Contemporary Computer Architecture TDSN13

LECTURE 6 – CUDA MISC

ANDREAS AXELSSON (ANDREAS.AXELSSON@JU.SE)

Developer Tools – Debuggers

Nsight

Nsight Systems

CUDA-GDB

CUDA MEMCHECK

NVIDIA Provided

arm FORGE

TotalView®

3rd Party

https://developer.nvidia.com/debugging-solutions

Compiler Flags

Remember there are two compilers being used

- NVCC: Device code
- Host Compiler: C/C++ code

NVCC supports some host compiler flags

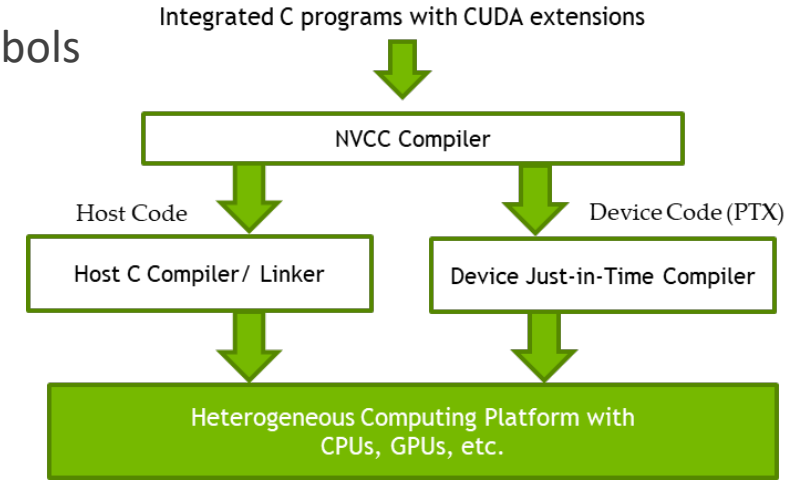
- If flag is unsupported, use –Xcompiler to forward to host
  - e.g. –Xcompiler –fopenmp

Debugging Flags

- -g: Include host debugging symbols
- -G: Include device debugging symbols
- -lineinfo: Include line information with symbols

Useful flag

- nvcc –x cu
  - Treat all input files as .cu files



CUDA-MEMCHECK

Memory debugging tool

- No recompilation necessary

```
%> cuda-memcheck ./exe
```

Can detect the following errors

- Memory leaks
- Memory errors (OOB, misaligned access, illegal instruction, etc)
- Race conditions
- Illegal Barriers
- Uninitialized Memory

For line numbers use the following compiler flags:

- -Xcompiler -rdynamic -lineinfo

http://docs.nvidia.com/cuda/cuda-memcheck

CUDA-GDB

cuda-gdb is an extension of GDB

- Provides seamless debugging of CUDA and CPU code

Works on Linux and Macintosh

- For a Windows debugger use NVIDIA Nsight Eclipse Edition or Visual Studio Edition

Instructions:

1. Run program in cuda-gdb

   %> cuda-gdb --args ./a.out

2. Run a few cuda-gdb commands:

```
(cuda-gdb) b main              //set break point at main
(cuda-gdb) r                       //run application
(cuda-gdb) l                       //print line context
(cuda-gdb) b foo                   //break at kernel foo
(cuda-gdb) c                       //continue
(cuda-gdb) cuda thread         //print current thread
(cuda-gdb) cuda thread 10      //switch to thread 10
(cuda-gdb) cuda block          //print current block
(cuda-gdb) cuda block 1        //switch to block 1
(cuda-gdb) d                       //delete all break points
(cuda-gdb) set cuda memcheck on     //turn on cuda memcheck
(cuda-gdb) r                       //run from the beginning
```

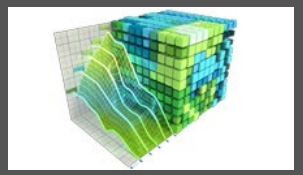
3. Fix Bug

http://docs.nvidia.com/cuda/cuda-gdb

Developer Tools – Profilers

NSIGHT

NVVP

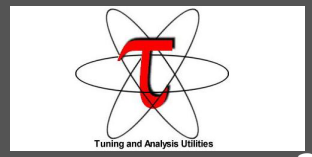


NVPROF

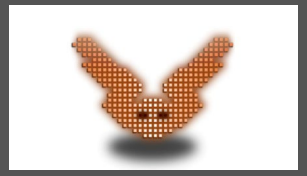
```
==20561== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.88%  866.69ms    504758  1.7170us  1.5040us  2.0160us  void th
int, thrust::detail::device_generate_functor<thrust::detail::fill_
 25.33%  440.05ms     252662  1.7410us  1.5360us  2.3680us  void th
t, thrust::detail::device_generate_functor<thrust::detail::fill_fu
 17.07%  296.60ms       200  1.4830ms  1.2040ms  1.7253ms  kerComp
  2.98%  51.819ms       200  259.09us  246.97us  264.83us  kerMake
  1.16%  20.173ms       501  40.265us     928ns  17.677ms  [CUDA m
  0.93%  16.198ms       200  80.991us  71.840us  90.751us  kerColl
  0.73%  12.636ms       400  31.589us  14.720us  50.432us  [CUDA m
  0.69%  12.075ms       200  60.376us  59.680us  62.304us  kerRena
  0.63%  10.993ms       200  54.963us  52.608us  58.208us  kerMake
  0.32%  5.5524ms       200  27.761us  22.559us  33.152us  [CUDA m
  0.12%  2.1342ms         1  2.1342ms  2.1342ms  2.1342ms  void th
```

