CUDA Memory

Hierarchy and Performance Considerations

Sergio Orts Escolano
Albert García García

sorts @ ua.es
agarcia @ dtic.ua.es

Contents

Memory Hierarchy

GPU Memories

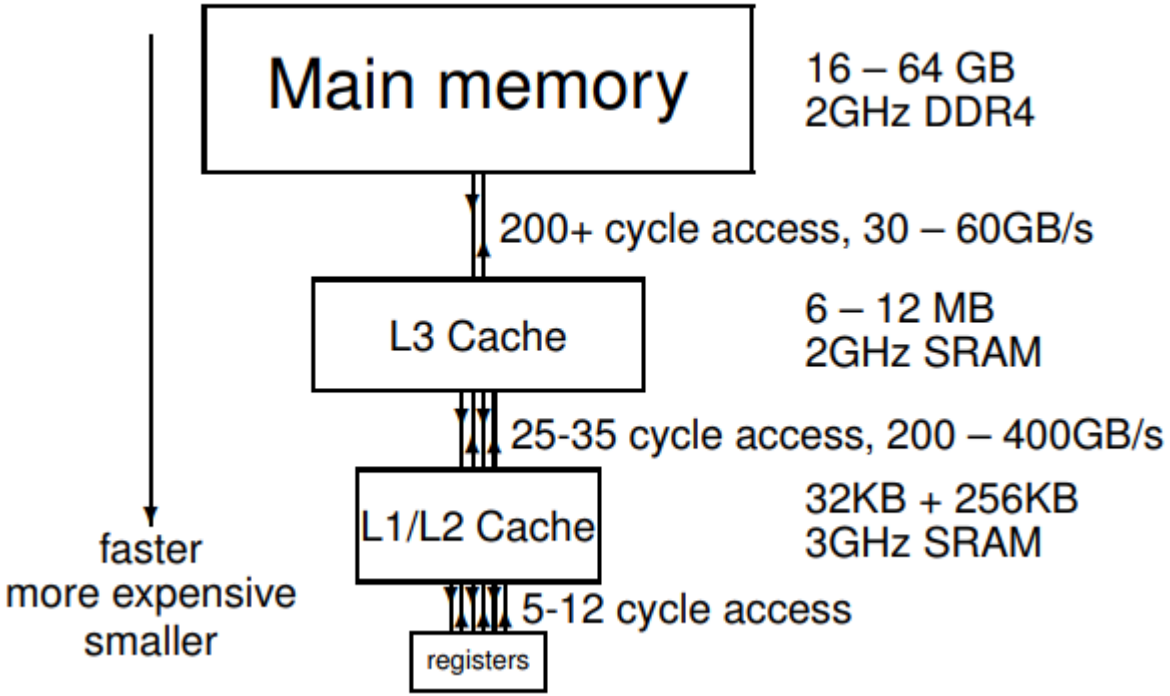
Performance Considerations

Unified Memory

Memory Hierarchy

Memory Hierarchy (CPU)

Trade-off between speed and cost



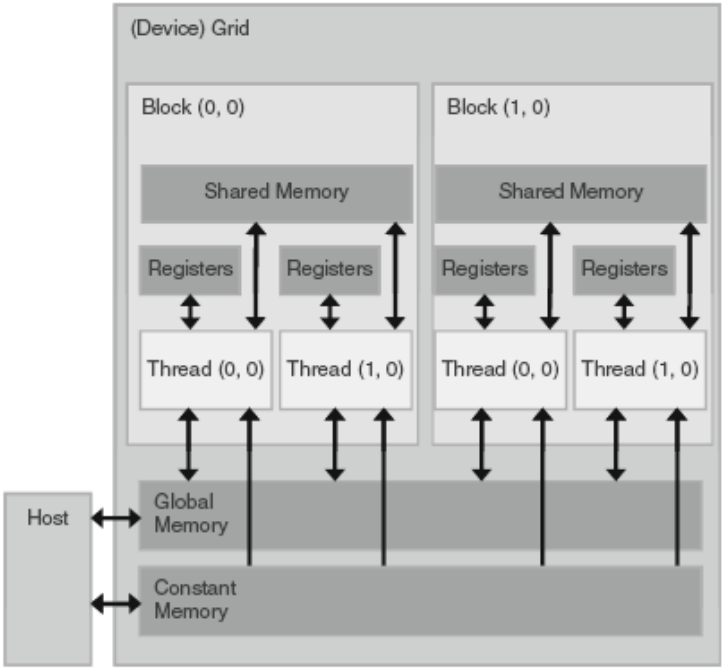
Why Do We Need a Hierarchy?

