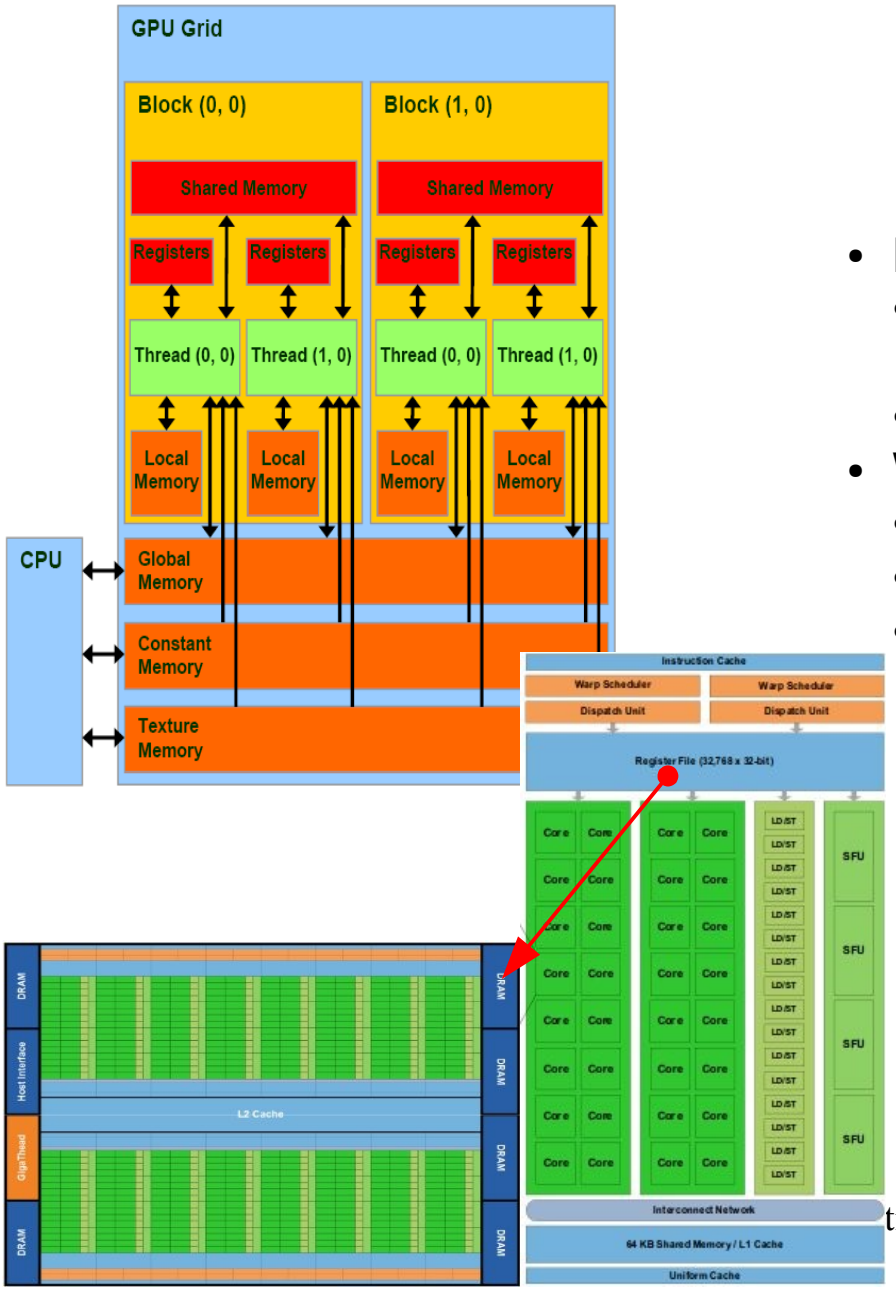


(image credit: NVIDIA)

- Location: On-Chip Memory (inside SM/SMX)
  - Small (64K 32-bit register ~ 256 KB)
  - Fast
- Variables of threads

As multiple blocks (~ 8/16 blocks) can be scheduled to run concurrently on one SM/SMX => optimization the number of thread inside a block (block size).

