Contemporary Computer Architecture TDSN13

LECTURE 4 – INTRODUCTION TO CUDA

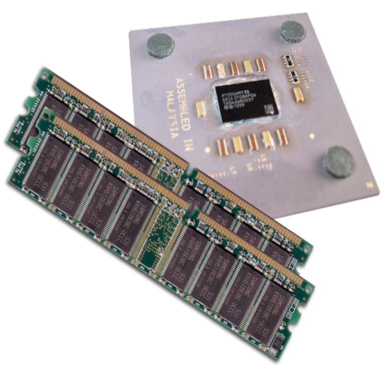
ANDREAS AXELSSON (ANDREAS.AXELSSON@JU.SE)

Introduction to CUDA

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.
- How to get started

Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host

Device

CUDA Parallel Computing Platform

Programming Approaches

Libraries – "Drop-in" Acceleration

OpenACC Directives – Easily Accelerate Apps

Programming Languages – Maximum Flexibility

Development Environment

Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

Open Compiler Tool Chain

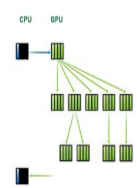
LLVM Compiler Infrastructure

Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

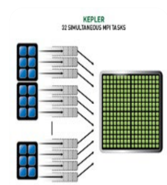
Hardware Capabilities

SMX

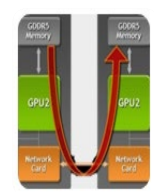
Dynamic Parallelism



HyperQ

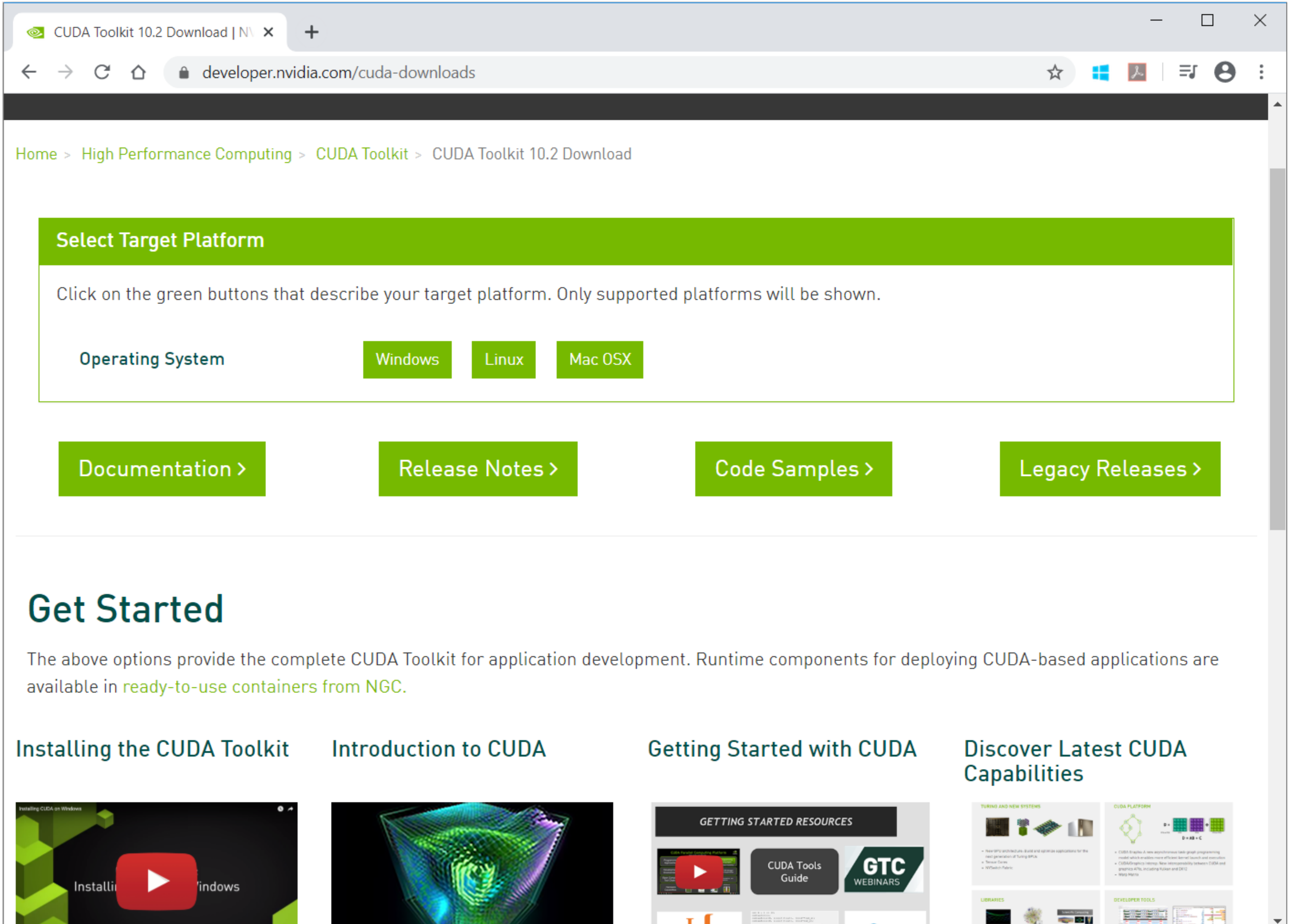


GPUDirect



CUDA GPU Programming

Fortran	CUDA Fortran