Locality

- "Temporal Locality" results that are referenced very closely together can be kept in the machine registers.
- "Spatial Locality" if one element is referenced, a few neighboring elements will also be brought into cache.

Memory Hierarchy (GPU)

Trade-off between speed and cost



Global Memory Bandwidth

Memory limits FLOPS!

Ideal

Reality

Global Memory Bandwidth

Memory limits FLOPS!

- All threads access global memory for their input matrix elements
  - One memory accesses (4 bytes) per floating-point addition.
  - 4B/s of memory bandwidth/FLOPS.
- Assume a GPU with
  - Peak floating-point rate 1,500 GFLOPS with 200 GB/s DRAM bandwidth.
  - 4*1,500 = 6,000 GB/s required to achieve peak FLOPS rating.
  - The 200 GB/s memory bandwidth limits the execution at 50 GFLOPS.
- This limits the execution rate to 3.3% (50/1500) of the peak floating-point execution rate of the device!
- Need to drastically cut down memory accesses to get close to the 1,500 GFLOPS!

GPU Memories

Memory Hierarchy (GPU)

Global Memory

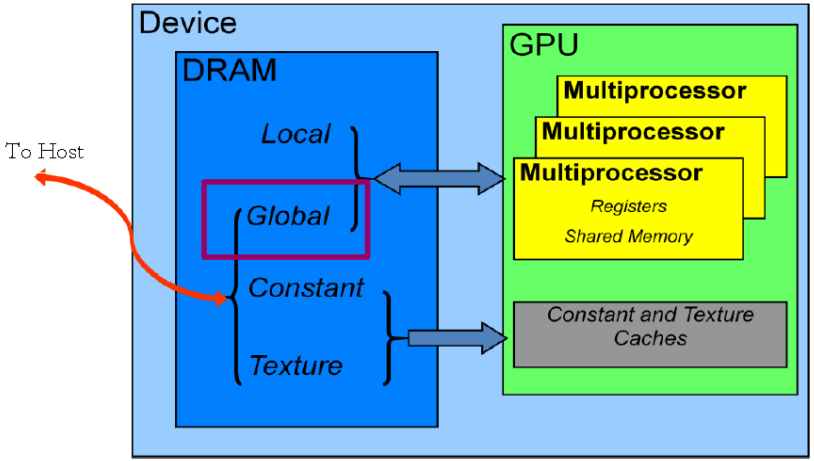
Shared across SMs

High latency

Stores application data

CPU managed
(cudaMalloc, cudaFree)

Requires CPU-GPU transfers
(cudaMempcy)



Memory Hierarchy (GPU)

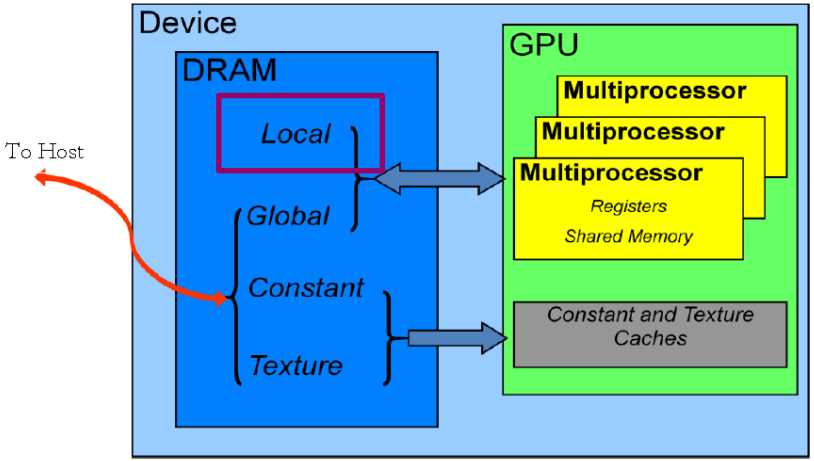
Local Memory

Local thread information

Phyisically DRAM (global) but cacheable (L1 y L2)

