

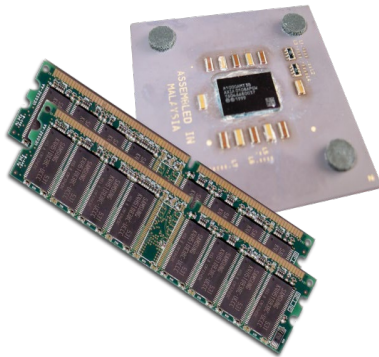
Contemporary Computer Architecture TDSN13

LECTURE 5 – CONT INTRO TO CUDA

ANDREAS AXELSSON (ANDREAS.AXELSSON@JU.SE)

Heterogeneous Computing

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

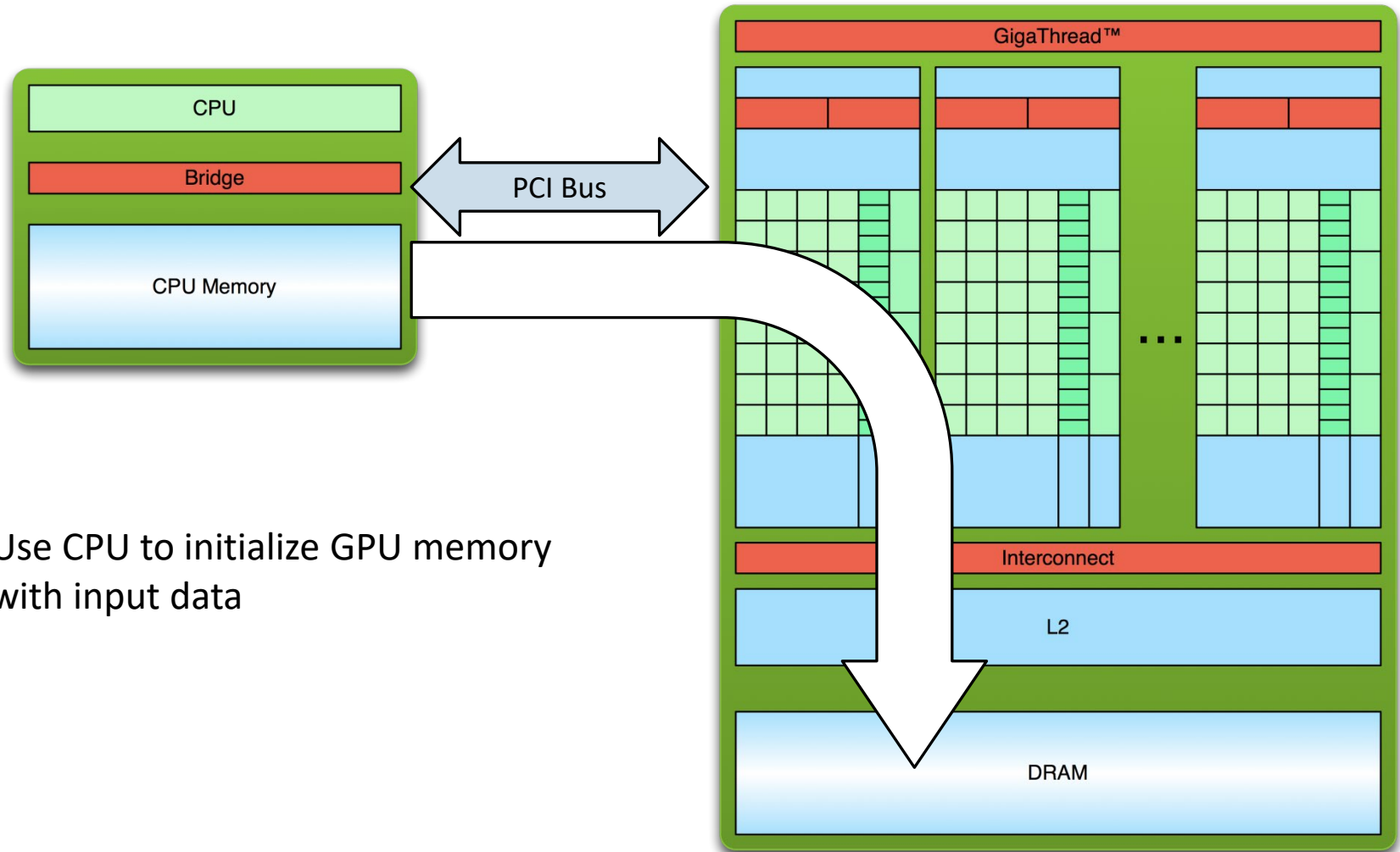


Host



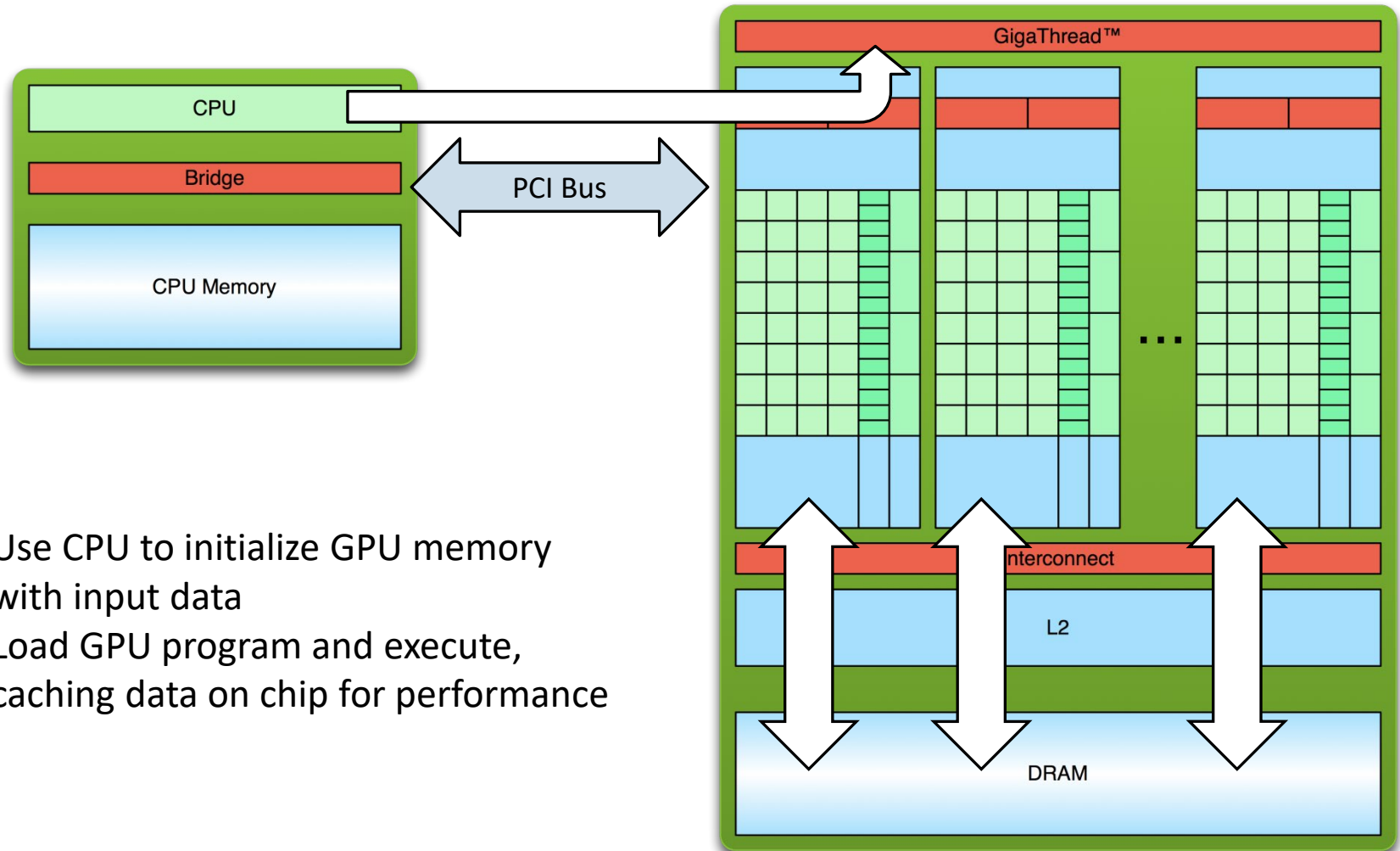
Device

Workflow

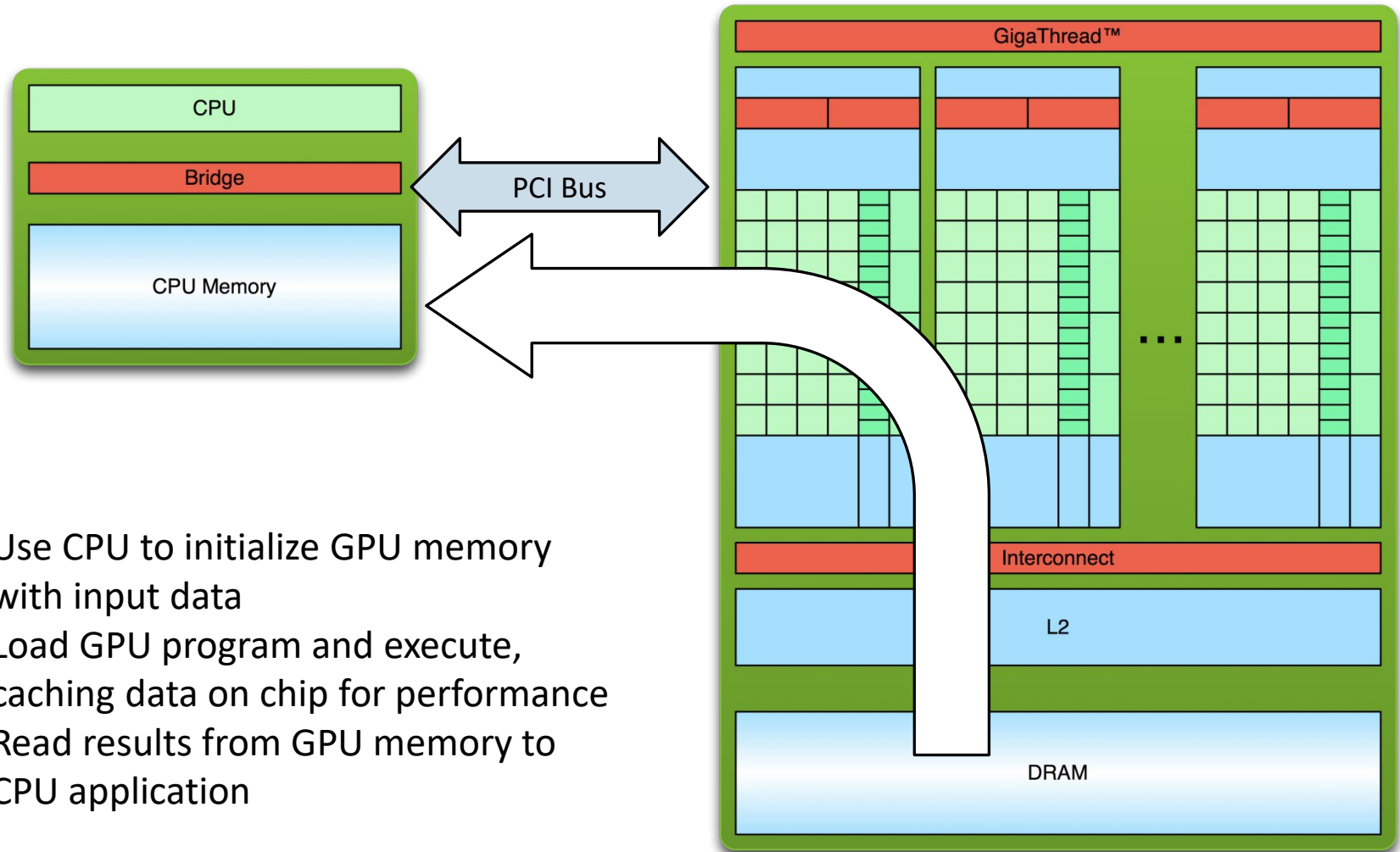