C	CUDA C
C++	CUDA C++
Python	PyCUDA



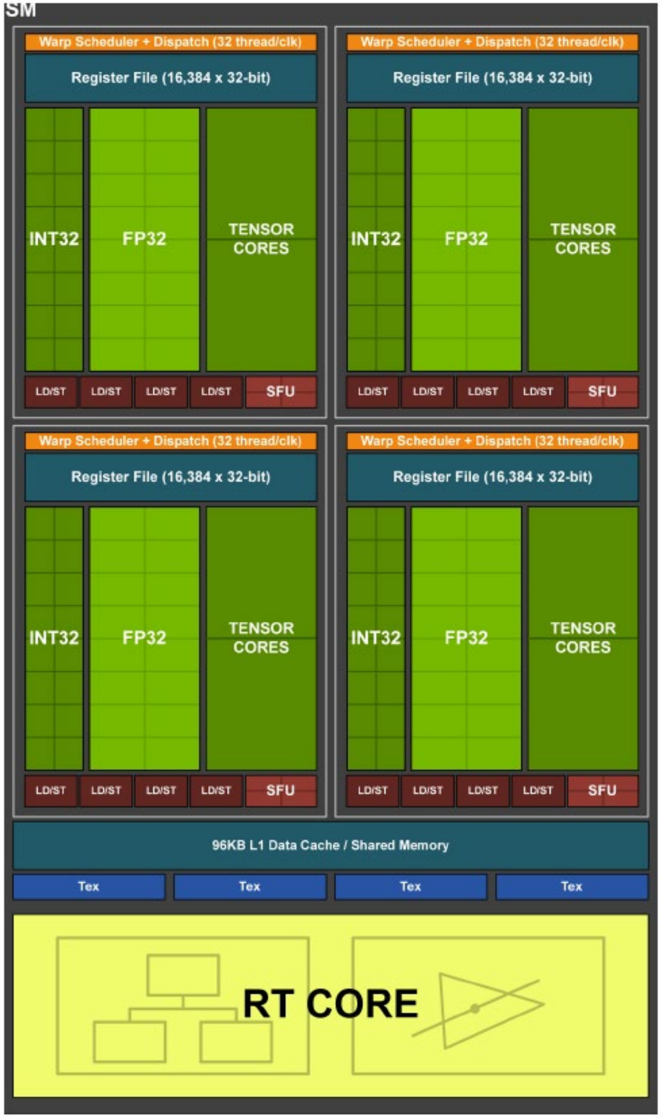
CUDA Toolkit 10.2 Download | N\
developer.nvidia.com/cuda-downloads
Home > High Performance Computing > CUDA Toolkit > CUDA Toolkit 10.2 Download
Select Target Platform
Click on the green buttons that describe your target platform. Only supported platforms will be shown.
Operating System
Windows
Linux
Mac OSX
Documentation >
Release Notes >
Code Samples >
Legacy Releases >
Get Started
The above options provide the complete CUDA Toolkit for application development. Runtime components for deploying CUDA-based applications are available in ready-to-use containers from NGC.
Installing the CUDA Toolkit
Introduction to CUDA
Getting Started with CUDA
Discover Latest CUDA Capabilities
GETTING STARTED RESOURCES
CUDA Tools Guide
GTC
WEBINARS

Nvidia – GPU Architectures

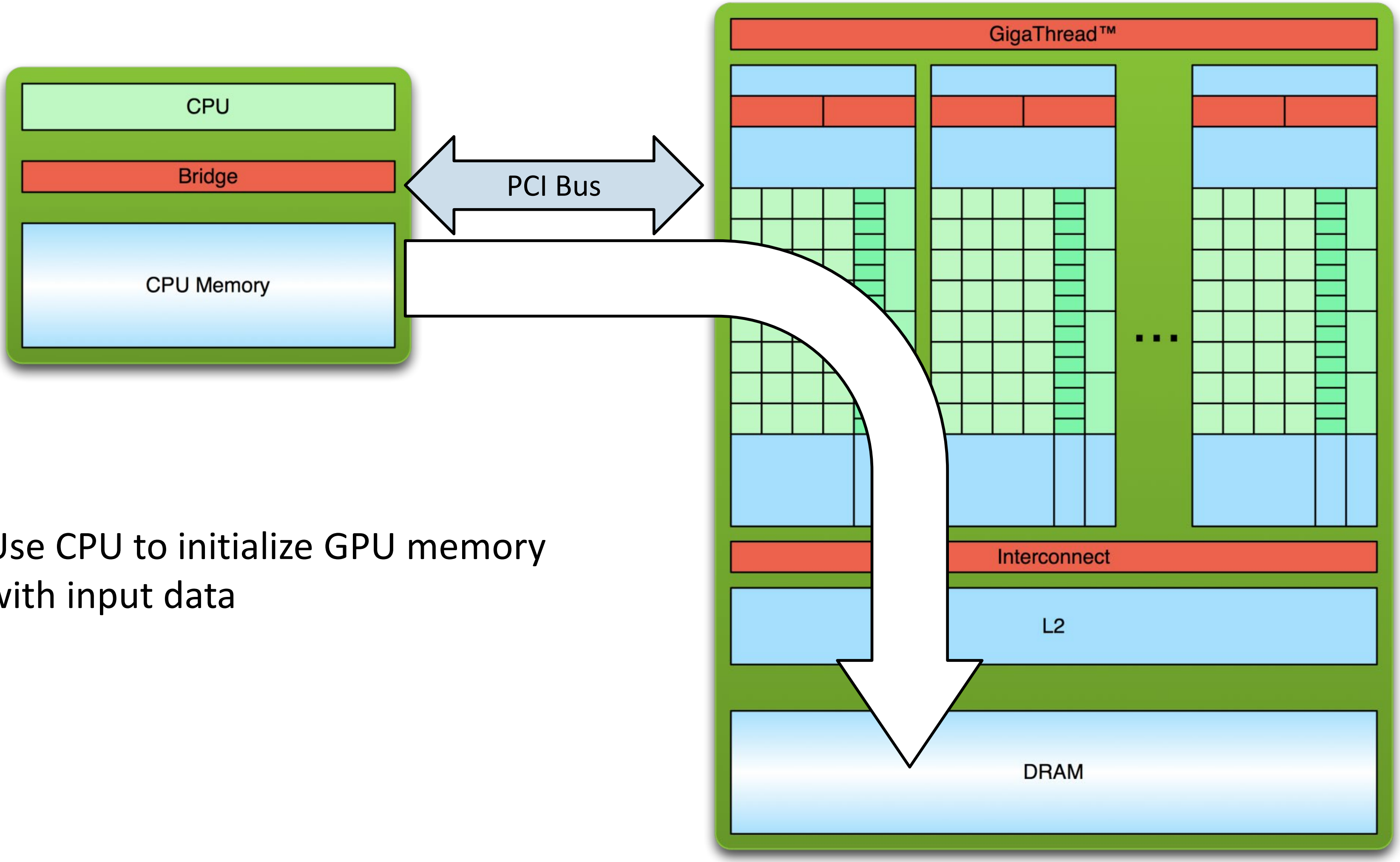
- Tesla
- Fermi
- Kepler
- Maxwell
- Pascal
- Volta
- Turing
- Ampere?

