Latency, Occupancy & Streams

UNIVERSITY OF
TORONTO

Homework 3

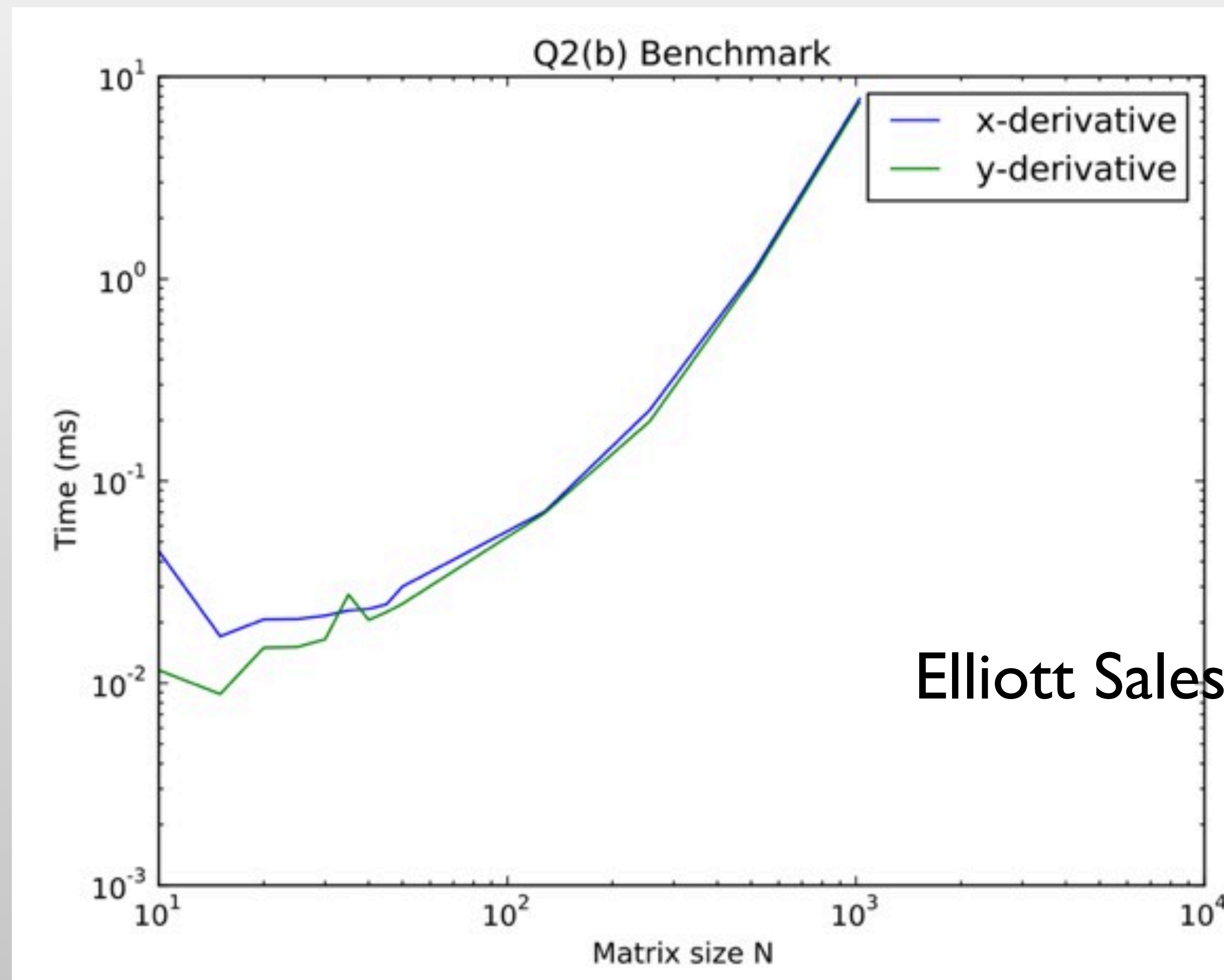
- In 2-D, both d/dx and d/dy collapse to single matrix-matrix multiply
  - dudx = u Dx -- sum over x (stride 1)
  - dudy = (Dy)^t u -- sum over y (stride Nx) ≡ left-multiply by (Dy)^t