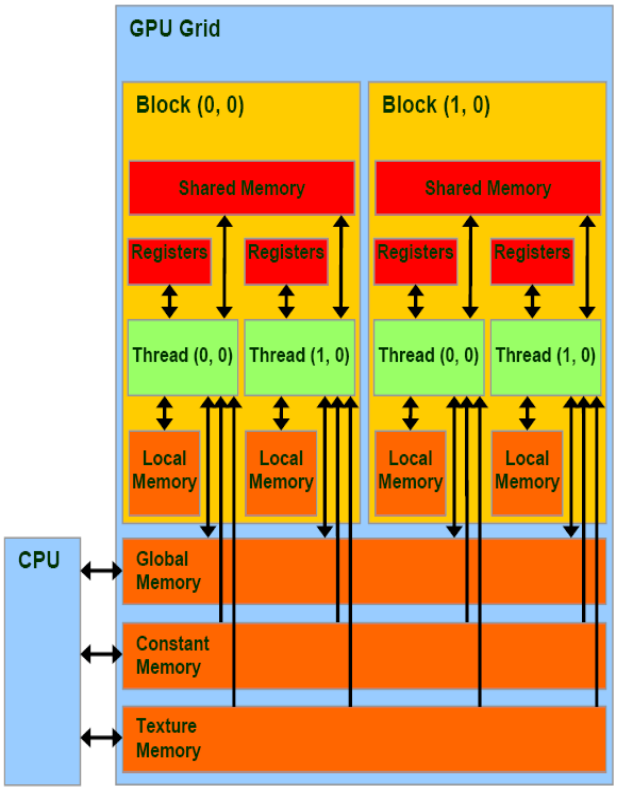
“Local Memory”

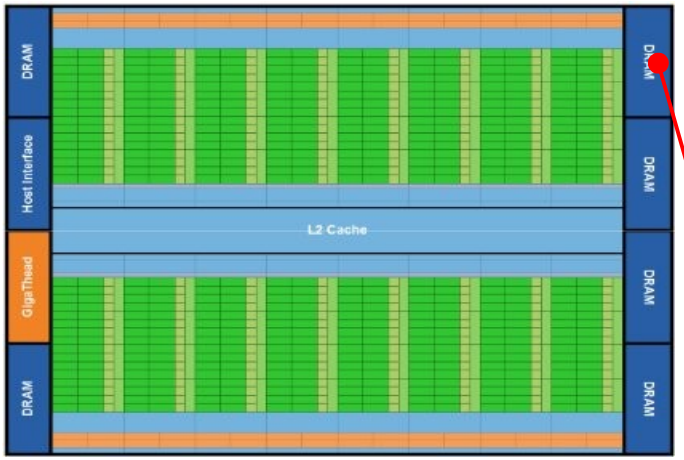


(image credit: NVIDIA)

- Location:
  - Compute capabilities 1.x and 2.x: Off-Chip Memory (global memory)
  - Compute capability 3.x: L1 cache
- When:
  - Spilled Registers
  - Stack
  - Array inside the kernel, with the non-specified size at the compiled time.