Turing Architecture

Turing Streaming Multiprocessor

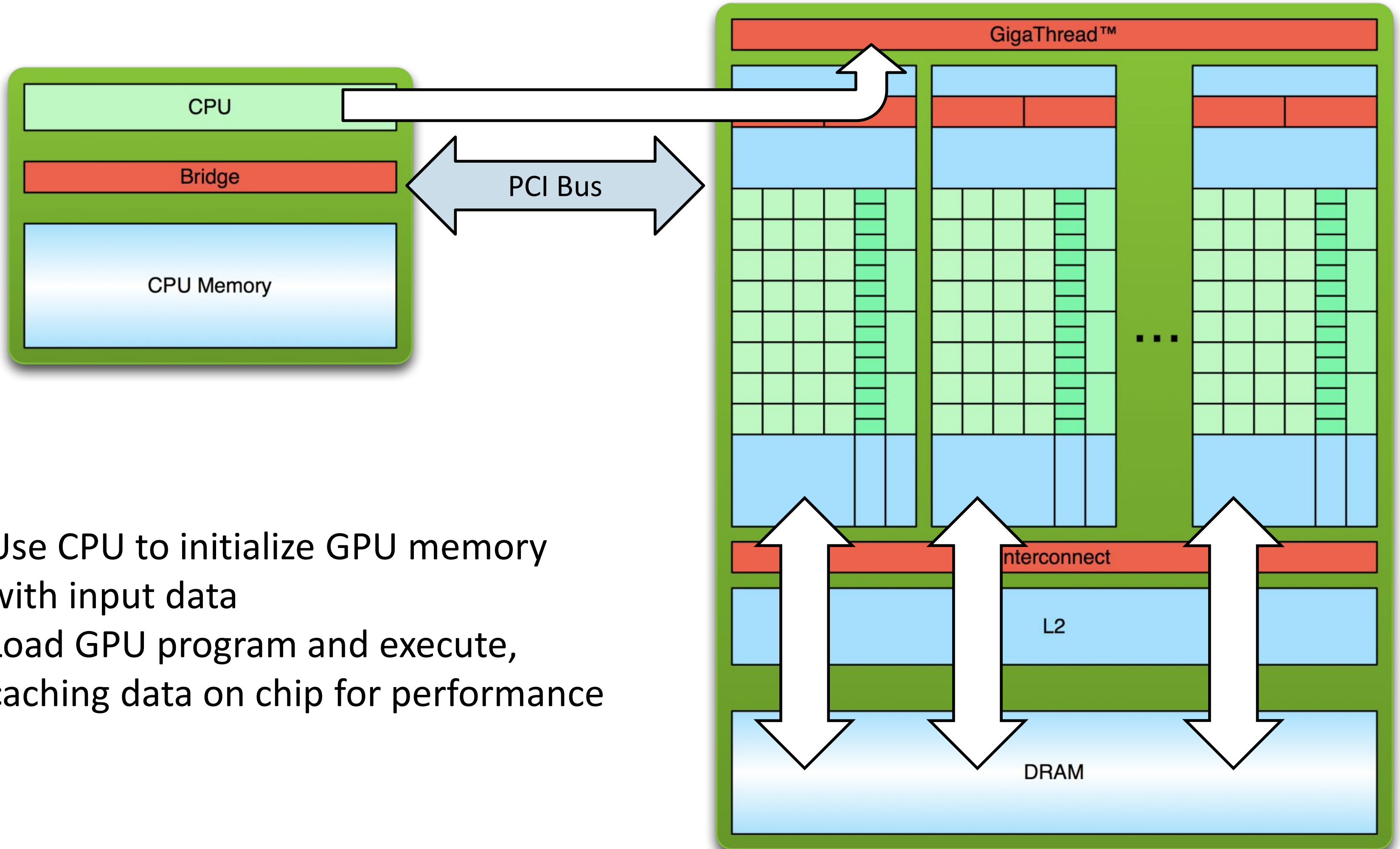


Workflow



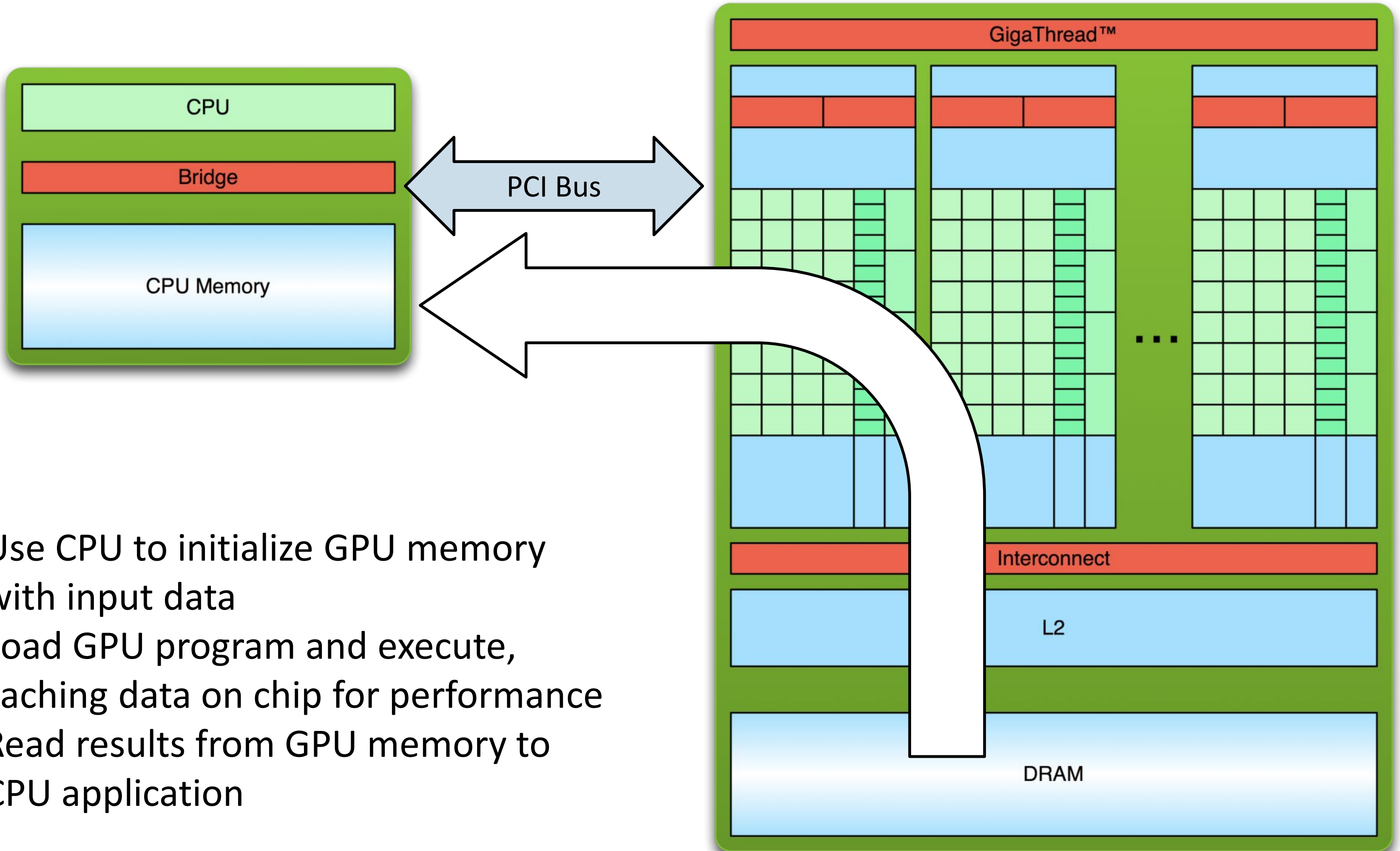
1. Use CPU to initialize GPU memory with input data

Workflow



1. Use CPU to initialize GPU memory with input data
2. Load GPU program and execute, caching data on chip for performance

Workflow



1. Use CPU to initialize GPU memory with input data
2. Load GPU program and execute, caching data on chip for performance
3. Read results from GPU memory to CPU application

Hello World!