- NB: 3-D is different:
  - dudx = u Dx -- stride 1
  - dudz = (Dz)^t u -- stride Nx*Ny, as above, good
  - dudy = ??? -- strides not consistent with BLAS syntax

- CuBLAS from CUDA 3 was *slower* than CPU-BLAS

Homework 3: Transpose

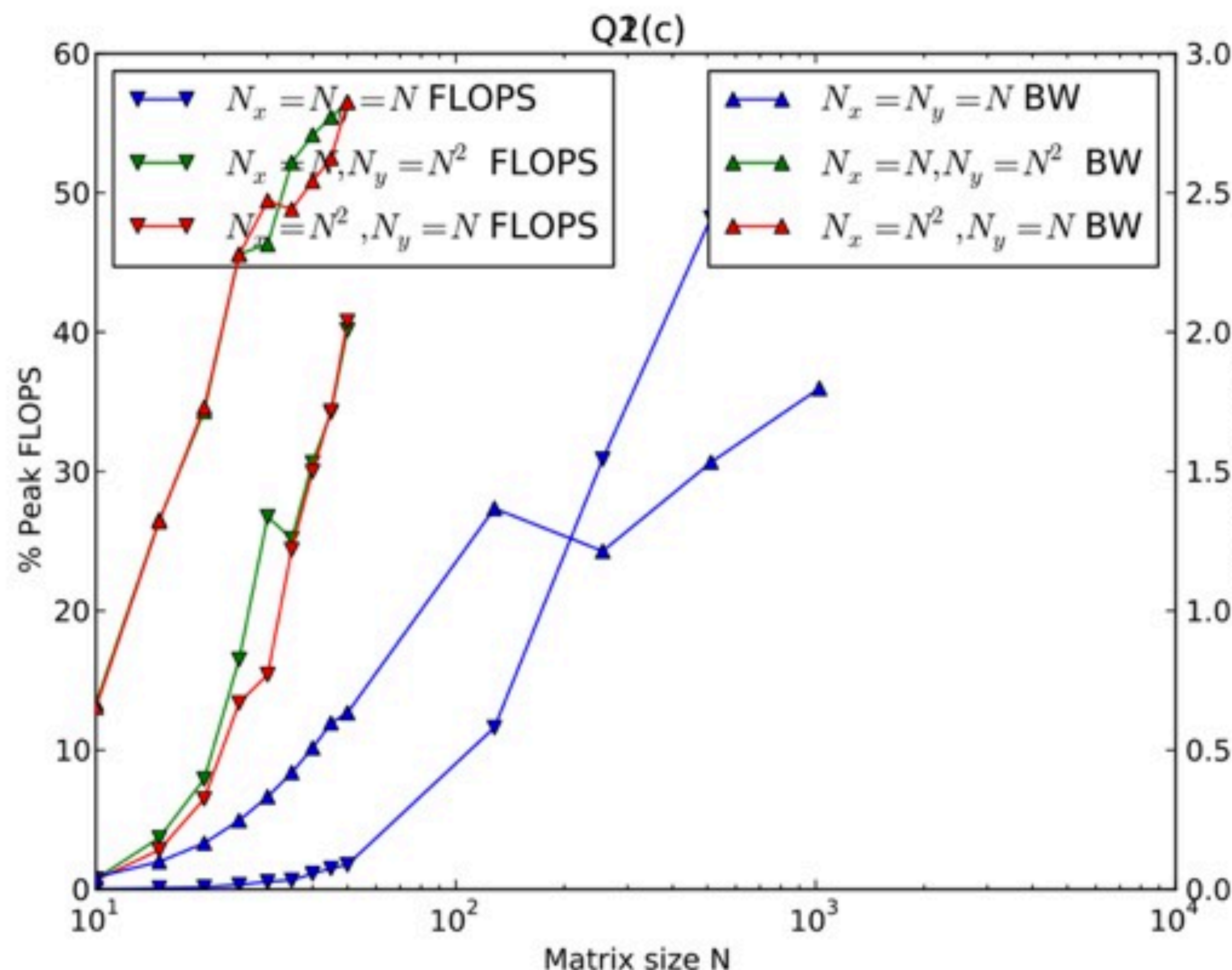
- ❖ CuBLAS: No speed-difference u Dx vs. (Dy)^t u
  - Striding/coalesced memory access taken care of