Constant Memory



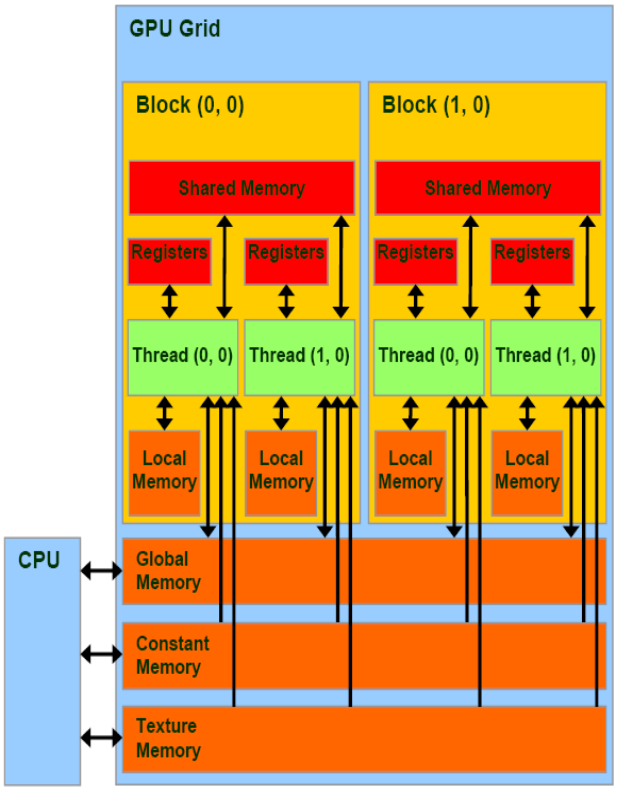


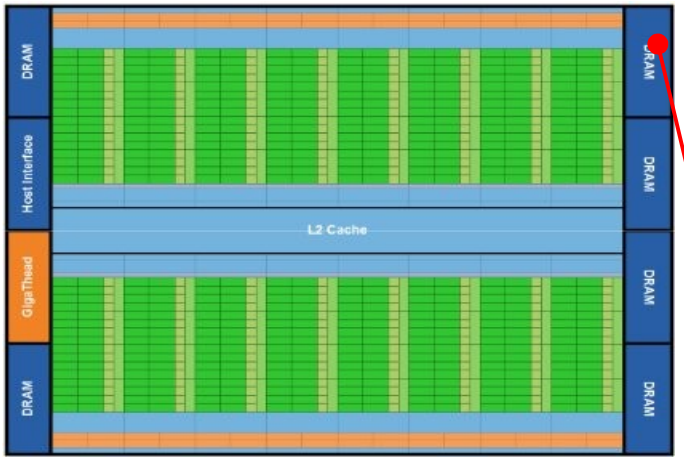
(image credit: NVIDIA)

- Location: Off-Chip Memory, cached in an on-chip 8KB cache (different from L1)
  - Read Only
  - Small Size (max 64 KB)
- *Cache Hit/Cache Missed*

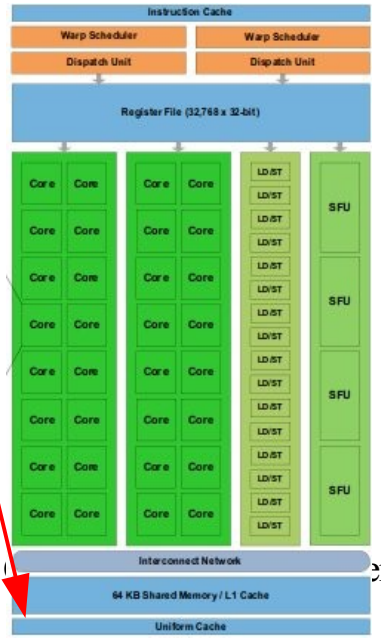
- ***More detail:*** *http://cuda-programming.blogspot.in/2013/01/what-is-constant-memory-in-cuda.html*

Texture Memory

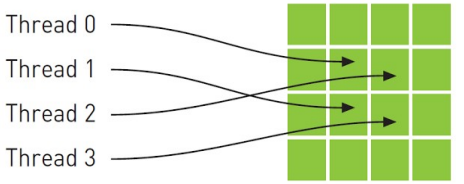




(image credit: NVIDIA)

