# Introduction to GPU Computing

Putt Sakdhnagool

putt.sakdhnagool@nectec.or.th

September 12, 2018

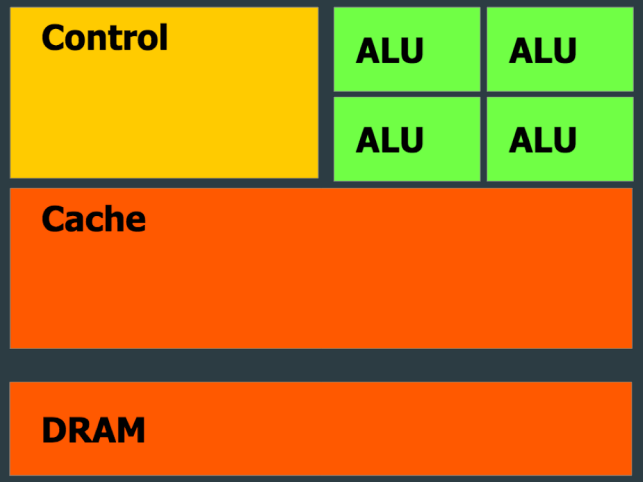


Some slides are adopted from “Introduction to GPU Computing” by Jeff Larkin, NVIDIA:
https://www.olcf.ornl.gov/wp-content/uploads/2018/06/intro_to_HPC_gpu_computing.pdf

# Fundamentals of GPU Computing

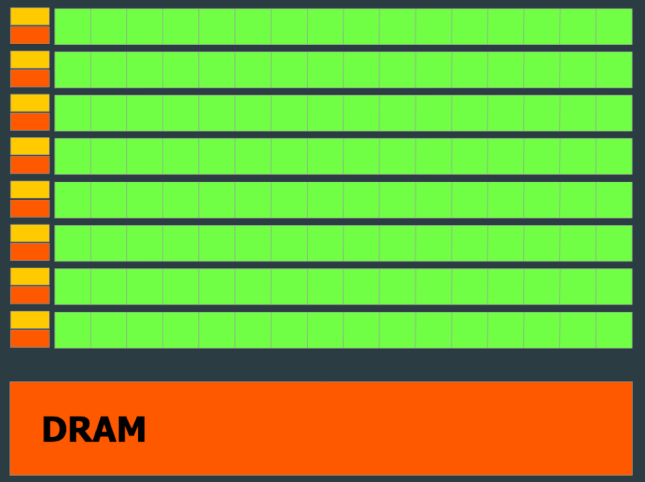
Introduction to accelerated computing

# CPUs vs GPUs at a Glance




CPU

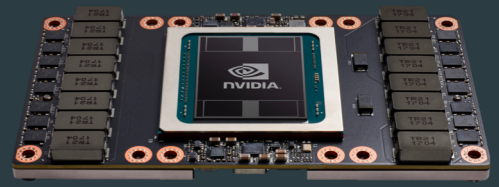
- Optimized for **latency**
  - Get a job done *fast*
- Lower memory bandwidth
- Larger memory size
- Suitable for *generic* workload




GPU

- Optimized for **bandwidth**
  - Get *more* jobs done
- Higher memory bandwidth
- Smaller memory size
- Suitable for *parallel* workload

# CPUs vs GPUs at a Glance

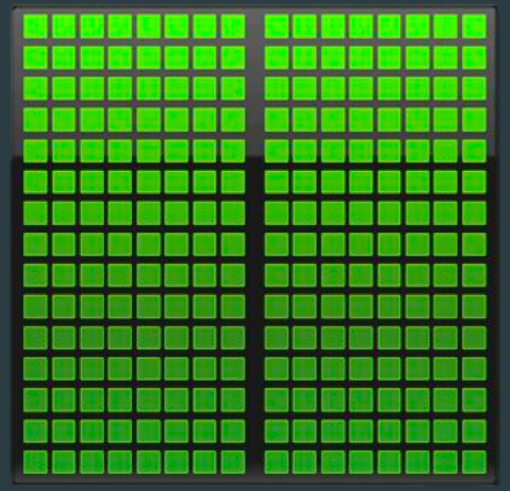

| | Intel Xeon Platinum 8180 | NVIDIA Tesla V100 SXM2 |
|---|---|---|
| Processing Cores | 28 | 5120 |
| Processor Clock Speed | 2.5 GHz | 1.2 GHz |
| Memory Type | DDR4-2666 | HBM2 |
| Memory Bandwidth | 120 GB/sec | 900 GB/sec |
| Memory Size | 768 GB | 32 GB |
| Max Power Consumption | 205W | 300W |