Elliott Sales de Andrade

HW 3: Saturation size

- Many small matrix-vector multiplies
  - even N=30 gives seizable fraction of peak



Elliott Sales de Andrade

Size	ms CPU	ms GPU
64x8	0.06	0.02
256x16	1.3	0.3
576x24	13	1.2
1024x32	54	6.7

Lukas Kontentis (on GT540m)

Latency

❖ time required to perform an operation
- ~20 cycles arithmetic
- 400-800 cycles global memory access
- cannot start *dependent* operation for this time
- can hide by overlapping with other operations

```
x = a + b;// takes ≈20 cycles to execute
y = a + c;// independent, can start anytime
(stall)
z = x + d;// dependent, must wait for completion
```

Latency hiding (Little's law, again)

CITA–ICAT

- register read-after-write latency ~24 cycles

  x = a + b;
  z = **x** + d;

- SM will perform other operations while waiting
- Need 24 warps to hide 24 cycles latency, i.e. 32*24=768 threads
- Or need code with independent operations:

```
x = a + b;// takes ≈20 cycles to execute
y = a + c;// independent, can start anytime
(stall)
z = x + d;// dependent, must wait for completion
```

Occupancy

- ❖ Rule of thumb: Use as many threads as possible
- ❖ **Occupancy**: # of threads on SM / max # of threads on SM
- ❖ Occupancy subject to various constraints
  - complete blocks assigned to SM
  - If combined register usage exceeds SM limits -> fewer blocks
  - If combined shared mem usage exceeds SM limits -> fewer blocks
  - number of blocks limited
- ❖ For Fermi:
  - 32768 32-bit registers/SM
  - 48KB shared memory/SM
  - max 48 warps/SM (48*32=1536 threads)
  - max 8 blocks/SM
    - blocksize<1536/8=192 can NEVER reach full occupancy

Learning about memory usage I

❖ Compiler diagnostics --ptx-options=-v

```
[pfeiffer@marten class6]$ make clean
rm -f *.o matmult bitreverse
[pfeiffer@marten class6]$ make matmult
nvcc -arch=sm_20 -O3 --ptxas-options=-v -c matmult.cu
ptxas info    : Compiling entry function '_Z14cuda_sgemm_regPKfS0_iPf' for 'sm_20'
ptxas info    : Used 14 registers, 64 bytes cmem[0], 4 bytes cmem[16]
ptxas info    : Compiling entry function '_Z17cuda_sgemm_sharedPKfS0_iPf' for 'sm_20'
ptxas info    : Used 24 registers, 64 bytes cmem[0]
nvcc -arch=sm_20 --ptxas-options=-v -o matmult matmult.o -lcublas
[pfeiffer@marten class6]$ 
```

Learing about memory usage II

CITA–ICAT

❖ Keep track of user-specified shared mem

- compiler cannot know this at compile time

```
__global__ void cuda_sgemm_shared(const float *ad, const float *bd,
                                  const int n, float *cd) {

    extern __shared__ float shared_data[];

    int loci = threadIdx.x;
    int locj = threadIdx.y;
    int tilesize = blockDim.x;
```

Dynamic shared memory

```
printf("cuda_sgemm_shared: shared mem per block: %i bytes\n",
       2*blocksize.x*blocksize.y*sizeof(float));
cudaEventRecord(start, 0);
cuda_sgemm_shared<<<gridsize, blocksize,
                    (2*blocksize.x*blocksize.y*sizeof(float))>>>(ad, bd, n, cd);
```

- here: 2 floats/thread, i.e. 8 bytes/thread.
- Shared Mem/Max(#threads)=48K/1538=32bytes/thread -- should always be safe

Learning about memory usage III

- CUDA visual profiler / nsight
  - Lectures 3/4
  - CUDA_Profiler_Users_Guide.pdf
  - Nsight_Eclipse_Edition_Getting_Started.pdf

