Introduction to GPU

Architecture & Programming

Intended Learning Outcomes

- After completing this course students should be able to:
  - Explain the difference between CPU & GPU architecture.
  - Understand GPU different components.
  - Evaluate the impact of memory hierarchy on GPU.
  - Learn the principle of accelerator designs (architecture features and constraints).
  - Implement and practice Cuda programming language.
  - Learn Principles and patterns of parallel algorithms.
  - Learn how to efficiently write parallel programs.

References

- Programming Massively Parallel Processors: A Hands-on Approach (5th edition).
- Cuda programming Guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- NVIDIA DLI Teaching Kit Program
- GPU Computing GEMs

Logistics

- TA: eng. Mohamed Abdallah
- Grades: (subjected to change)
  - Final 60%
  - Midterm 10%
  - Quizzes 5%
  - Labs & Assignments 15%
  - Big Assignment 10%
- Cheating is Strongly penalized.
- Course Page on elearn http://www.elearn.eng.cu.edu.eg/course/view.php?id=182

Course Modules

- Part I : Fundamentals
  - Introduction
  - GPU Architecture & Scheduling
  - Memory Architecture & Data Locality
  - Performance Consideration
  - Dynamic Parallelism
- Part II: Parallel Patterns
  - Convolution
  - Histogram
  - Reduction
  - Sparse Matrix computation
  - Graph traversal
  - Parallel Programming & Computational Thinking

Throughput: refers to the rate at which a system can process a certain amount of work over a given period of time => (Amount of completed work / unit time)
Usually we use the throughput when we want to talk about a system which can handle multiple tasks or operations concurrently, or the rate it can process a stream of data.
we represent is using OPS -> operations per second or IPC -> Instruction per cycle.
Usually we need Higher Throughput

Latency: refers to the delay between initiating a request, and recieving the corresponding response -> it measures the time taken to finish single task.
Usually we need lower Latency.
Usually CPUs have a better Latency than GPUs, however, GPUs are much more better in the Throughput.

General rule ->
GPUs -> Better Throughput (Higher Throughput) CPUs -> Better Latency (lower Latency)

so if we used both we can have a very fast system -> Hybrid Systems.

Agenda

- Why GPU
- CPU vs GPU
- Ways of Acceleration
- Cuda Computational Model

read about Tensor Cores: https://www.nvidia.com/en-us/data-center/tensor-cores/

Why GPU?

There are several reasons why GPUs (Graphics Processing Units) are often preferred over CPUs (Central Processing Units) for certain types of computations and workloads:

Parallel Processing Power:
GPUs are designed with a highly parallel architecture featuring thousands of smaller, more efficient cores compared to CPUs. This design makes GPUs exceptionally well-suited for parallelizable tasks that can be divided into many smaller computations, such as graphics rendering, scientific simulations, and deep learning.

Massive Parallelism:
GPUs excel at executing a large number of computations simultaneously, leveraging their numerous cores to process data in parallel. This enables GPUs to handle massive workloads more efficiently than CPUs, especially for tasks that can be decomposed into many independent subtasks.

High Performance for Vectorized Operations:
GPUs are optimized for performing vectorized operations on large datasets, such as matrix multiplications and element-wise computations. Their architecture allows for efficient execution of these operations across multiple elements simultaneously, resulting in significantly higher performance compared to CPUs for such tasks.

Specialized Compute Capabilities:
Modern GPUs often include specialized compute units, such as Tensor Cores for accelerating deep learning computations or dedicated hardware for handling specific types of mathematical operations. These features make GPUs well-suited for a wide range of compute-intensive applications beyond graphics processing.

Energy Efficiency for Parallel Workloads:
While GPUs consume more power than CPUs, they often deliver higher performance per watt for parallel workloads due to their ability to efficiently utilize their large number of cores. This energy efficiency can translate to lower operating costs and reduced environmental impact, especially in data centers and high-performance computing environments.

Cost-Effectiveness:
GPUs can offer better price-performance ratios compared to CPUs for certain tasks, particularly those that benefit from parallel processing. While GPUs may have higher upfront costs, their ability to handle large workloads efficiently can result in significant cost savings over time, especially for tasks like deep learning training or scientific simulations.

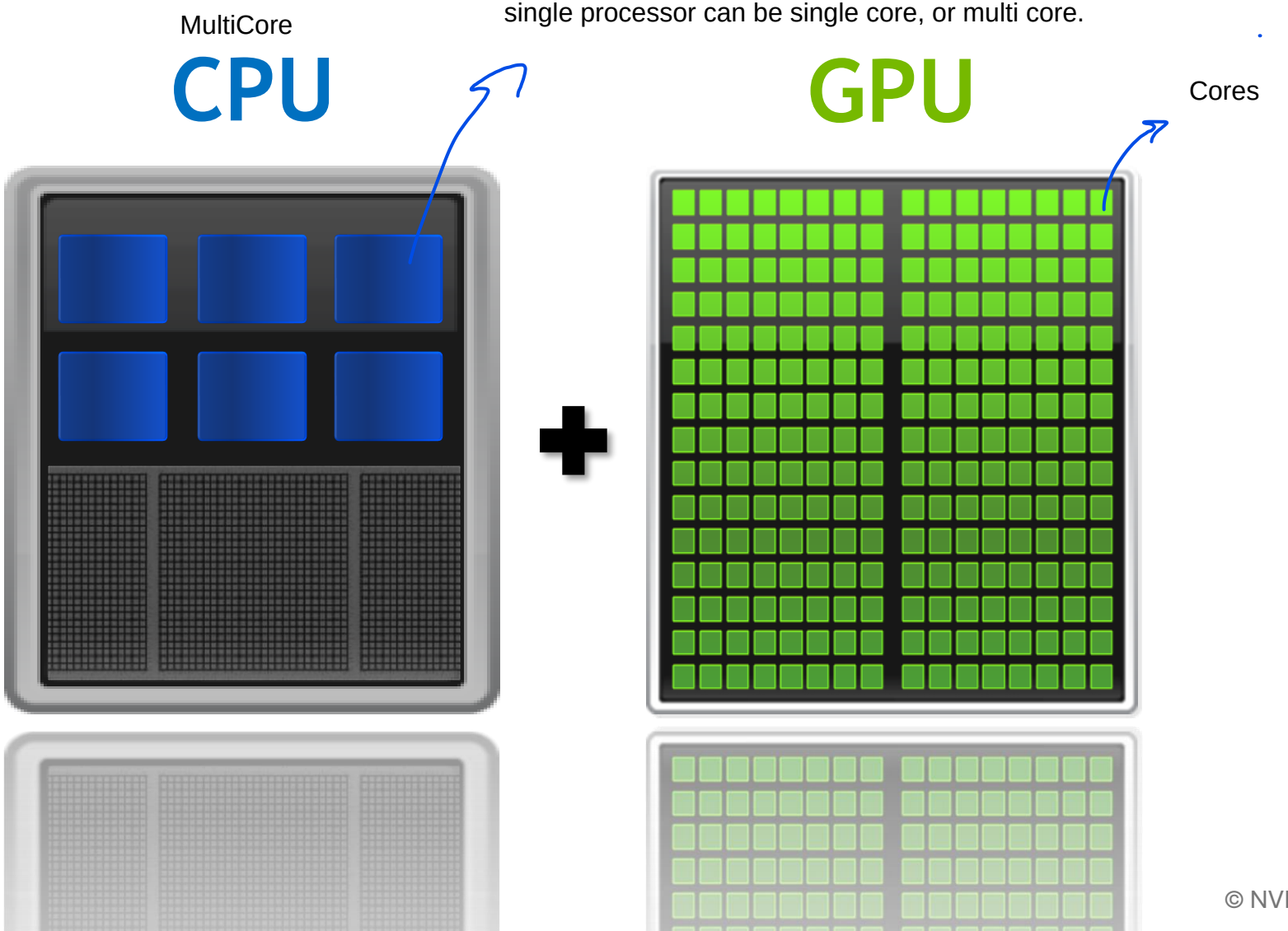
Scalability:
GPU clusters can be easily scaled by adding more GPUs to a system, allowing for linear increases in computational power. This scalability makes GPUs a preferred choice for applications that require high-performance computing resources and can benefit from distributed parallel processing.