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc
hello_world.
cu
$ a.out
Hello World!
$
```

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

- Two new syntactic elements…

Hello World! with Device Code

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

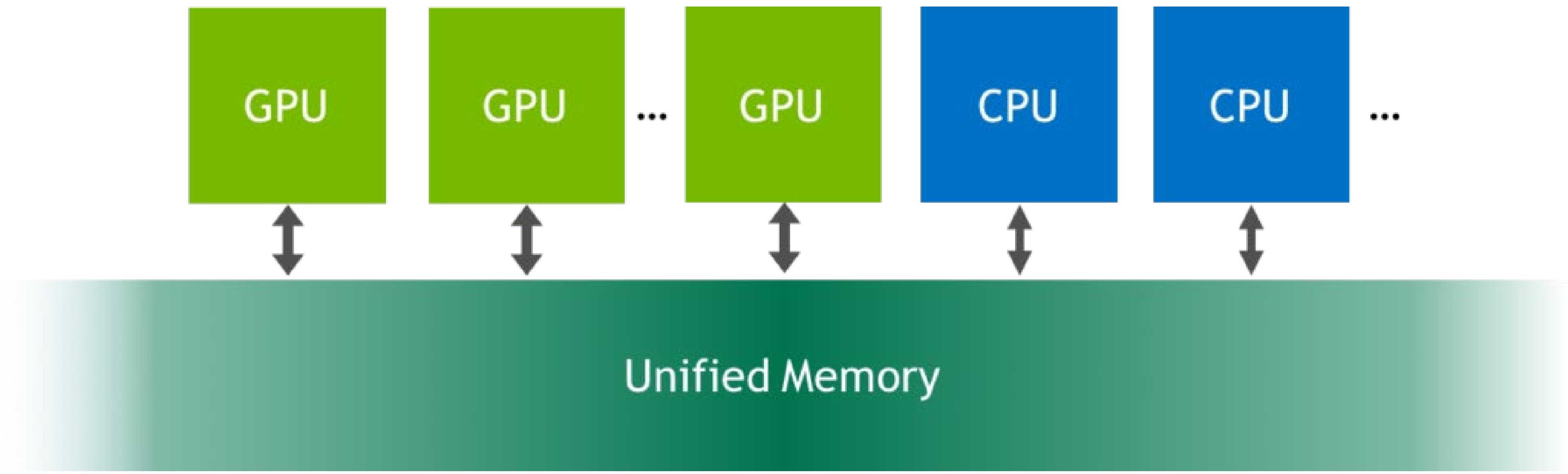
Hello World! with Device Code

```
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - **`gcc`, `cl.exe`**

Unified Memory




```
__host__cudaError_t cudaMallocManaged ( void** devPtr, size_t size, unsigned int  flags = cudaMemAttachGlobal )
```

Allocates memory that will be automatically managed by the Unified Memory system.

Use the same address pointer in CPU and GPU

https://devblogs.nvidia.com/unified-memory-cuda-beginners/

Unified Memory

Windows PowerShell

```
PS C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.2\bin\win64\Release> .\bandwidthTest.exe
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Quadro P1000
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     13.0

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     12.9

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     82.1

Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost
 is enabled.
PS C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.2\bin\win64\Release> _
```

Compute Capability

CUDA-Enabled Jetson Products

Jetson Products

GPU	Compute Capability
Jetson AGX Xavier	7.2
Jetson Nano	5.3
Jetson TX2	6.2
Jetson TX1	5.3
Tegra X1	5.3