Occupancy calculator

- ❖ developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

/Developer/NVIDIA/CUDA-5.0/tools/CUDA_Occupancy_Calculator.xls

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	2.0	(Help)

2.) Enter your resource usage:		
Threads Per Block	64	(Help)
Registers Per Thread	24	
Shared Memory Per Block (bytes)	256	

(Don't edit anything below this line)

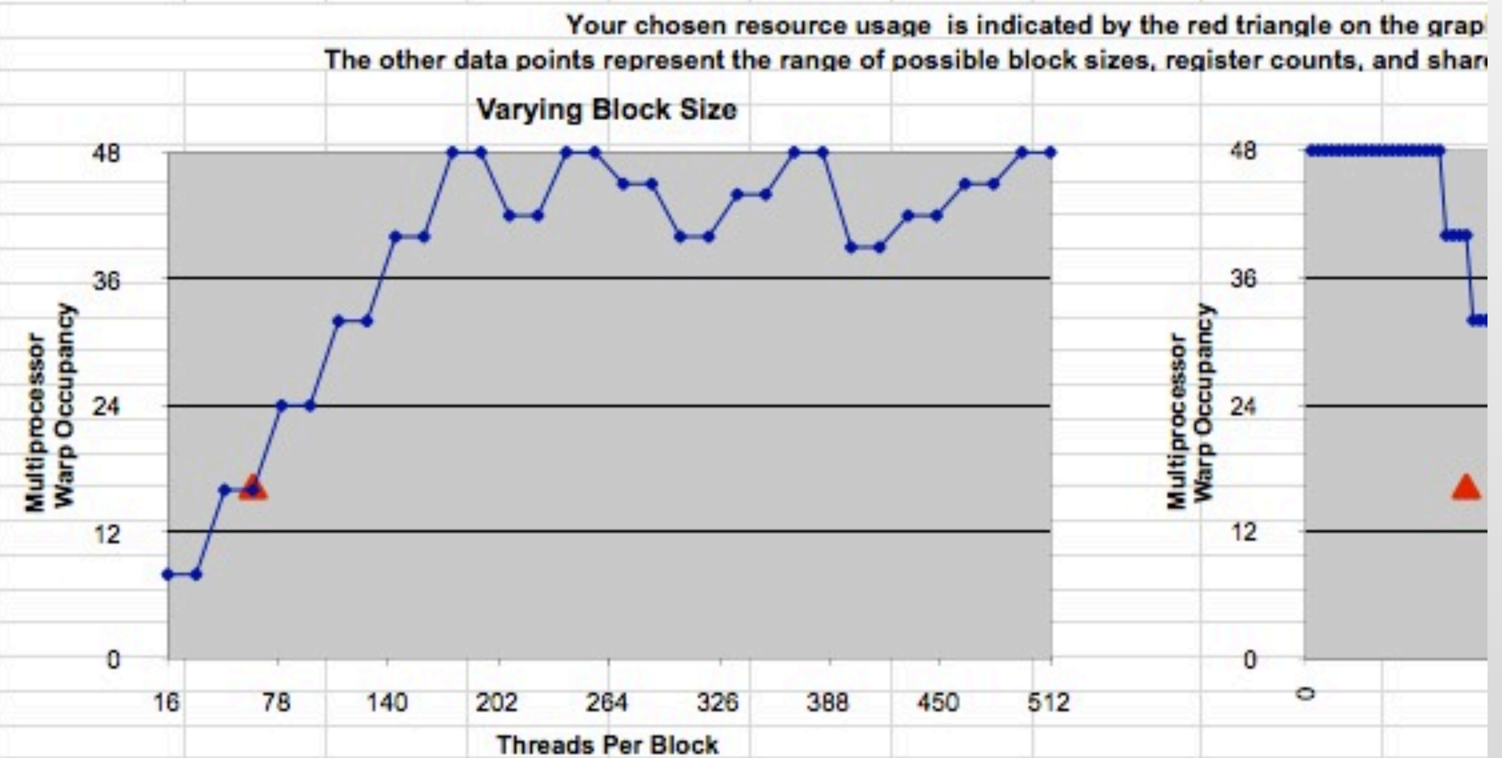
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	512	(Help)
Active Warps per Multiprocessor	16	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	33%	