Overall, GPUs offer significant advantages over CPUs for certain types of computations, particularly those that are highly parallelizable and require processing large datasets efficiently. By leveraging their parallel processing power, specialized compute capabilities, and energy efficiency, GPUs have become indispensable tools for accelerating a wide range of applications across various domains, including scientific research, artificial intelligence, computer graphics, and data analytics.

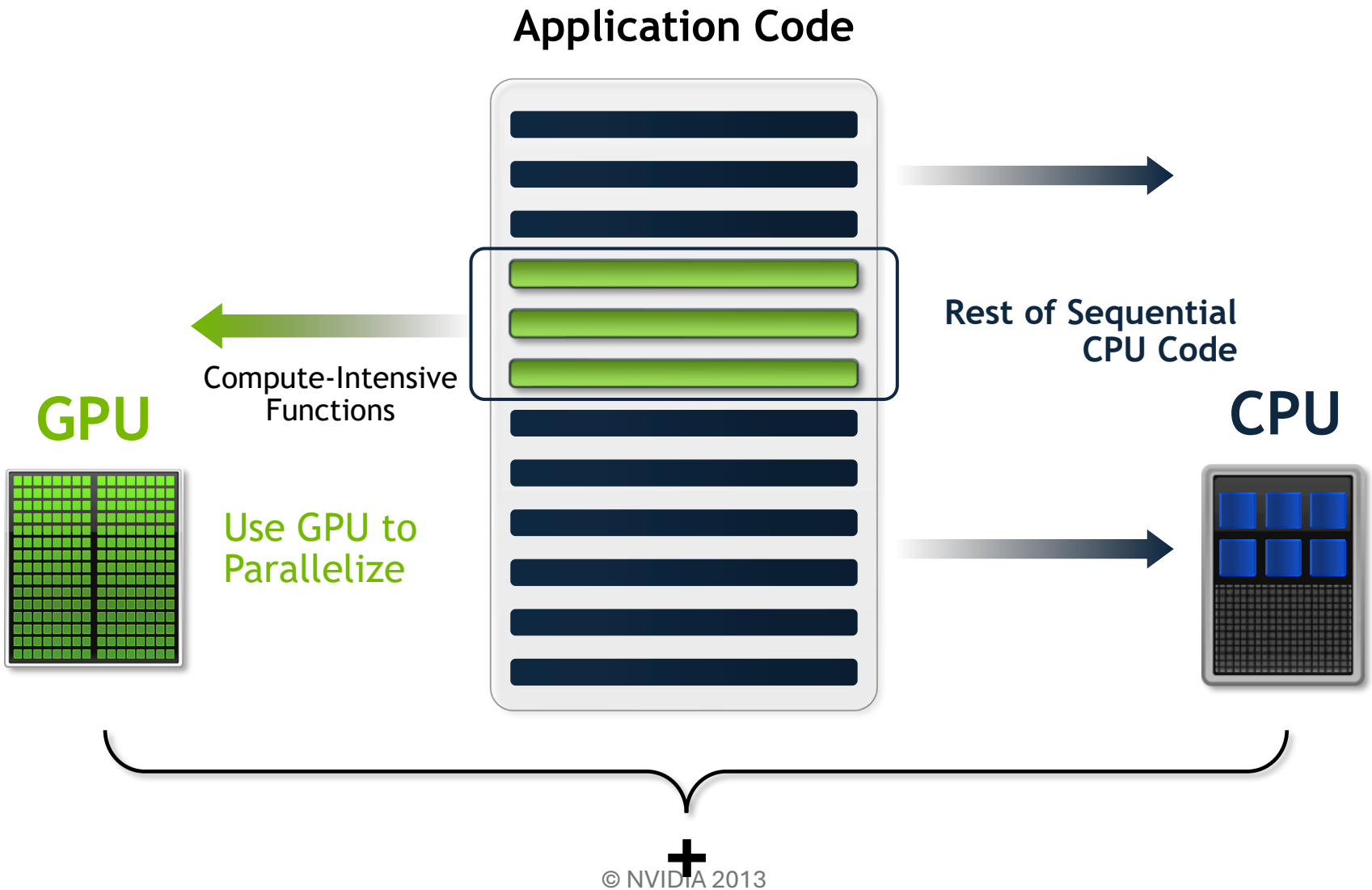
usually we use the GPU when we need to apply the same operation on multiple different data -> zy fl convelution keda msln 3ndk mask, w 3auz ttb2o 3la matrix, fa hya nfs el operations (dot product + summation) 3la kol el matrix, de bnsameha (SIMD -> Single instruction, multiple Data)