# Accelerated Computing

- Heterogeneous Computing
- Complementary processors work together

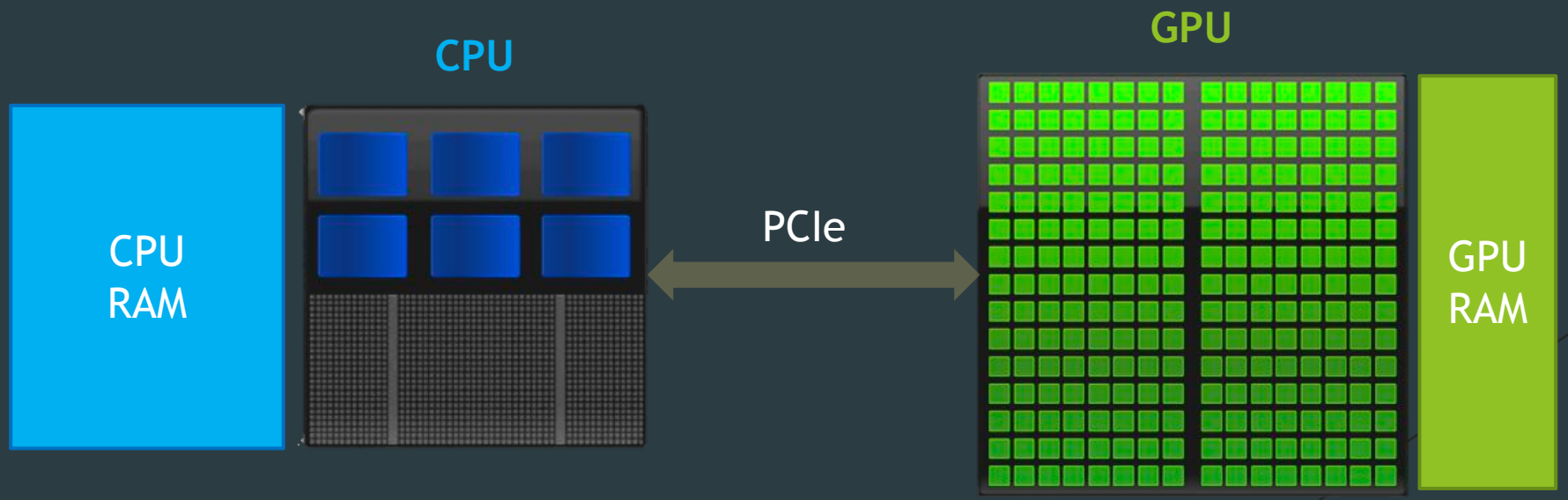
**CPU**
Serial Tasks

**GPU Accelerator**
Parallel Tasks

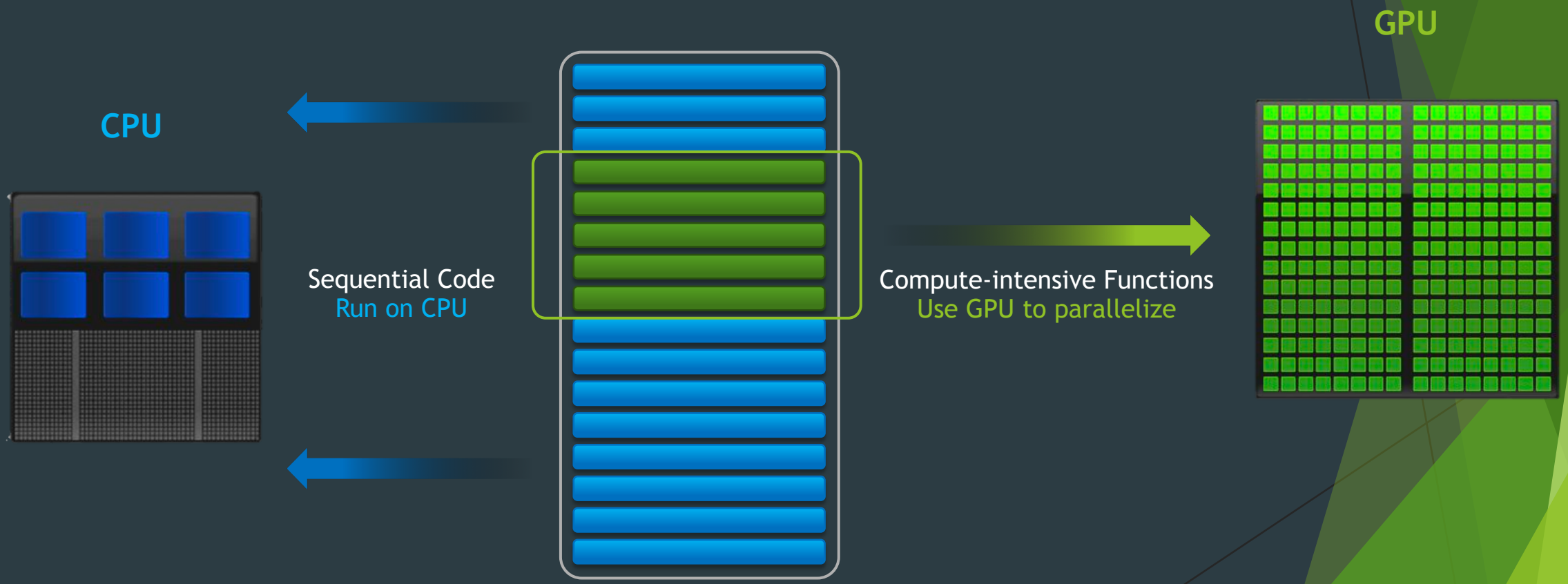


# Accelerated Computing

- CPU and GPU are separated entities
  - CPU and GPU have *distinct memories*
- Execution begins on CPU
  - Data and computation are *offloaded* to GPU
- CPU and GPU communicate via PCIe
  - PCIe bandwidth is much *lower* than memories



# Accelerated Computing



# GPU Programming

A short introduction

# 3 Ways to Program GPUs

Simplicity

- **Libraries**: "*Drop-in*" acceleration
  - Little or no code change for standard libraries; high performance
  - Limited by what libraries are available
- **OpenACC**: Directive-based programming for rapid acceleration
  - Rapidly accelerate code; only need small changes to existing code