NVIDIA Provided

TAU



VampirTrace



3rd Party

https://developer.nvidia.com/performance-analysis-tools

NVPROF

Command Line Profiler

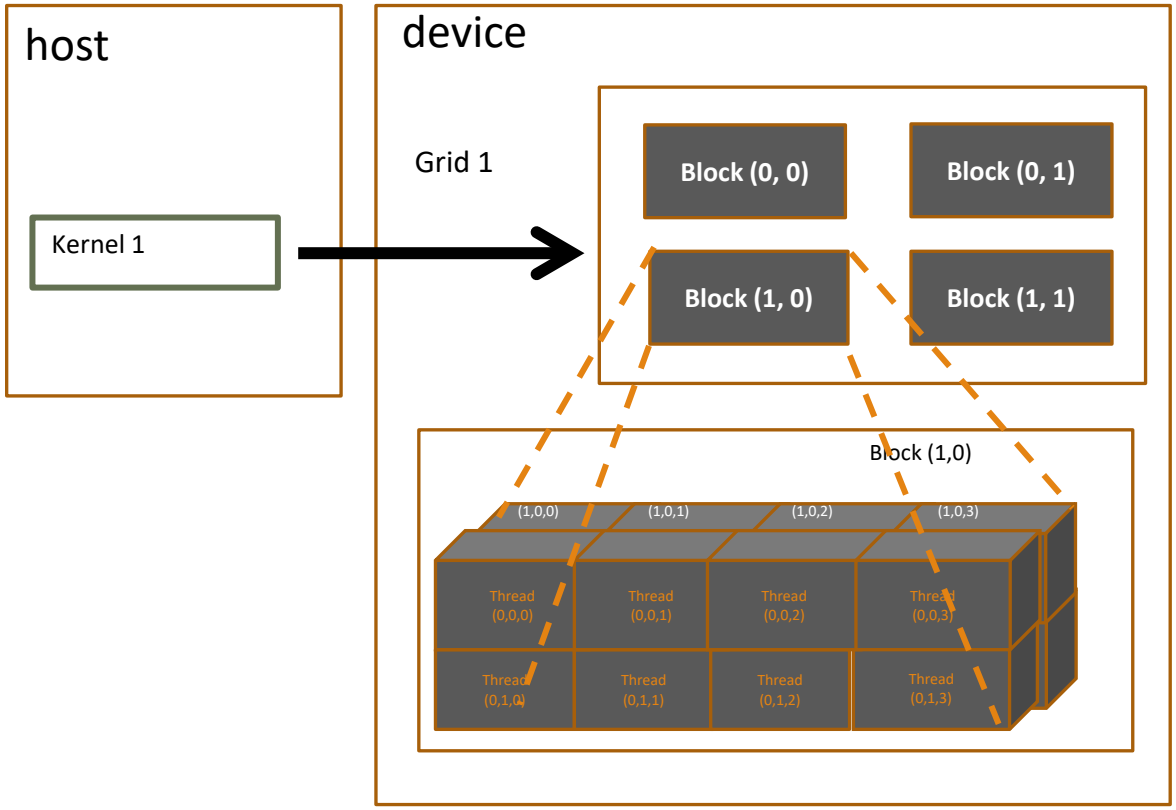
- Compute time in each kernel
- Compute memory transfer time
- Collect metrics and events
- Support complex process hierarchy's
- Collect profiles for NVIDIA Visual Profiler
- No need to recompile

CUDA Function declaration again

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- **`__global__`** defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return **`void`**
- **`__device__`** and **`__host__`** can be used together
- **`__host__`** is optional if used alone

Multi-Dimensional Grid Example



2D Access Pattern

```
__global__
 void PictureKernel(float * in, float * out, int w, int h)
{
  int Col  = blockIdx.x * blockDim.x + threadIdx.x;
  int Row  = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
     ... // Rest of our kernel
  }
}
```

Source code of PictureKernel

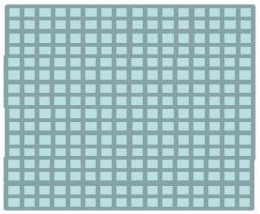
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                              int height, int width)
{

  // Calculate the row # of the d_Pin and d_Pout element
  int Row = blockIdx.y*blockDim.y + threadIdx.y;

  // Calculate the column # of the d_Pin and d_Pout element
  int Col = blockIdx.x*blockDim.x + threadIdx.x;

  // each thread computes one element of d_Pout if in range
  if ((Row < height) && (Col < width)) {
    d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
  }
}
```

Scale every pixel value by 2.0

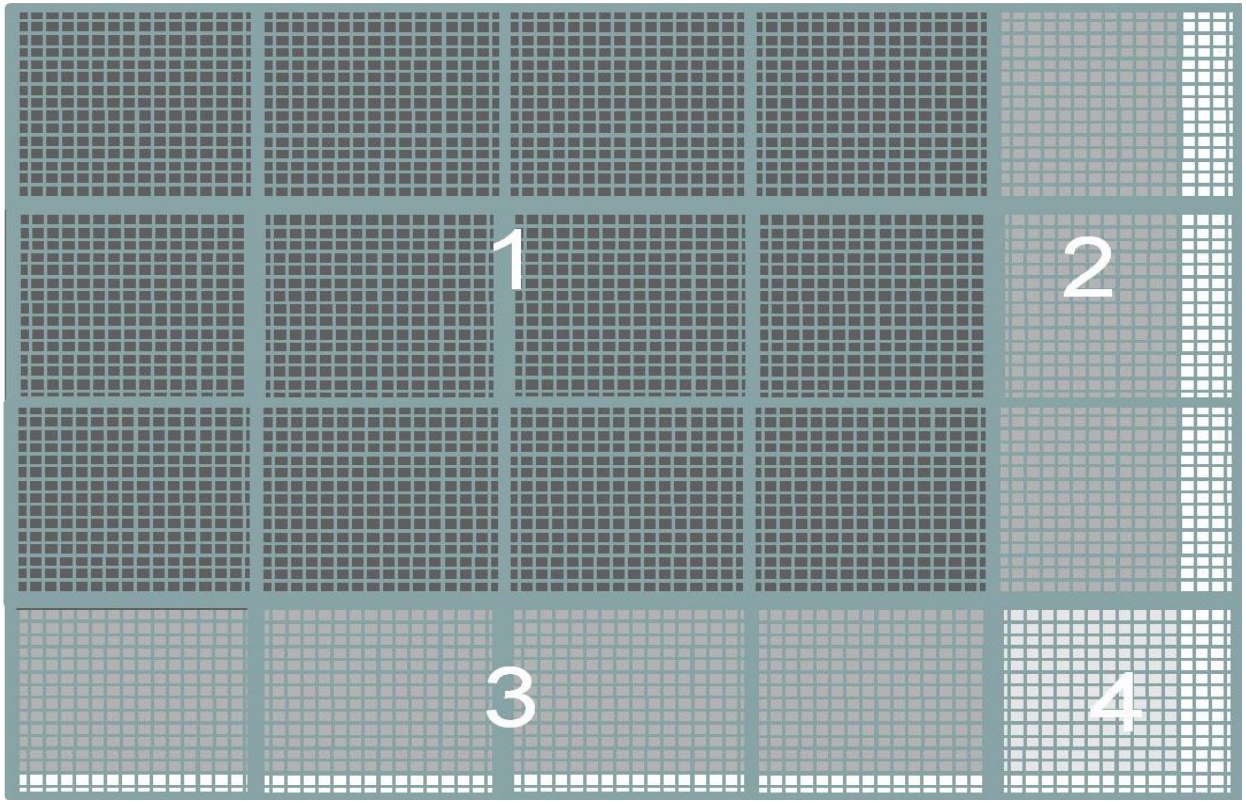
Launching PictureKernel