watch this amazing video!
https://www.youtube.com/watch?v=-P28LKWTzrI

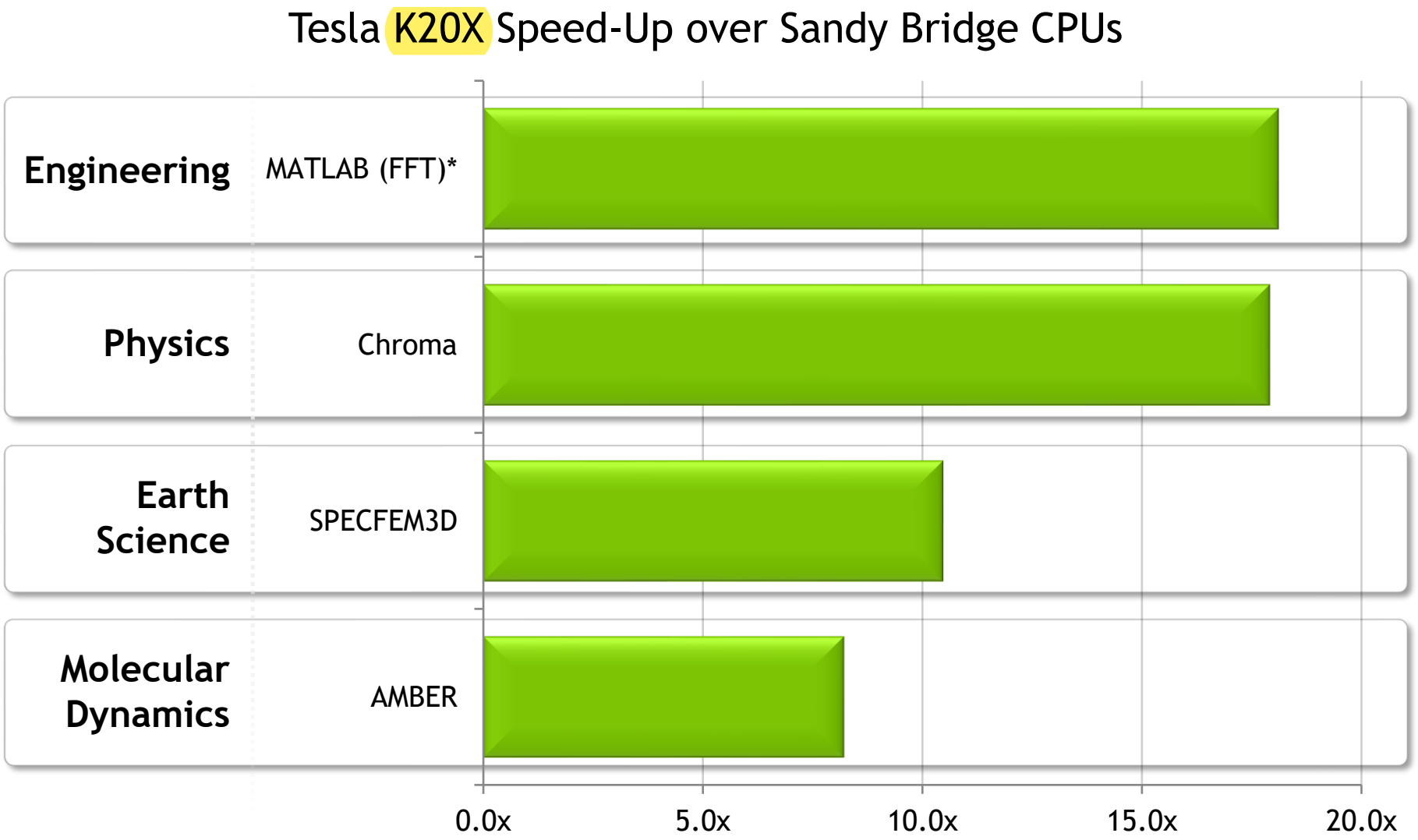
Core -> individual processing unit -> elly bn7ot feh baa el ALU wl registers w keda, single processor can be single core, or multi core.



Small Changes, Big Speed-up



Fastest Performance on Scientific Applications



CPU results: Dual socket E5-2687w, 3.10 GHz, GPU results: Dual socket E5-2687w + 2 Tesla K20X GPUs
*MATLAB results comparing one i7-2600K CPU vs with Tesla K20 GPU
Disclaimer: Non-NVIDIA implementations may not have been fully optimized

Why Computing Perf/Watt Matters?