  - Less control over performance
- **Programming Language**: Maximum flexibility
  - Expose low-level details for maximum performance
  - Steep learning curve and time consuming

Performance

# GPU-accelerated Libraries

Linear Algebra

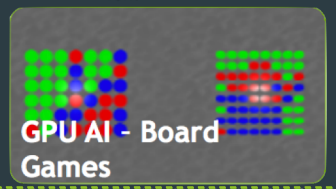


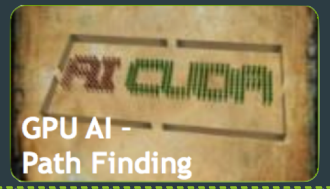


Numerical & Math

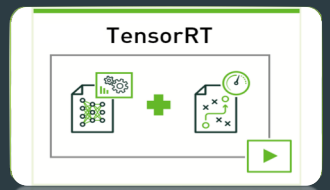


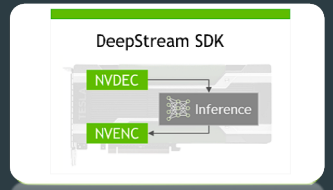

Data Structure & AI


Deep Learning


CUDA-aware
memory allocation

# Drop-in Acceleration
## With Automatic Data Management

### Original

```
int N = 1 << 20; // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

### With cuBLAS

```
int N = 1 << 20; // 1M elements

cudaMallocManaged(&x, N * sizeof(float));
cudaMallocManaged(&y, N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
cublasSaxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