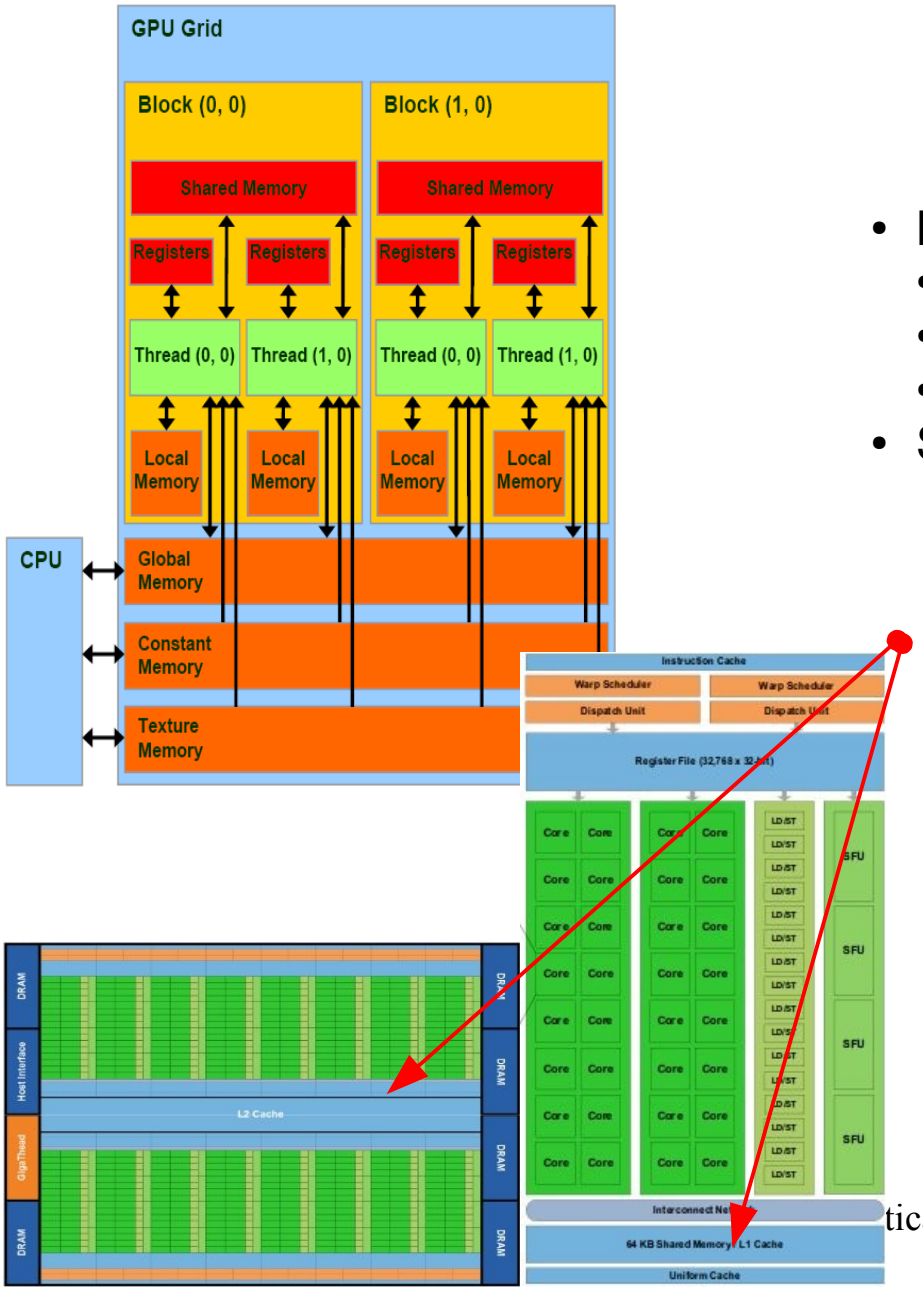


- Location: Off-Chip Memory, cached in an on-chip cache (6-8 KB)(different from L1`
  - Read only
  - Size is limited (hundreds MB)
- Optimal for 2D spactial access



- More detail: http://cuda-programming.blogspot.de/2013/02/texture-memory-in-cuda-what-is-texture.html

Shared Memory



(image credit: NVIDIA)

- Location: On-Chip Memory (inside SM/SMX)
  - Using the same memory with the L1 cache
  - Small (16, 32 or 48 KB)
  - Fast
- Shared among threads in the same block

Shared Memory Bank Conflict

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

- Access to shared memory is performed through the “banks”
- GPU with computed capability 2.x and 3.x: there are 32 banks => can perform 32 access to the shared memory simultaneously.
- Width of bank: 32 bits (4 bytes) = 1 word

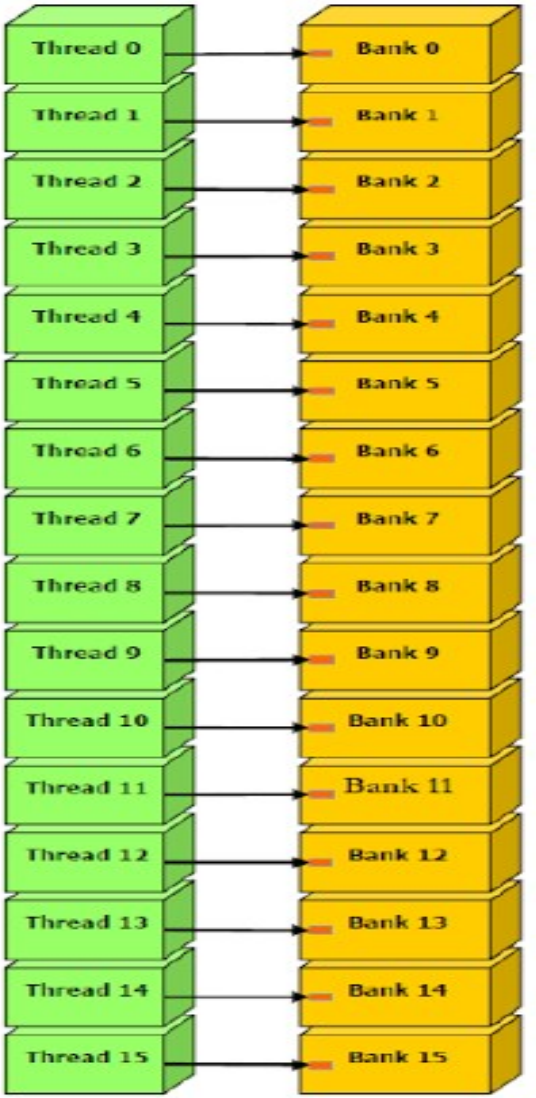
- Given a based index: B => the word number B and the world number (B+32) are accessed through a same bank

