Outline

1. Organization
2. Motivation
3. NVIDIA Fermi Architecture
4. CUDA
    - Basics
    - Optimization
5. NVIDIA Kepler Architecture
6. LA Libraries

Evalutation & Feedback

- Help me to help you . . .

`https://docs.google.com/document/d/1Dxim2gU2zEMYGOpnTO2W_CptoKwa3IowM-v-1qIiXyU/edit?pli=1`

Thread divergence

```
if( isOdd( idx ) )
    data[idx] = sin( data[idx] );
else
    data[idx] = 1.0 / data[idx];
```

- Threads within the same warp can follow different execution paths
- Why is it important?
    - Performance penalty of up to 32x
- How can we avoid this?
    - Keep divergence to threads belonging to different warps

SDOT example

Pinned Memory

- *cudaMallocHost* allocates pinned memory
- Pinned memory is required for
  - Asynchronous memory transfers
  - Overlapping computation with communication
- Higher bandwidth than non-pinned memory
- Warning: Too much pinned memory can decrease performance!

Pinned Memory example

Pinned Memory

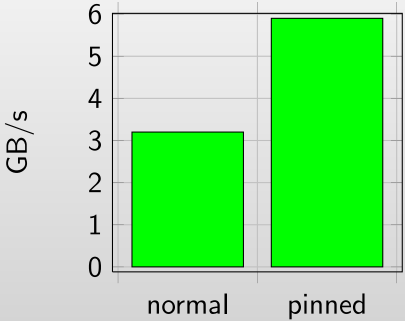
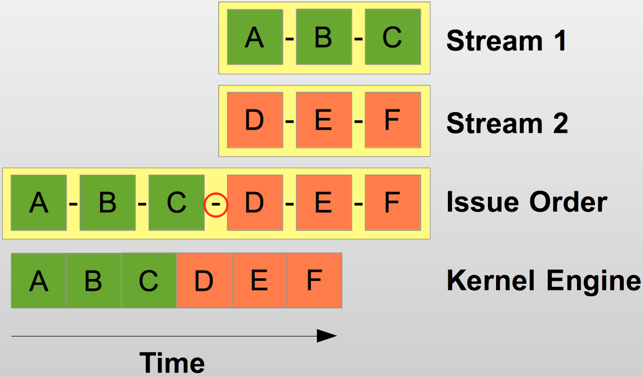


Figure: HtD data transfer for NVIDA Quadro 6000.

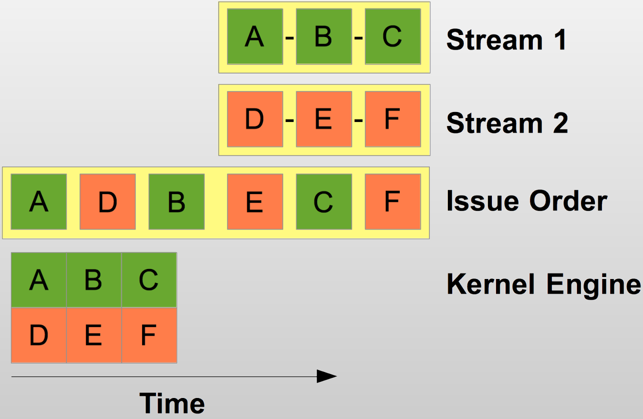
CUDA Streams

- Tasks in different streams may run concurrently
- Tasks in the same stream are executed in-order
- CC 2.*x*
  - Up to 16 kernel simultaneously
  - Up to 2 simultaneous cudaMemcpyAsync (must be in different directions)
    - Query *asyncEngineCount* of *cudaDeviceProp*
- Asynchronous data transfers require . . .
  - . . . non-default streams
  - . . . pinned memory