# Drop-in Acceleration
## With Automatic Data Management

Original

```
int N = 1 << 20; // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

With cuBLAS

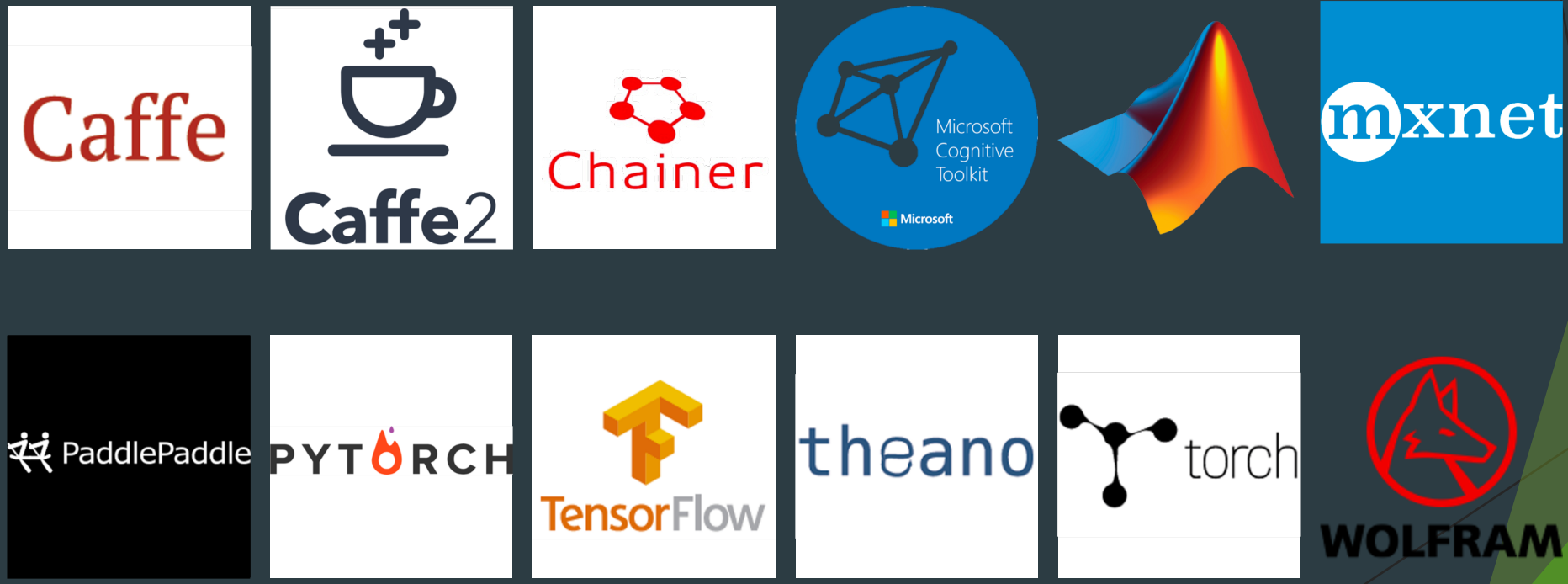
```
int N = 1 << 20; // 1M elements

cudaMallocManaged(&x, N * sizeof(float));
cudaMallocManaged(&y, N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
cublasSaxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

CUDA-aware memory allocation

Call cuBLAS API

# GPU Accelerated Deep Learning

- cuDNN accelerated framework






For more information: https://developer.nvidia.com/cudnn

# OpenACC Programming

- Directive-based programming model designed for **performance** and **portability** on CPUs and GPUs for HPC
- Single code for multiple platforms
  - POWER
  - x86
  - NVIDIA GPUs
  - PEZY-SC

```
void saxpy(int n,
           float a,
           float *x,
           float *y){

  #pragma acc kernels
  //automatically run on GPUs
  {
    for(int i = 0; i < n; i++)
      y[i] = a*x[i] + y[i];
  }
}


// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```




For more information: http://developer.nvidia.com/openacc and http://openacc.org

# Programming Languages: CUDA C/C++

- C/C++ Language extension for programming and managing CUDA GPUs
- Expose GPU architecture through APIs

```
void main(){
    float *a, *b, *out;
    a = (float*)malloc(sizeof(float) * N);

