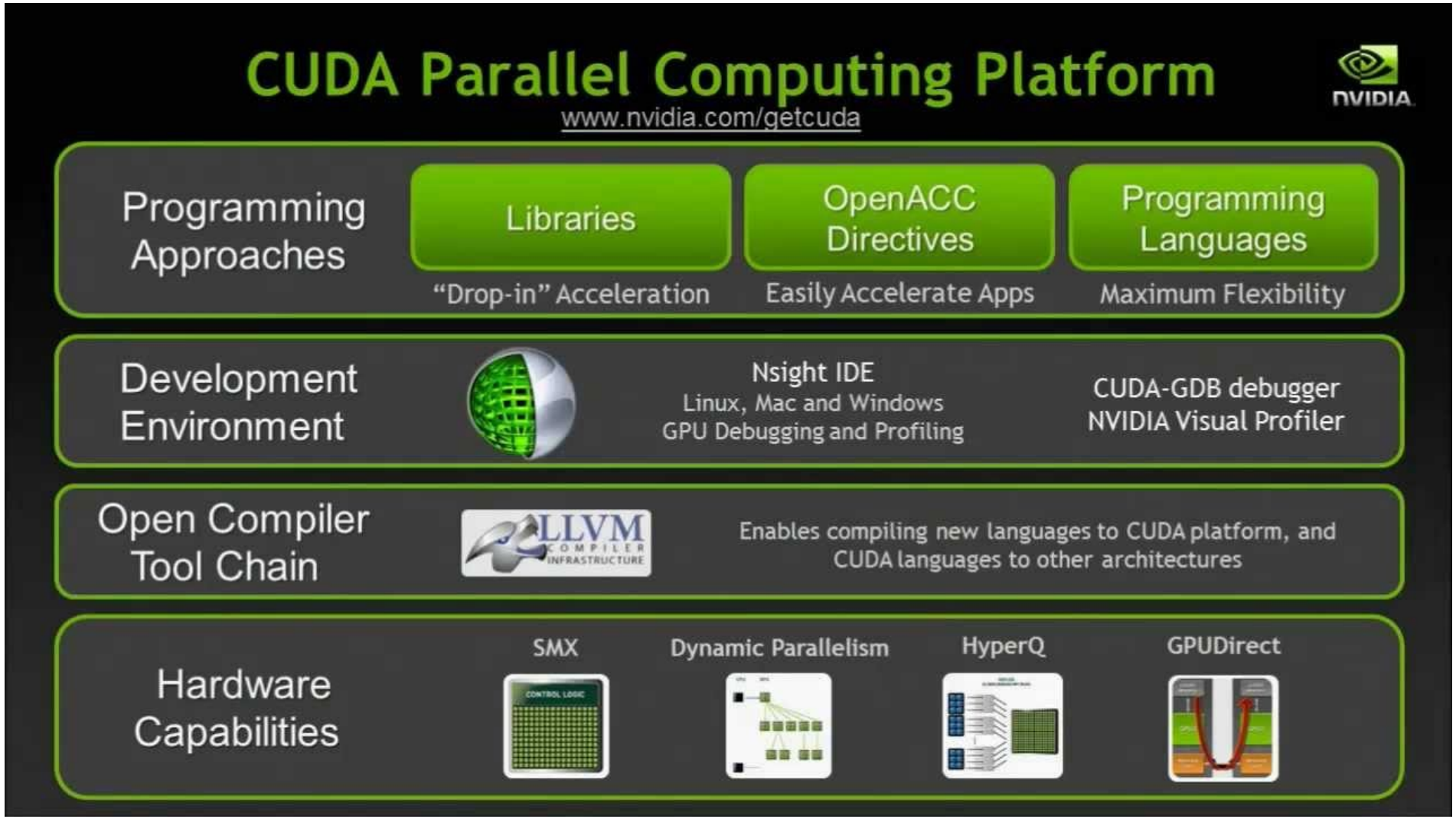
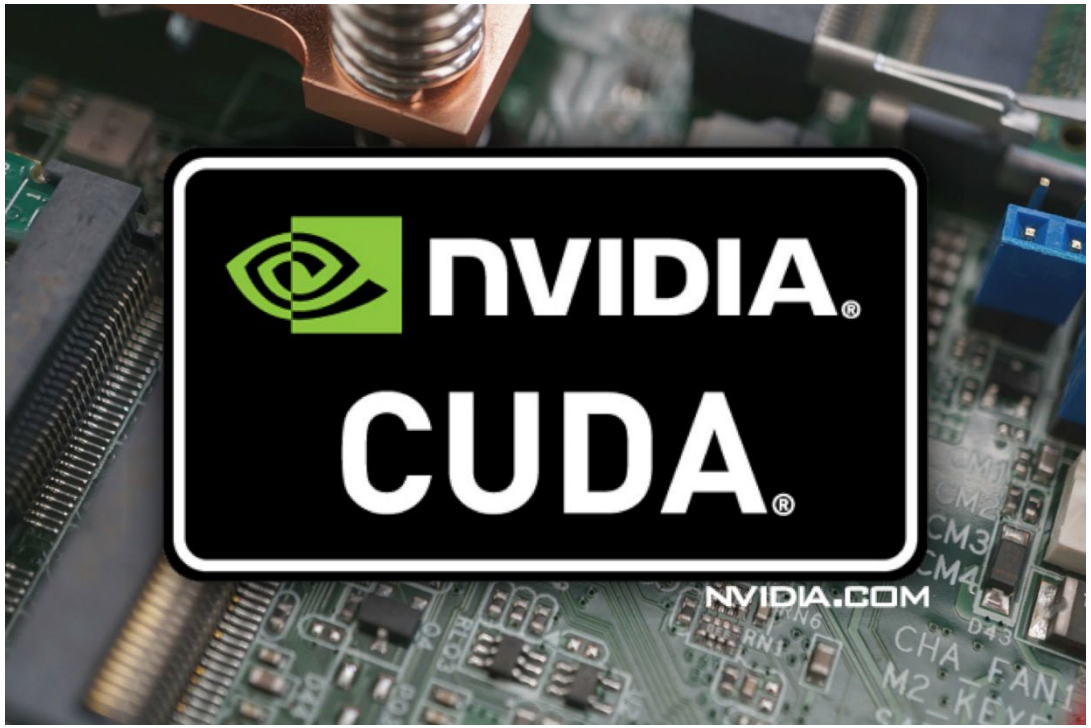
CUDA Programming

Hamdy Sultan



CUDA Programming

- **C**ompute **U**nified **D**evice **A**rchitecture (**CUDA**).
  - Parallel computing platform and application programming interface (API).
  - GPGPU (General-Purpose computing on Graphics Processing Units).