```
// assume that the picture is m X n pixels,
// m pixels in y dimension and n pixels in x dimension
// input d_Pin has been allocated on and copied to device
// output d_Pout has been allocated on device
…
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
…
```

62×76 Picture with 16×16 Blocks



16×16 block

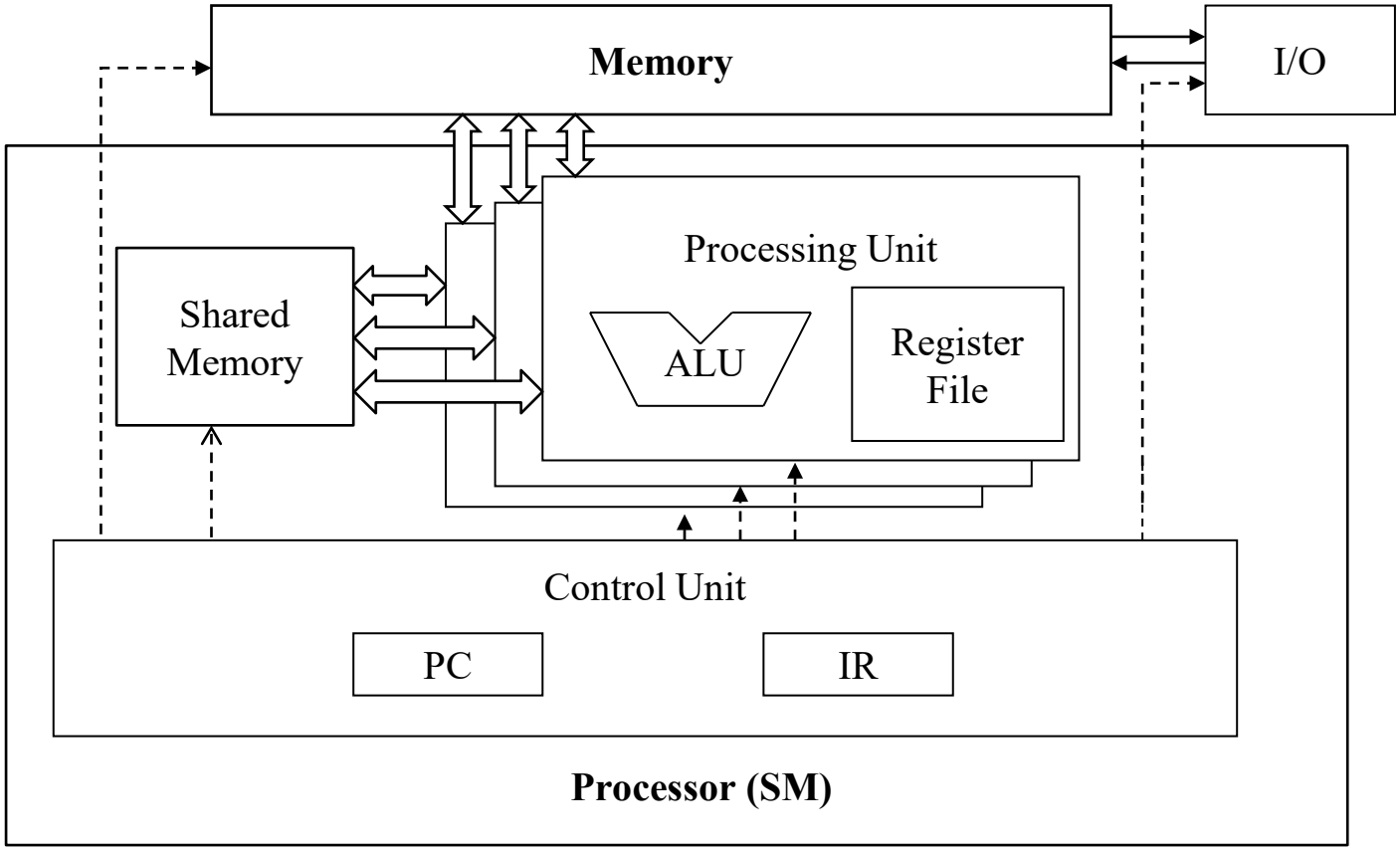


Not all threads in a Block will follow the same control flow path.
Will we suffer from warp divergence?

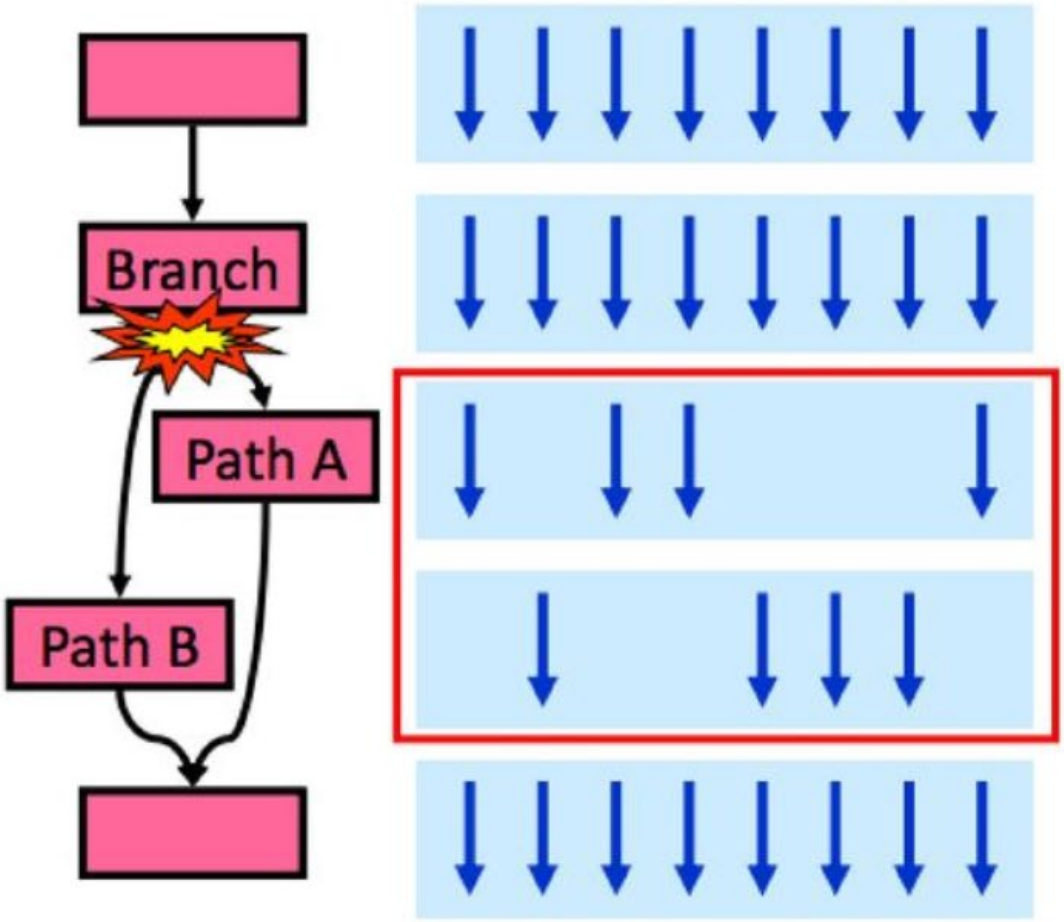
SMs are SIMD Processors

Control unit for instruction fetch, decode, and control is shared among multiple processing units

- Control overhead is minimized (Module 1)