    //allocate device memory for a
    float *d_a;
    cudaMalloc((void**)&d_a, sizeof(float) * N);

    //transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

    …
    vector_add<<<256,1024>>>(out, d_a, d_b, N);
    …

    //cleanup after kernel execution
    cudaFree(d_a);
    free(a);
}
```

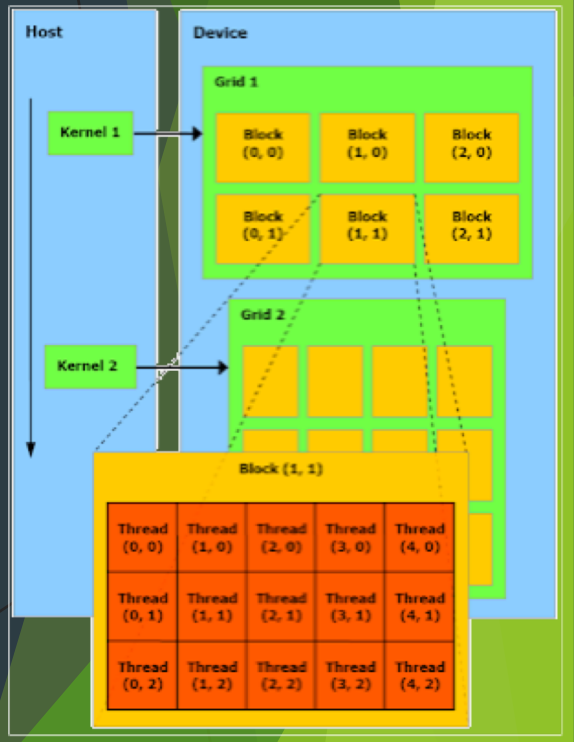
# CPU vs GPU Programming

# CPU vs GPU Programming

- GPU is a specialized hardware designed for executing thousands of threads
- There are several issues programmers need to consider when writing GPU programs
- In this talk
  - Execution model
  - Memory hierarchy
  - Parallelism limiting factor
- There are several other issues to consider: coalesced memory accesses, shared memory bank conflict, and etc.
- Suggested read
  - CUDA programming guide (https://docs.nvidia.com/cuda/cuda-c-programming-guide)
  - CUDA documentation (https://docs.nvidia.com/cuda/index.html)

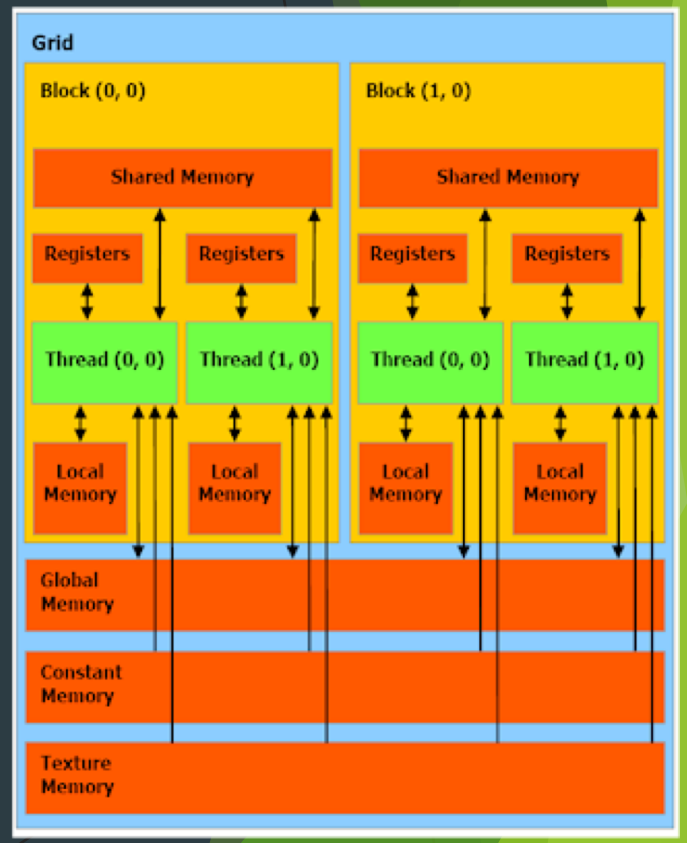
# CPU vs GPU Programming: Execution model

| | CPU | GPU |
|---|---|---|
| Execution Model | • MIMD | • SIMT |
| Thread Execution | • Threads are executed independently | • A group of 32 threads, called *warp*, are executed in lockstep |
| Threads Divergence | • Allow divergence<br>• No performance impact | • Allow some degree of divergence<br>• Impact performance |
| Thread Hierarchy | • Threads are completely independent | • Threads are organized in a group called *thread block*<br>• Kernel can be executed by multiple thread blocks organized in a *grid* |



# CPU vs GPU Programming: Memory Hierarchy

- CPU memory hierarchy is simpler and is hardware managed
- GPU consists of several memory spaces
- On-chip memory spaces
  - Register: thread-private, very fast
  - Shared memory: user-managed, shared across all threads in the same block
- Off-chip memory spaces
  - Local memory: thread-private memory
  - Global memory: shared across all threads
  - Constant memory: read-only, cached in constant cache
  - Texture memory: read-only, cached in texture cache, optimized for 2D spatial locality.



# CPU vs GPU Programming: Occupancy

- CPU: # of cores is a parallelism limiting factor
- GPU: On-chip resources could be parallelism limiting factor
- **Occupancy**: the ratio of active warps to the maximum number of active warps supported by an *Streaming Multiprocessor (SM)*
- Occupancy limiting factors:
  - Size of thread blocks: SM has a maximum number of tasks that can be active at once.
  - Register usage: SM has a register file that is shared by all active threads. If the # of required register is too high, the SM can host less # of threads.
  - Shared memory usage: Higher usage of shared memory could limit the # of thread blocks.

# Choosing GPUs

Understanding GPU specification

# Understanding GPU Specification: Compute Capability

- Identify the features supported by the GPU hardware