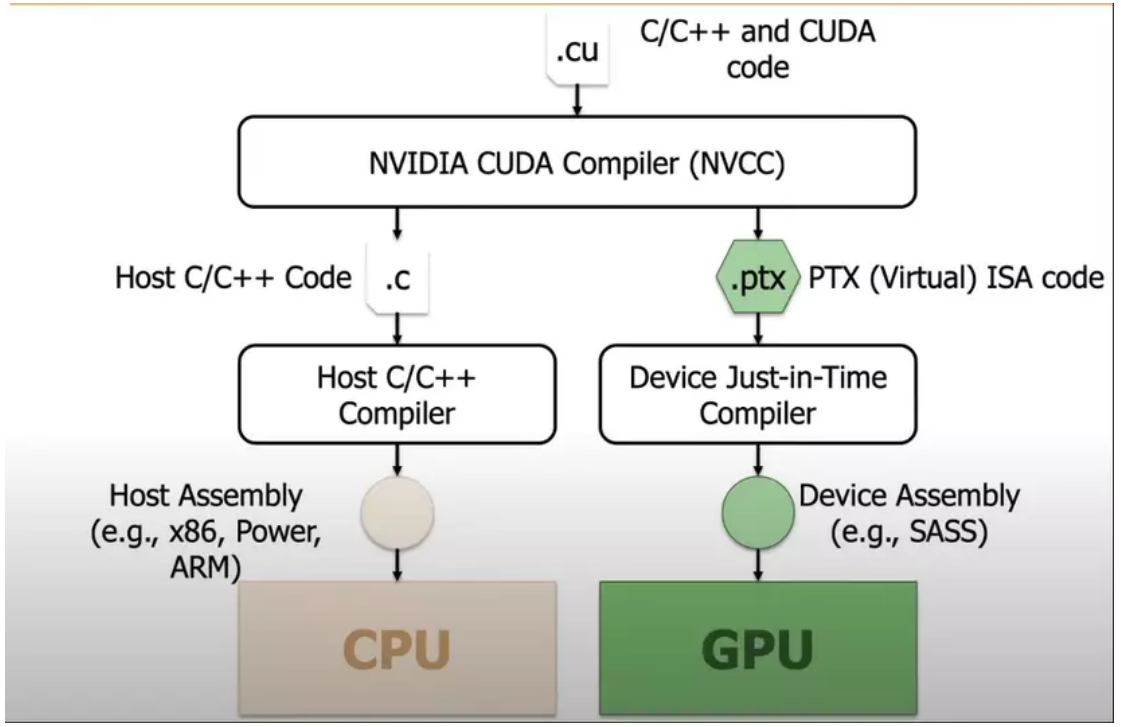


https://www.spo-comm.de/en/blog/know-how/nice-to-know-what-is-cuda

CUDA Programming

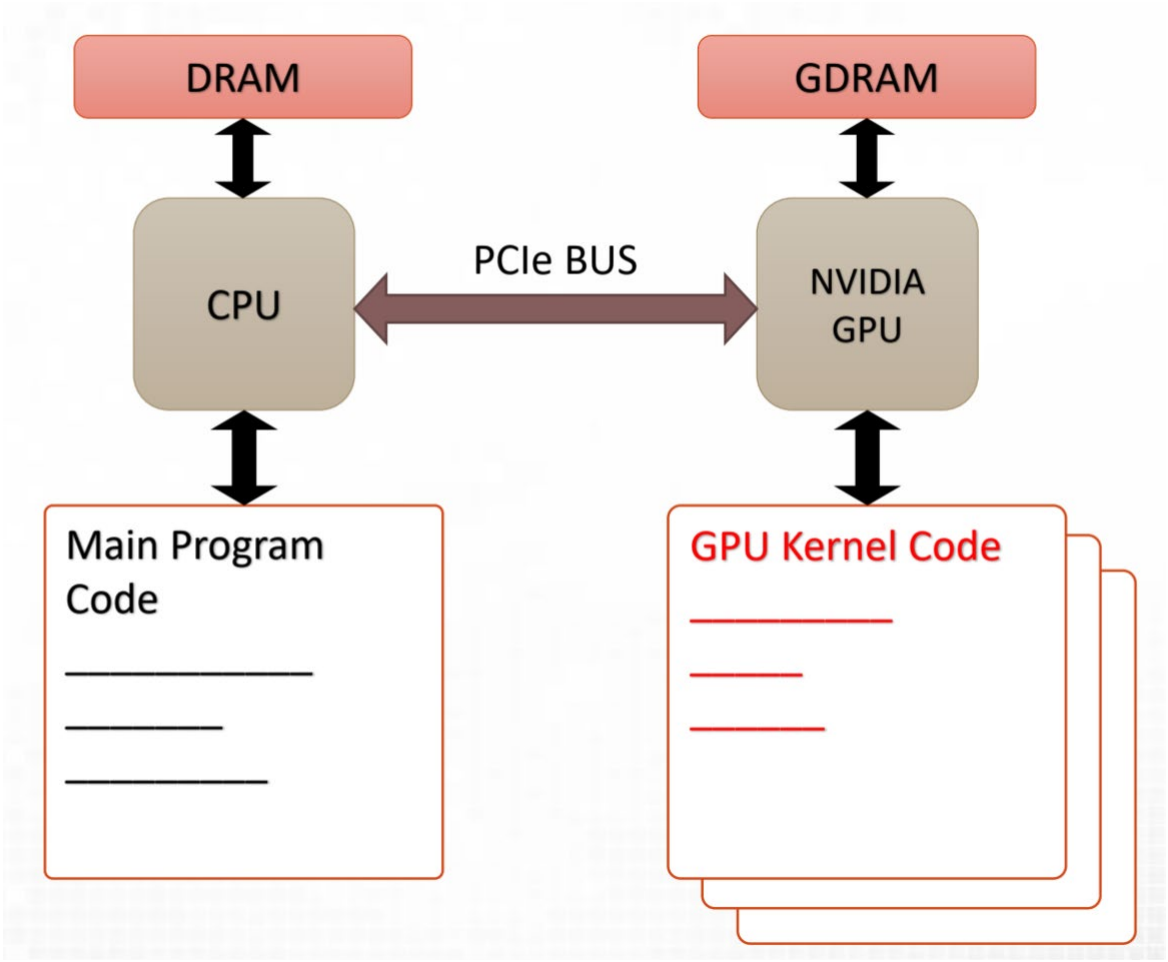
- **C**ompute **U**nified **D**evice **A**rchitecture (**CUDA**).

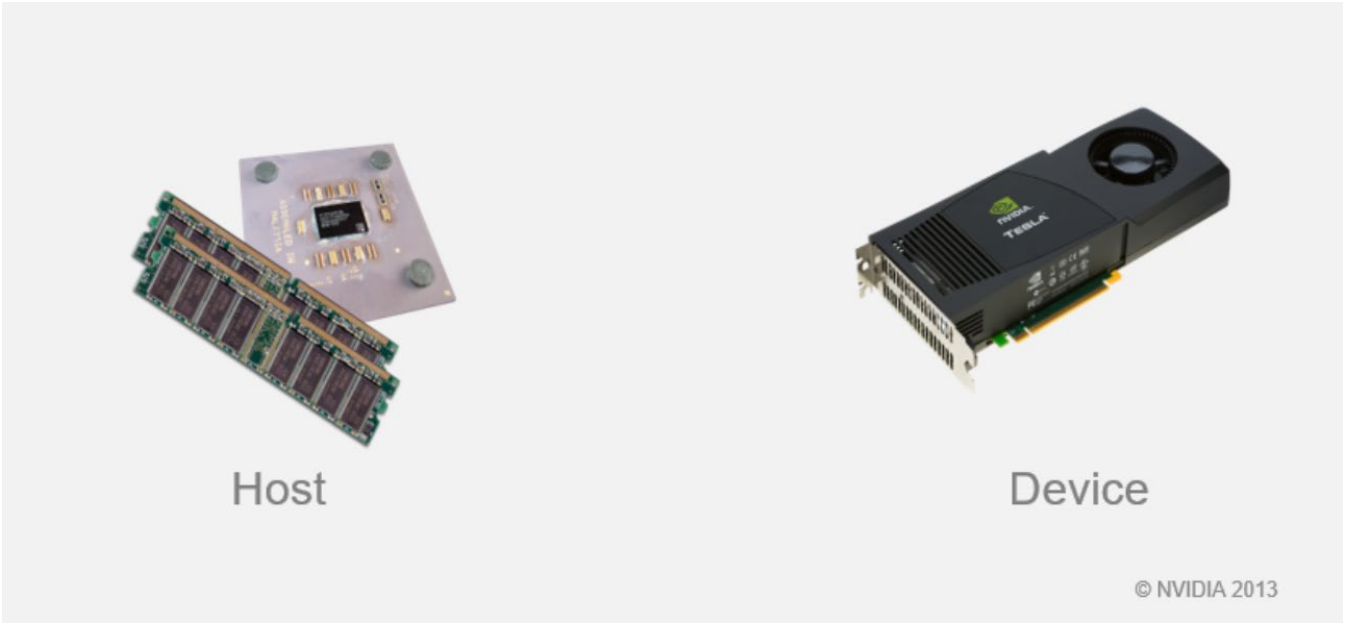
1. Based on the C programming language.
2. The CUDA compiler is called NVCC.
   - Not just a compiler.
   - Fully utilizing the GPU resources.