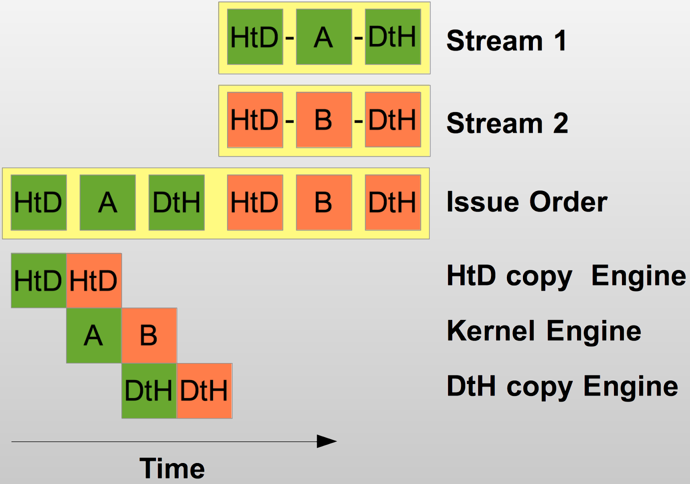
CUDA Streams



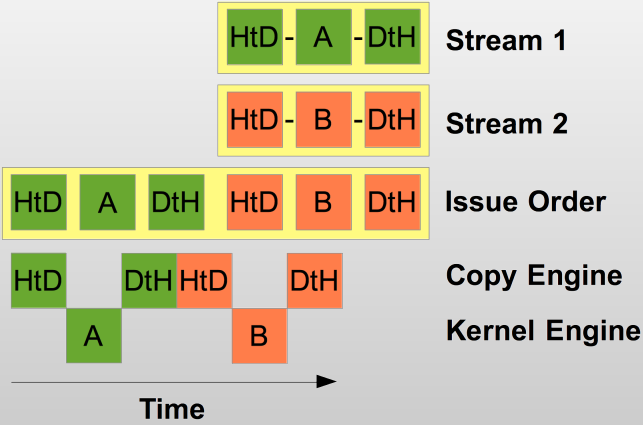
CUDA Streams



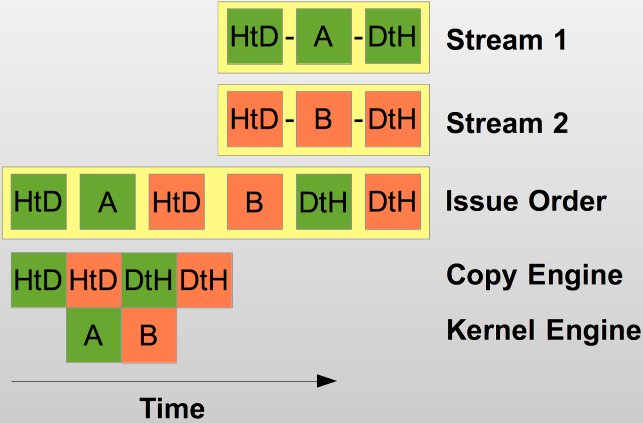
CUDA Streams



CUDA Streams

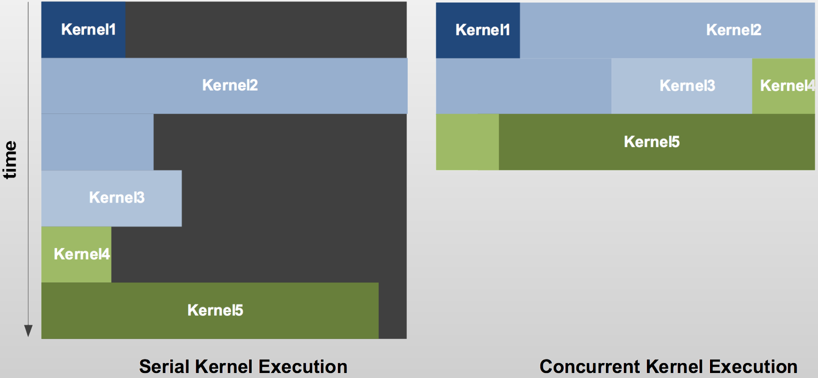


CUDA Streams



Issue order matters!

CUDA Streams



©NVIDIA Fermi Whitepaper

Stream example

CUDA Streams

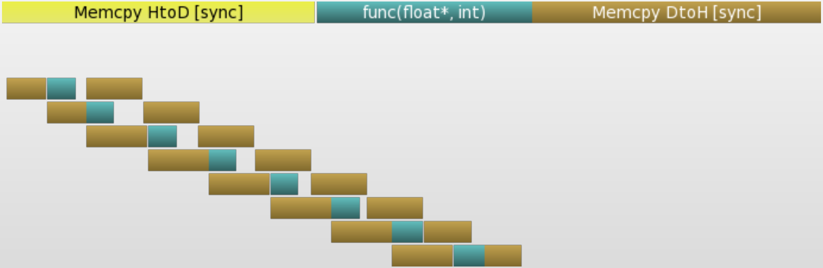


Figure: NVIDIA Quadro 6000. Upper: No streams. Lower: Multiple streams

18ms vs 12 ms (i.e. 1.5x speedup)