Compute Capability

https://en.wikipedia.org/wiki/CUDA

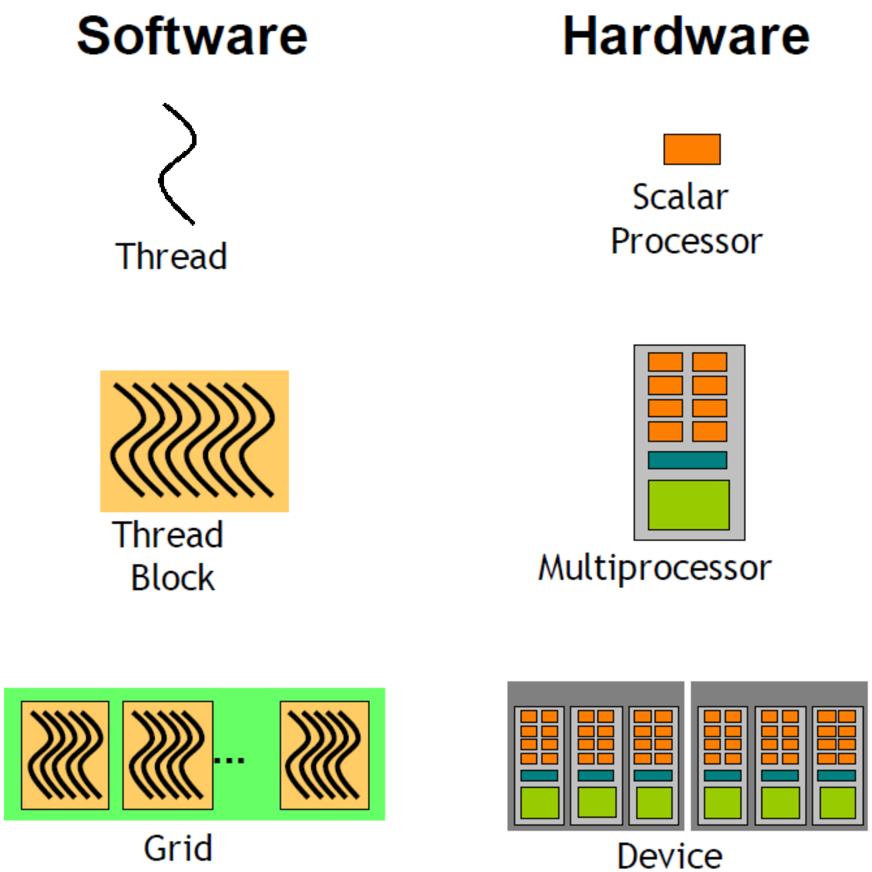
<table>
<tr><th rowspan="2">Technical specifications</th><th colspan="17">Compute capability (version)</th></tr>
<tr><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th><th>3.0</th><th>3.2</th><th>3.5</th><th>3.7</th><th>5.0</th><th>5.2</th><th>5.3</th><th>6.0</th><th>6.1</th><th>6.2</th><th>7.0 (7.2?)</th><th>7.5</th></tr>
<tr><td>Maximum number of resident grids per device (concurrent kernel execution)</td><td colspan="4">t.b.d.</td><td colspan="2">16</td><td>4</td><td colspan="4">32</td><td>16</td><td>128</td><td>32</td><td>16</td><td colspan="2">128</td></tr>
<tr><td>Maximum dimensionality of grid of thread blocks</td><td colspan="4">2</td><td colspan="13">3</td></tr>
<tr><td>Maximum x-dimension of a grid of thread blocks</td><td colspan="5">65535</td><td colspan="12">$2^{31} - 1$</td></tr>
<tr><td>Maximum y-, or z-dimension of a grid of thread blocks</td><td colspan="17">65535</td></tr>
<tr><td>Maximum dimensionality of thread block</td><td colspan="17">3</td></tr>
<tr><td>Maximum x- or y-dimension of a block</td><td colspan="4">512</td><td colspan="13">1024</td></tr>
<tr><td>Maximum z-dimension of a block</td><td colspan="17">64</td></tr>
<tr><td>Maximum number of threads per block</td><td colspan="4">512</td><td colspan="13">1024</td></tr>
<tr><td>Warp size</td><td colspan="17">32</td></tr>
<tr><td>Maximum number of resident blocks per multiprocessor</td><td colspan="5">8</td><td colspan="4">16</td><td colspan="7">32</td><td>16</td></tr>
<tr><td>Maximum number of resident warps per multiprocessor</td><td colspan="2">24</td><td colspan="2">32</td><td>48</td><td colspan="11">64</td><td>32</td></tr>
<tr><td>Maximum number of resident threads per multiprocessor</td><td colspan="2">768</td><td colspan="2">1024</td><td>1536</td><td colspan="11">2048</td><td>1024</td></tr>
<tr><td>Number of 32-bit registers per multiprocessor</td><td colspan="2">8 K</td><td colspan="2">16 K</td><td>32 K</td><td colspan="3">64 K</td><td>128 K</td><td colspan="8">64 K</td></tr>
<tr><td>Maximum number of 32-bit registers per thread block</td><td colspan="4">N/A</td><td>32 K</td><td>64 K</td><td>32 K</td><td colspan="4">64 K</td><td>32 K</td><td colspan="2">64 K</td><td>32 K</td><td colspan="2">64 K</td></tr>
<tr><td>Maximum number of 32-bit registers per thread</td><td colspan="4">124</td><td colspan="2">63</td><td colspan="11">255</td></tr>
<tr><td>Maximum amount of shared memory per multiprocessor</td><td colspan="4">16 KB</td><td colspan="4">48 KB</td><td>112 KB</td><td>64 KB</td><td>96 KB</td><td colspan="2">64 KB</td><td>96 KB</td><td>64 KB</td><td>96 KB (of 128)</td><td>64 KB (of 96)</td></tr>
<tr><td>Maximum amount of shared memory per thread block</td><td colspan="15">48 KB</td><td>48/96 KB</td><td>64 KB</td></tr>
<tr><td>Number of shared memory banks</td><td colspan="4">16</td><td colspan="13">32</td></tr>
<tr><td>Amount of local memory per thread</td><td colspan="4">16 KB</td><td colspan="13">512 KB</td></tr>
<tr><td>Constant memory size</td><td colspan="17">64 KB</td></tr>
<tr><td>Cache working set per multiprocessor for constant memory</td><td colspan="12">8 KB</td><td>4 KB</td><td colspan="4">8 KB</td></tr>
<tr><td>Cache working set per multiprocessor for texture memory</td><td colspan="4">6 – 8 KB</td><td colspan="2">12 KB</td><td colspan="3">12 – 48 KB</td><td>24 KB</td><td>48 KB</td><td>N/A</td><td>24 KB</td><td>48 KB</td><td>24 KB</td><td>32 – 128 KB</td><td>32 – 64 KB</td></tr>
<tr><td>Maximum width for 1D texture reference bound to a CUDA array</td><td colspan="4">8192</td><td colspan="13">65536</td></tr>
<tr><td>Maximum width for 1D texture reference bound to linear memory</td><td colspan="17">2^{27}</td></tr>
<tr><td>Maximum width and number of layers for a 1D layered texture reference</td><td colspan="4">8192 × 512</td><td colspan="13">16384 × 2048</td></tr>
<tr><td>Maximum width and height for 2D texture reference bound to a CUDA array</td><td colspan="4">65536 × 32768</td><td colspan="13">65536 × 65535</td></tr>
</table>

CUDA Device Query

Windows PowerShell

```
PS C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.2\bin\win64\Release> .\deviceQuery.exe
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.2\bin\win64\Release\deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Quadro P1000"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 4096 MBytes (4294967296 bytes)
  ( 4) Multiprocessors, (128) CUDA Cores/MP:     512 CUDA Cores
  GPU Max Clock rate:                            1519 MHz (1.52 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 5 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.2, CUDA Runtime Version = 10.2, NumDevs = 1
Result = PASS
PS C:\ProgramData\NVIDIA Corporation\CUDA Samples\v10.2\bin\win64\Release> _
```

Execution Model



Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

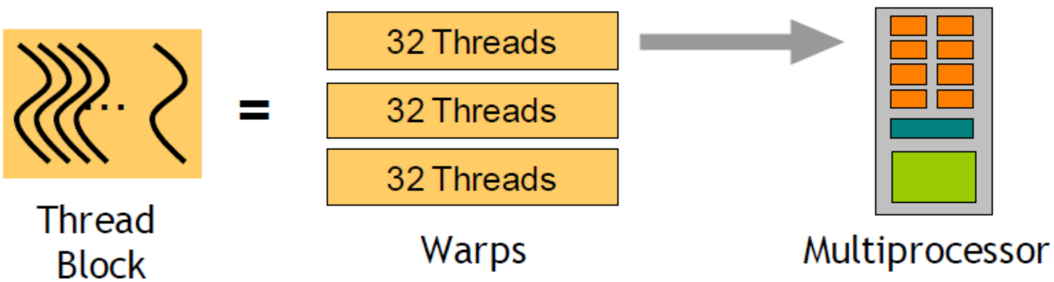
Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/GPU_Fundamentals.pdf

Execution Model - Warps



A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMT) on a multiprocessor

SIMT: Single Instruction, Multiple Thread

http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/GPU_Fundamentals.pdf

Memory Coalescing

Global memory access happens in transactions of 32 or 128 bytes

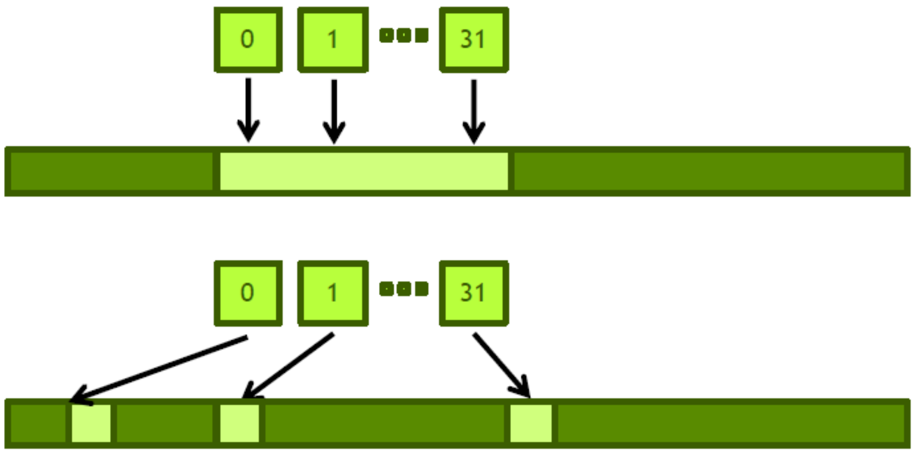
The hardware will try to reduce to as few transactions as possible