To start,

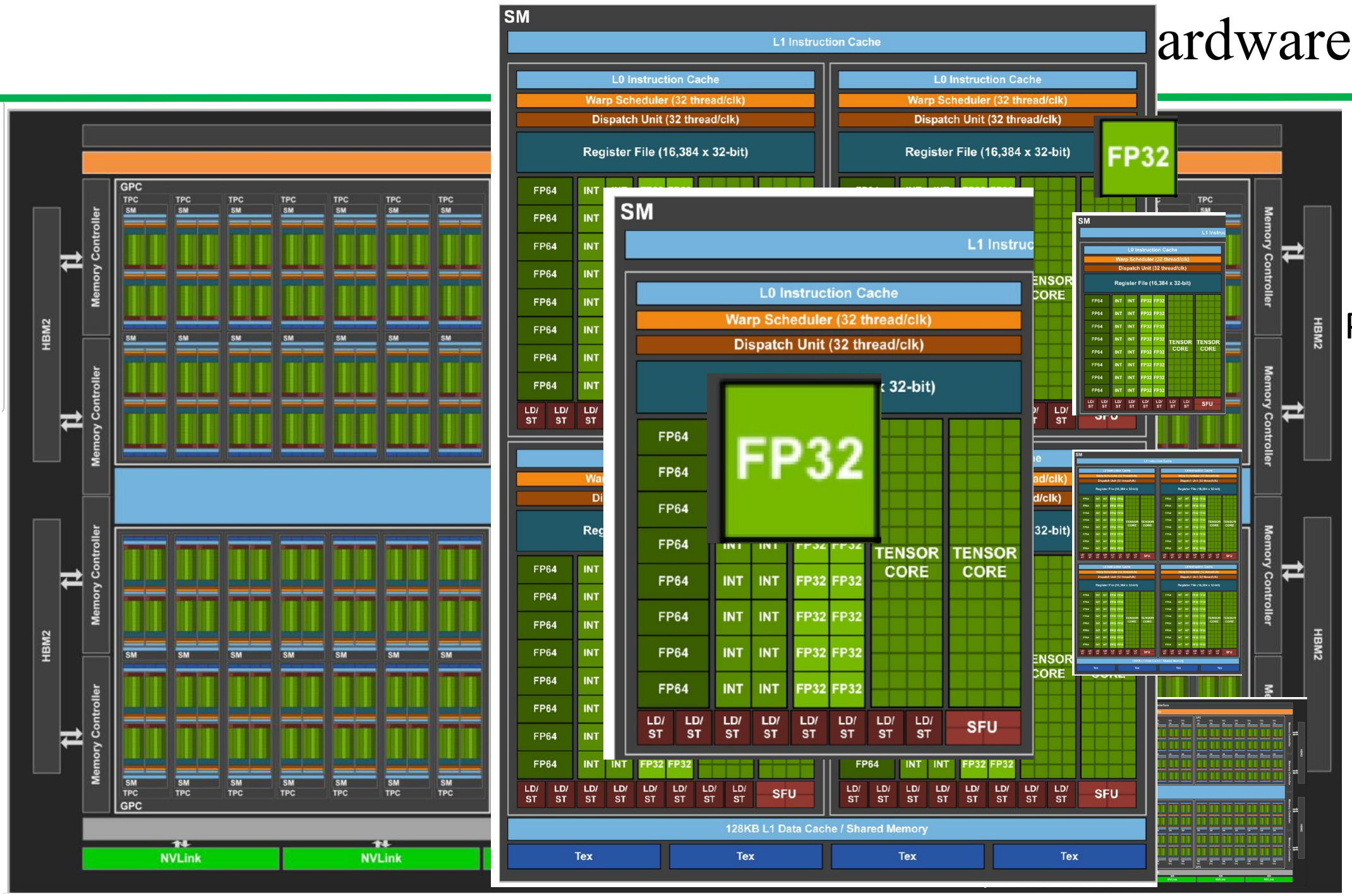
- **Host and Device?**





Src: Lecture 08 - Part 01 - CUDA Programming Model.pdf - Google Drive

ardware



Core

Partition

SM

GPU

Software

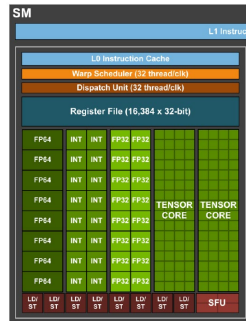
Hardware

GPU application

Block	Block	Block	Block
Block	Block	Block	Block
Block	Block	Block	Block
Block	Block	Block	Block
Block	Block	Block	Block
Block	Block	Block	Block