Outline

1. Organization
2. Motivation
3. NVIDIA Fermi Architecture
4. CUDA
   - Basics
   - Optimization
5. NVIDIA Kepler Architecture
6. LA Libraries

Optimization Guidelines

- Maximize utilization
- Maximize memory throughput
- Maximize instruction throughput

Utilization

- Overlap data transfers with kernel execution
- Overlap host and device computations
- Choose appropriate launch configurations
    - At least as many threadblocks as SMs (preferably many more)
    - Keep threadblock size a multiple of 32

Occupancy

- Occupancy $= \frac{\#active\ warps}{\#maximum\ warps}$
- Metric for parallel efficiency
- Low occupancy typically results in poor performance
- Caveat: High Occupancy is not always required[1]

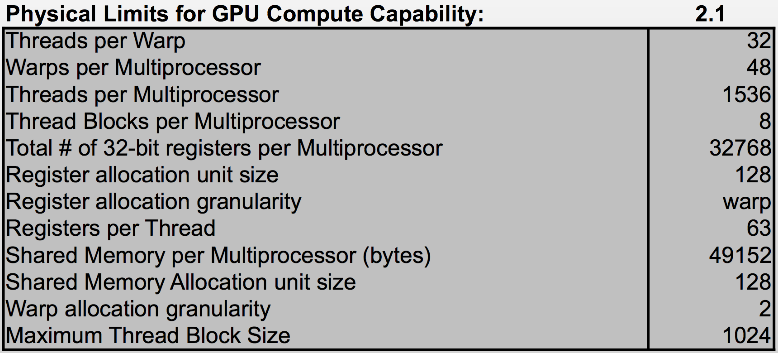
Limiting factors for high occupancy

- Register usage
- Shared memory usage
- Maximum #warps and #threadblocks per SM

---

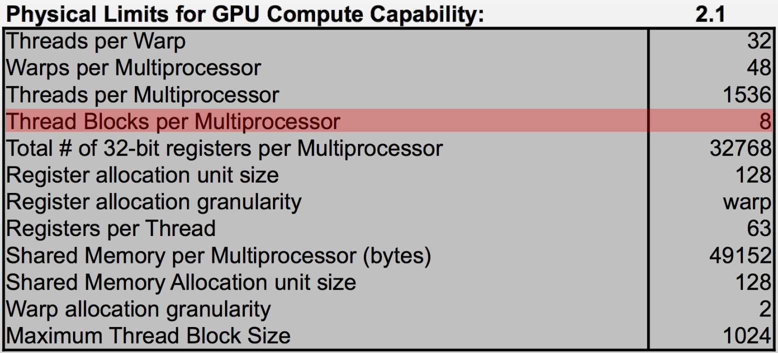
[1] Vasily Volkov. "Better performance at lower occupancy". In: *Proceedings of the GPU Technology Conference, GTC*. Vol. 10. 2010.

Occupancy



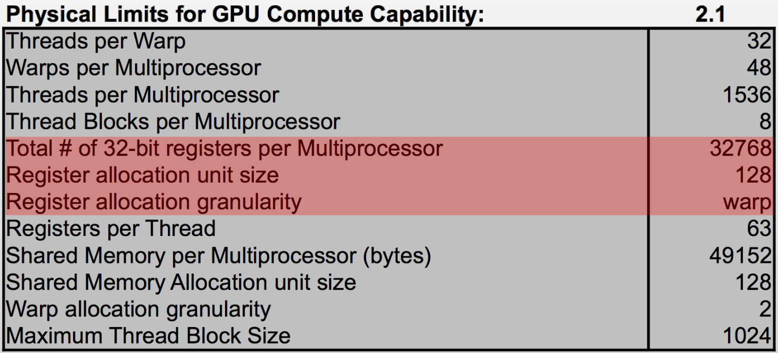
Physical Limits for GPU Compute Capability:	**2.1**
Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Register allocation unit size	128
Register allocation granularity	warp
Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128
Warp allocation granularity	2
Maximum Thread Block Size	1024

