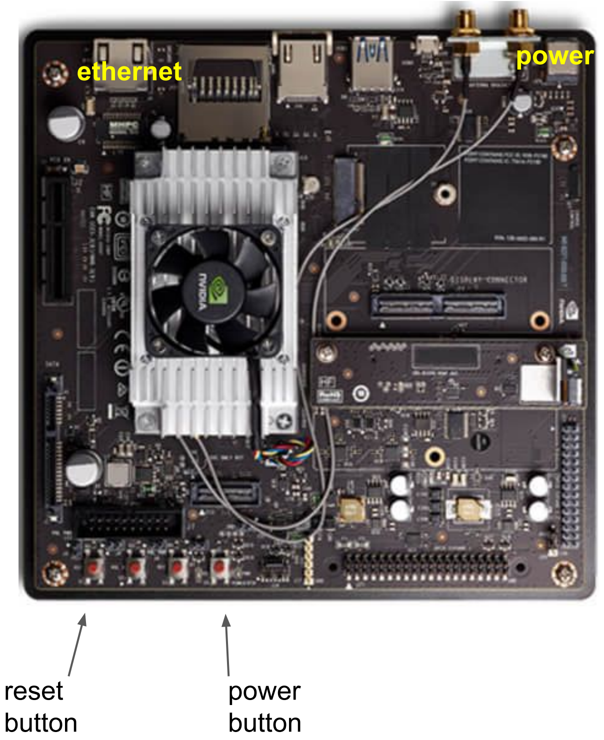
WES237B – SU22

Lab4

GPU Acceleration with Jetson TX2

- Dual-core NVIDIA Denver2 + quad-core ARM Cortex-A57
- 256-core Pascal GPU
- 8GB LPDDR4, 128-bit interface
- 32GB eMMC
- 4kp60 H.264/H.265 encoder and decoder
- Dual ISPs (Image Signal Processors)
- 1.4 Gpps MIPI CSI camera ingest

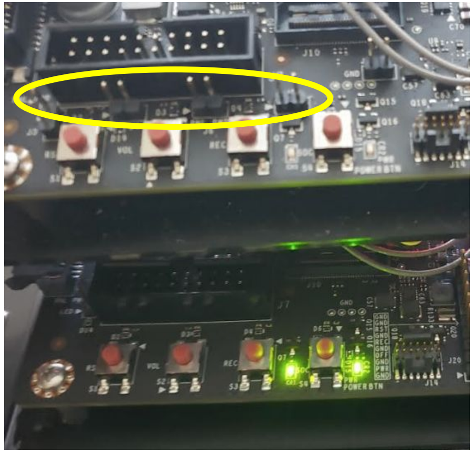


Jetson TX2

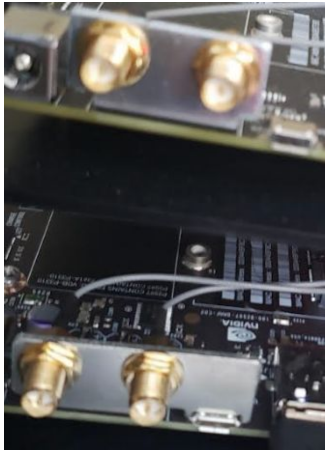
Jumpers

Old board

New board



BNC plate



CUDA

- CUDA is a *parallel* computing platform and programming model that makes using a GPU for general purpose computing simple and elegant.
- CUDA is NOT a programming language or an API (Application Programming Interface)
- Developers use CUDA in C, C++, etc. and incorporate extensions of these languages in the form of a few basic keywords.

CUDA and C/C++

- Lab Work 1 (hello world)
  - Create two files and compile them as following:

main.cc

```
#include <cstdio>

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

g++ main.cc -o hello_cpu

main.cu

```
#include <cstdio>

__global__ void mykernel(void){
}

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

nvcc main.cu -o hello_gpu

CUDA and C/C++

- Lab Work 1 (hello world)
  - Create two files and compile them as following:

main.cu

```
#include <cstdio>

__global__ void mykernel(void){
}

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

__*global*__ defines a CUDA kernel that can be called from the host

CUDA compiler