GigaThread

Core



Partition

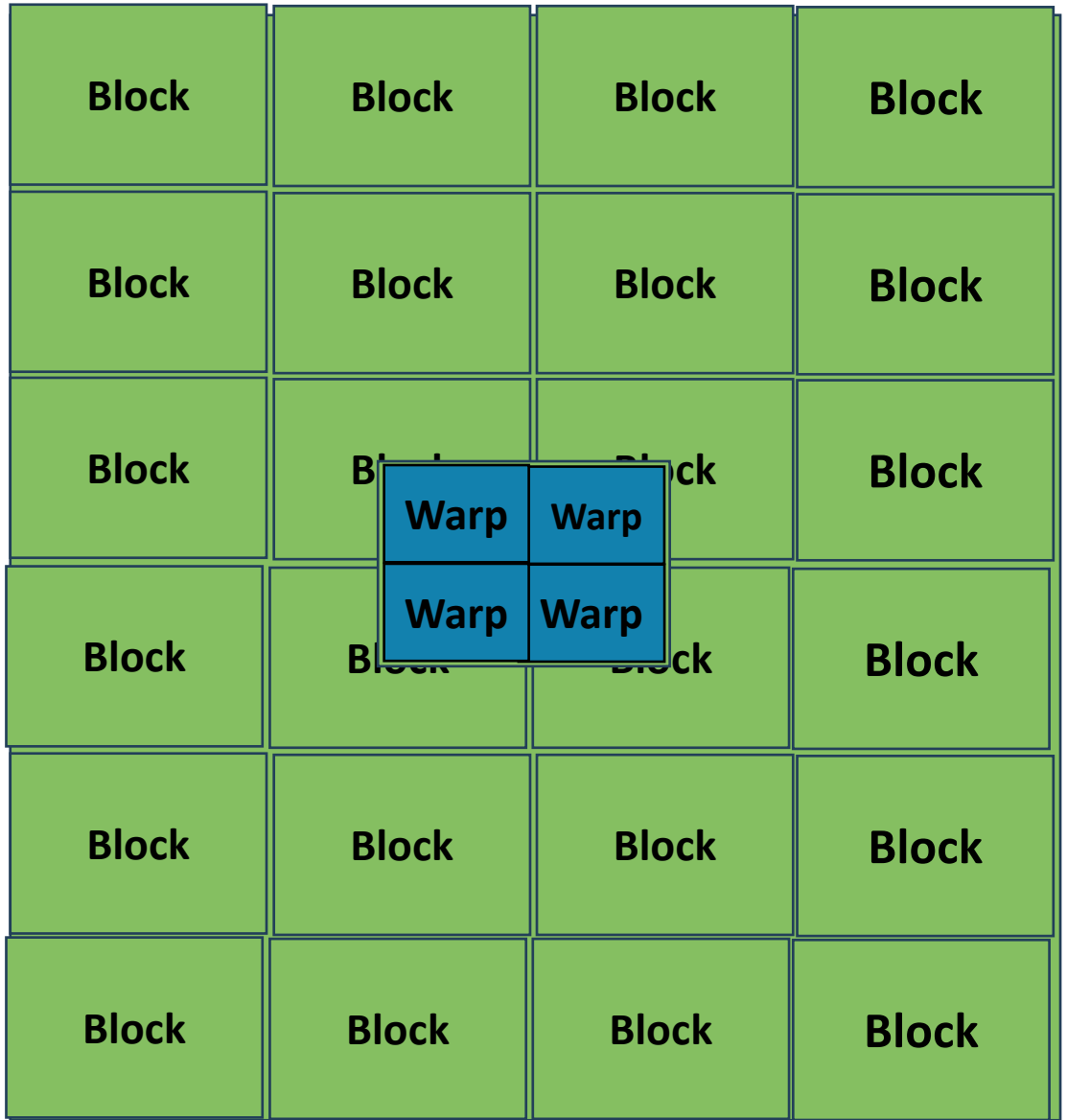
SM

GPU

Software

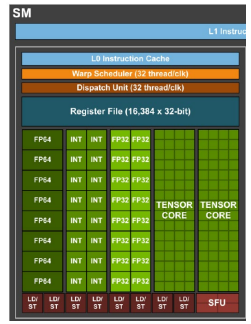
Hardware

GPU application



GigaThread

Core



Partition

SM

GPU

Software

Hardware

GPU application

Core

Warp Scheduler

Partition

GigaThread

SM

GPU

Software

Hardware

GPU application

Dispatcher

Core

Warp Scheduler

Partition

GigaThread

SM

GPU

CUDA Programming

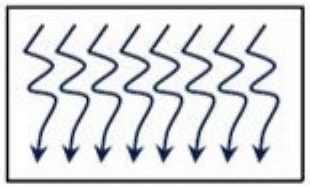
- **Step**

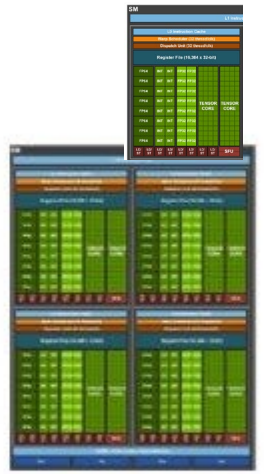
Software | Hardware

Threads run on a cores — Thread — FP32 — Core

Warp: Group of 32 threads executed simultaneously in CUDA.

Blocks run on SMs — Thread Block



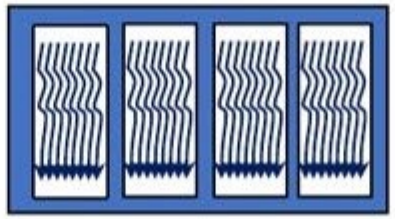


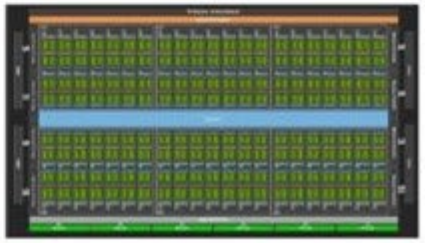
SM

Grids runs on the GPU

In CUDA this is a kernel launch.

Grid





Device

Src: Oxford Cuda Course

To start any Cuda application:

- **Two parameters are required:**

1. Grid Size = The total number of blocks.
2. Block Size = Number of threads per block.
3. Warp = 32 threads. (configured automatically)
4. Warps_per_block = Threads_per_block/32

CUDA Programming

Software

Hardware

Threads run on a cores

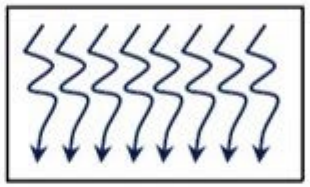
Thread

Core

Warp: Group of 32 threads executed simultaneously in CUDA.

Blocks run on SMs

Thread Block

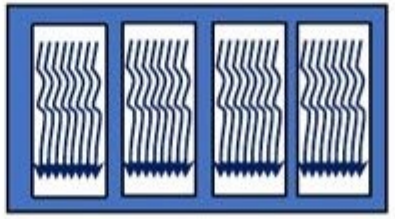


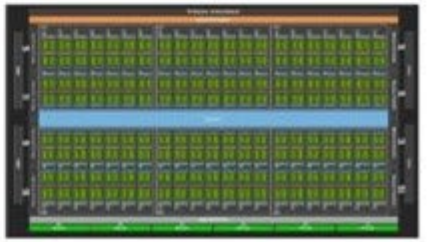
SM

Grids runs on the GPU

In CUDA this is a kernel launch.

Grid





Device

Src: Oxford Cuda Course

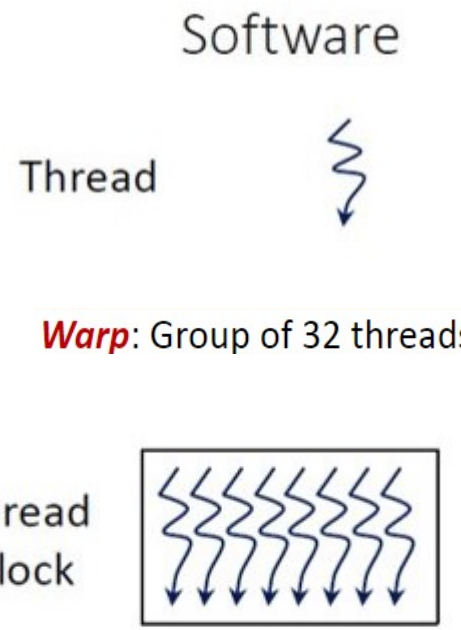
GPU Architecture and CUDA programming introduction

- Purpose:
  - Printing the warp IDs.
  - Understanding the gridDim and blockDim.

threadIdx.x

?

blockIdx.x



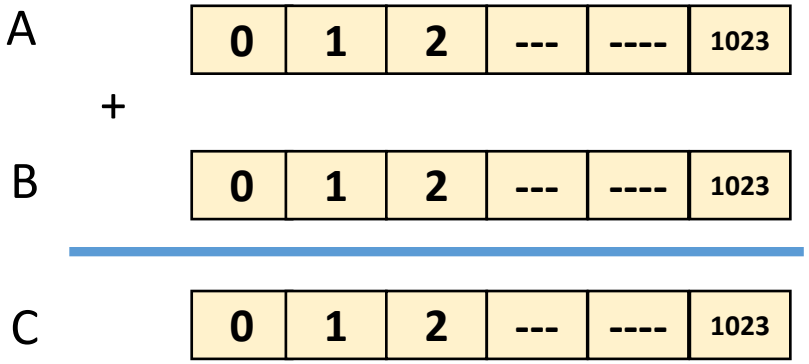

```
The block ID is 1 --- The thread ID is 0
The block ID is 1 --- The thread ID is 1
The block ID is 1 --- The thread ID is 2
The block ID is 1 --- The thread ID is 3
```

Project_002

- some special variables:
  - `gridDim` size (or dimensions) of grid of blocks
  - `blockDim` size (or dimensions) of each block
  - `blockIdx` index (or 2D/3D indices) of block
  - `threadIdx` index (or 2D/3D indices) of thread
  - `warpSize` always 32 so far, but could change

Project 003: Vector Addition (CPU)

- Adding two vectors with around 1024 elements each.

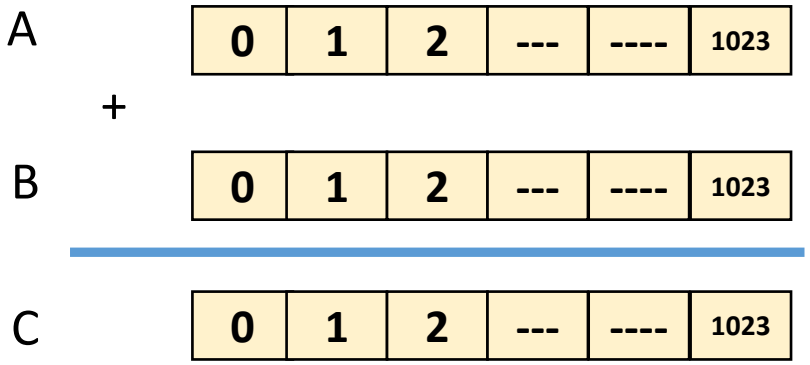


CPU sequential execution

```
for (int i = 0; i < SIZE; i++) {
        C[i] = A[i] + B[i];
    }
```

Project 003: Vector Addition (GPU)

- Adding two vectors with around 1024 elements each.
- We choose the **block**#=1 and the **threads** per block = 1024



C[TID0] = A[TID0] + B[TID0]
C[TID1] = A[TID1] + B[TID1]
C[TID2] = A[TID2] + B[TID2]

C[TID1023] = A[TID1023] + B[TID1023]