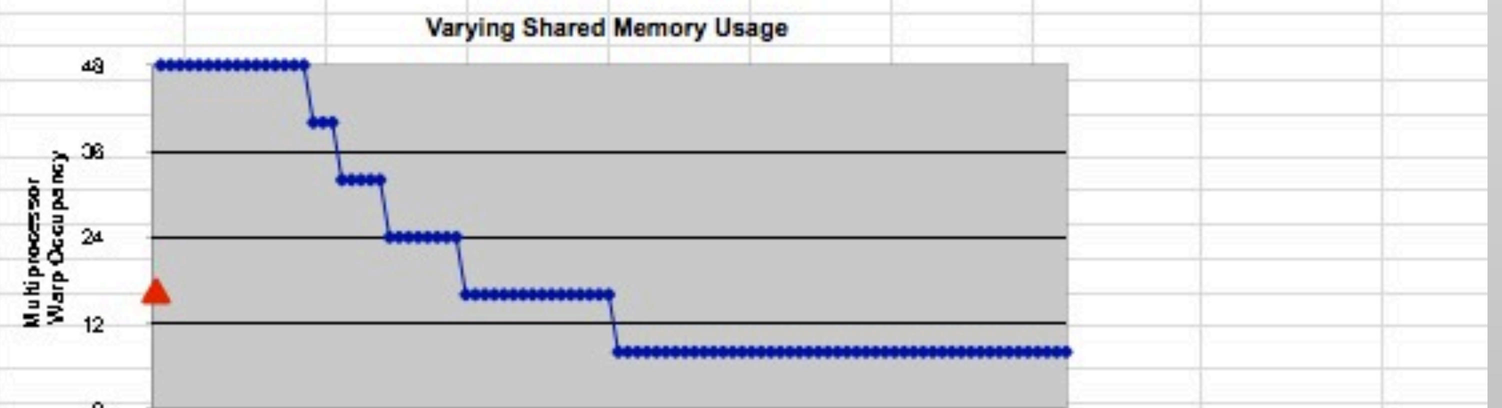
Physical Limits for GPU Compute Capability:	2.0
Threads per Warp	32.0
Warps per Multiprocessor	48.0
Threads per Multiprocessor	1536.0
Thread Blocks per Multiprocessor	8.0
Total # of 32-bit registers per Multiprocessor	32768.0
Register allocation unit size	64.0
Register allocation granularity	warp
Shared Memory per Multiprocessor (bytes)	49152.0
Shared Memory Allocation unit size	128.0
Warp allocation granularity (for register allocation)	0.0

Allocation Per Thread Block	
Warps	2
Registers	1536
Shared Memory	256

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	8
Limited by Registers per Multiprocessor	21
Limited by Shared Memory per Multiprocessor	192





Occupancy more suggestion than rule

❖ In practice, important to try

matmult --matsize=1024 --nblocks=$(NBLK)

$(NBLK)	block-size	sgem_shared (msec)	cuBLAS (msec)
512	4	896	4.4
256	16	156	4.4
128	64	36	4.4
64	256	52	4.4
32	1024	153	4.4

CUDA default behavior

CITA–ICAT

- Kernel calls go into a "pipeline"

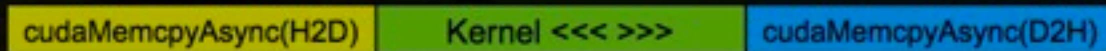
```
func1<<<... >>>(...)
func2<<<...>>>(...)
func3<<<...>>>(...)
```

  - later kernels will only execute when earlier ones complete
  - only *one* kernel executes at a time. Data-transfers are in same pipeline

- low performance, if kernels cannot fill entire GPU
- lost performance, because GPU idle during Memcpy

Concurrency

- **Serial** (1x)



- **2-way concurrency** (up to 2x)

- **3-way concurrency** (up to 3x)

- **4-way concurrency** (3x+)