1. Use CPU to initialize GPU memory with input data

Workflow



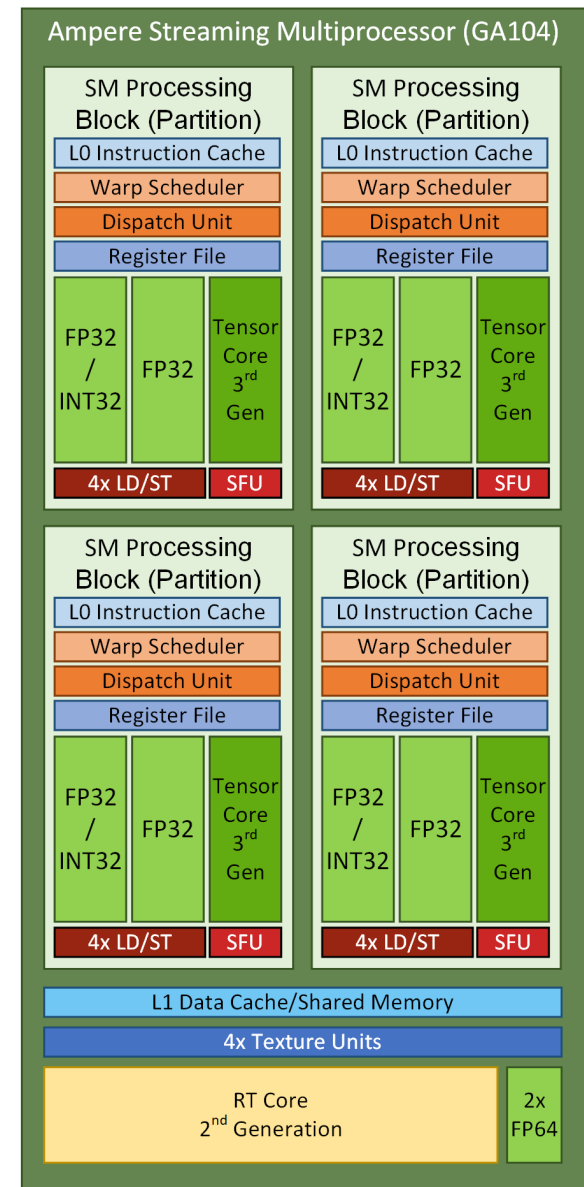
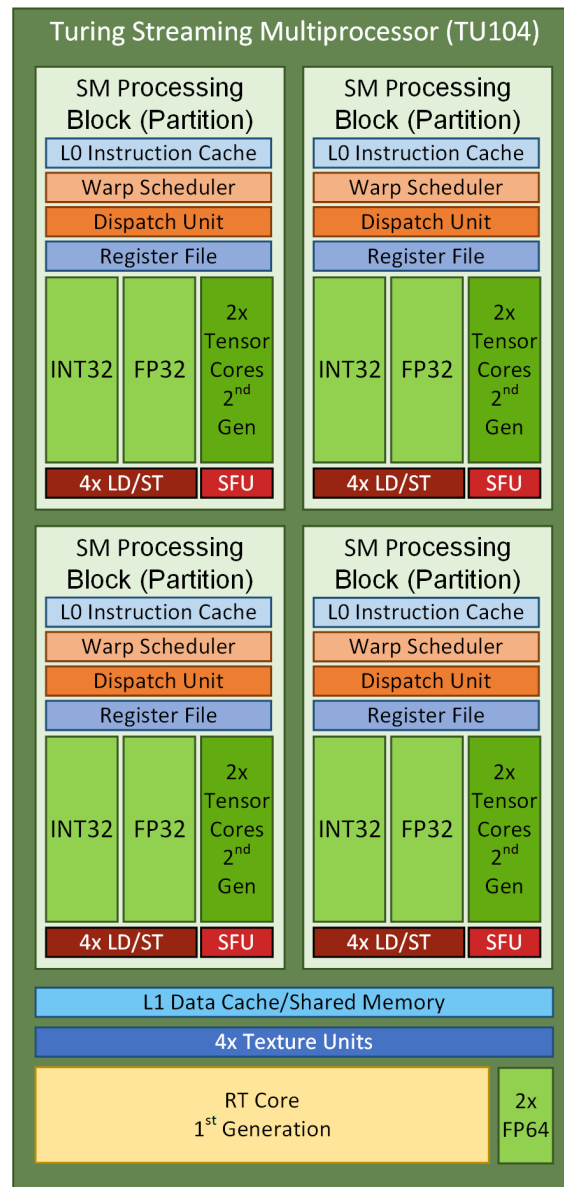
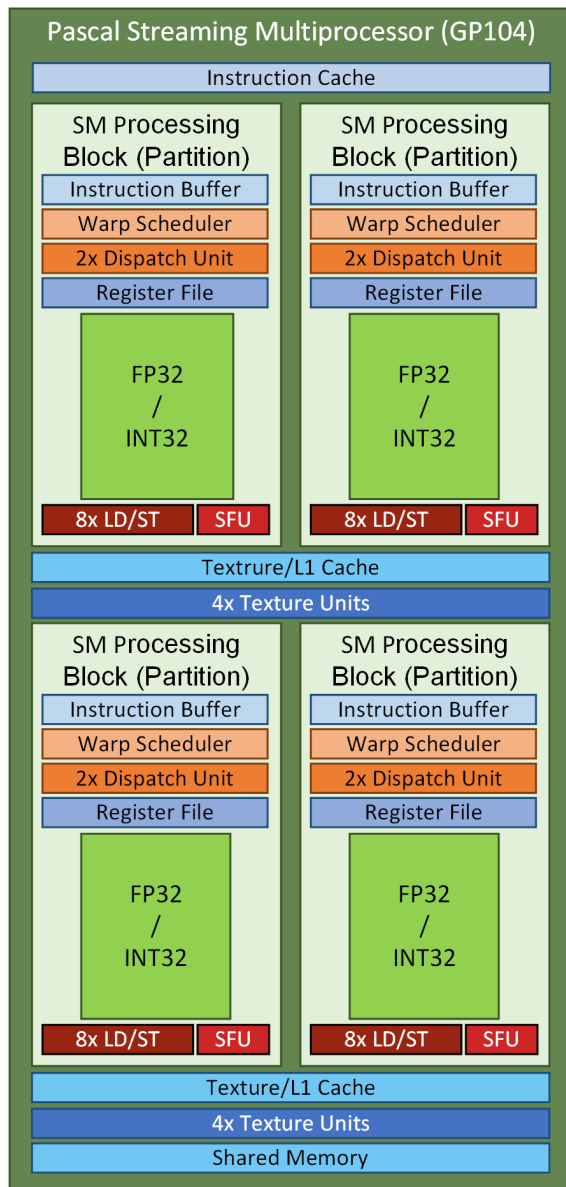
Workflow