- Comprises a major revision number X and a minor revision number Y and is denoted by *X.Y*
- Do not confuse with CUDA version.
  - Current CUDA version: 9.2

| | Compute Capability | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 3.0, 3.2 | 3.5, 3.7 | 5.0, 5.2 | 5.3 | 6.0 | 6.1 | 6.2 | 7.0 |
| 16-bit floating-point add, multiply, multiply-add | N/A | N/A | N/A | 256 | 128 | 2 | 256 | 128 |
| 32-bit floating-point add, multiply, multiply-add | 192 | 192 | 128 | 128 | 64 | 128 | 128 | 64 |
| 64-bit floating-point add, multiply, multiply-add | 8 | 64[2] | 4 | 4 | 32 | 4 | 4 | 32 |
| 32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (__log2f), base 2 exponential (exp2f), sine (__sinf), cosine (__cosf) | 32 | 32 | 32 | 32 | 16 | 32 | 32 | 16 |
| 32-bit integer add, extended-precision add, subtract, extended-precision subtract | 160 | 160 | 128 | 128 | 64 | 128 | 128 | 64 |
| 32-bit integer multiply, multiply-add, extended-precision multiply-add | 32 | 32 | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | 64 |
| 24-bit integer multiply (__[u]mul24) | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. |
| 32-bit integer shift | 32 | 64[4] | 64 | 64 | 32 | 64 | 64 | 64 |
| compare, minimum, maximum | 160 | 160 | 64 | 64 | 32 | 64 | 64 | 64 |
| 32-bit integer bit reverse, bit field extract/insert | 32 | 32 | 64 | 64 | 32 | 64 | 64 | Multiple Instruct. |
| 32-bit bitwise AND, OR, XOR | 160 | 160 | 128 | 128 | 64 | 128 | 128 | 64 |
| count of leading zeros, most significant non-sign bit | 32 | 32 | 32 | 32 | 16 | 32 | 32 | 16 |
| population count | 32 | 32 | 32 | 32 | 16 | 32 | 32 | 16 |
| warp shuffle | 32 | 32 | 32 | 32 | 32 | 32 | 32 | 32 |
| sum of absolute difference | 32 | 32 | 64 | 64 | 32 | 64 | 64 | 64 |
| SIMD video instructions vabsdiff2 | 160 | 160 | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. |
| SIMD video instructions vabsdiff4 | 160 | 160 | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | 64 |
| All other SIMD video instructions | 32 | 32 | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. | Multiple instruct. |
| Type conversions from 8-bit and 16-bit integer to 32-bit types | 128 | 128 | 32 | 32 | 16 | 32 | 32 | 16 |
| Type conversions from and to 64-bit types | 8 | 32[5] | 4 | 4 | 16 | 4 | 4 | 16 |
| All other type conversions | 32 | 32 | 32 | 32 | 16 | 32 | 32 | 16 |

CUDA programming guide (https://docs.nvidia.com/cuda/cuda-c-programming-guide)

# Understanding GPU Specification: GeForce vs Quadro vs Tesla GPUs

- GeForce: Designed for gaming
  - Low FP64 and FP16 performance due to the lack of computing units
  - Least memory space and performance
  - Do not support RDMA
  - Active cooling
- Quadro: Designed for workstation workload, e.g. CAD, CGI
  - Some model has similar DP performance to Tesla GPUs
  - Use ECC memory. Memory size can be as large as Tesla card.