Holds variables and vectors which do not fit in registers

Automatically managed by the GPU



Memory Hierarchy (GPU)

Constant Memory

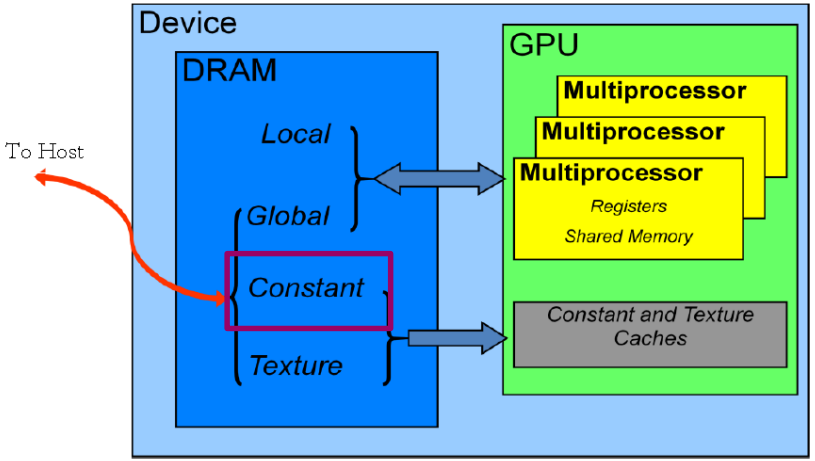
Shared across SMs

Read-only

Stores constant data
(can be modified from the CPU)

CPU managed
Static allocation (__constant__)

Requires CPU-GPU transfers
(cudaMempcyToSymbol
cudaMemcpyFromSymbol)



Memory Hierarchy (GPU)

Texture Memory

Shared across SMs

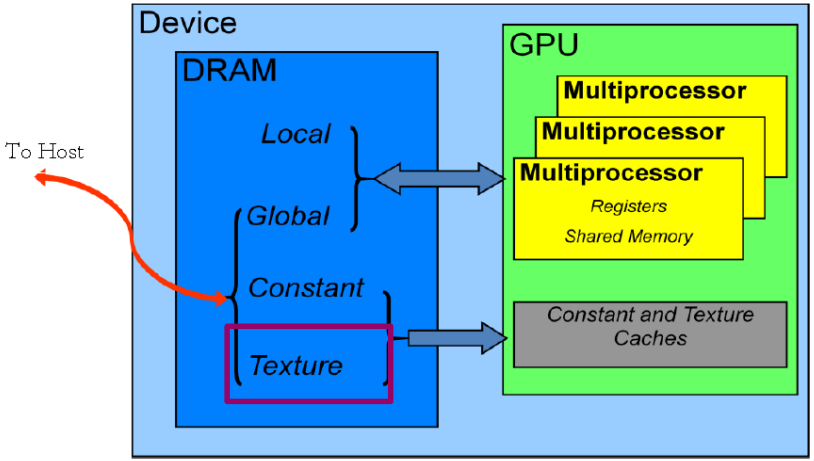
Cacheable (read-only)

Optimized for non-coalesced access

Stores constant data

Very efficient for data normalization or interpolation

Managed using the datatype cudaArray



Memory Hierarchy (GPU)