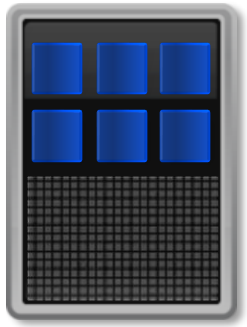
2.3 PFlops

7.0 Megawatts

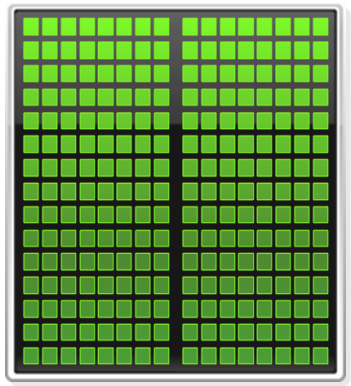
7000 homes

7.0 Megawatts

CPU
Optimized for Serial Tasks



GPU Accelerator
Optimized for Many Parallel Tasks



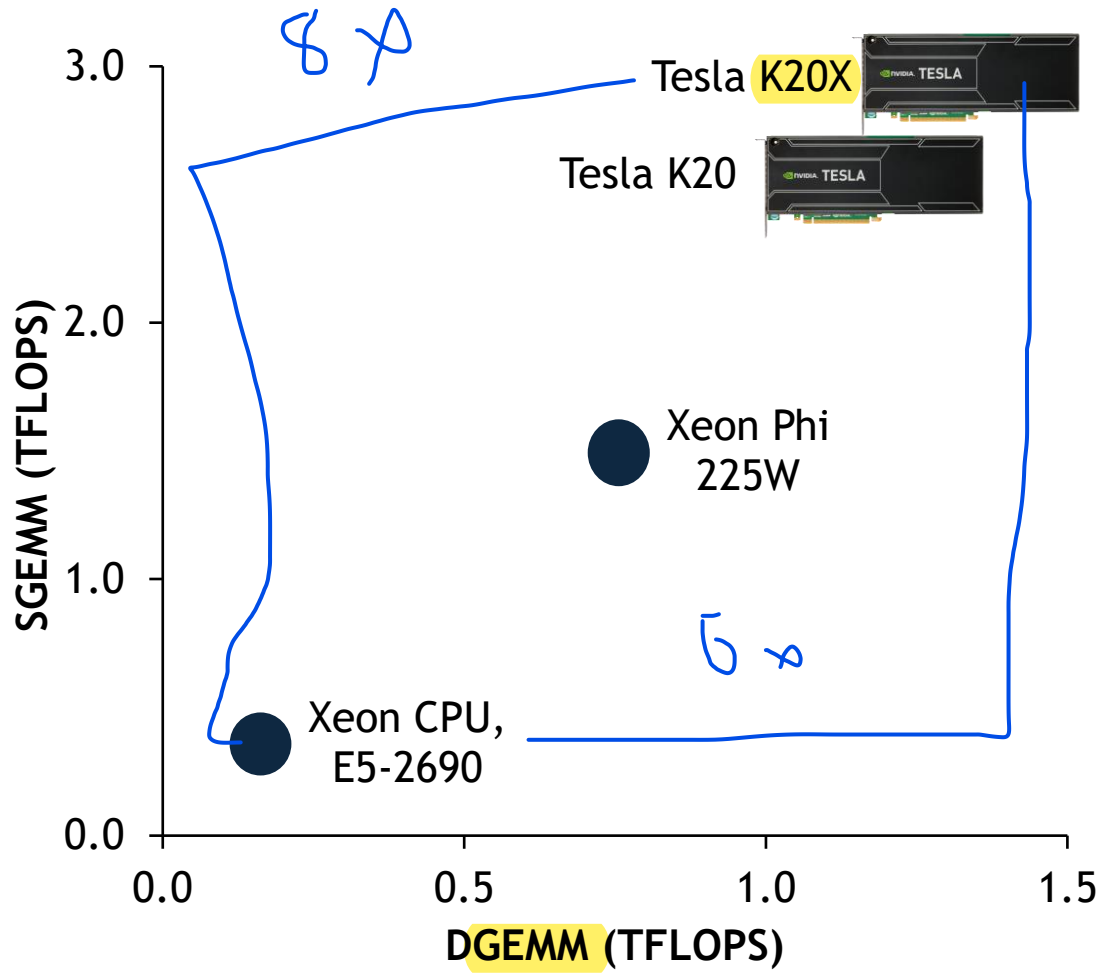
10x performance/socket

> 5x energy efficiency

Traditional CPUs are not economically feasible

Era of GPU-accelerated computing is here

World's Fastest, Most Energy Efficient Accelerator



Tesla K20X vs Xeon CPU

8x Faster SGEMM

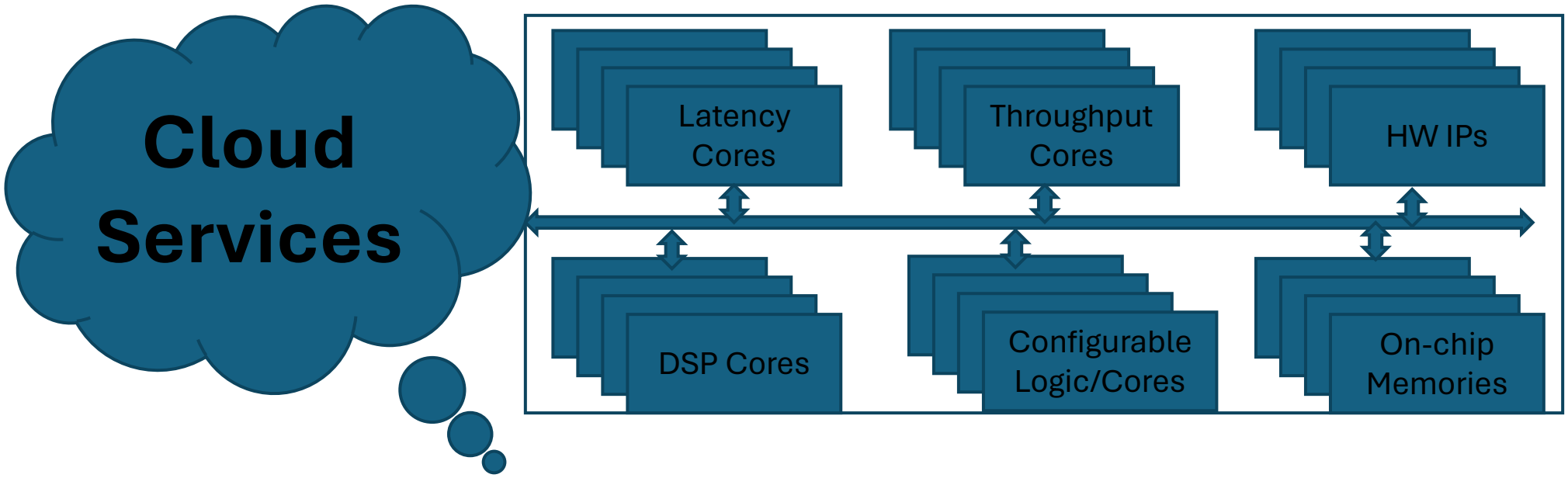
6x Faster DGEMM

Tesla K20X vs Xeon Phi

90% Faster SGEMM

60% Faster DGEMM

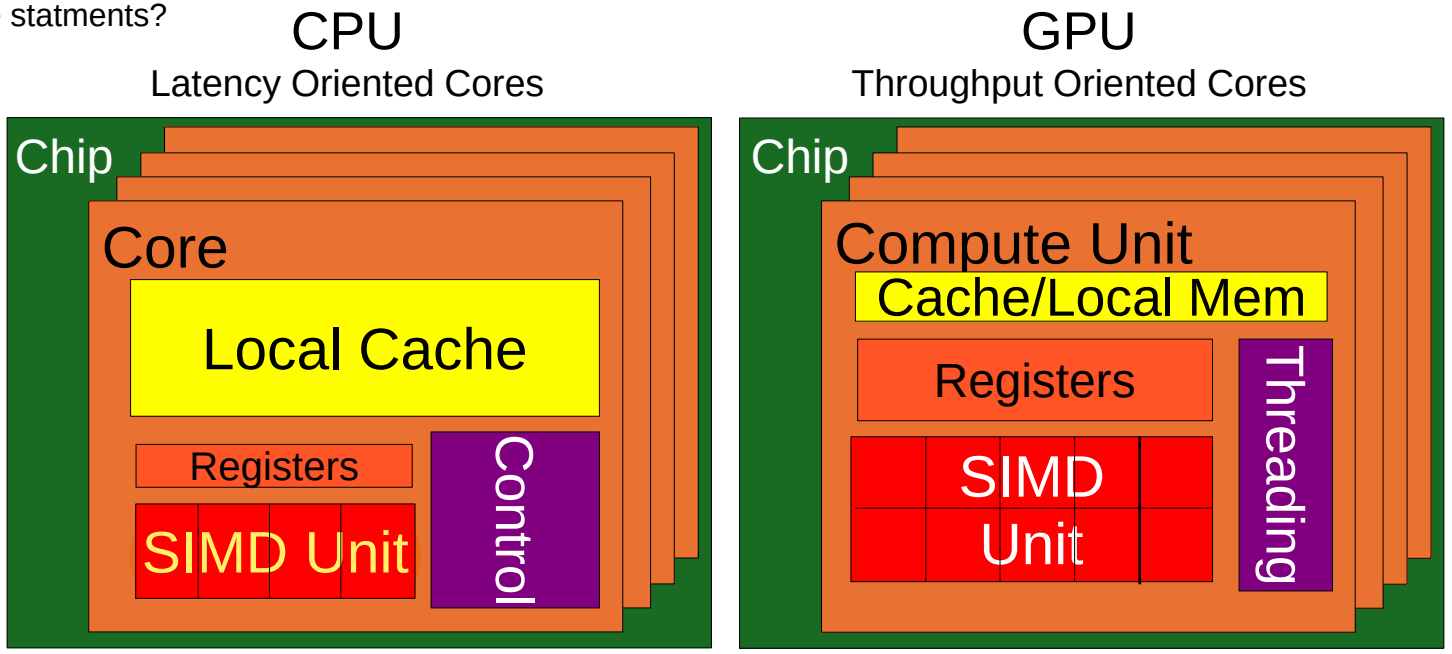
– Use the best match for the job (heterogeneity in mobile SOC)



What are the diffrences between the two SOC?

Local Caches are larger at CPUs, than on GPUs
Registers on CPUs are less than GPUs
There is a Control Block on CPUs, but Threding Block on GPUs
SIMD Units in CPUs are less than on GPUs

Give a reason for each of the above statments?



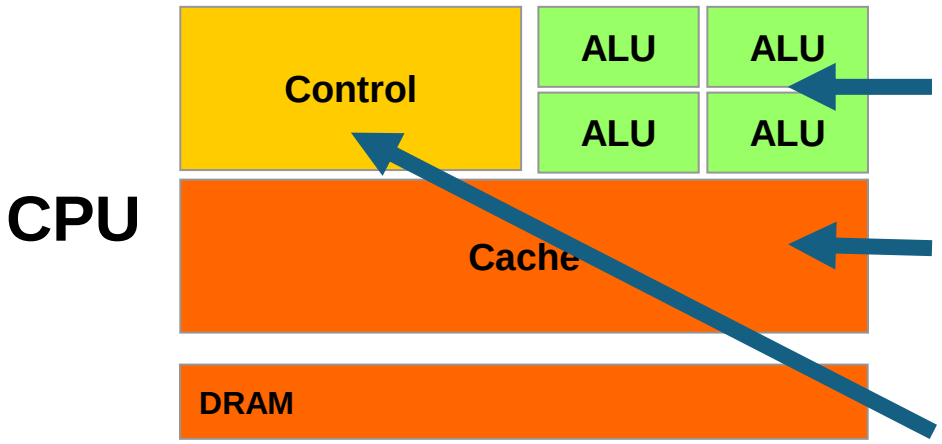
1. The small size of Cache on the GPUs is mainly due to the tradeoff of having a small area, so we prefere to give this area to other components rather than the cache, even if having a larger cache will enhance our data locality. moreover, GPUs are mainly designed to apply parallel workloads, the parallel nature of the computations and the large number of processing elements, make it less critical to have a large local cache for each individual processing unit.