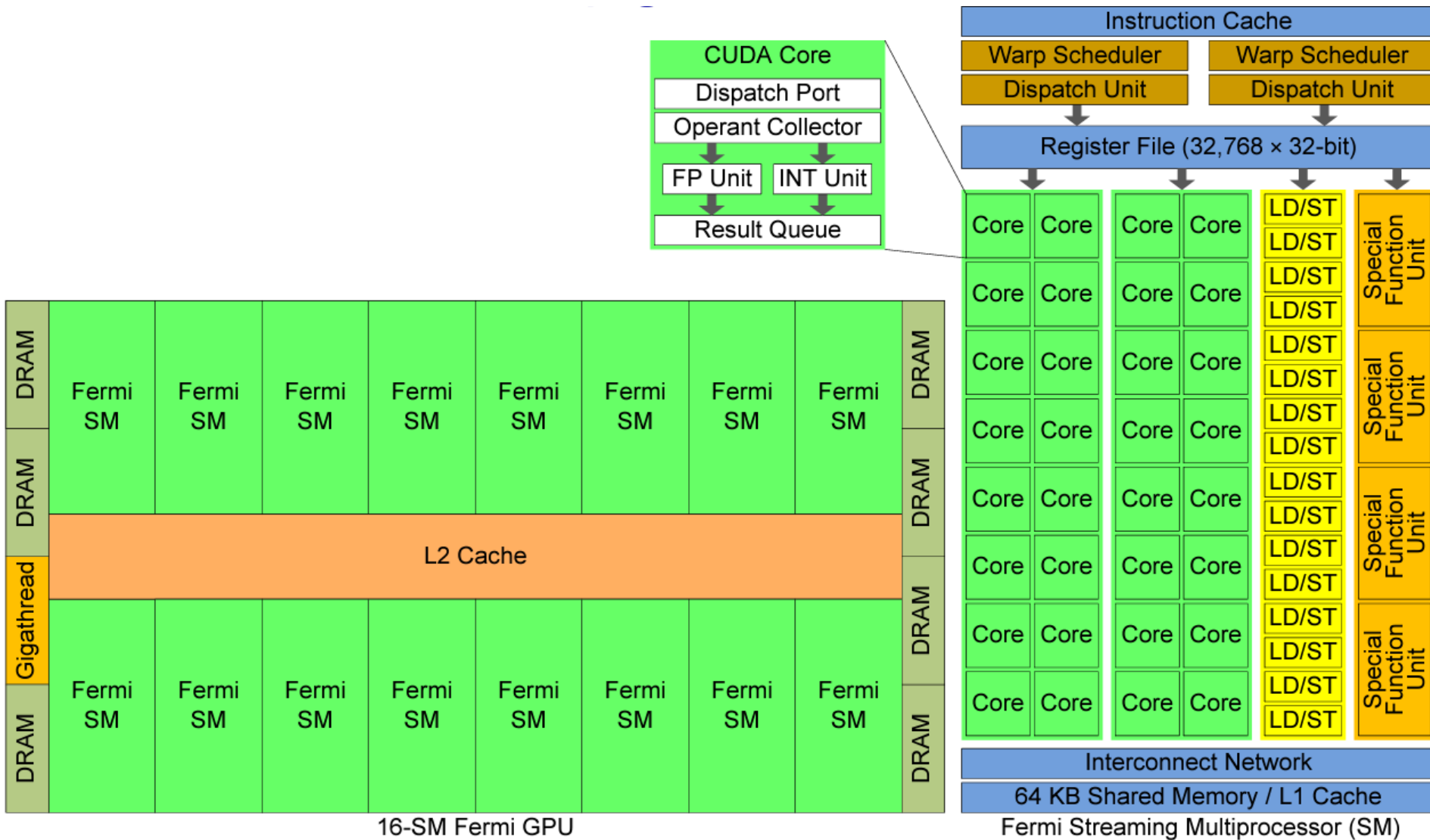
GPU Architecture (Pascal)