  - Active cooling
- Tesla: Designed for data center
  - Best overall performance
  - Use ECC memory; Largest and fastest memory across all variants
  - Passive cooling
  - No graphics capability
  - NVLink could connect up to 8 GPUs together and be used in placed of PCIe (Power9)

# What to choose ?

- For GPU computing, typically it comes down to either GeForce or Tesla
- Several questions to ask
  - What is the characteristic of your computation? Is it bandwidth-bound? Is it computation-bound ?
  - What kind of form factor you are running ?
  - Do you need double-precision performance ?
  - Do you need large memory ?
  - How many GPUs do you need per system ?
  - Do you need RDMA ?
  - …
- With the introduction of Titan V and RTX 2080 Ti, it becomes more complicated

# NVIDIA DGX Systems

- NVIDIA-designed systems built for AI and deep learning
  - DGX Station: Development workstation with 4 Tesla V100
  - DGX-1: 1 PFLOPS server with 8 Tesla V100, connected with NVLink
  - DGX-2: 2 PFLOPS server with 16 Tesla V100, connected with NVSwitch
- What's special with DGX system?
  - NVIDIA GPU Cloud (NGC): A comprehensive catalog of GPU-optimized Docker containers.
    - DGX systems natively support NGC and NVIDIA Docker
    - Programmers are expected to run their programs on NGC ecosystem
  - NVLink, NVSwitch: High performance interconnect

# What's About AMD ?

- Performance: The latest Radeon Vega Frontier GPU performs similar to NVIDIA GPUs [1] for deep-learning application
- Programmability: AMD provides ROCm platform [2] and HIP [3] programming language, which is similar to CUDA
- Documentation and Support: AMD seems a bit lacking compared to NVIDIA

- Radeon Vega seems to have better performance to price ratio
- Still no direct competition to Tesla V100 for HPC application.

- The new Vega 20 will be a direct competitor to Tesla V100 for deep learning [4]
  - Aim at deep learning workload
  - Will have Tensor-core like processor

[1] http://blog.gpueater.com/en/2018/04/23/00011_tech_cifar10_bench_on_tf13/
[2] https://github.com/RadeonOpenCompute/ROCm
[3] https://github.com/ROCm-Developer-Tools/HIP/tree/roc-1.8.x
[4] https://wccftech.com/amd-7nm-vega-20-20-tflop-compute-estimation/

# NSTDA-HPC

Our GPU Resources

# NSTDA-HPC: GPU Resources

- Phase 1: 12x Tesla V100 are available (pilot: Jan 2019, launch Apr 2019)
  - 3x DGX-1 (2 are reserved)
  - 2x Development machines, each has 2 Tesla V100
- Phase 2: 44x Tesla V100 (Tentative)
  - 4x DGX-1 (2 are reserved)
  - 2x Development machines, each has 2 Tesla V100
  - 12x Production machines, each has 2 Tesla V100
- System is time-shared
  - Users execute programs through a job submission system
  - Users can specified the number of resources and wall time for their jobs
  - More details will be announced in upcoming workshops

# Q&A

Thanks you