Registers

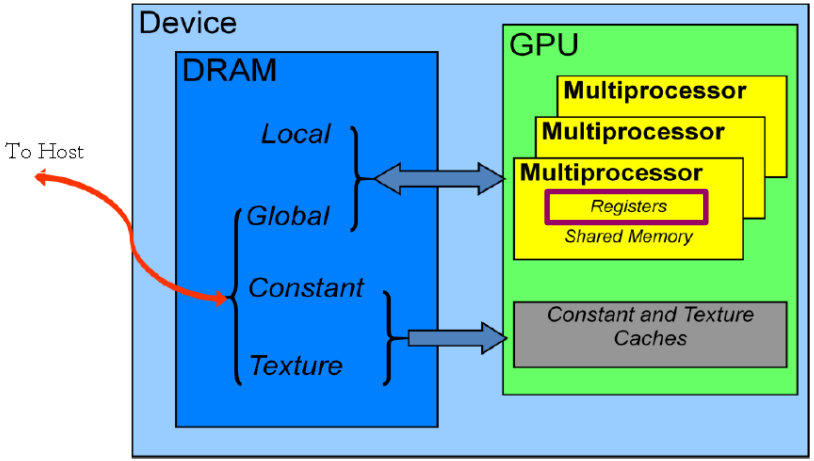
Each SM has a fixed amount of registers

Shared across threads in a SM

Low latency – fastest memory

Stores data allocated in variables (CUDA kernels)

Relatively limited in quantity (depends on the number of threads in execution)



Memory Hierarchy (GPU)

Shared Memory

Each SM has a limited amount of Shared memory

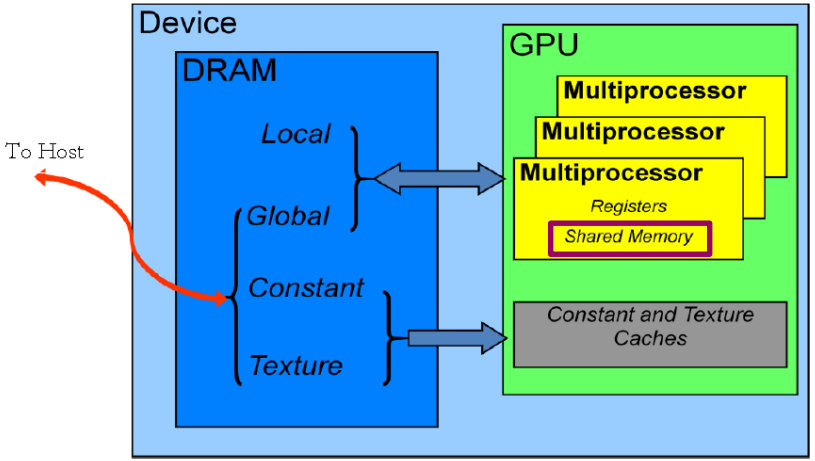
Shared across threads in a block

Low latency

Useful for data re-use

Can be allocated from the GPU (__shared__)

Manual management from CUDA Kernels



Declaring CUDA Variables

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- **__device__** is optional when used with **__shared__**, or **__constant__**
- Automatic variables reside in a register
  - Except per-thread arrays that reside in global memory

Performance Considerations

Performance Considerations

Limitations on GTX1080 Pascal

For each SM:

64 K registers

96 KiB shared memory

48 KiB L1 cache

16 KiB constant cache

2048 threads

Performance Considerations

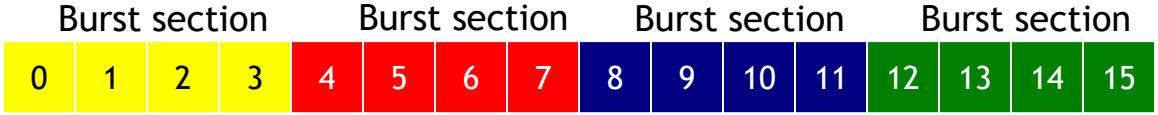
Register spilling

- If the kernel needs more registers per thread than available:
  - They are spilled to local memory.
  - Certain variables might be converted into arrays (maybe global memory!) to store one element per thread.

- POSSIBLE PERFORMANCE LOSS!