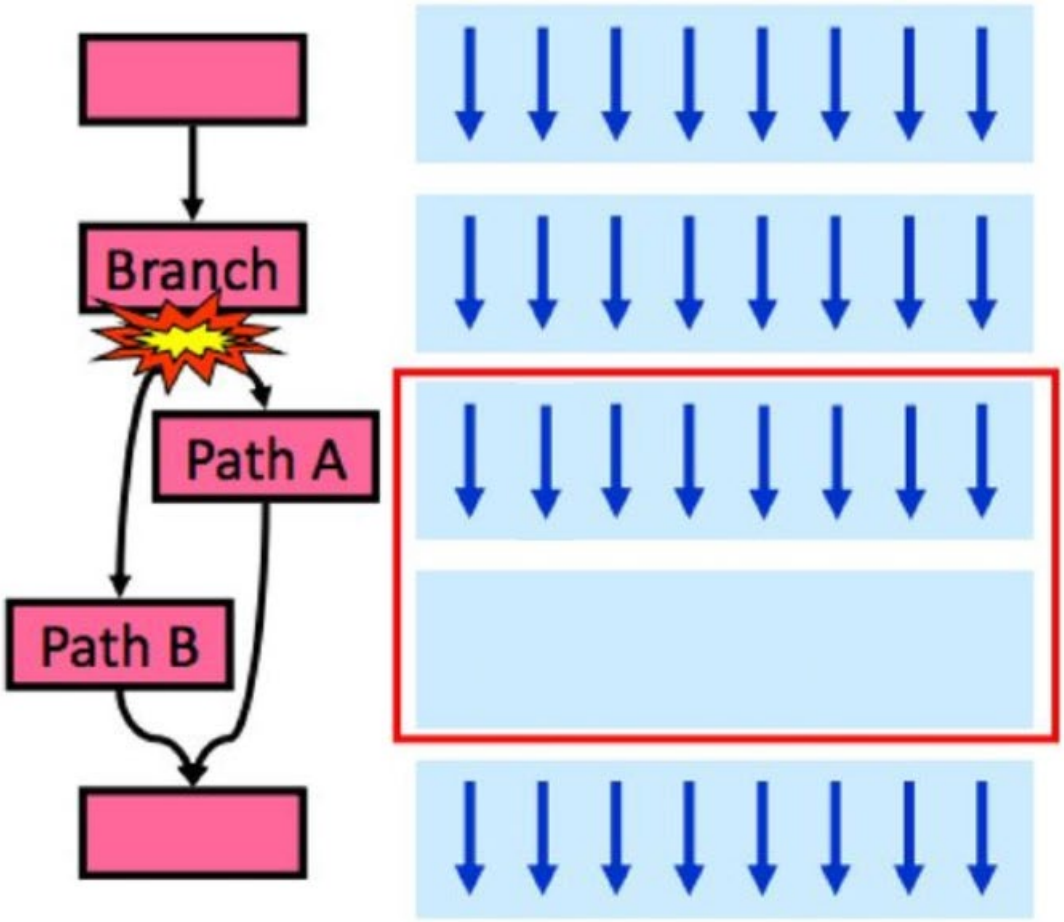
Warp Divergence



50% performance hit!

https://slideplayer.com/slide/12245403/

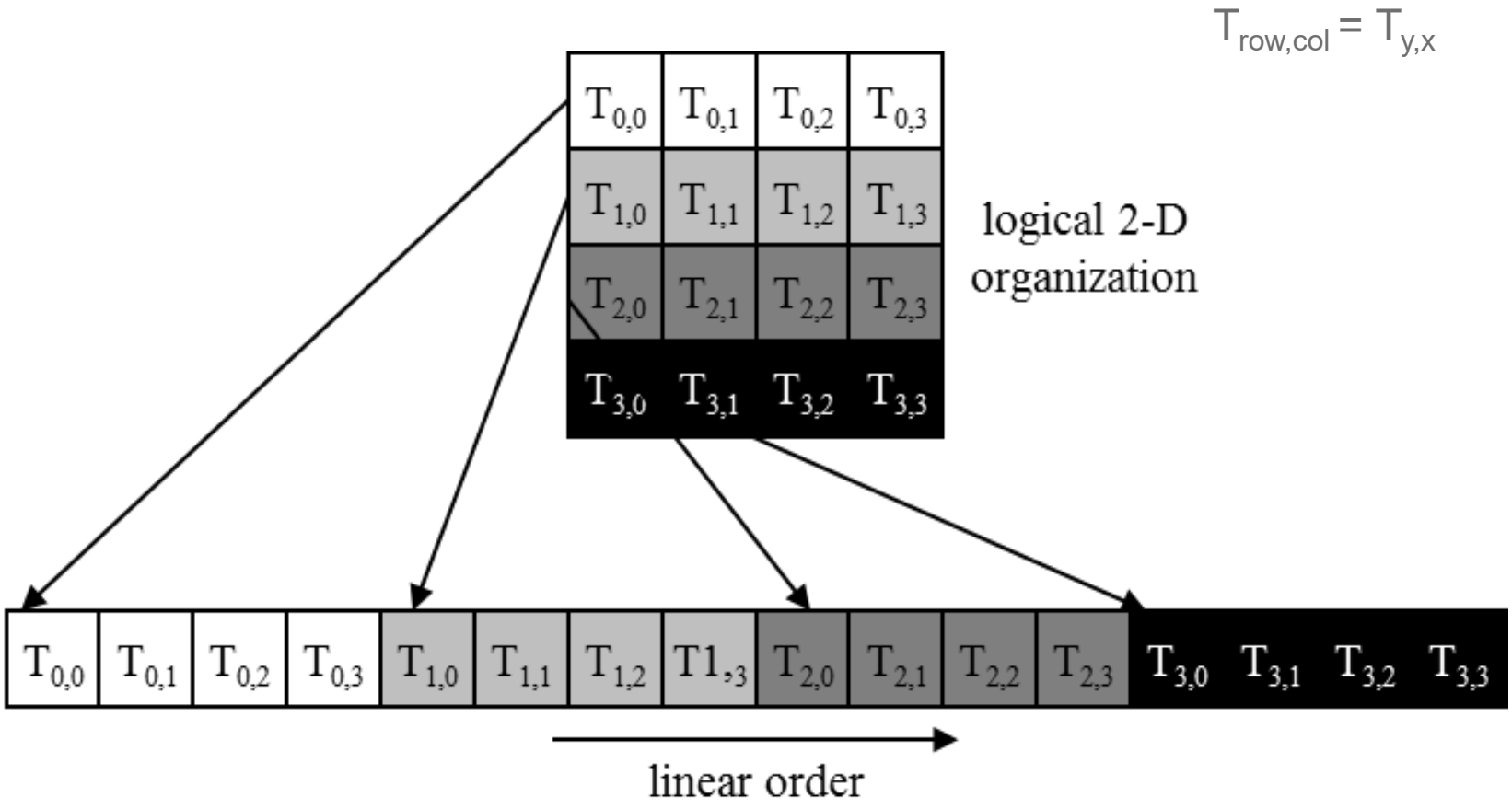
Warp Divergence



0% performance hit!

Warps in Multi-Dim Thread Blocks

- The thread blocks are first linearized into 1D in row major order
  - In x-dimension first, y-dimension next, and z-dimension last



Warps in Multi-Dim Thread Blocks

Linearized thread blocks are partitioned

- Thread indices within a warp are consecutive and increasing
- Warp 0 starts with Thread 0

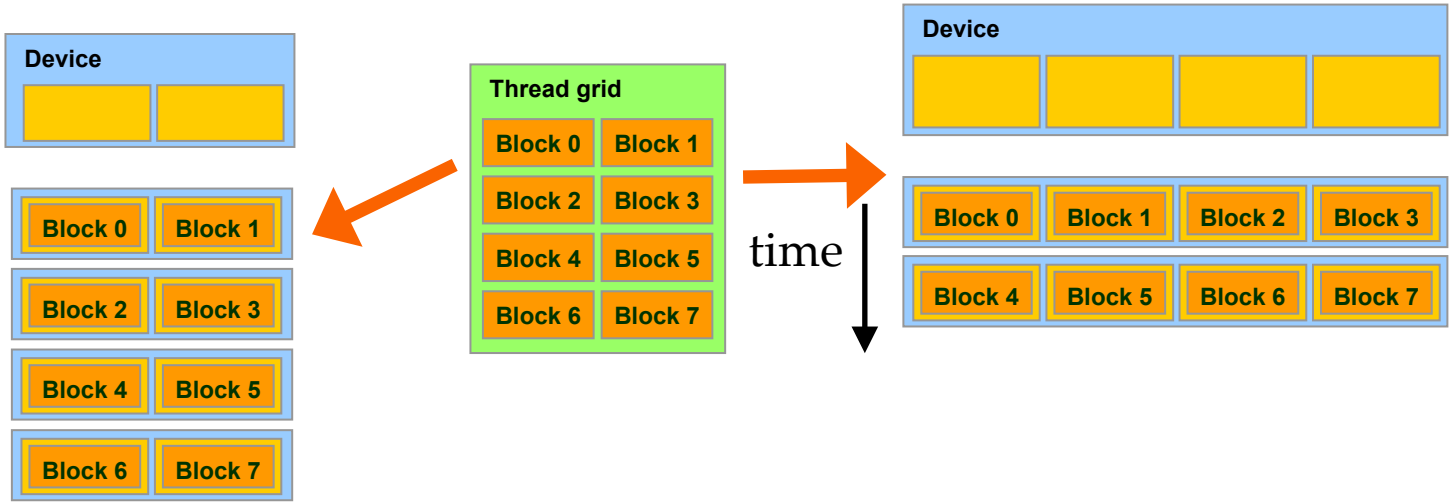
Partitioning scheme is consistent across devices

- Thus, you can use this knowledge in control flow