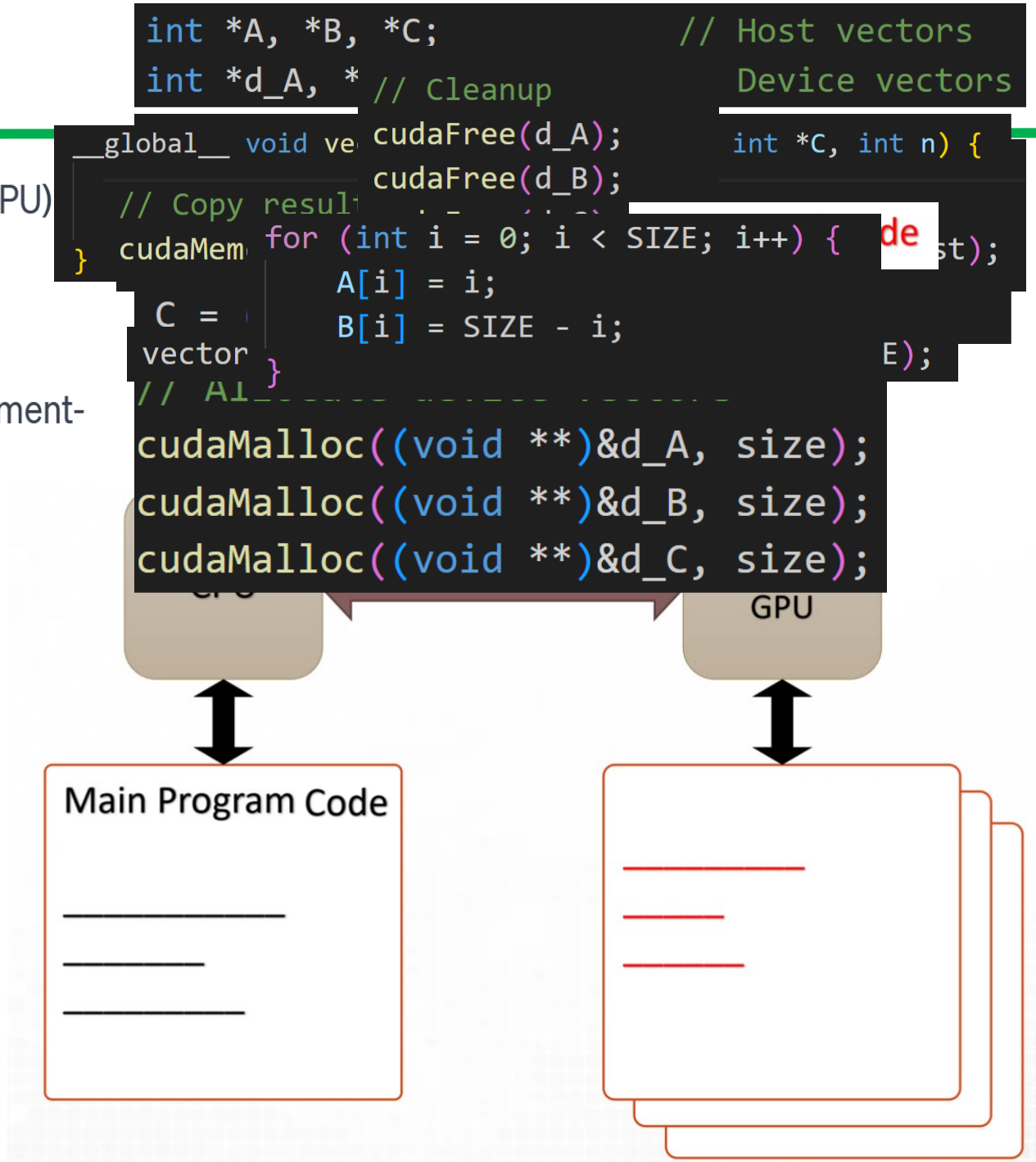
C[0] = A[0] + B[0]
C[1] = A[1] + B[1]
C[2] = A[2] + B[2]

C[1023] = A[1023] + B[1023]

```
__global__ void vectorAdd(int *A, int *B, int *C, int n) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

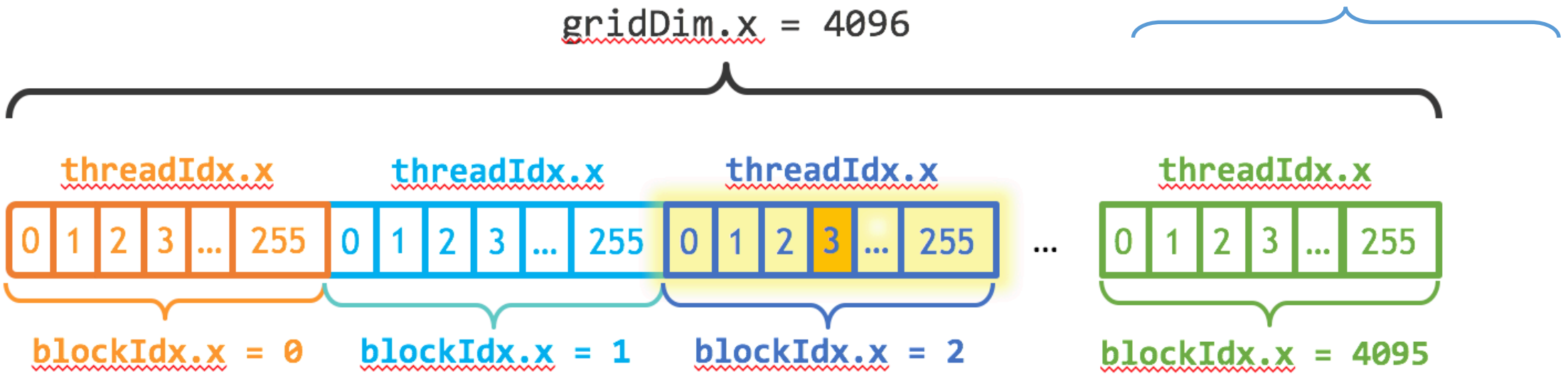
Project 003: Vector Addition

1. Allocate memory for vectors A, B, and C on the host (CPU) and (GPU)
2. Initialize vectors A and B with values.
3. copy the host vectors to the device.
4. Define the CUDA kernel `vectorAdd`, which adds the vectors element-wise.
5. Launch the kernel with a suitable number of blocks and threads.
6. Copy the result back to the host vector C.
7. Free the allocated memory on both the host and device.



Project 003: Vector Addition

- Adding two vectors with around 4096*256 elements each.
- We choose the block#=4096 and the threads per block = 256




```
index = blockIdx.x  * blockDim.x + threadIdx.x

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

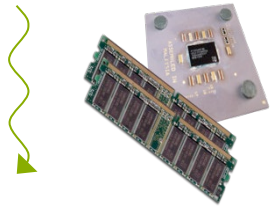
General CUDA code (Host & device)

```
// DAXPY in CUDA
__global__ void daxpy(int n, double a, double *x, double *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int main(void) {
     int n = 1024;
    double a;
    double *x, *y; /* host copy of x and y */
    double *x_d; *y_d; /* device copy of x and y */
    int size = n * sizeof(double)
    // Alloc space for host copies and setup values
    x = (double *)malloc(size); fill_doubles(x, n);
    y = (double *)malloc(size); fill_doubles(y, n);

    // Alloc space for device copies
    cudaMalloc((void **)&d_x,  size);
    cudaMalloc((void **)&d_y, size);

    // Copy to device
    cudaMemcpy(d_x,  x,  size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);

    // Invoke DAXPY with 256 threads per Block
    int nblocks = (n+ 255) / 256;
    daxpy<<<nblocks, 256>>>(n, 2.0, x_d, y_d);

    // Copy result back to host
    cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(x); free(y);
    cudaFree(d_x); cudaFree(d_y);
    return 0;
}
```

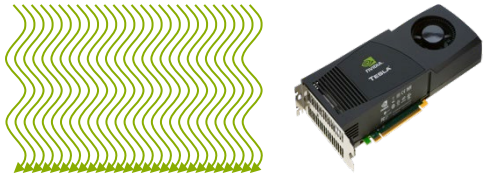
CUDA kernel

Memory allocation and data copy-in

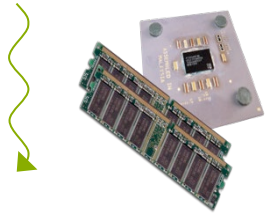
serial code



Offloading computation



parallel exe on GPU



Data copy-out and deallocation

serial code

CUDA Execution Abstraction

- Thread – 1D,2D and 3D
- Block: Multi-dimensional array of threads
  - 1D, 2D, or 3D
  - Threads in a block can synchronize among themselves
  - Threads in a block can access shared memory
- Grid: Multi-dimensional array of blocks
  - 1D or 2D
  - Blocks in a grid can run in parallel, or sequentially.