Coalesced access:

- A group of 32 contiguous threads ("warp") accessing adjacent words
- Few transactions and high utilization

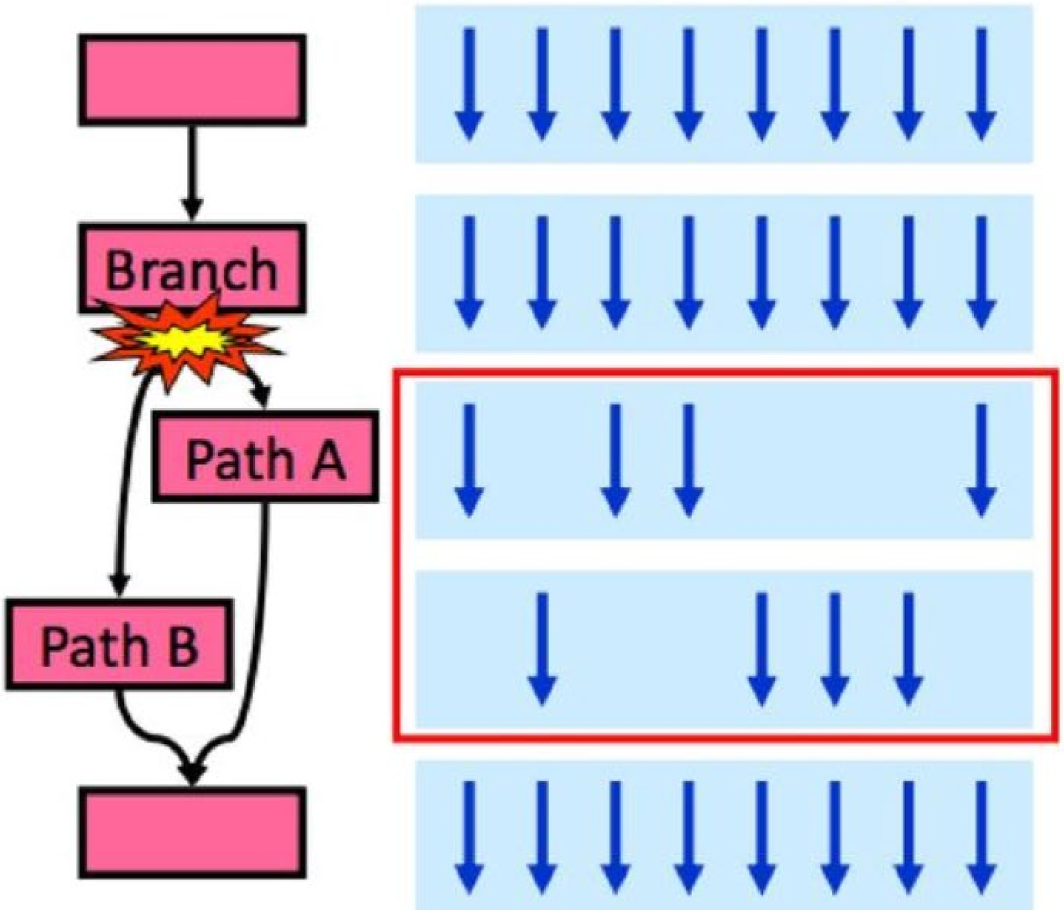
Uncoalesced access:

- A warp of 32 threads accessing scattered words
- Many transactions and low utilization