2. It is obvious that we need more registers in GPUs than in CPUs, because in GPUs, we have multiple threads, so each thread needs its own register to avoid resources preservation, lazm kol thread ykon m3aha el registers bto3ha, 34an te2dr enha t3ml kol el operations, mn gher ma tefdl mestnya 7d tany y5ls 34an takhud el registers bto3o.

3. Control Block in CPU is used to generate the control signal to allow the CPU to perform certain tasks, which should go the ALU, when we should apply forwarding, and so on. on the other hand, the Threading block in the GPU is responsible for executing group of threads together on the GPU's processors, they are managed and schaduled by the GPU's hardware schadular, which schadule which thread should work now, and what is the algorithm that should be applied and so on.

4. We need more SIMD Units in GPUs because the SIMD allows us to apply the same instruction on multiple Data, which is exactly what we created GPUs for, so it helps us in performing better parallelism.

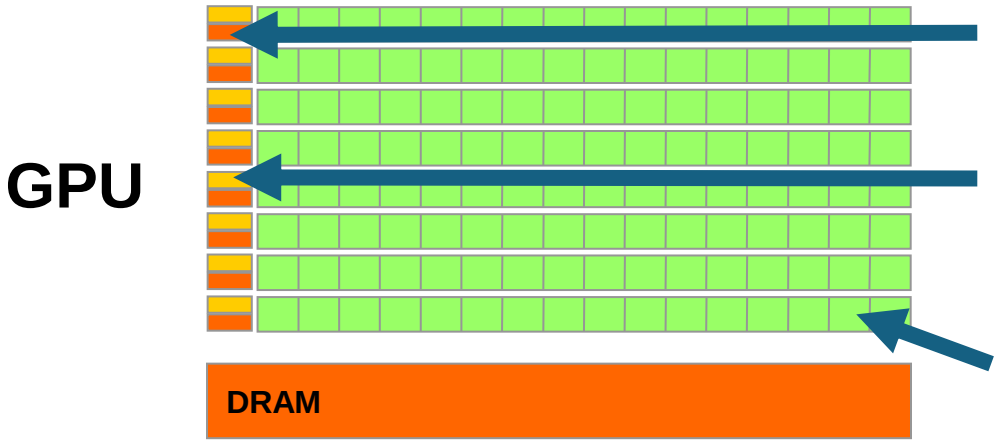
CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

follow the coloring schemes.