```
nvcc main.cu -o hello_gpu
```

CUDA and C/C++

- Lab Work 1 (hello world)
    - Create two files and compile them as following:

__host__ defines a function that runs on the host

__global__ defines a CUDA kernel that can be called from the host

__device__ defines a CUDA function that can not be called from the host

main.cu

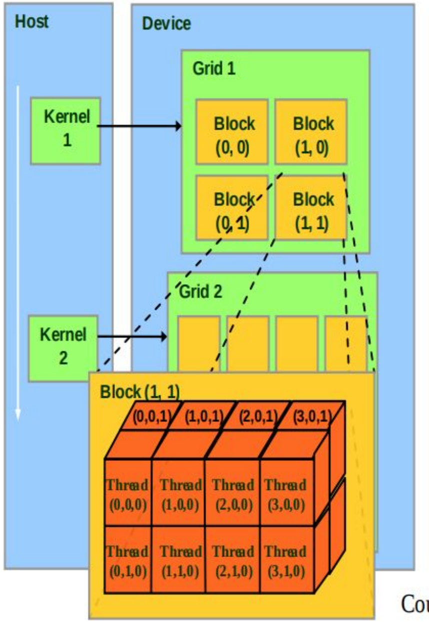
```
#include <cstdio>

__global__ void mykernel(void){
}

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

CUDA compiler → nvcc main.cu -o hello_gpu

NVIDIA GPU Memory Hierarchy

- Hardware implementation: TX2
  - SM (streaming multiprocessor) 2
  - SP (streaming processor) 128
  - Total number of cores: SM*SP 256
  - Warp a set of threads that execute together 32

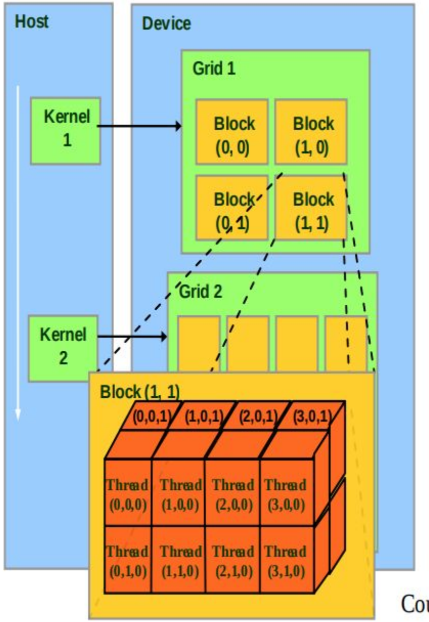
- From a programmer's perspective:
  - Grid - An array of blocks
    all threads in a grid execute the same kernel function
  - Block - An array of threads
  - Threads
  - Kernels - do not return a value



Courtesy: NDVIA

CUDA Pre-defined Variables

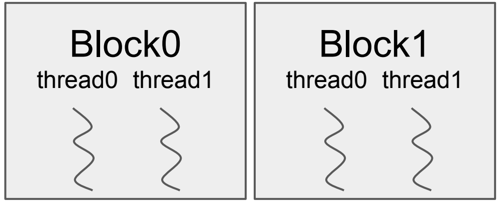
- Structures:
    - dim3 (x, y, z) - if (y, z) are not initialized, they will be set to 1
    - uint3 (x, y, z) - if (y, z) are not initialized, they will be set to 1
- dim3 gridDim - dimensions of grid
- dim3 blockDim - dimensions of block
- uint3 blockIdx - block index within grid
- uint3 threadIdx - thread index within block



Courtesy: NDVIA

Example 1 - (ex1.cu)

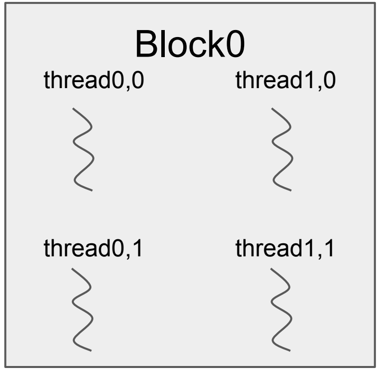
```
#include <cstdio>
#include <stdio.h>
#include <stdlib.h>
#include <string>

__global__ void myKernel(int *a){
    uint thread_global_idx = blockIdx.x * blockDim.x + threadIdx.x;
    printf("block[%d], thread[%d]: a[%d]=%d\n", blockIdx.x, threadIdx.x, thread_global_idx, a[thread_global_idx]);
}

int main(int argc, char* argv[]){
    int a[4] = {0,1,2,3};
    int *dev_a;
    uint size = 4*sizeof(int);

    cudaMalloc((void**)&dev_a, size);  // allocating memory in device

    cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);  // copy from host to device memory

    uint b = 2; // dim3 b(2,1,1);
    uint t = 2; // dim3 t(2,1,1);
    myKernel<<<b,t>>>(dev_a);
    cudaDeviceSynchronize(); // blocks until the device has completed all preceding requested tasks
    cudaFree(dev_a);  // free the device memory
    return 0;
}
```




```
$./ex1
block[0], thread[0]: a[0]=0
block[0], thread[1]: a[1]=1
block[1], thread[0]: a[2]=2
block[1], thread[1]: a[3]=3
```