http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/GPU_Fundamentals.pdf

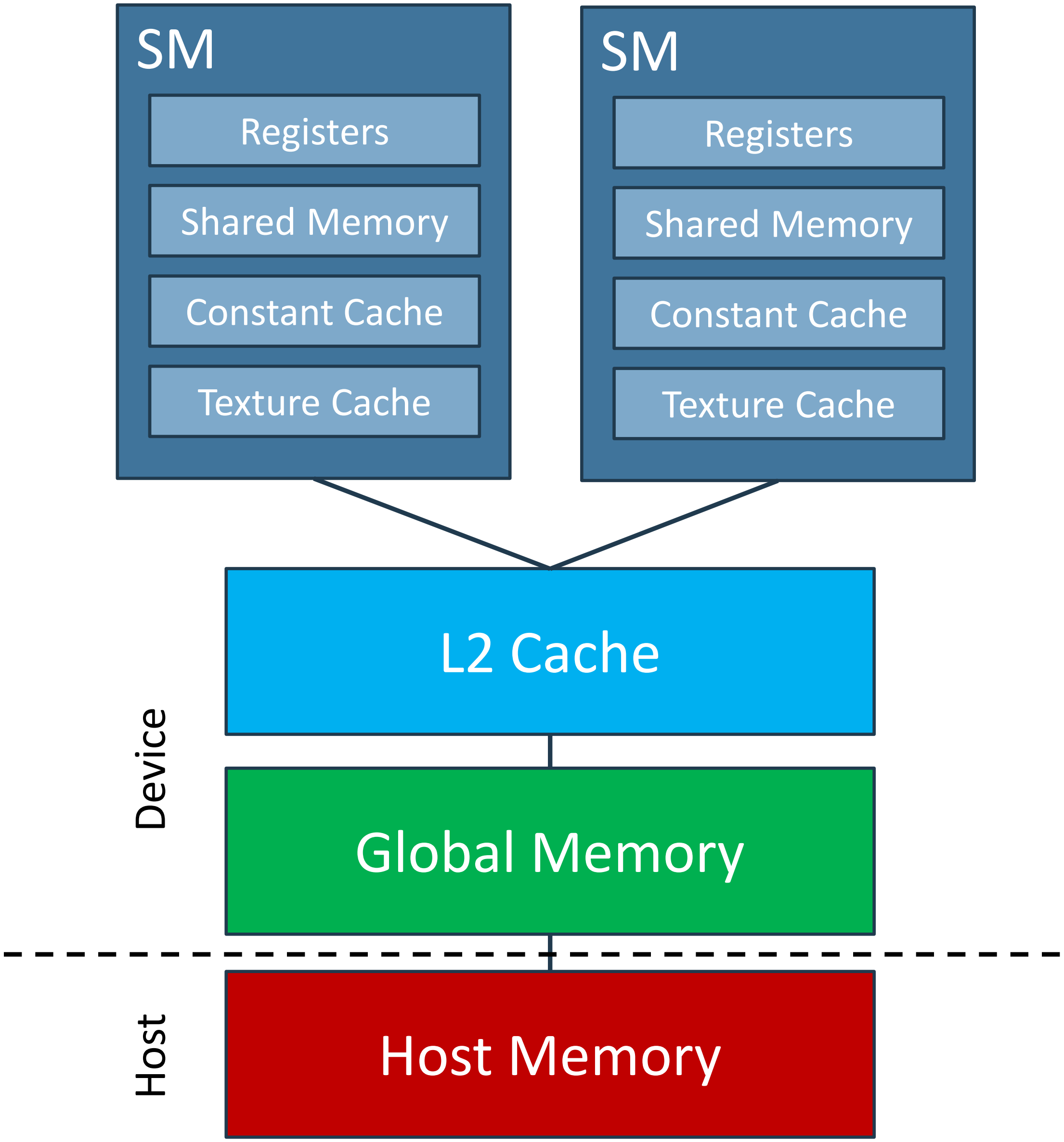
Warp Divergence



50% performance hit!

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

Memory Hierarchy



- Global Memory: Large and high latency
- L2 Cache: Medium latency
- SM Caches: Lower latency
- Registers: Lowest latency

Back to coding again…

```
// Host code
MyKernel<<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>(...);
```

BLOCKS_PER_GRID and THREADS_PER_BLOCK are of type *dim3*

Total number of blocks are Dg.x * Dg.y * Dg.z

And total number of threads in the block are Db.x * Db.y * Db.z

Note: Number of threads per block must be multiple of 32

```
__global__ void MyKernel(...)
{

}
```

Back to coding again...

```
// Variable in shared memory
__shared__ float sum;


// Device local constant
__constant__ float growth_rate;


// Device function to add the elements of two arrays
__device__ void add(int n, float* x, float* y)
{
       for (int i = 0; i < n; i++)
              y[i] = x[i] + y[i];
}
```

```cpp
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__ void add(int n, float* x, float* y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1 << 20;
    float* x, * y;

    // Allocate Unified Memory – accessible from CPU or GPU
    cudaMallocManaged(&x, N * sizeof(float));
    cudaMallocManaged(&y, N * sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Run kernel on 1M elements on the GPU
    add << <1, 1 >> > (N, x, y);

    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();

    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i] - 3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    cudaFree(x);
    cudaFree(y);

    return 0;
}
```

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__ void add(int n, float* x, float* y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

int main(void)
{
```

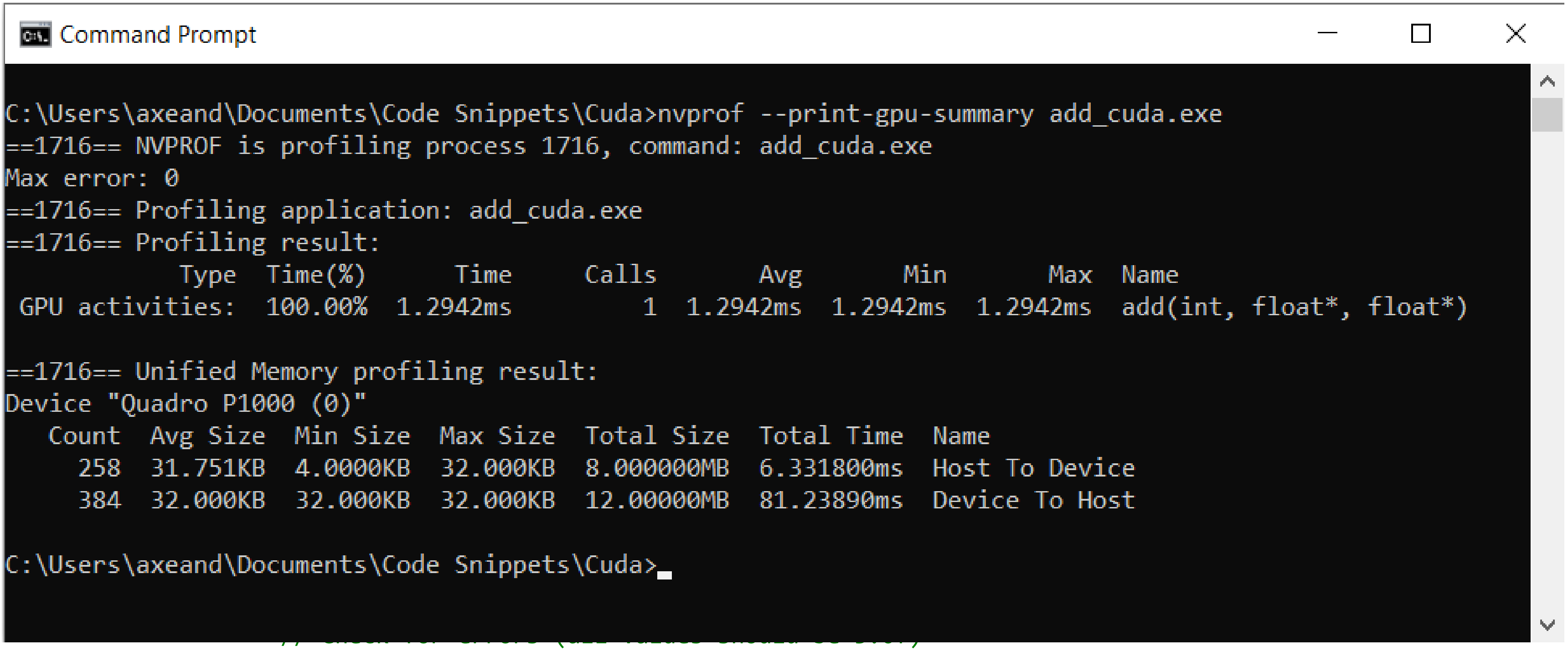
Command Prompt

```
C:\Users\axeand\Documents\Code Snippets\Cuda>nvprof --print-gpu-summary add_cuda.exe
==38564== NVPROF is profiling process 38564, command: add_cuda.exe
Max error: 0
==38564== Profiling application: add_cuda.exe
==38564== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  176.56ms         1  176.56ms  176.56ms  176.56ms  add(int, float*, float*)

==38564== Unified Memory profiling result:
Device "Quadro P1000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
     258  31.751KB  4.0000KB  32.000KB  8.000000MB  6.310000ms  Host To Device
     384  32.000KB  32.000KB  32.000KB  12.00000MB  80.83200ms  Device To Host

C:\Users\axeand\Documents\Code Snippets\Cuda>
```

```
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i] - 3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    cudaFree(x);
    cudaFree(y);

    return 0;
}
```

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__ void add(int n, float* x, float* y)
{
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1 << 20;
    float* x, * y;

    // Allocate Unified Memory – accessible from CPU or GPU
    cudaMallocManaged(&x, N * sizeof(float));
    cudaMallocManaged(&y, N * sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    // Run kernel on 1M elements on the GPU
    add << <1, 256 >> > (N, x, y);

    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();

    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i] - 3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    cudaFree(x);
    cudaFree(y);

    return 0;
}
```