- **4+ way concurrency**

Slides with black background from NVidia
StreamsAndConcurrencyWebinar.pdf

Synchronous

```
cudaMalloc ( &dev1, size ) ;
double* host1 = (double*) malloc ( &host1, size ) ;
…

cudaMemcpy ( dev1, host1, size, H2D ) ;
kernel2 <<< grid, block, 0 >>> ( …, dev2, … ) ;
kernel3 <<< grid, block, 0 >>> ( …, dev3, … ) ;
cudaMemcpy  ( host4, dev4, size, D2H ) ;
...
```

completely synchronous

- **All CUDA operations in the default stream are synchronous**

Asynchronous between GPU and CPU

```
cudaMalloc ( &dev1, size ) ;
double* host1 = (double*) malloc ( &host1, size ) ;
...

cudaMemcpy ( dev1, host1, size, H2D ) ;
kernel2 <<< grid, block >>> ( ..., dev2, ... ) ;
some_CPU_method ();
kernel3 <<< grid, block >>> ( ..., dev3, ... ) ;
cudaMemcpy ( host4, dev4, size, D2H ) ;
...
```

potentially overlapped

- **GPU kernels are asynchronous with host by default**

❖ kernel3 executes *after* kernel2

Asynchronous kernels: Streams

❖ Each *Stream* is a separate execution pipeline

```
cudaStream_t stream1, stream2, stream3, stream4 ;
cudaStreamCreate ( &stream1) ;
...
cudaMalloc ( &dev1, size ) ;
cudaMallocHost ( &host1, size ) ;                    // pinned memory required on host
...
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;
kernel2 <<< grid, block, 0, stream2 >>> ( …, dev2, … ) ;
kernel3 <<< grid, block, 0, stream3 >>> ( …, dev3, … ) ;
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 ) ;
some_CPU_method ();
...
```

potentially overlapped

- **Fully asynchronous / concurrent**
- **Data used by concurrent operations should be independent**

Synchronization

CITA–ICAT

- **Synchronize everything**
  - **cudaDeviceSynchronize ()**
  - **Blocks host until all issued CUDA calls are complete**
- **Synchronize w.r.t. a specific stream**
  - **cudaStreamSynchronize** (streamid)
  - **Blocks host until all CUDA calls in streamid are complete**
- **Synchronize using Events**
  - **Create specific 'Events', within streams, to use for synchronization**
  - **cudaEventRecord** (event, streamid)
  - **cudaEventSynchronize** (event)
  - **cudaStreamWaitEvent** (stream, event)
  - **cudaEventQuery** (event)

Fineprint: *One* compute engine queue

Stream Scheduling

- Fermi hardware has 3 queues
  - 1 Compute Engine queue
  - 2 Copy Engine queues – one for H2D and one for D2H
- CUDA operations are dispatched to HW in the sequence they were issued
  - Placed in the relevant queue
  - Stream dependencies between engine queues are maintained, but lost within an engine queue
- A CUDA operation is dispatched from the engine queue if:
  - Preceding calls in the same stream have completed,
  - Preceding calls in the same queue have been dispatched, and
  - Resources are available
- CUDA kernels may be executed concurrently if they are in different streams
  - Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have been scheduled and there still are SM resources available
- Note a blocked operation blocks all other operations in the queue, even in other streams