Streaming Multiprocessor

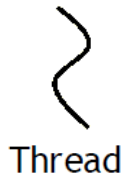


Fermi GPU



Execution Model

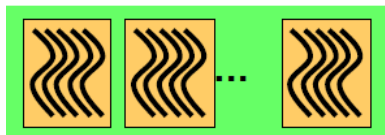
Software



Thread



Thread Block

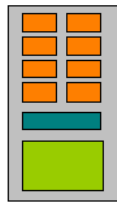


Grid

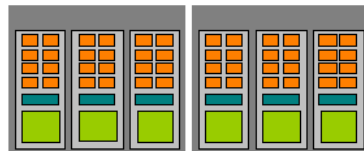
Hardware



Scalar
Processor



Multiprocessor



Device

Threads are executed by scalar processors

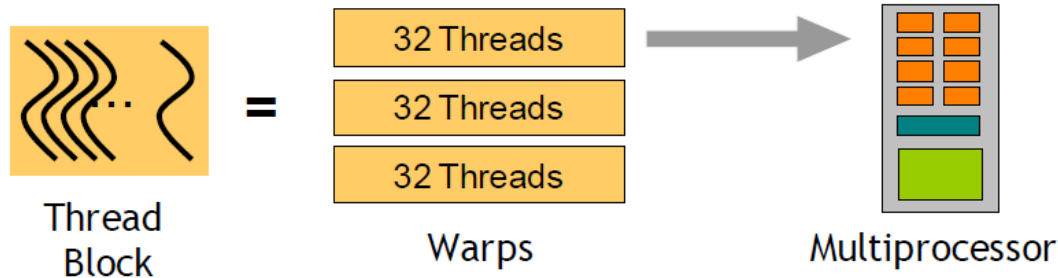
Thread blocks are executed on multiprocessors

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

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

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:

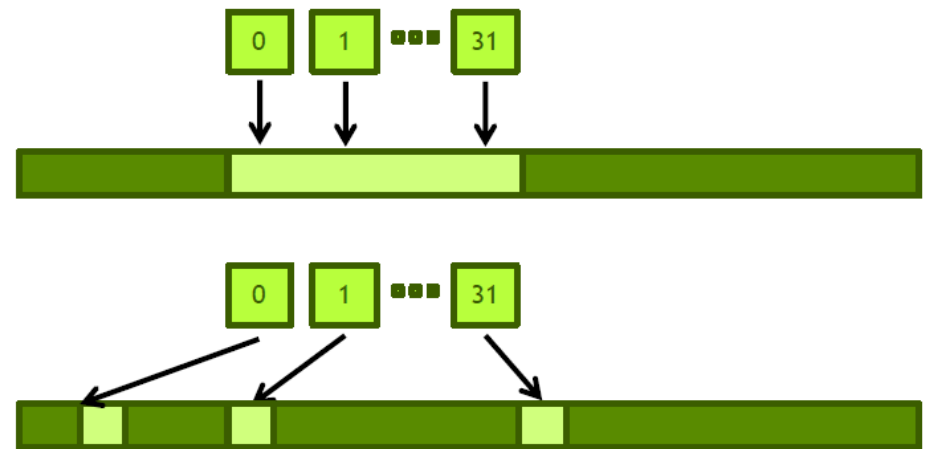
A group of 32 contiguous threads (“warp”) accessing adjacent words

Few transactions and high utilization

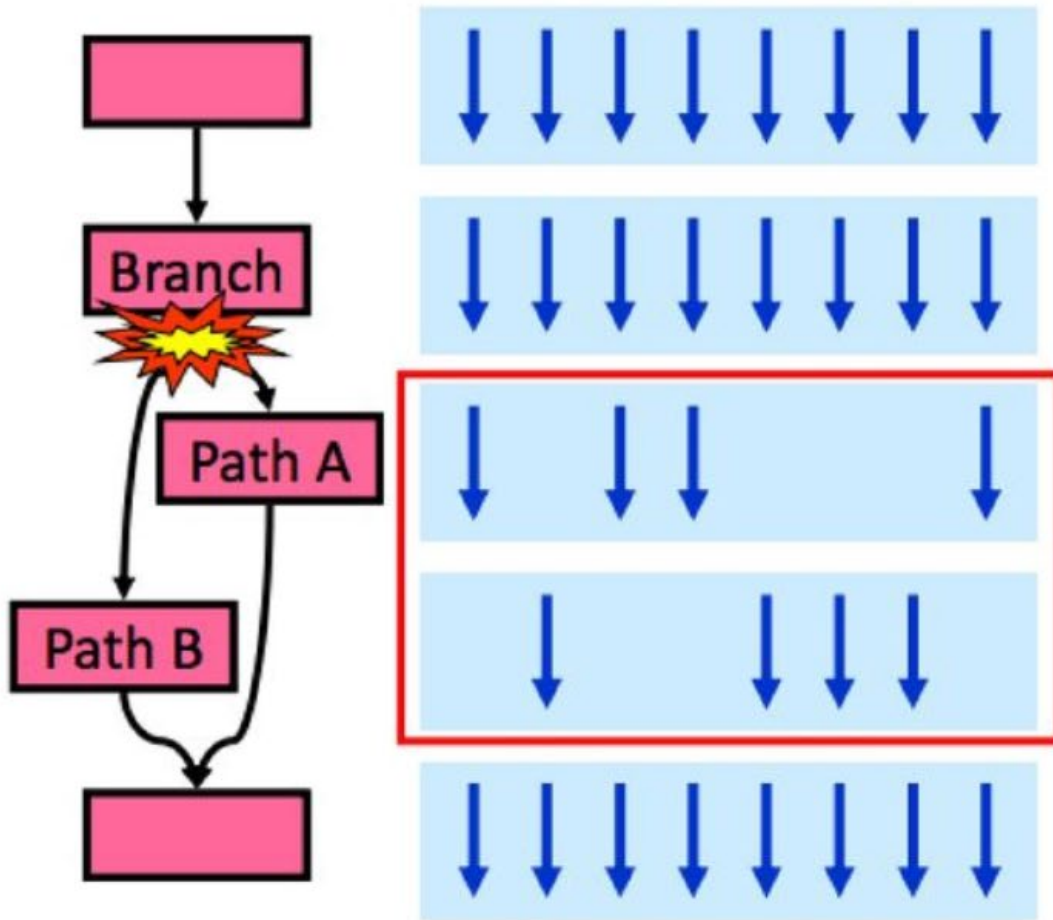
Uncoalesced access:

A warp of 32 threads accessing scattered words

Many transactions and low utilization

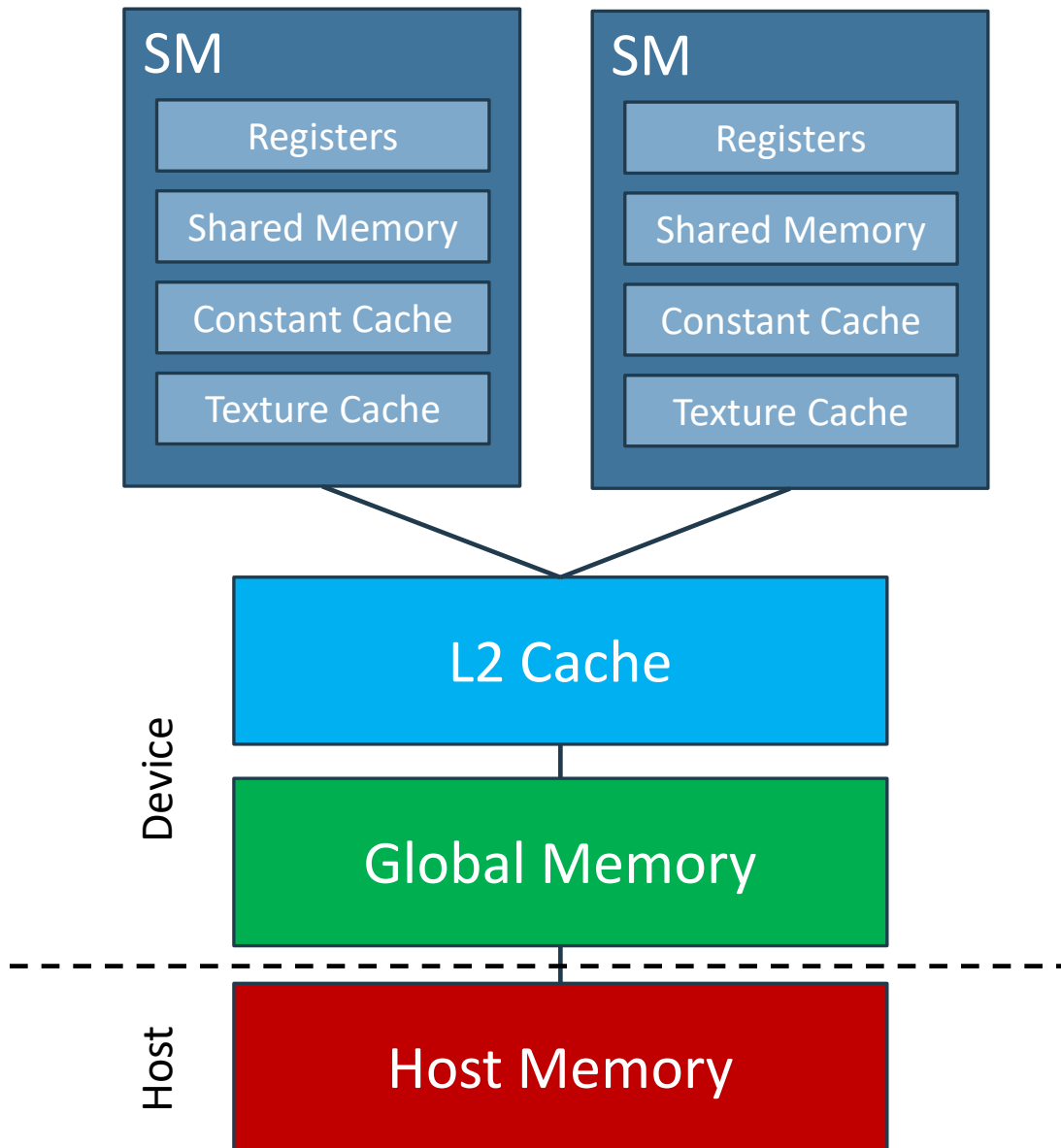


Warp Divergence



50% performance hit!

Memory Hierarchy



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

Memory Hierarchy

CUDA Memory Types

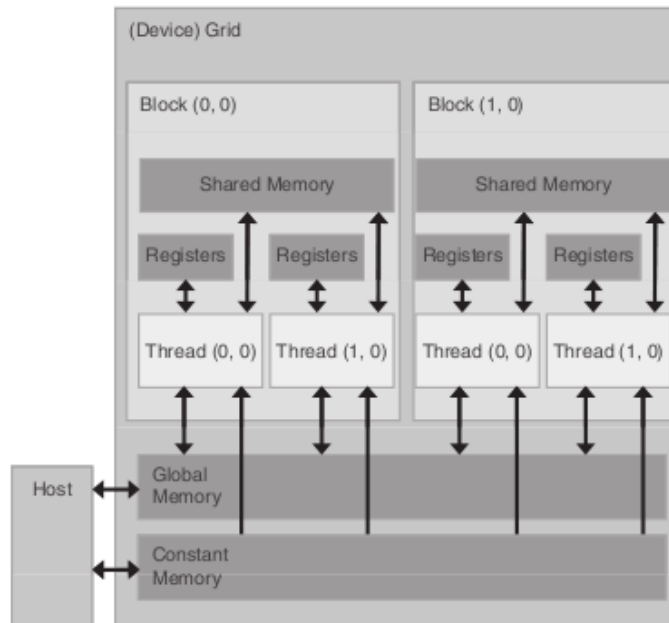
Memory	Scope of Access	Lifetime	R/W ability	Speed	Declaration
Register	Thread	Kernel	R/W	Fast	Automatic Variables
Local	Thread	Kernel	R/W	Fast	Automatic Arrays
Shared	Block	Kernel	R/W	Fast	<code>__shared__</code>
Global	Grid	Host	R/W	Slow	<code>__device__</code>
Constant	Grid	Host	Read only	Fast	<code>__constant__</code>

- Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

- Host code can

- Transfer data to/from per-grid global and constant memories



Memory Hierarchy

```
// 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];
```

```
}
```

```

#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;
}

```

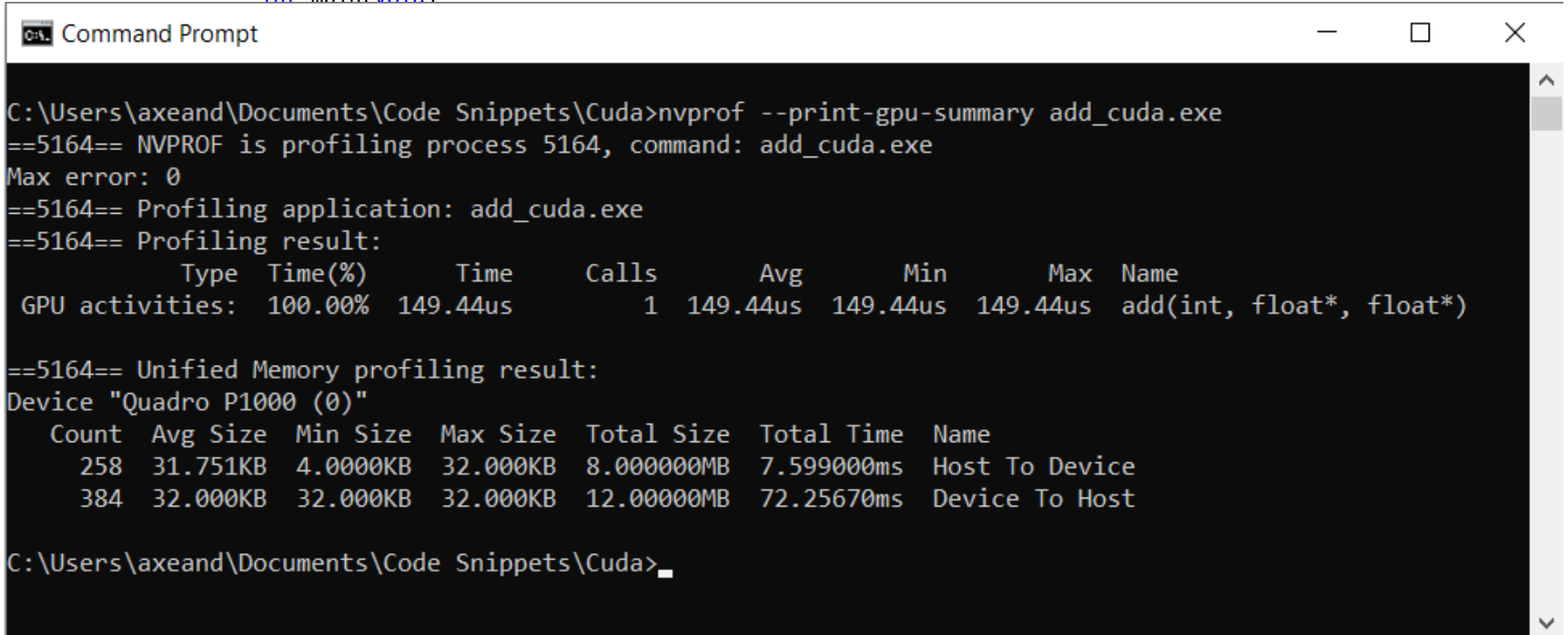


```

#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)

```



```

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  12.000000MB  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);

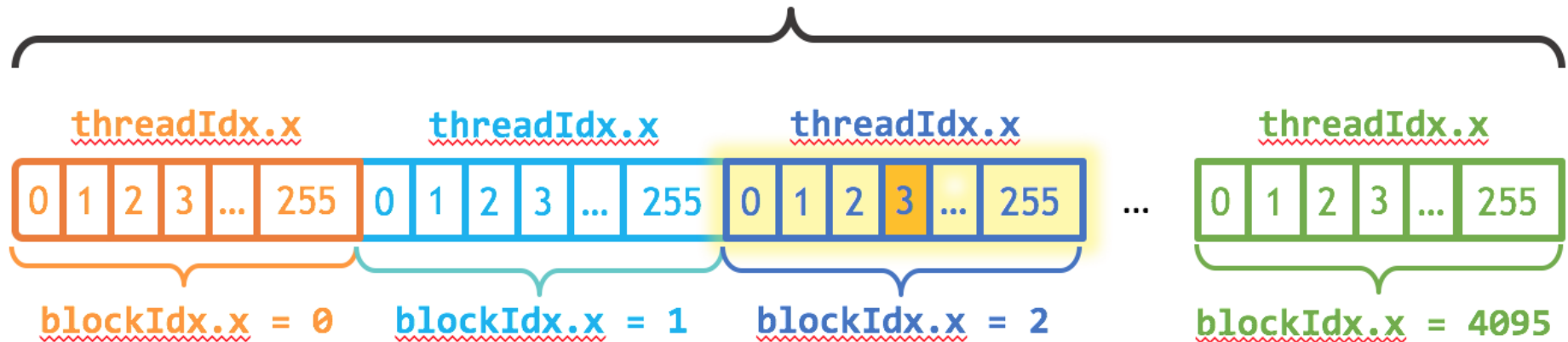
return 0;
}

```

Indexing

```
int blockSize = 256;  
int numBlocks = (N + blockSize - 1) / blockSize;  
add<<<numBlocks, blockSize>>>(N, x, y);
```

gridDim.x = 4096



$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$

index = (2) * (256) + (3) = 515

Some tips for the road...

- In the kernel invocation, `<<<Blocks, Threads>>>`, try to choose 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.

Another one for the road

NOTE:

Both registers and shared memory is divided among the threads in a block

A too large block size might not be scheduled if the kernel program requires many registers (due to large and complex problem)

How many registers per streaming multiprocessor?