Performance Considerations

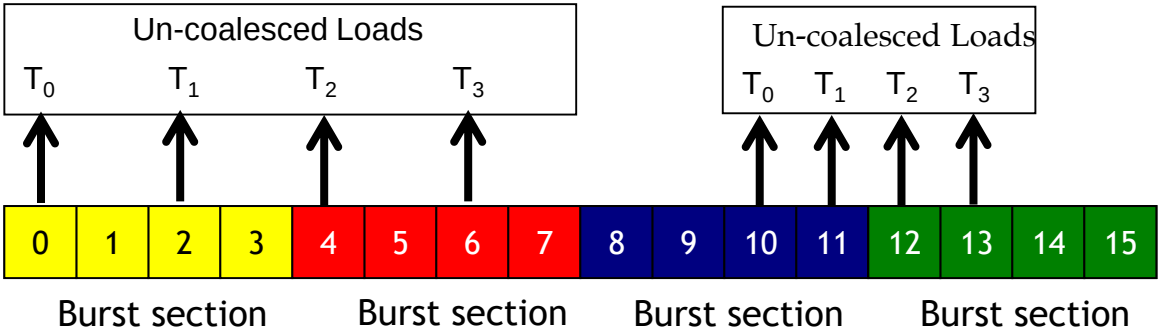
Memory bursting



- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor.
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more.

Performance Considerations

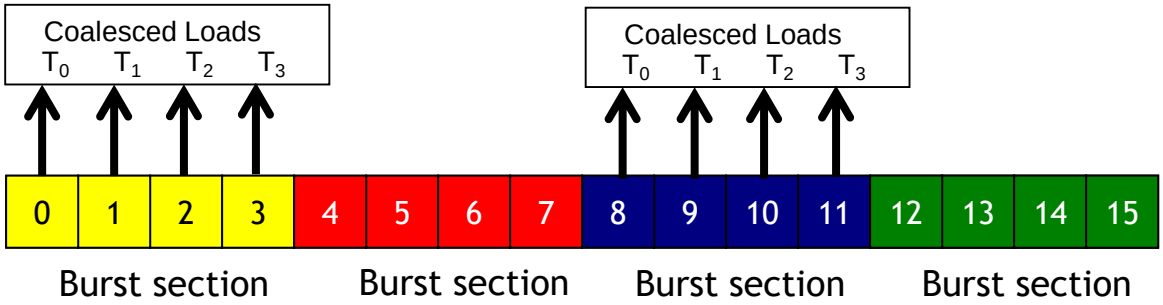
Memory bursting: Uncoalesced access



- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

Performance Considerations

Memory bursting: Coalesced access



– When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

Performance Considerations

Colaesced access example

```
__global__ coalesced_access(float*x)
{
  int tid= threadIdx.x+ blockDim.x*blockIdx.x;
  x[tid] = threadIdx.x;
}
```