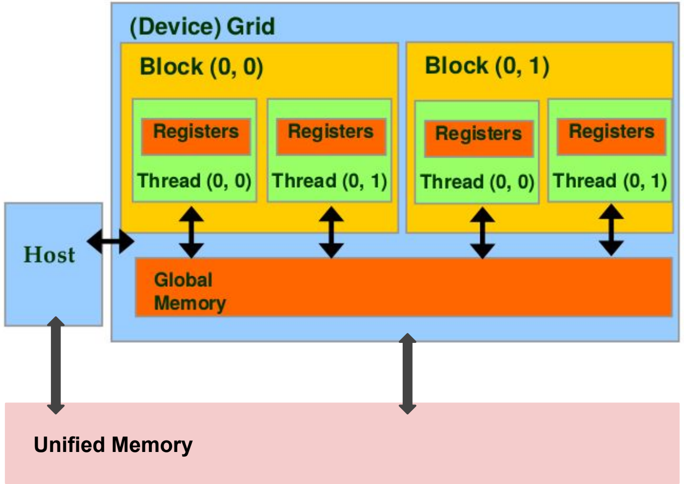
Example 2 - (ex2.cu)

```
#include <cstdio>
#include <stdio.h>
#include <stdlib.h>
#include <string>

__global__ void myKernel(int *a){
    uint thread_idx = threadIdx.y * blockDim.x + threadIdx.x;
    printf("thread[%d, %d]: a[%d]=%d\n", threadIdx.x, threadIdx.y, thread_idx, a[thread_idx]);
}

int main(int argc, char* argv[]){
    int a[4] = {0,1,2,3};
    int *dev_a;
    uint size = 4*sizeof(int);

    cudaMalloc((void**)&dev_a, size); // allocating memory in device

    cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice); // copy from host to device memory

    uint b = 1; // dim3 b(1,1,1);
    dim3 t(2, 2); // dim3 t(2,2,1);
    myKernel<<<b,t>>>(dev_a);
    cudaDeviceSynchronize(); // blocks until the device has completed all preceding requested tasks
    cudaFree(dev_a); // free the device memory
    return 0;
}
```




```
$./ex2
thread[0,0]: a[0]=0
thread[1,0]: a[1]=1
thread[0,1]: a[2]=2
thread[1,1]: a[3]=3
```

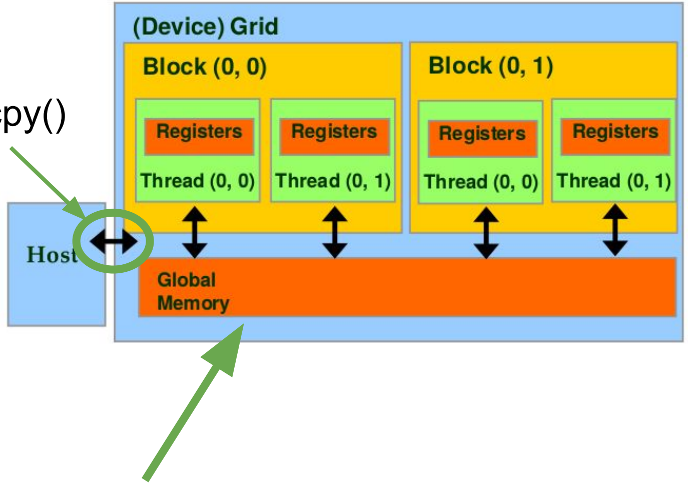
cudaMalloc() vs. cudaMallocManaged()



cudaMallocManaged()

example: lw_managed.cu

cudaMemcpy()



cudaMalloc()

example: lw.cu

Lab Work - (lw.cu)