Occupancy



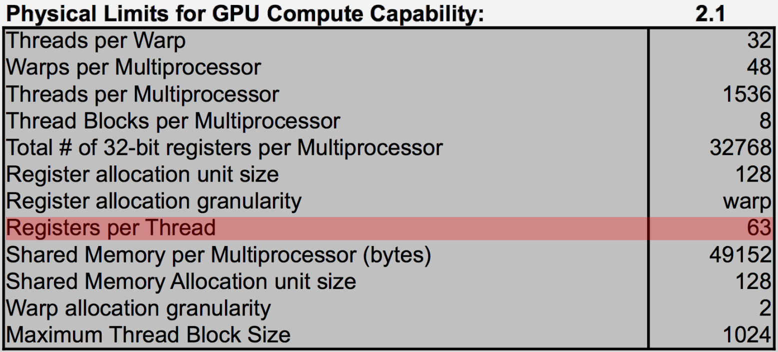
Physical Limits for GPU Compute Capability:	2.1
Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Register allocation unit size	128
Register allocation granularity	warp
Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128
Warp allocation granularity	2
Maximum Thread Block Size	1024

Occupancy



Physical Limits for GPU Compute Capability:	2.1
Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Register allocation unit size	128
Register allocation granularity	warp
Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128
Warp allocation granularity	2
Maximum Thread Block Size	1024

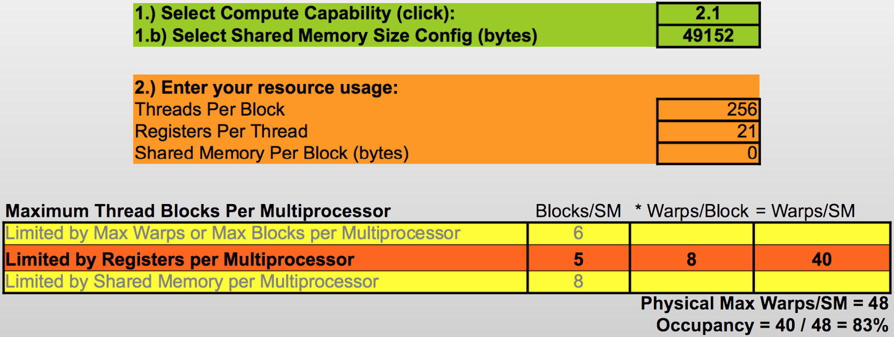
Occupancy



Physical Limits for GPU Compute Capability:	2.1
Threads per Warp	32
Warps per Multiprocessor	48
Threads per Multiprocessor	1536
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	32768
Register allocation unit size	128
Register allocation granularity	warp
Registers per Thread	63
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	128
Warp allocation granularity	2
Maximum Thread Block Size	1024

- Restrict max registers per thread: *-maxrregcount*
- Show resource usage: *–ptxas options=v*

Occupancy



1.) Select Compute Capability (click):	**2.1**
1.b) Select Shared Memory Size Config (bytes)	**49152**

2.) Enter your resource usage:

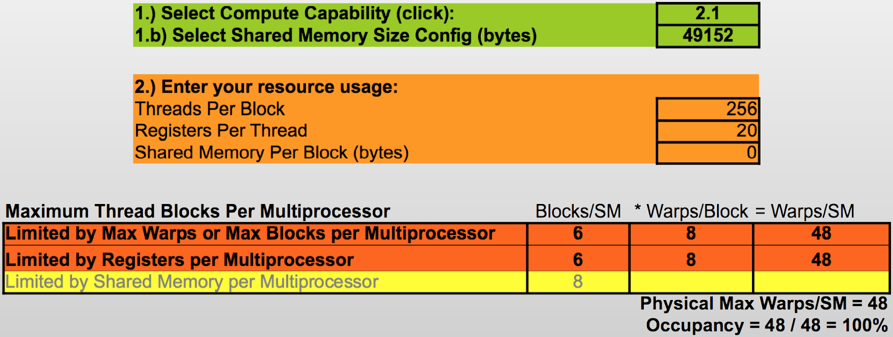
Threads Per Block	256
Registers Per Thread	21
Shared Memory Per Block (bytes)	0