- However, the exact size of warps may change from generation to generation

DO NOT rely on any ordering within or between warps

- If there are any dependencies between threads, you must __*syncthreads()* to get correct results (more later).

Transparent Scalability (Grids)



- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of parallel processors

GPU Performance Consideration

Assume a more complex algorithm using pixel values many times

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 1,600 GFLOPS with 600 GB/s DRAM bandwidth
- 4*1,600 = 6,400 GB/s required to achieve peak FLOPS rating
- The 600 GB/s memory bandwidth limits the execution at 150 GFLOPS

This limits the execution rate to 9.3% (150/1600) of the peak floating-point execution rate of the device!

Need to drastically cut down memory accesses to get close to the 1,600 GFLOPS

Shared memory??

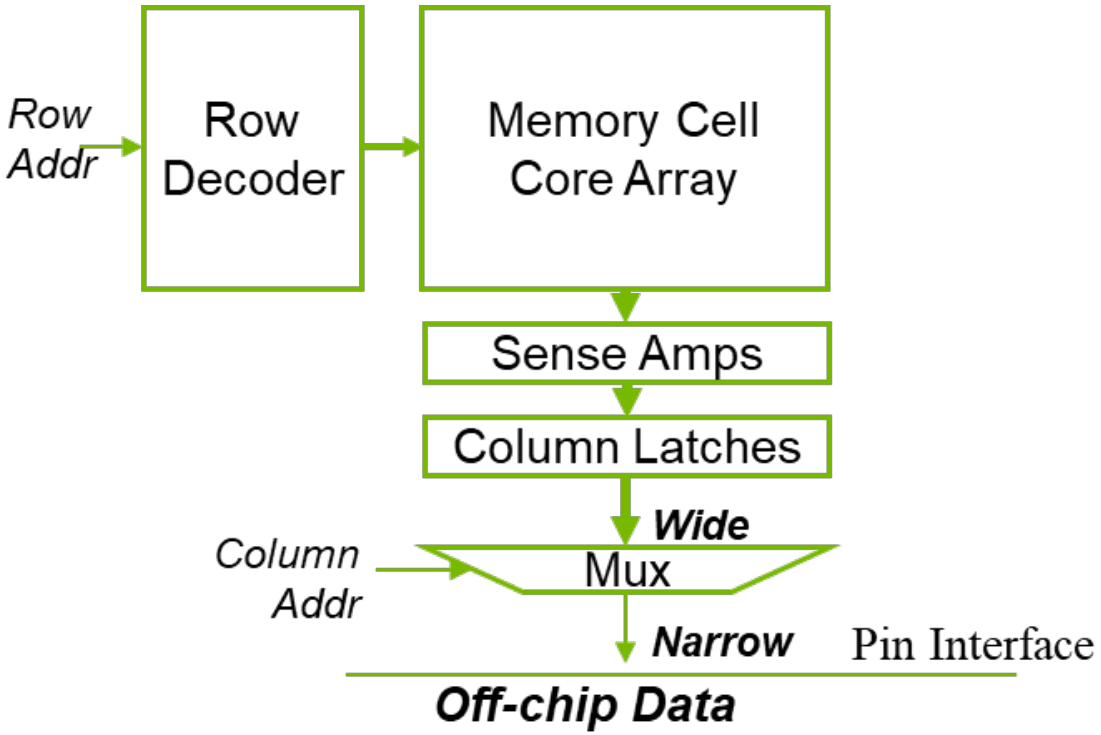
Global Memory DRAM Bandwidth

Ideal

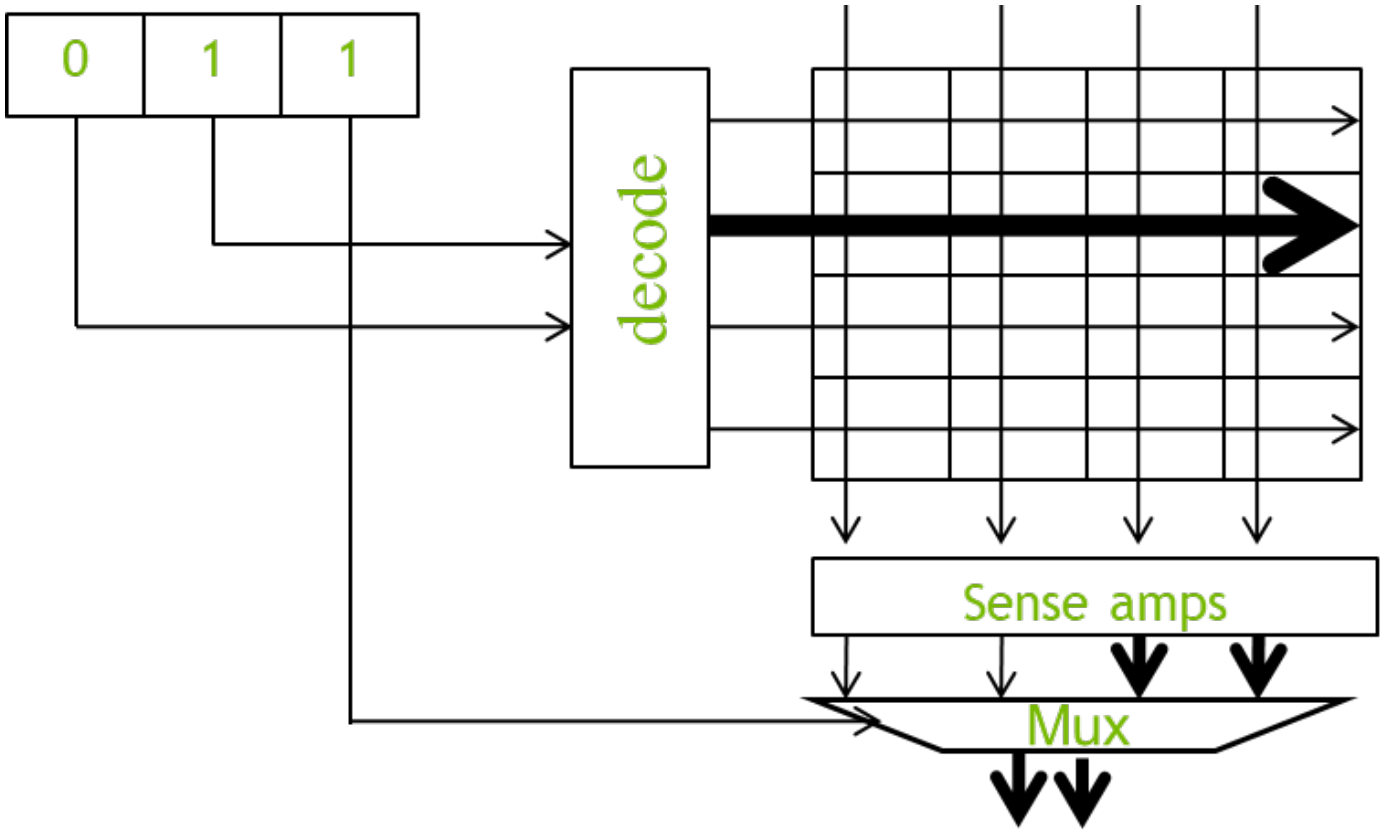
Reality

DRAM Core Array Organization

- Each DRAM core array has about 16M bits
- Each bit is stored in a tiny capacitor made of one transistor

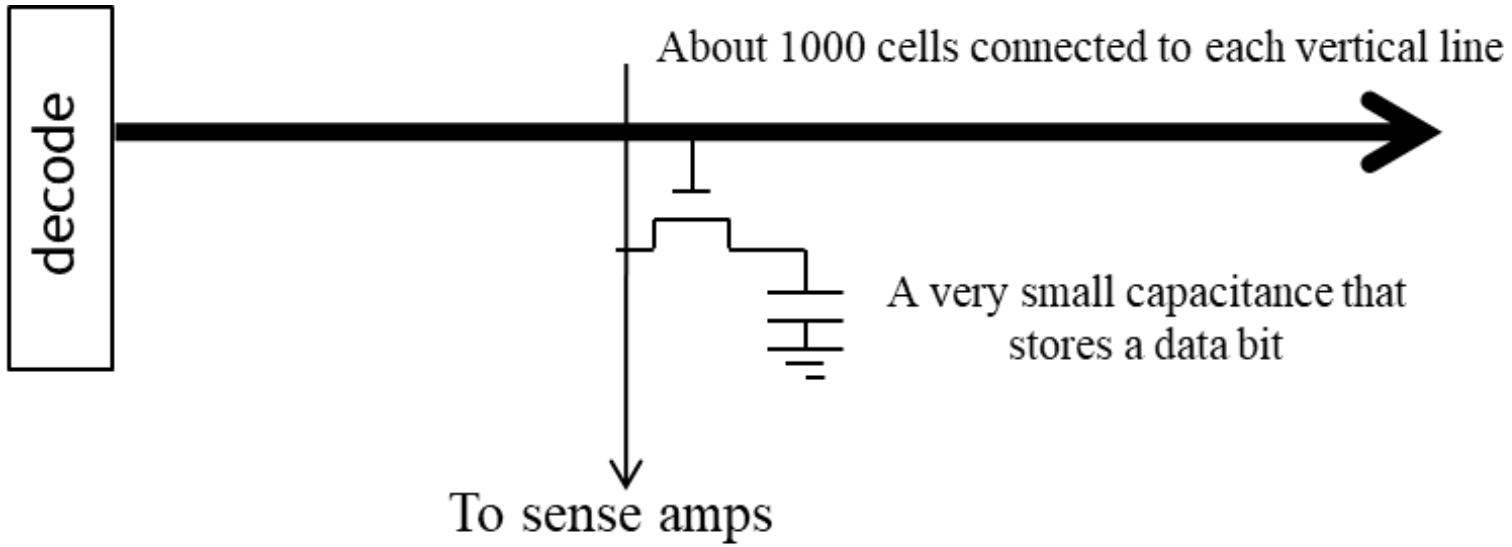


A very small (8x2-bit) DRAM Array



DRAM Core Arrays are Slow

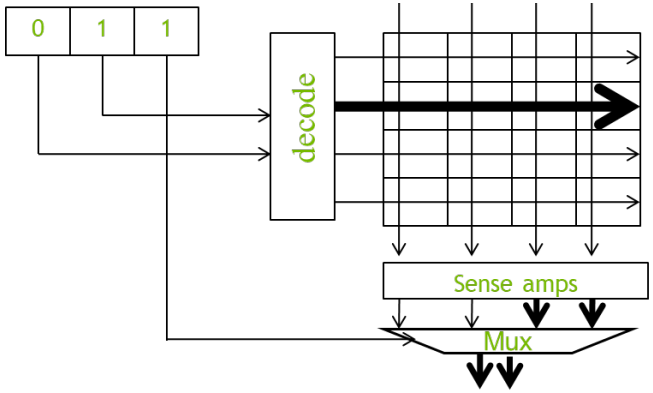
- Reading from a cell in the core array is a very slow process