- Implement the following Matrix Vector Multiplication (MVM) in CUDA
  - Use three threads to calculate r(0), r(1), and r(2) in parallel
  - Note the indices to complete your implementation

m(0,0)	m(0,1)	m(0,2)
m(1,0)	m(1,1)	m(1,2)
m(2,0)	m(2,1)	m(2,2)

×

v(0)
v(1)
v(2)

=

r(0) = m(0,0)*v(0) + m(0,1)*v(1) + m(0,2)*v(2)
r(1) = m(1,0)*v(0) + m(1,1)*v(1) + m(1,2)*v(2)
r(2) = m(2,0)*v(0) + m(2,1)*v(1) + m(2,2)*v(2)

r(0) = m(0)*v(0) + m(1)*v(1) + m(2)*v(2)
r(1) = m(3)*v(0) + m(4)*v(1) + m(5)*v(2)
r(2) = m(6)*v(0) + m(7)*v(1) + m(8)*v(2)

Lab Work - (lw.cu)

- Implement the following Matrix Vector Multiplication (MVM) in CUDA
  - Use three threads to calculate r(0), r(1), and r(2) in parallel
  - Note the indices to complete your implementation

m(0,0)	m(0,1)	m(0,2)
m(1,0)	m(1,1)	m(1,2)
m(2,0)	m(2,1)	m(2,2)

×

v(0)
v(1)
v(2)

=

r(0) = m(0,0)*v(0) + m(0,1)*v(1) + m(0,2)*v(2)
r(1) = m(1,0)*v(0) + m(1,1)*v(1) + m(1,2)*v(2)
r(2) = m(2,0)*v(0) + m(2,1)*v(1) + m(2,2)*v(2)

r(0) = m(0)*v(0) + m(1)*v(1) + m(2)*v(2)
r(1) = m(3)*v(0) + m(4)*v(1) + m(5)*v(2)
r(2) = m(6)*v(0) + m(7)*v(1) + m(8)*v(2)

r(0) = m(0+t)*v(0) + m(1+t)*v(1) + m(2+t)*v(2) t=0*3
r(1) = m(0+t)*v(0) + m(1+t)*v(1) + m(2+t)*v(2) t=1*3
r(2) = m(0+t)*v(0) + m(1+t)*v(1) + m(2+t)*v(2) t=2*3

Image Indexing

Image Indexing 2D (Review)

- OpenCV and other major image processing tools use row-major memory storage.

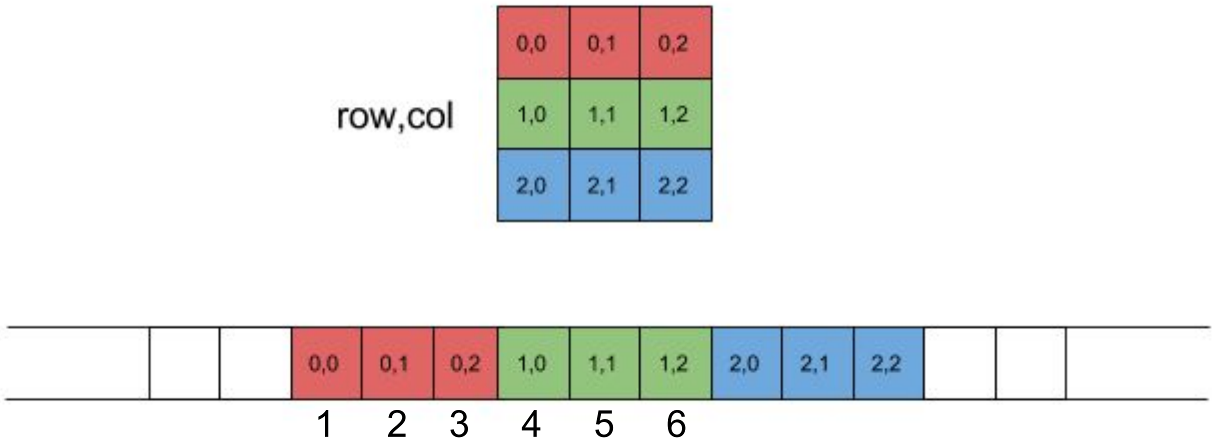


Image Indexing 3D

- r, g, b pixels are next to each other.