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6		
Limited by Registers per Multiprocessor	**5**	**8**	**40**
Limited by Shared Memory per Multiprocessor	8		

Physical Max Warps/SM = 48

Occupancy = 40 / 48 = 83%

Occupancy



1.) Select Compute Capability (click):	**2.1**
1.b) Select Shared Memory Size Config (bytes)	**49152**

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	20
Shared Memory Per Block (bytes)	0

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	**6**	**8**	**48**
Limited by Registers per Multiprocessor	**6**	**8**	**48**
Limited by Shared Memory per Multiprocessor	8		

Physical Max Warps/SM = 48

Occupancy = 48 / 48 = 100%

Occupancy

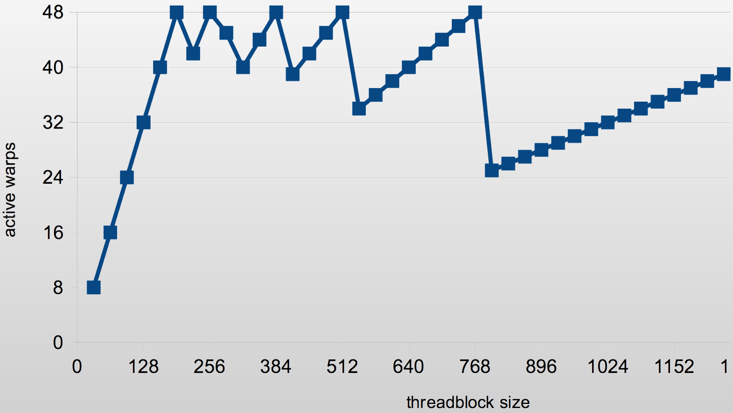


Figure: Impact of varying threadblock size on occupancy for 20 registers per thread.

Memory Throughput

- Use coalesced global memory accesses[2]
- Use shared memory whenever possible
    - Avoid bank conflicts
    - Caveat: Can affect launch the launch configuration
- Use asynchronous data transfer and kernel execution
- Avoid register spilling
- Use pinned memory

---

[2] NVIDIA. *CUDA C Best Practices Guide*. Version 5.0. 2012.

Instruction Throughput

- Use the fast math library whenever speed trumps precision
  - E.g. *__sinf(x)* instead of *sinf(x)*
  - *-use-fast-math*
- Prefer faster, more specialized math functions over slower, more general ones when possible
  - E.g. *exp10(x)* instead of *pow(x,10)*
  - E.g. *x*x* instead of *pow(x,2)*

Instruction Throughput

- Avoid divergence within a warp
- Use *__restrict__* whenever possible
- Avoid automatic conversion between doubles and floats[2]

```
__global__ void foo( const float* a,
                     const float* b,
                           float* c)
{
   int idx = ...;
   c[idx] = a[idx] * b[idx];
   c[idx + blockDim.x] = a[idx] * b[idx];
}
```

---

[2] NVIDIA. *CUDA C Best Practices Guide*. Version 5.0. 2012.

Instruction Throughput

- Avoid divergence within a warp
- Use __*restrict*__ whenever possible
- Avoid automatic conversion between doubles and floats[2]

```
__global__ void foo( const float* __restrict__ a,
                     const float* __restrict__ b,
                           float* __restrict__ c)
{
   int idx = ...;
   c[idx] = a[idx] * b[idx];
   c[idx + blockDim.x] = a[idx] * b[idx];
}
```

---

[2] NVIDIA. *CUDA C Best Practices Guide.* Version 5.0. 2012.

Instruction Throughput

- Avoid divergence within a warp
- Use *__restrict__* whenever possible
- Avoid automatic conversion between doubles and floats[2]

```
__global__ void foo(float* __restrict__ a)
{
   int idx = ...;
   a[idx] *= sqrt(5.0);
}
```

---

[2] NVIDIA. *CUDA C Best Practices Guide.* Version 5.0. 2012.

Instruction Throughput

- Avoid divergence within a warp
- Use *__restrict__* whenever possible
- Avoid automatic conversion between doubles and floats[2]

```
__global__ void foo(float* __restrict__ a)
{
    int idx = ...;
    a[idx] *= sqrt(5.0f);
}
```

---

[2] NVIDIA. *CUDA C Best Practices Guide*. Version 5.0. 2012.