GPUs: Throughput Oriented Design



- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy-efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require a massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Winning Applications Use Both CPU and GPU

CPUs for sequential parts where latency matters

CPUs can be 10X+ faster than GPUs for sequential code

GPUs for parallel parts where throughput wins

GPUs can be 10X+ faster than CPUs for parallel code

SIMPLE EXAMPLE

Add two arrays, A and B to produce array C

- A[ ] + B[ ] -> C[ ]

On the CPU:

```
float *C = malloc(N * sizeof(float));
for (int i = 0; i < N; i++)
C[i] = A[i] + B[i];
return C;
```

*On CPUs the above code operates **sequentially**, but can we do better, still on CPUs?*

what is the difference between multi-Threding, and multi-cores processor?

MutliThreding -> on the same core, we have multiple threads, and usually up to 8 threads.

MultiCores -> we have multiple cores on the same chip.

SIMPLE EXAMPLE

- On the CPU (multi-threaded, pseudocode):

```
(allocate memory for array C)
Create # of threads equal to number of cores on processor (around
2, 4, perhaps 8?)
(Indicate portions of arrays A, B, C to each thread...)


...


In each thread,
For (i from beginning region of thread)
C[i] <- A[i] + B[i]
//lots of waiting involved for memory reads, writes, ...
Wait for threads to synchronize...
```

*This is **slightly** faster – 2-8x (can be slightly more with other tricks)*

SIMPLE EXAMPLE

- How many threads are available on the CPUs? How can the performance scale with thread count?
  - (Each CPU can generally have two threads).

- Context switching:
  - The action of switching which thread is being processed
  - **High penalty** on the CPU (main computer)
  - Not a big issue on the GPU

SIMPLE EXAMPLE

- On the GPU:

  (allocate memory for arrays A, B, C on GPU)
  Create the “kernel” – where each thread will perform one (or a few) additions
  Specify the following kernel operation:

  For all i‘s (indices) assigned to this thread:
  C[i] <- A[i] + B[i]

  Start ~**20000 (!)** threads all at the same time!
  Wait for threads to synchronize...

How to get benefit of GPU?

3 Ways to Accelerate Applications

Easy to use
Most Performance

Easy to use
Portable code

Most Performance
Most Flexibility

Libraries: Easy, High-Quality Acceleration

– Ease of use: Using libraries enables GPU acceleration without in-depth knowledge of GPU programming

– "Drop-in": Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes

– Quality: Libraries offer high-quality implementations of functions encountered in a broad range of applications

NVIDIA GPU Accelerated Libraries

DEEP LEARNING

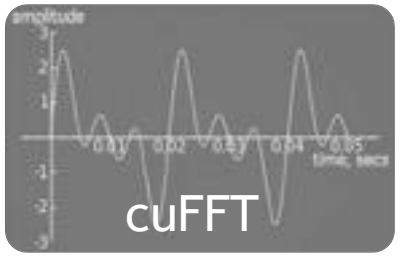
cuDNN


TensorRT

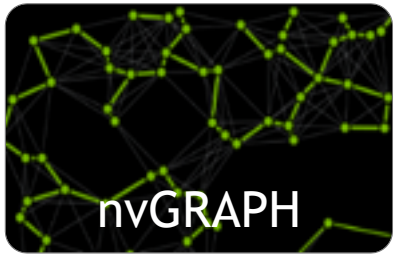
DeepStream SDK

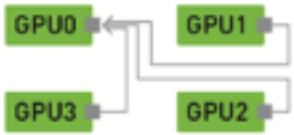
LINEAR ALGEBRA

SIGNAL, IMAGE, VIDEO



PARALLEL ALGORITHMS





NCCL

Vector Addition in Thrust

```
#include <thrust/device_vector.h>
#include <thrust/copy.h>

int main(void) {
  size_t inputLength = 500;
  thrust::host_vector<float> hostInput1(inputLength);
  thrust::host_vector<float> hostInput2(inputLength);
  thrust::device_vector<float> deviceInput1(inputLength);
  thrust::device_vector<float> deviceInput2(inputLength);
  thrust::device_vector<float> deviceOutput(inputLength);

  thrust::copy(hostInput1.begin(), hostInput1.end(), deviceInput1.begin());
  thrust::copy(hostInput2.begin(), hostInput2.end(), deviceInput2.begin());

  thrust::transform(deviceInput1.begin(), deviceInput1.end(),
                    deviceInput2.begin(), deviceOutput.begin(),
                    thrust::plus<float>());
}
```

Compiler Directives: Easy, Portable Acceleration

- Ease of use: Compiler takes care of details of parallelism management and data movement
- Portable: The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- Uncertain: Performance of code can vary across compiler versions

OpenACC

– Compiler directives for C, C++, and FORTRAN

```
#pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLength]),
              copyout(output[0:inputLength])
    for(i = 0; i < inputLength; ++i) {
        output[i] = input1[i] + input2[i];
    }
```

Programming Languages: Most Performance and Flexible Acceleration

- Performance: Programmer has best control of parallelism and data movement
- Flexible: The computation does not need to fit into a limited set of library patterns or directive types
- Verbose: The programmer often needs to express more details