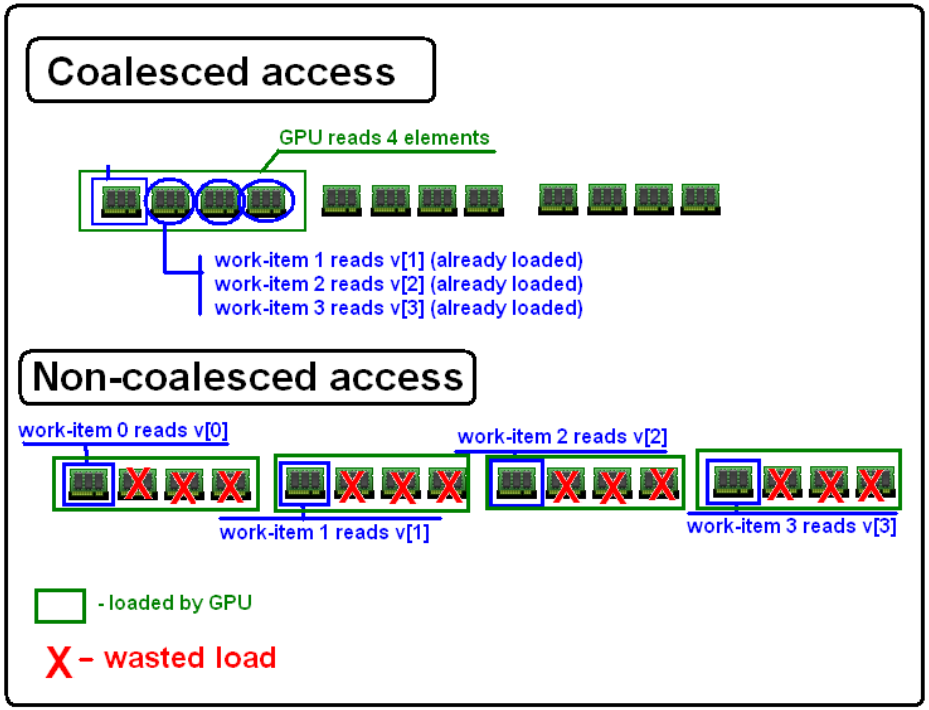
Performance Considerations

Non-colaesced access example

```
__global__ non_coalesced_access(float*x)
{
  int tid= threadIdx.x+ blockDim.x*blockIdx.x;
  x[tid*100] = threadIdx.x;
}
```

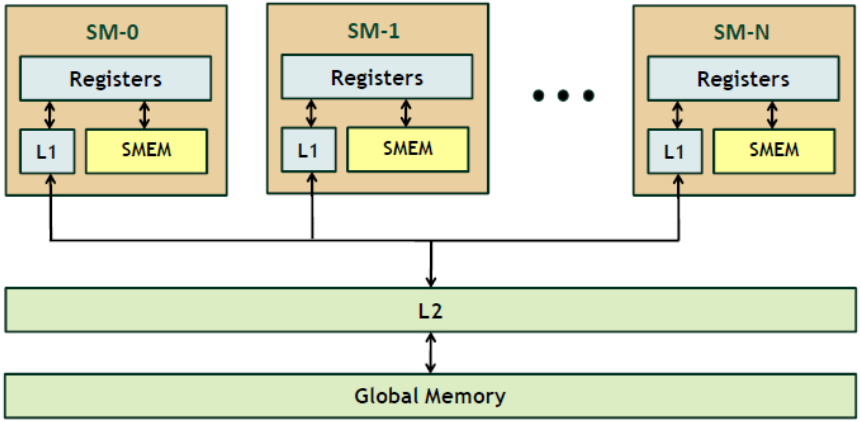
Performance Considerations

Coalesced vs. non-coalesced: side-by-side comparison



Performance Considerations

Shared memory



- Serves a way of communicating or synchronizing threads in a block.
- Takes advantage of data reuse to reduce global/local memory accesses.
- Potentially reduces the number of registers needed for each thread.

Performance Considerations

Shared memory declaration

```
__global__ void medianFilter2D_sm
        (unsigned char *d_output, unsigned char *d_input)
{
        […]
        __shared__ unsigned char d_input_sm_[shmem_size];
        […]
}
```

Performance Considerations

Shared memory copy from global memory

```
__global__ void medianFilter2D_sm
            (unsigned char *d_output, unsigned char *d_input)
{
            […]
            __shared__ unsigned char d_input_sm_[shmem_size];
             d_input_sm[idx_shared_] = d_input[idx_global_];
            […]
}
```

Performance Considerations

Shared memory synchronization

```
__global__ void medianFilter2D_sm
            (unsigned char *d_output, unsigned char *d_input)
{
            […]
            __shared__ unsigned char d_input_sm_[shmem_size];
             d_input_sm[idx_shared_] = d_input[idx_global_];
            __syncthreads();
            […]
}
```