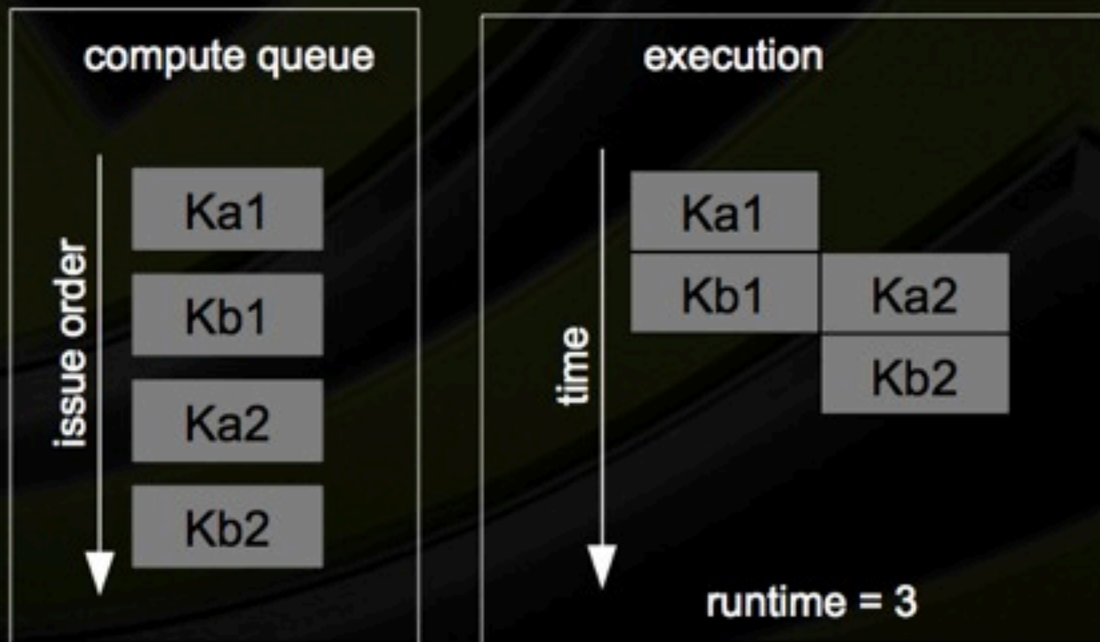
Fineprint: *One* compute engine queue

- The *one* compute-engine queue dispatches kernels *in order*
- Once dispatched, kernels in different streams run in parallel

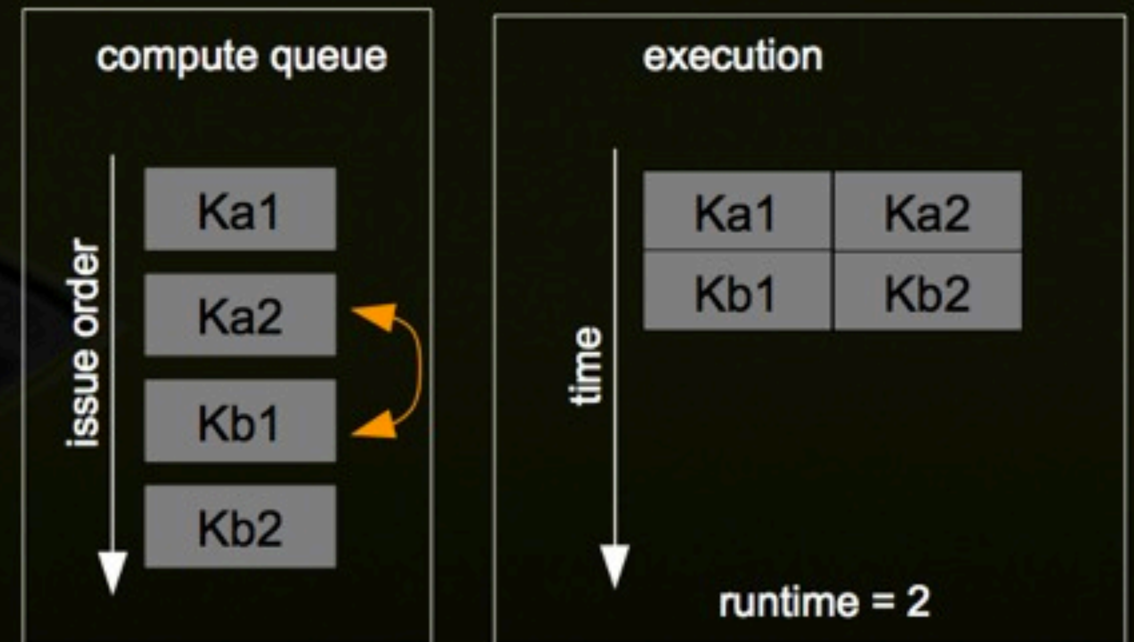
- **Two streams – just issuing CUDA kernels**
  - **Stream 1 : Ka1, Kb1**
  - **Stream 2 : Ka2, Kb2**
  - **Kernels are similar size, fill ½ of the SM resources**

issue order matters!

- **Issue depth first**



- **Issue breadth first**



Homework 5: Streams

- Code two independent kernels that each use approximately half the GPU
- execute them without, and with streams, and observe the speed-up