  - DDR: Core speed = ½ interface speed
  - DDR2/GDDR3: Core speed = ¼ interface speed
  - DDR3/GDDR4: Core speed = ⅛ interface speed
  - … likely to be worse in the future



DRAM Bursting

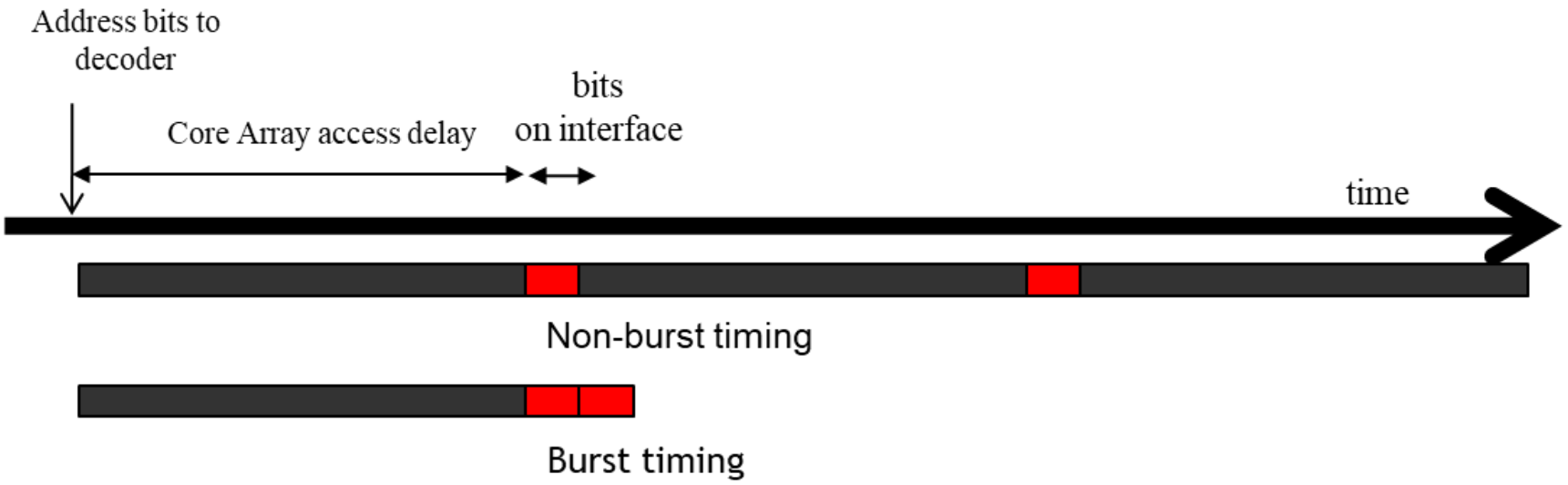
For DDR{2,3} SDRAM cores clocked at 1/N speed of the interface:

- Load (N × interface width) of DRAM bits from the same row at once to an internal buffer, then transfer in N steps at interface speed
- DDR3/GDDR4: buffer width = 8X interface width



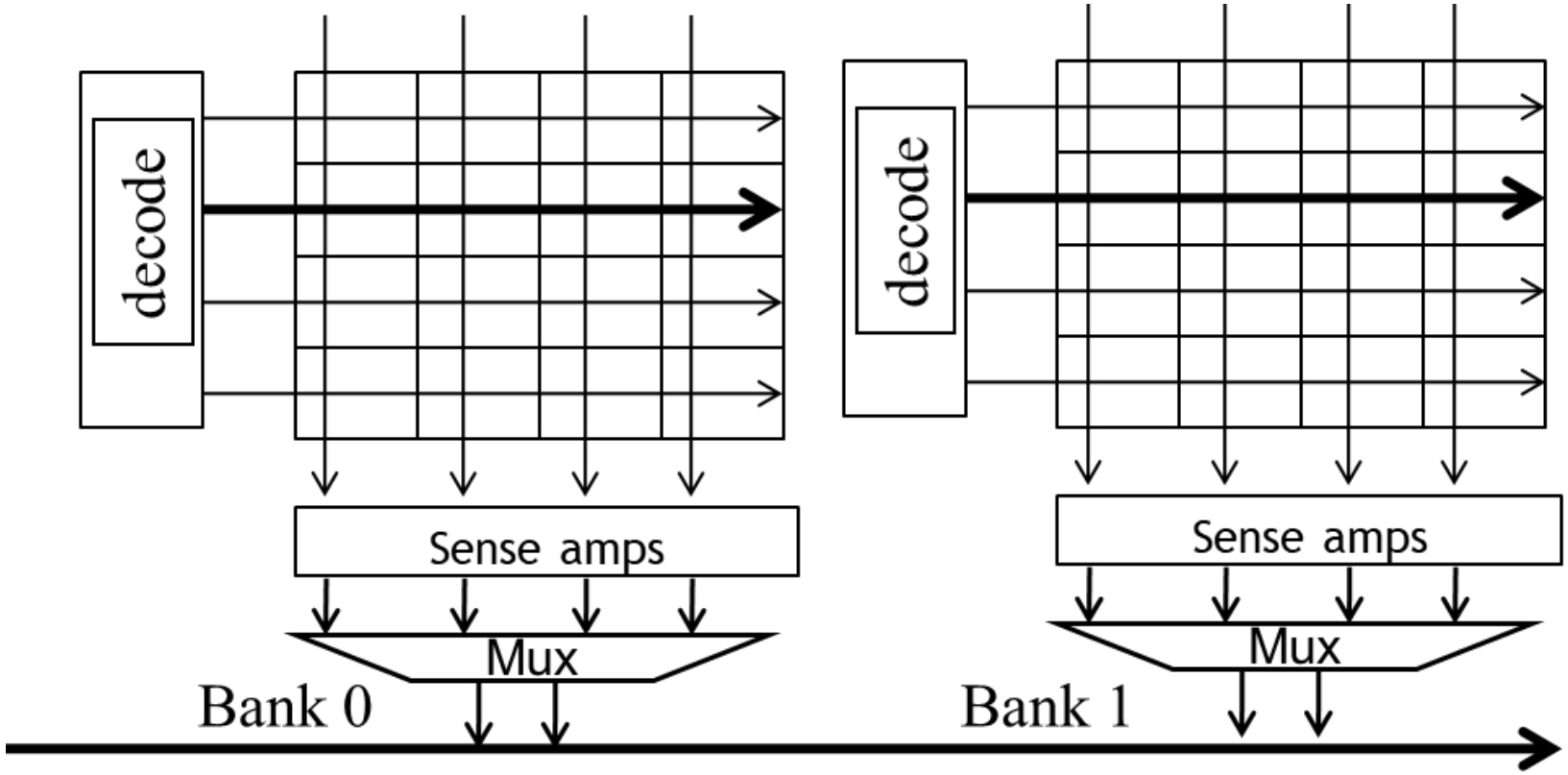
Add internal buffer

DRAM Bursting Timing Example



Modern DRAM systems are designed to always be accessed in burst mode. Burst bytes are transferred to the processor but discarded when accesses are not to sequential locations.