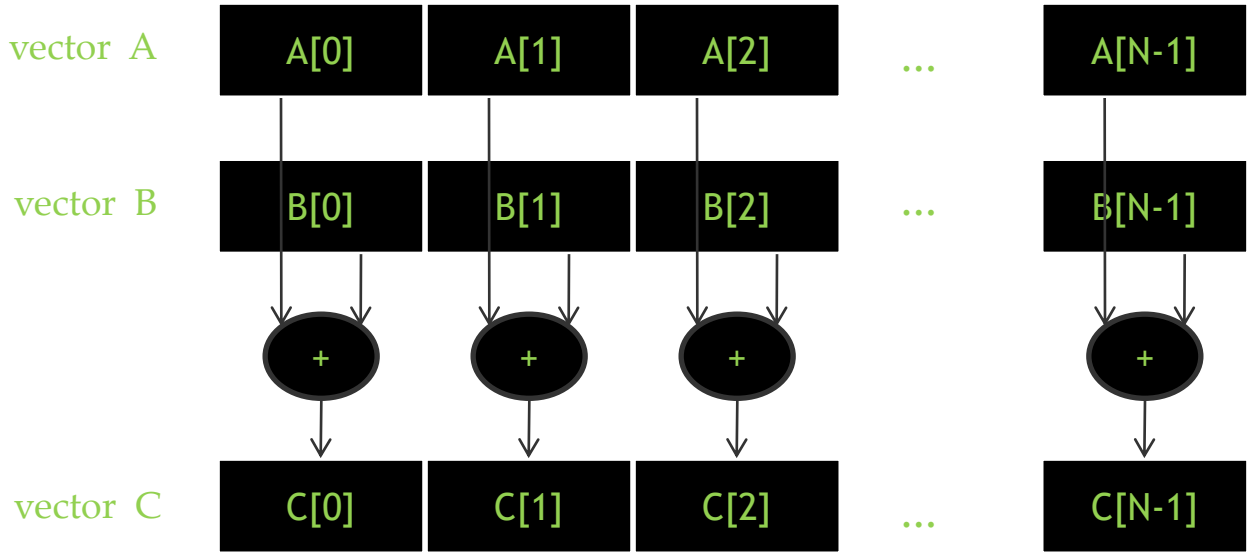
CUDA - C

Easy to use
Most Performance

Easy to use
Portable code

Most Performance
Most Flexibility

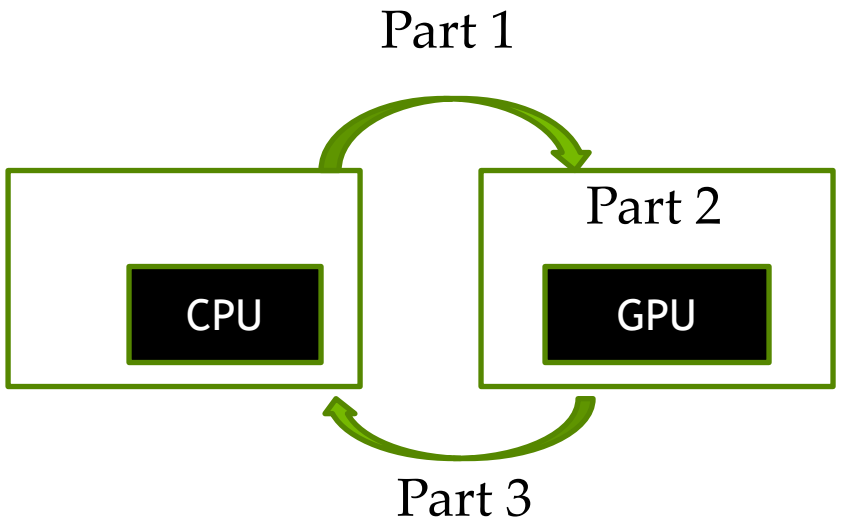
Data Parallelism - Vector Addition Example



Vector Addition – Traditional C Code

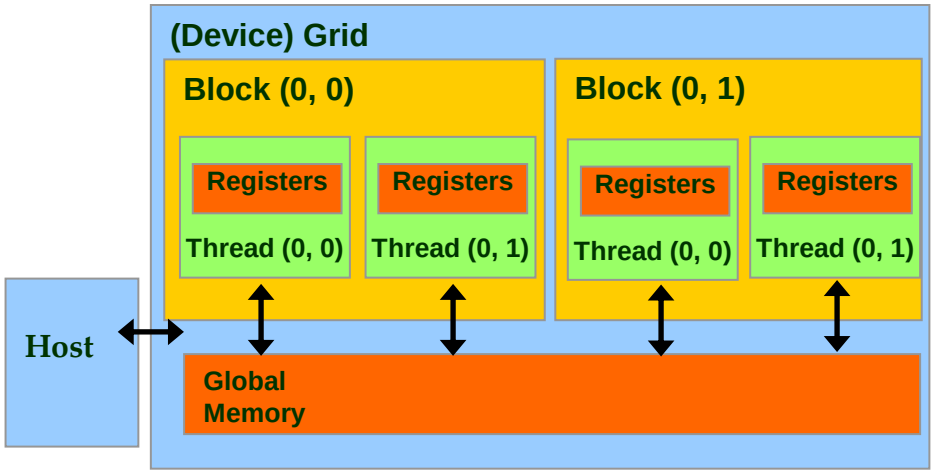
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

Heterogeneous Computing vecAdd CUDA Host Code




```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory

    // Part 2
    // Kernel launch code – the device performs the actual vector addition

    // Part 3
    // copy C from the device memory
    // Free device vectors
}
```