How much shared memory per streaming multiprocessor?

CUDA API Introduction

Global Memory (Device):

*cudaMalloc(void** devPtr, size_t size)*: Allocates *size* bytes of linear memory on the device and returns a pointer in *devPtr*.

cudaFree(void devPtr)*: Frees the memory space pointed to by *devPtr*, which must have been returned by a previous call to *cudaMalloc*.

Unified Memory:

*cudaMallocManaged(void** devPtr, size_t size, unsigned int flags = cudaMemAttachGlobal)*: Allocates *size* bytes of managed memory that is accessible from both the host and device, potentially enabling a unified view of the memory.

cudaMemAttachGlobal: Attaches the memory to the global address space.

cudaDeviceSynchronize(void): function is a host function that waits for the completion of all preceding commands in all streams on the current device.

CUDA API Introduction

Pinned (Page-Locked) Host Memory:

*cudaHostAlloc(void** pHost, size_t size, unsigned int flags)*: Allocates *size* bytes of host memory that is page-locked and accessible to the device. The *flags* parameter enables options like *cudaHostAllocMapped* (to map the allocation into the CUDA address space) or *cudaHostAllocWriteCombined* (write-combined memory).

*cudaMallocHost(void** ptr, size_t size)*: Simplified version of *cudaHostAlloc* that allocates page-locked host memory without additional flags.

cudaFreeHost(void ptr)*: Frees the memory space pointed to by *ptr*, which must have been returned by a previous call to *cudaHostAlloc* or *cudaMallocHost*.

CUDA API Introduction

Shared Memory:

Shared memory is declared within a kernel using the `__shared__` qualifier and does not have explicit API calls for allocation, as it is automatically managed by the CUDA runtime.

Constant Memory:

Constant memory is typically declared using the `__constant__` qualifier in CUDA kernel code. To copy data to and from constant memory, you use the following functions:

cudaMemcpyToSymbol(const void symbol, const void* src, size_t count, size_t offset = 0, cudaMemcpyKind kind = cudaMemcpyHostToDevice):*

Copies data from the host memory pointed to by *src* to the constant memory symbol *symbol*.

cudaMemcpyFromSymbol(void dst, const void* symbol, size_t count, size_t offset = 0, cudaMemcpyKind kind = cudaMemcpyDeviceToHost):*

Copies data from the constant memory symbol *symbol* to the host memory pointed to by *dst*.

CUDA API Introduction

Error Handling:

Almost all CUDA api functions returns an error code *cudaError_t*, where *cudaSuccess* means the call was successful. If other values are returned there was an error.

Some error helper functions:

cudaGetLastError(void): Returns the last error from a runtime call.

cudaPeekAtLastError(void): Returns the last error from a runtime call without resetting the error code to *cudaSuccess*.

const char cudaGetErrorString(cudaError_t error)*: Returns the error in clear text for printout etc.

CUDA Samples

cuda-samples/Samples at master · NVIDIA · GitHub

github.com/NVIDIA/cuda-samples/tree/master/Samples

Apps Embedded Progra... JU e-meeting - Zoom Launch Meeting - Z... DH-Self Service

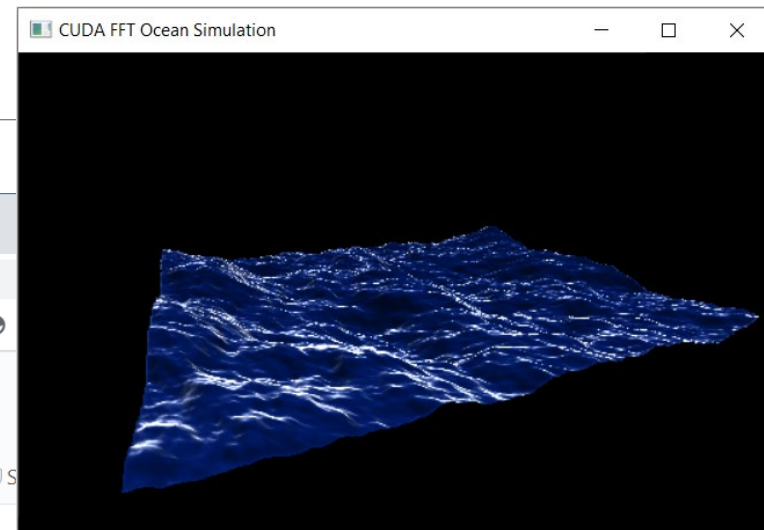
NVIDIA / cuda-samples

Code Issues 6 Pull requests 4 Actions Projects 0 Wiki S

Branch: master cuda-samples / Samples /

mdoijade Add and update samples for cuda 10.2 support Latest commit 6be5146 on Oct 23, 2019

..		
EGLStream_CUDA_Interop	Add and update samples for cuda 10.2 support	7 months ago
MersenneTwisterGP11213	Add and update samples for cuda 10.2 support	7 months ago
NV12toBGRandResize	Add and update samples for cuda 10.2 support	7 months ago
UnifiedMemoryPerf	Add and update samples for cuda 10.2 support	7 months ago
bandwidthTest	Add and update samples for cuda 10.2 support	7 months ago
boxFilterNPP	Add and update samples for cuda 10.2 support	7 months ago
cannyEdgeDetectorNPP	Add and update samples for cuda 10.2 support	7 months ago
conjugateGradientCudaGraphs	Add and update samples for cuda 10.2 support	7 months ago
conjugateGradientMultiBlockCG	Add and update samples for cuda 10.2 support	7 months ago
conjugateGradientMultiDeviceCG	Add and update samples for cuda 10.2 support	7 months ago
cuSolverDn_LinearSolver	Add and update samples for cuda 10.2 support	7 months ago



Create new file Upload files Find file History

Introduction to CUDA

Questions?

Contact information

Andreas Axelsson

Email: andreas.axelsson@ju.se

Mobile: 0709-467760