CUDA Basics

- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...);   // kernel callable from host
__device__ void DeviceFunc(...);   // function callable on device
__device__ int  GlobalVar;         // variable in device memory
__shared__ int  SharedVar;         // in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...);     // 500 blocks, 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads();                   // barrier synchronization
```

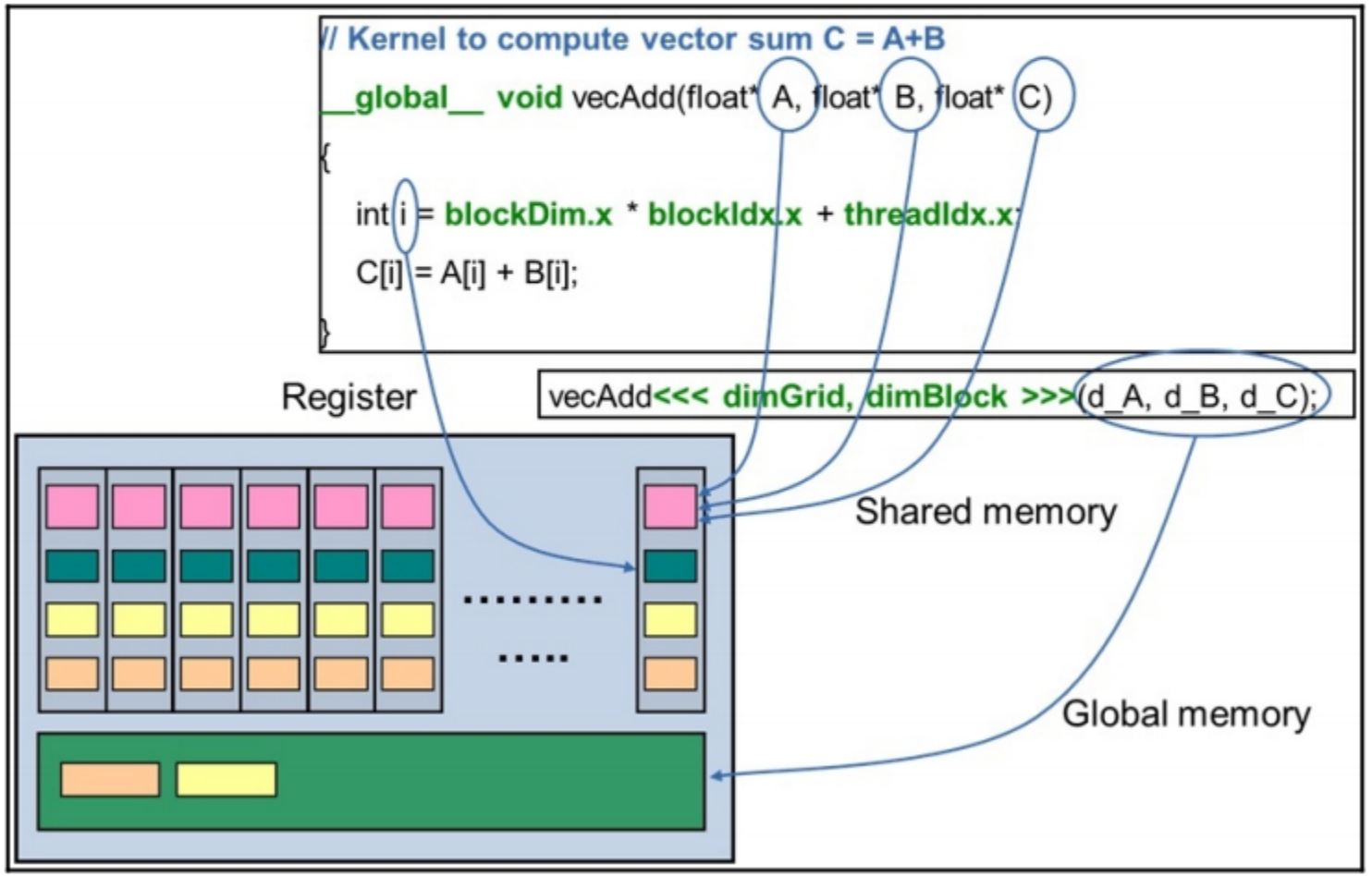
CUDA Memory Management

```
// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```

Mapping software to hardware

- I need to study each memory level
  - For Example, studying L2, we should run

A variable that's declared as part of the `vecAdd` kernel is stored in register memory. The arguments that are passed to the kernel, that is, `A`, `B`, and `C`, point to global memory, but the variable themselves are stored either in the shared memory of registers based on the GPU architecture. The following diagram shows the UDA memory hierarchy and the default locations of the different variable types:



Memory

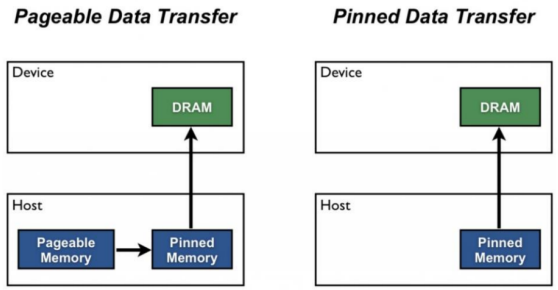
Key challenge in modern computer architecture

- no point in blindingly fast computation if data can't be moved in and out fast enough
- need lots of memory for big applications
- very fast memory is also very expensive
- end up being pushed towards a hierarchical design

Page-locked / Pinned memory

Section 6.2.6 of the cuda programming guide:

- Host memory is usually paged, so run-time system keeps track of where each page is located.
- For higher performance, pages can be fixed (fixed address space, always in RAM), but means less memory available for everything else.
- CUDA uses this for better host <–> GPU bandwidth, and also to hold "device" arrays in host memory.
- Can provide up to 100% improvement in bandwidth
- You must use page-locked memory with `cudaMemcpyAsync();`
- Page-locked memory is allocated using `cudaHostAlloc()`, or registered by `cudaHostRegister();`



Pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory.

https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

Non-blocking loads/stores

What happens with the following code?

```
__global__ void lap(float *u1, float *u2) {
  float a;

  a = u1[threadIdx.x + blockIdx.x*blockDim.x]
  ...
  ...
  c = b*a;
  u2[threadIdx.x + blockIdx.x*blockDim.x] = c;
  ...
  ...
}
```

Load doesn't block until needed; store also doesn't block

Active blocks per SM

Each block require certain resources:

- threads
- registers (registers per thread $\times$ number of threads)
- shared memory (static + dynamic)

Together these determine how many blocks can be run simultaneously on each SM – up to a maximum of 32 blocks

Warp divergence

Threads are executed in warps of 32, with all threads in the warp executing the same instruction at the same time.

What happens if different threads in a warp need to do different things?

```
if (x<0.0)
  z = x-2.0;
else
  z = sqrt(x);
```

This is called *warp divergence* – CUDA will generate correct code to handle this, but to understand the performance you need to understand what CUDA does with it

Warp divergence

Note that:

- `sqrt(x)` would usually produce a `NaN` when `x<0`, but it's not really executed when `x<0` so there's no problem
- all threads execute both conditional branches, so execution cost is sum of both branches
  $\Longrightarrow$ potentially large loss of performance

Active blocks per SM

My general advice:

- number of active threads depends on number o registers each needs
- good to have at least 4 active blocks per SM, ea with at least 128 threads
- smaller number of blocks when each needs lots shared memory
- larger number of blocks when they don't need a shared memory

Warp divergence

This is not a new problem.