Multiple DRAM Banks



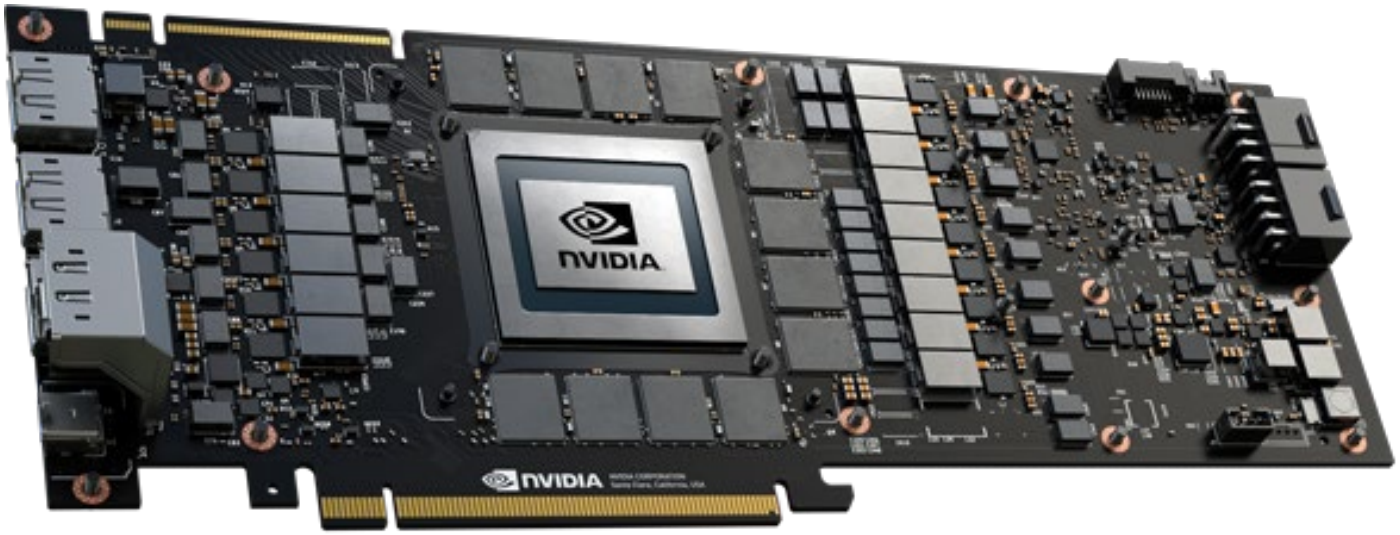
DRAM Bursting with Banking

Single-Bank burst timing, dead time on interface

Multi-Bank burst timing, reduced dead time

GPU off-chip memory subsystem

- NVIDIA RTX6000 GPU:
  - Peak global memory bandwidth = 672GB/s
- Global memory (GDDR6) clocked @ 1750 MHz
  - 14 Gbps pin speed (SERDES x 8)
  - For GDDR6 32-bit interface, we can sustain only about 56 GB/s
  - We need a lot more bandwidth (672 GB/s) – thus 12 memory channels



Introduction to CUDA

Questions?

Contact information

Andreas Axelsson

Email: andreas.axelsson@ju.se

Mobile: 0709-467760