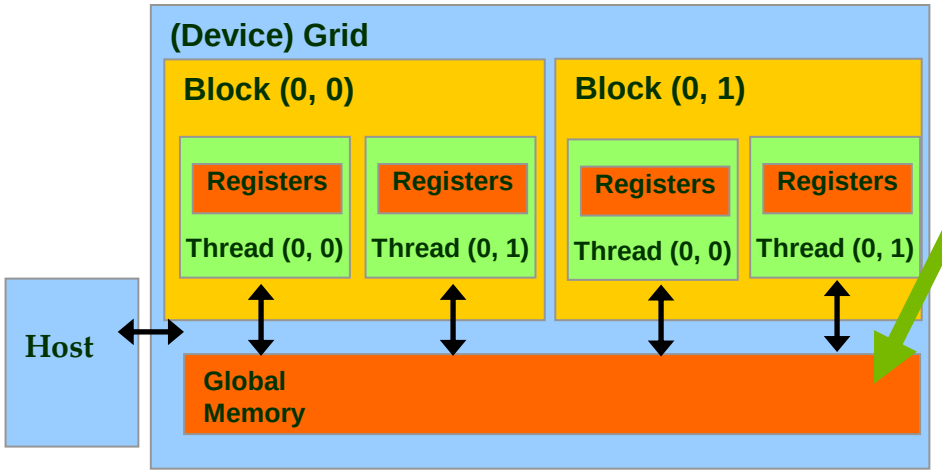
Partial Overview of CUDA Memories



- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

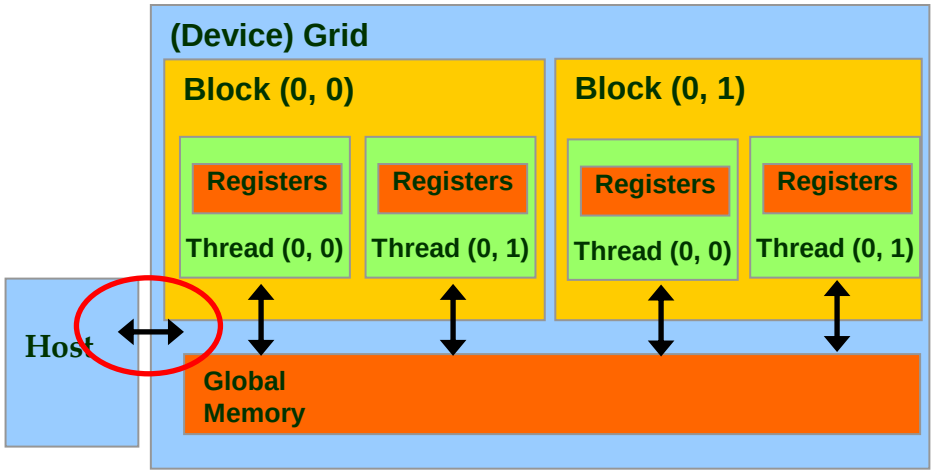
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

Host-Device Data Transfer API functions



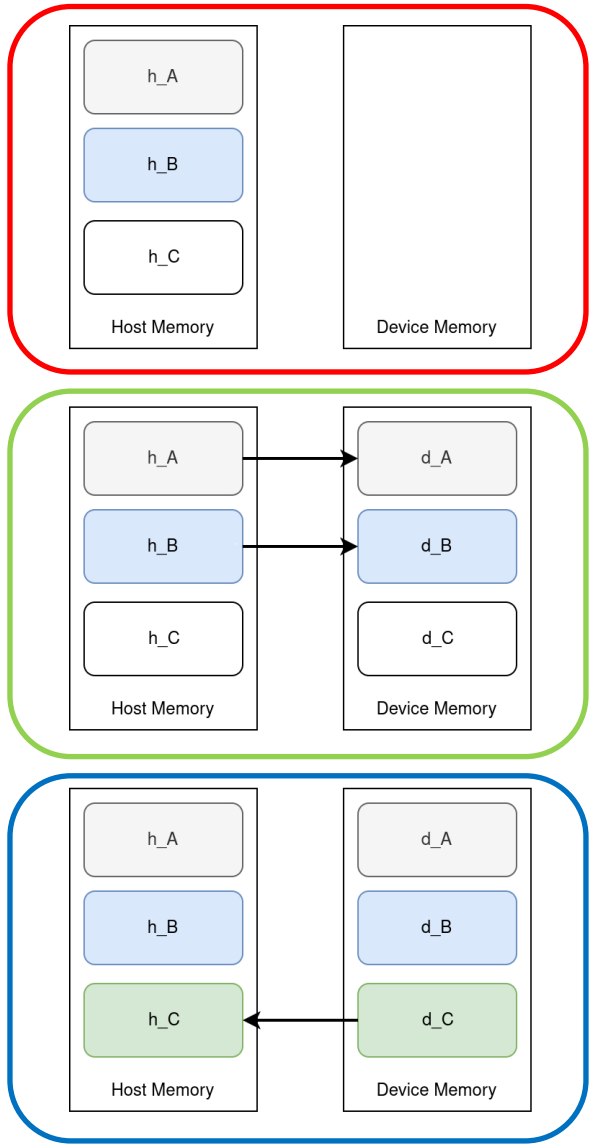
- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is synchronous with respect to the host

Vector Addition, Explicit Memory Management

… Allocate h_A, h_B, h_C …

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

     // Kernel invocation code – to be shown later

     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
     cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

… Free h_A, h_B, h_C …



In Practice, Check for API Errors in Host Code

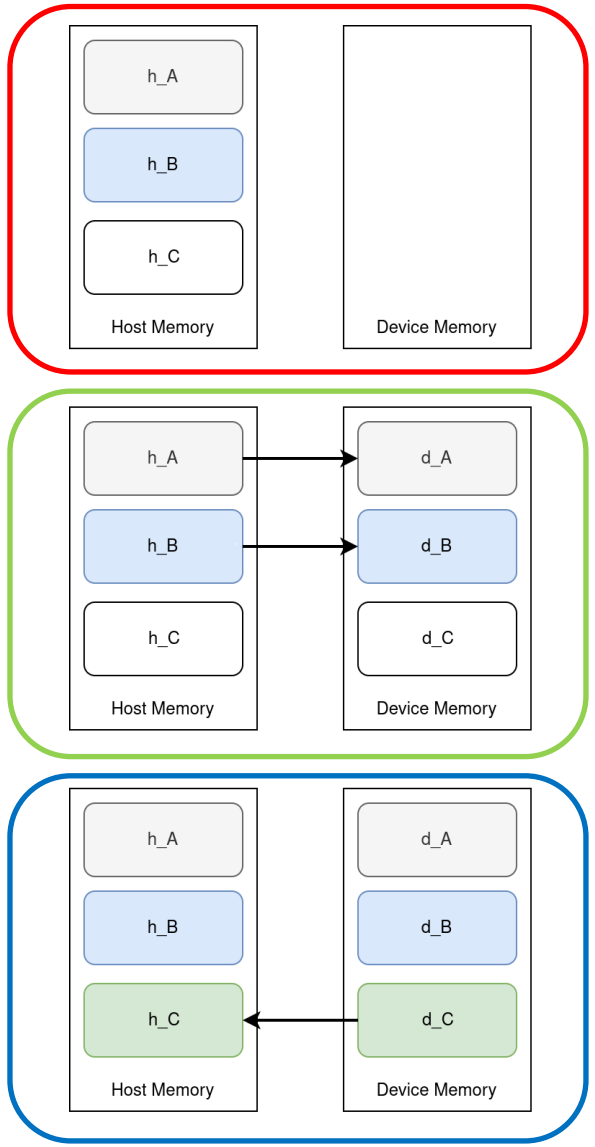
```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf("%s in %s at line %d\n",   cudaGetErrorString(err), __FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```

Vector Addition, Explicit Memory Management

… Allocate h_A, h_B, h_C …

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

     // Kernel invocation code – to be shown later

     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
     cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

… Free h_A, h_B, h_C …



How to write the Kernel?

To be continued. ...

Summary

- Why GPU
- CPU vs GPU
- Ways of Acceleration
- Cuda Computational Model