Old CRAY vector supercomputers had a logical merge vector instruction

```
z = p ? x : y;
```

which stored the relevant element of the input vectors `x,y` depending on the logical vector `p`, equivalent to

```
for(i=0; i<I; i++) {
  if (p[i]) z[i] = x[i];
  else      z[i] = y[i];
}
```

Active blocks per SM

On Volta:

- maybe 4 big blocks (512 threads) if each needs a lot of shared memory
- maybe 12 small blocks (128 threads) if no shared memory needed
- or 4 small blocks (128 threads) if each thread needs lots of registers

Warp divergence

Similarly, NVIDIA GPUs have *predicated* instructions which are carried out only if a logical flag is true.

```
 p:  a = b + c;  // computed only if p is true
```

In the previous example, all threads compute the logical predicate and two predicated instructions

```
     p = (x<0.0);
 p:  z = x-2.0;      // single instruction
!p:  z = sqrt(x);
```

openmpi

- temporal locality: a data item just accessed is likely to be used again in the near future, so keep it in the cache
- spatial locality: neighbouring data is also likely to be used soon, so load them into the cache at the same time using a ‘wide’ bus (like a multi-lane motorway)

Different levels of parallizem.

within a single GPU

Sw vs hardware comparison (GPU-SM-Paritions-Units) vs (grid-blocks-warps-threads)

Asynccopy (using the shared memory) to swap between memory access and computations

Streams, to swap between the kernel execution and data transfere from CPU to the GPU or vice versa

Using tensor cores for executing matrix operations (tens of operations/cycle) instead of one-by-one operations using normal cores

using multiple GPUs

I want to prepare a talk for

Prepare talk about the diffe

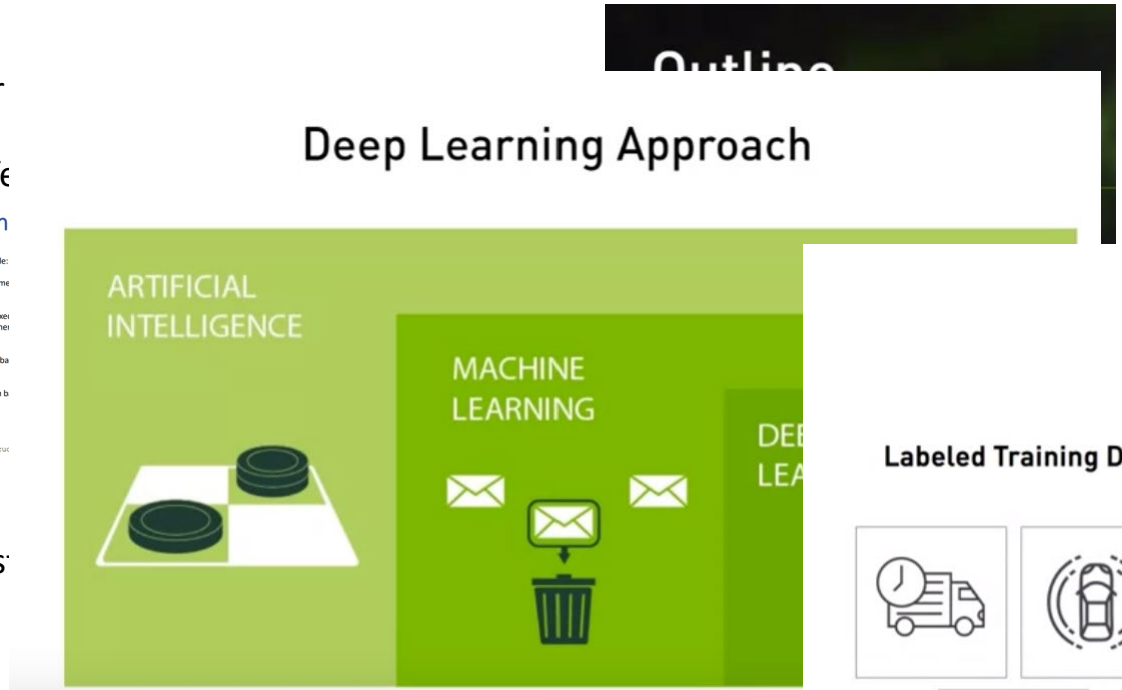
I need to read and unders

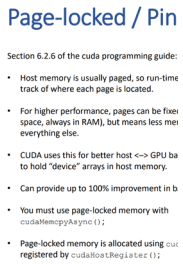
Atomic operations

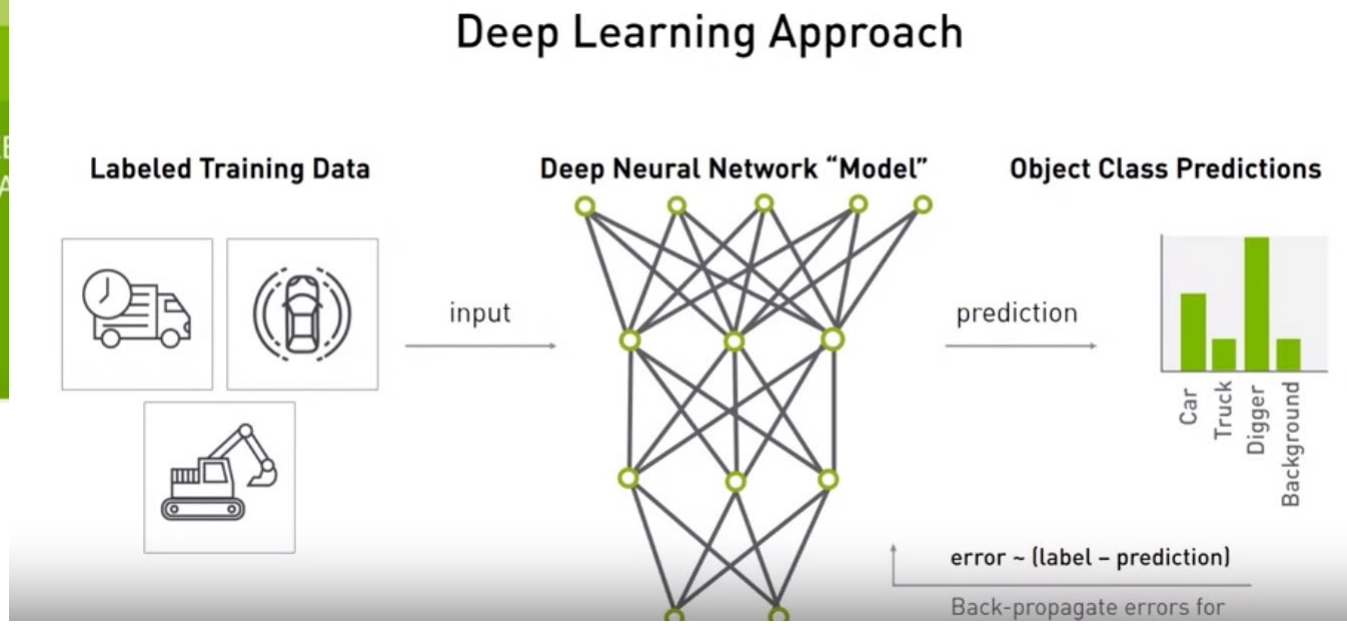
Occasionally, an application needs threads to update a counter in shared memory.

```
__shared__ int count;

...

if ( ... ) count++;
```

In this case, there is a problem if two (or more) threads try to do it at the same time