Performance Considerations

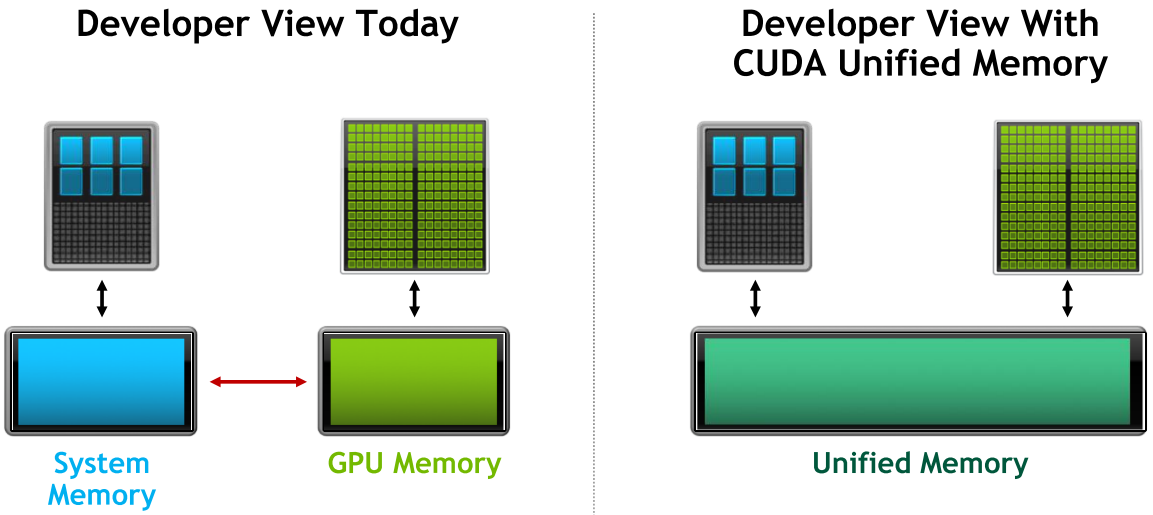
Shared memory usage

```
__global__ void medianFilter2D_sm
        (unsigned char *d_output, unsigned char *d_input)
{
        […]
        __shared__ unsigned char d_input_sm_[shmem_size];
         d_input_sm[idx_shared_] = d_input[idx_global_];
        __syncthreads();
        result = d_input_sm[pos] + d_input_sm[pos+1];
        […]
}
```

Unified Memory

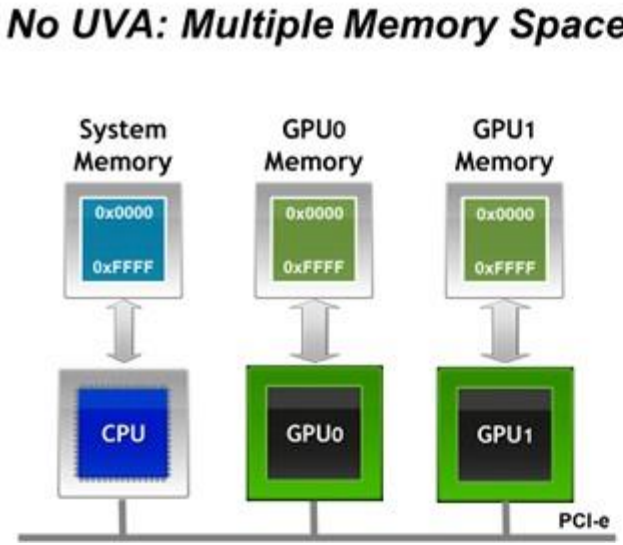
Unified Memory

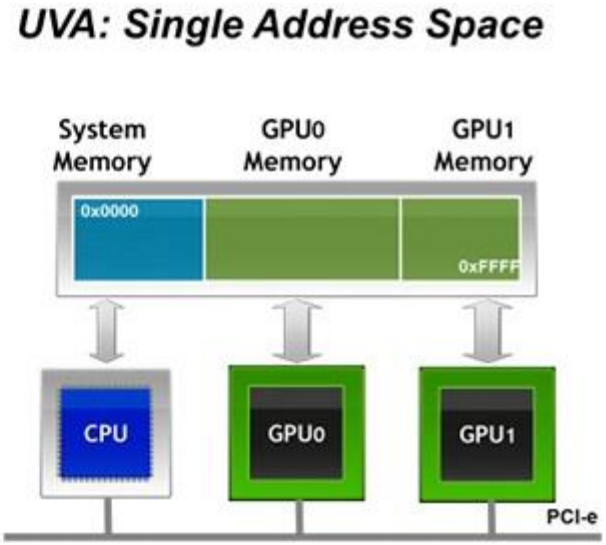
Since CUDA 6.0 (and supported GPUs)!



Unified Memory

UVA (Unified Virtual Addressing)





CUDA Memory

Hierarchy and Considerations

Thanks for your attention!

Sergio Orts Escolano
Albert García García

sorts @ ua.es
agarcia @ dtic.ua.es