Bank Conflict free

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

```
__global__ void kernel(int* gData)
{
    __shared__  int  sData[32];
    sData[threadIdx.x] = gData[threadIdx.x];
}

//1 int = 4 bytes = 1 word
```

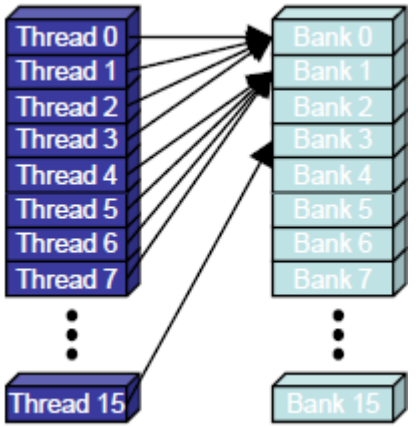


(image credit: NVIDIA)

4-way bank conlict

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

```
__global__ void kernel(char* gData)
{
    __shared__  char  sData[32];
    sData[threadIdx.x] = gData[threadIdx.x];
}
```



(image credit: NVIDIA)

```
//1 char = 1 byte
//=> the first 4 chars are accessed through the first bank, etc
```

2-ways bank conflict

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

```
__global__ void kernel(int* gData)
{
    __shared__  double  sData[32];
    sData[threadIdx.x] = gData[threadIdx.x];
}


//1 double = 8 bytes = 2 words
//=> the first double and the 16th double
//are accessed through the same bank
```

Using the shared memory inside the CUDA code

The shared memory can be declared and allocated **inside** or **outside** the kernel:

Inside the kernel, the size must be fixed

```
#define LEN 8
__global__ kernel1(…)
{
    __shared__ int  arr1[LEN];
    __shared__ int  arr2[16];
    ...
}
```

- `__shared__` : CUDA keyword, to define shared memory
- For shared memory decleared **ouside** the kernel:
  - `Extern`: C keyword, to tell the compiler that this variable has been declared outside this function
  - `sharedSize` is the size of the shared memory in 1 block, not a total shared memory size of the whole grid

Outside the kernel, the size can be set in the host code,
by using the third configuration argument when invoking the kernel:

```
#define LEN 8
__global__ kernel1(…)
{
    extern __shared__ int  arr[];
    ...
}
//
int main(int argc, char** argv)
{
    ...
    n = 24;
    sharedSize = n * sizeof(int);
    kernel1<<<nBlock,nThread,sharedSize>>>(...)
    ...
}
```

JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Part 2:

Implementation the reduction algorithm on GPU (cont.)

Continue the work of Day2:

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

- Based on the given skeleton: cudaReduction.cu (or from the source code of the squareArray application in Day 1)
- Using the ideas and the pseudo-codes in:

  [1] Mark Harris, Optimizing Parallel Reduction in CUDA,

  http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

- **Basic implementation**: the tree-based approach with 1 thread block, with small data (up to 1024 elements), by using 2 algorithms:
  - Interleaved addressing with divergent branching [1, page 7]
  - Interleaved addressing without divergent branching [1, page 11]
- **Advanced implementation**: based on the basic implementation for small data, write the full CUDA application to process the large data of any size, by using recursive kernel invocation [1, page 7]. We suppose that the size of the input array is the power of 2, and up to 32M (33554432 elements).
- **Extend implementation**: the tree-based approach with 1 thread block, with small data (up to 1024 elements), by using the other ideas and optimizations in [1],

Algorithm Analysis

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

- Implement and analyze 3 versions of the CUDA Parallel Reduction presented in [1]:
  - Ver-1: Interleaved addressing with divergent branching [page 7]
  - Ver-2: Interleaved addressing without divergent branching [page 11]
  - Ver-3: Sequential Addresing [page 15]
- Why Ver-2 can reduce the divergent branches, in comparison with Ver-1 ?
- Why Ver-3 is free of shared memory bank conflict, in comparision with Ver-1 and Ver-2 ?

Extend Implementation: Prefix Parallel Scan

We use the idea and the pseudo-code in:

[2] **M. Harris, S. Sengupta and J. D. Owens**, Prefix Parallel Scan (Sum) with CUDA,

http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Basic implementation: using 1 block with small data (up to 1024 elements), by using:

- A naïve parallel scan [2, section 39.2.1]
- The work efficient parallel scan [2, section 39.2.2]

Advanced implementation: based on the basic implementations for small data, write the full CUDA application to process the large data of any size by using the algorithms in [2, 39.2.4]. We suppose that the size of the input array is the power of 2, and up to `32M` (`33554432` elements).

JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Question ?