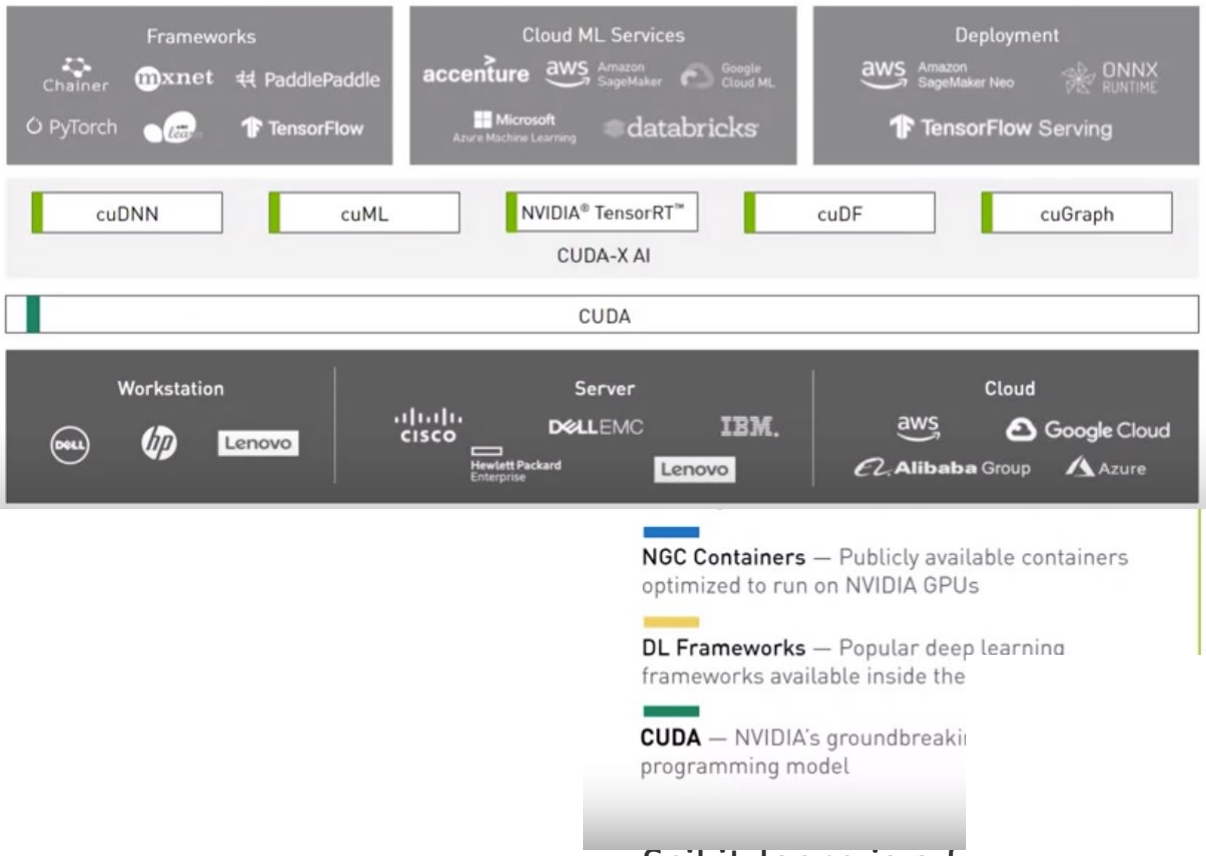
domain experts in computer vision,

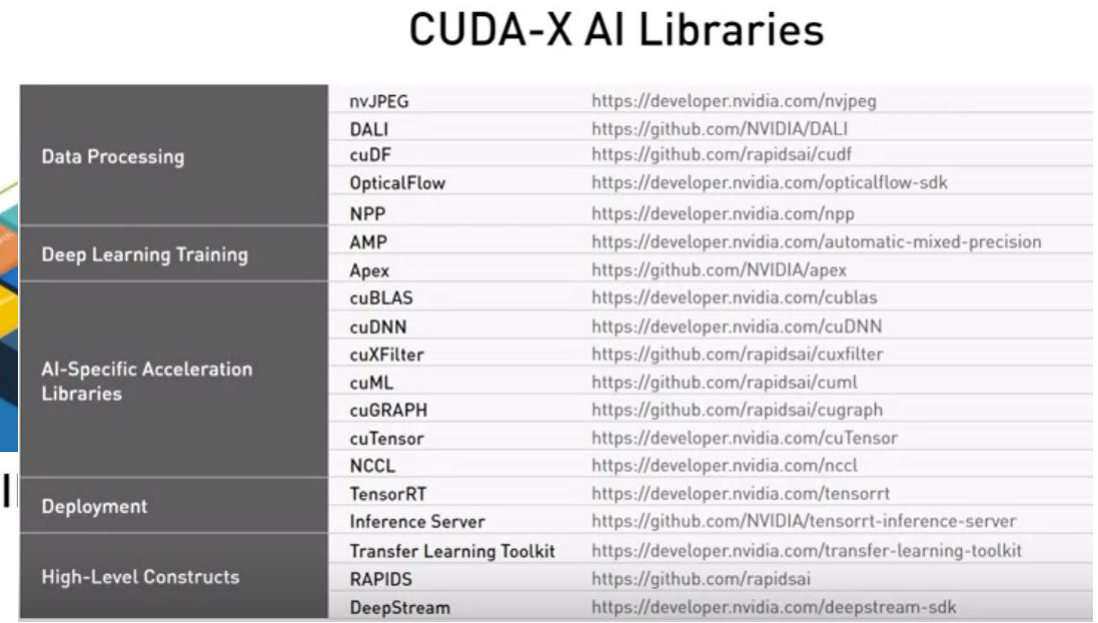
robotics,

ry or tool

ıore

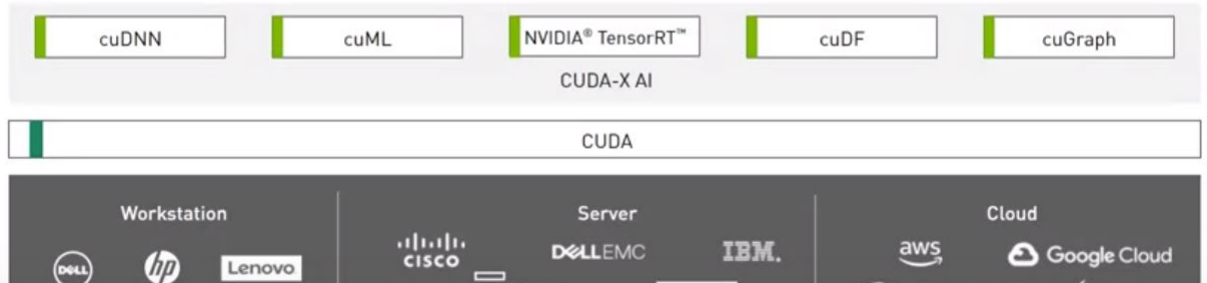
ls.





NVI

Scikit-learn is a fi
for the Python pr
It features variou
and clustering al
is designed to int
numerical and sc
Tensorflow is a p



Debugging – handling errors

- LLVM - NVHPC

nvprof --print-gpu-trace ./vadd01

```
#define checkCudaErrors(err) { \
    if (err != cudaSuccess) {  \

        fprintf(stderr, "checkCudaErrors() API error = %04d \"%s\" from
file <%s>, line %i.\n", \
                err, cudaGetErrorString(err), __FILE__, __LINE__); \
        exit(-1); \
    } \
}
#endif
```

```
checkCudaErrors(cudaMalloc((void **)&d_A, N * K * sizeof(float)));
checkCudaErrors(cudaMalloc((void **)&d_B, K * M * sizeof(float)));
checkCudaErrors(cudaMalloc((void **)&d_C, N * M * sizeof(float)));

checkCudaErrors(cudaGetLastError());
```

Important tools

CUDA Occupancy Calculator,
is a simple tool that tells us how “occupied” our GPU is,

Occupancy

- SM resources (typical values)
  - Maximum number of warps per SM (64)
  - Maximum number of blocks per SM (32)
  - Register usage (256KB)
  - Shared memory usage (64KB)
- Occupancy calculation
  - Number of threads per block
  - Registers per thread
  - Shared memory per block
- The number of registers per thread is known in compile time

CUDA Extended Features

- Standard ma

  `sinf`, `powf`, a

- Built-in vecto

  Asynchronous C

- /

  The A100 GPU includ
  global memory into SI
  usage. Async-copy re
  and reduces power c
  background while the

- To set dimensions:

```
dim3 grid(16,16);       // grid = 16 x 16 blocks
dim3 block(32,32);      // block = 32 x 32 threads
myKernel<<<grid, block>>>(...);
```

- which sets:

```
grid.x = 16;
grid.y = 16;
block.x = 32;
block.y = 32;
block.z = 1;
```

1. Allocate memory for vectors A, B, and C on the host (CPU) and (GPU)
2. Initialize vectors A and B with values.
3. copy the host vectors to the device.
4. Define the CUDA kernel `**vectorAdd**`, which adds the vectors element-wise.
5. Launch the kernel with a suitable number of blocks and threads.
6. Copy the result back to the host vector C.
7. Free the allocated memory on both the host and device.