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__ void add(int n, float* x, float* y)
{
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}
```




```
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i] - 3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    cudaFree(x);
    cudaFree(y);

    return 0;
}
```

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__ void add(int n, float* x, float* y)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1 << 20;
    float* x, * y;

    // Allocate Unified Memory – accessible from CPU or GPU
    cudaMallocManaged(&x, N * sizeof(float));
    cudaMallocManaged(&y, N * sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    int blockSize = 256;
    int numBlocks = (N + blockSize - 1) / blockSize;

    // Run kernel on 1M elements on the GPU
    add << <numBlocks, blockSize >> > (N, x, y);

    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();

    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i] - 3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    cudaFree(x);
    cudaFree(y);

    return 0;
}
```

```
#include <iostream>
#include <math.h>
// Kernel function to add the elements of two arrays
__global__ void add(int n, float* x, float* y)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] = x[i] + y[i];
}

int main(void)
```

Command Prompt

```
C:\Users\axeand\Documents\Code Snippets\Cuda>nvprof --print-gpu-summary add_cuda.exe
==5164== NVPROF is profiling process 5164, command: add_cuda.exe
Max error: 0
==5164== Profiling application: add_cuda.exe
==5164== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  149.44us         1  149.44us  149.44us  149.44us  add(int, float*, float*)

==5164== Unified Memory profiling result:
Device "Quadro P1000 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
     258  31.751KB  4.0000KB  32.000KB  8.000000MB  7.599000ms  Host To Device
     384  32.000KB  32.000KB  32.000KB  12.00000MB  72.25670ms  Device To Host

C:\Users\axeand\Documents\Code Snippets\Cuda>_
```

```
    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i] - 3.0f));
    std::cout << "Max error: " << maxError << std::endl;

    // Free memory
    cudaFree(x);
    cudaFree(y);

    return 0;
}
```

Some tips for the road…

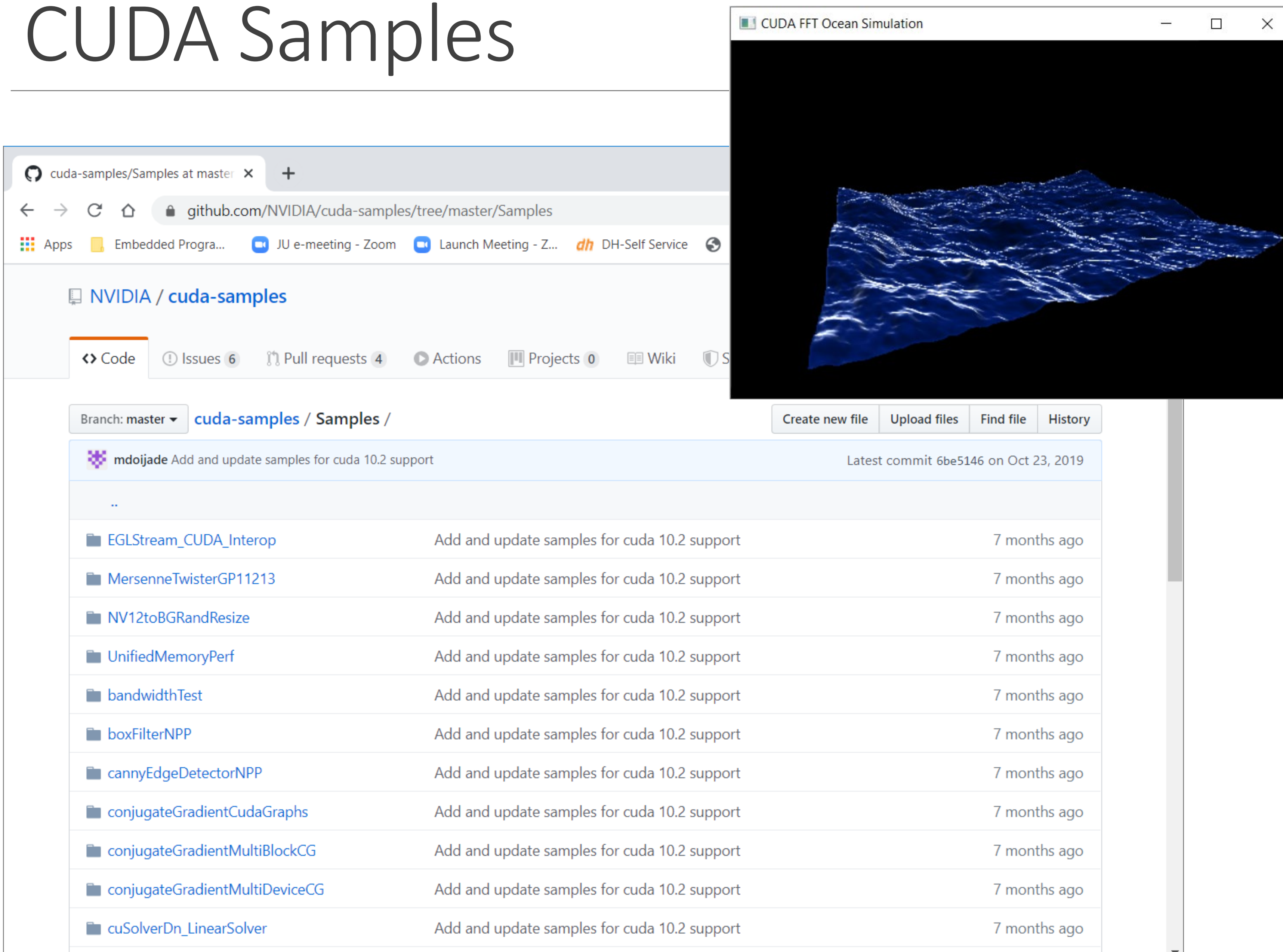
- In the kernel invocation, `<<<Blocks, Threads>>>`, try to chose a number of threads that divides evenly with the number of threads in a warp. If you don't, you end up with launching a block that contains inactive threads.
- In your kernel, try to have each thread in a warp follow the same code path. If you don't, you get what's called warp divergence. This happens because the GPU has to run the entire warp through each of the divergent code paths.
- In your kernel, try to have each thread in a warp load and store data in specific patterns. For instance, have the threads in a warp access consecutive 32-bit words in global memory.

CUDA in Python - PyCUDA

```
import pycuda.compiler as comp
import pycuda.driver as drv
import numpy
import pycuda.autoinit

mod = comp.SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1))

print dest-a*b
```

https://pypi.org/project/pycuda/

CUDA Samples



Introduction to CUDA

Questions?

Contact information

Andreas Axelsson

Email: andreas.axelsson@ju.se

Mobile: 0709-467760