[CSE4152] 고급소프트웨어 실습 I
Week 13

서강대학교 공과대학 컴퓨터공학과
교수 임 인 성

GPGPU(General-Purpose Computing on GPU)

- **"General-purpose computing on graphics processing units** (**GPGPU**) is the means of using a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit(CPU)." (*from Wikipedia*)

- 최근 그래픽스 프로세서는 massively-parallel streaming processing을 위한 프로세서로서 비약할 만한 성능 향상을 이루었으며, 그러한 추세는 계속될 것으로 예상됨.
  - "A *stream* is simply a set of records that require similar computation. Streams provide SIMD parallelism. *Kernels* are the functions that are applied to each element in the stream." (*from Wikipedia*)

- 최근 GPU의 놀라운 성능 향상으로 인하여 그래픽스 분야의 문제 뿐만 아니라 "compute-intensive data-parallel" 성질을 가지는 **일반 응용 문제를 해결**하는데 유용하게 쓰이고 있음.

NVIDIA Graphics Processing Units

- **Turing Architecture (2018)**
  - Chips: TU102, TU104, TU106, TU116, TU117
  - Graphics card: GeForce 16/20 series
    - GeForce RTX 2080 Ti with **4,352 CUDA cores** (**max 13.45 TFLOPS**)
  - CUDA cores/Ray Tracing cores/Tensor cores
  - CUDA Compute Capability : 7.5

- **Ampere Architecture (2020)**
  - Chips: GA100, GA102, GA104
  - Graphics card: GeForce 30 series
    - GeForce RTX 3090 with **10,496 CUDA cores** (**max 35.68 TFLOPS**)
  - CUDA cores/Ray Tracing cores/Tensor cores
  - CUDA Compute Capability : 8.0/8.6

Supercomputer와 성능 비교: FLOPS (FLoating-point OPerations per Second)

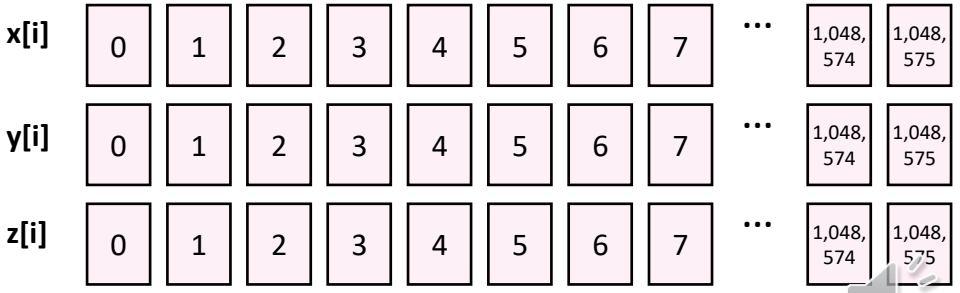
1993	Thinking Machines CM-5/1024	**65.5 GFLOPS**	DoE-Los Alamos National Laboratory; National Security Agency
	Fujitsu Numerical Wind Tunnel	**124.50 GFLOPS**	National Aerospace Laboratory, Tokyo, Japan
	Intel Paragon XP/S 140	**143.40 GFLOPS**	DoE-Sandia National Laboratories, New Mexico, USA
1994	Fujitsu Numerical Wind Tunnel	**170.40 GFLOPS**	National Aerospace Laboratory, Tokyo, Japan
1996	Hitachi SR2201/1024	**220.4 GFLOPS**	University of Tokyo, Japan
	Hitachi/Tsukuba CP-PACS/2048	**368.2 GFLOPS**	Center for Computational Physics, University of Tsukuba, Tsukuba, Japan
1997	Intel ASCI Red/9152	**1.338 TFLOPS**	DoE-Sandia National Laboratories, New Mexico, USA
1999	Intel ASCI Red/9632	**2.3796 TFLOPS**	
2000	IBM ASCI White	**7.226 TFLOPS**	DoE-Lawrence Livermore National Laboratory, California, USA
2002	NEC Earth Simulator	**35.86 TFLOPS**	Earth Simulator Center, Yokohama, Japan

© Wikipedia

SIMD (Single Instruction, Multiple Data)

```
#define MAX_N_ELEMENTS 1048576 // pow(2,20)

void combine_two_arrays(float *x, float *y, float *z, int n) {
  int i;
  for (i = 0; i < n; i++) {
    z[i] = 1.0f/(sin(x[i])*cos(y[i]) + cos(x[i])*sin(y[i]));
  }
}

void main(void) {
  int n_elements;
  float A[MAX_N_ELEMENTS], B[MAX_N_ELEMENTS], C[MAX_N_ELEMENTS];
  n_elements = MAX_N_ELEMENTS;
          :
  combine_two_arrays(A, B, C, n_elements);
          :
}
```

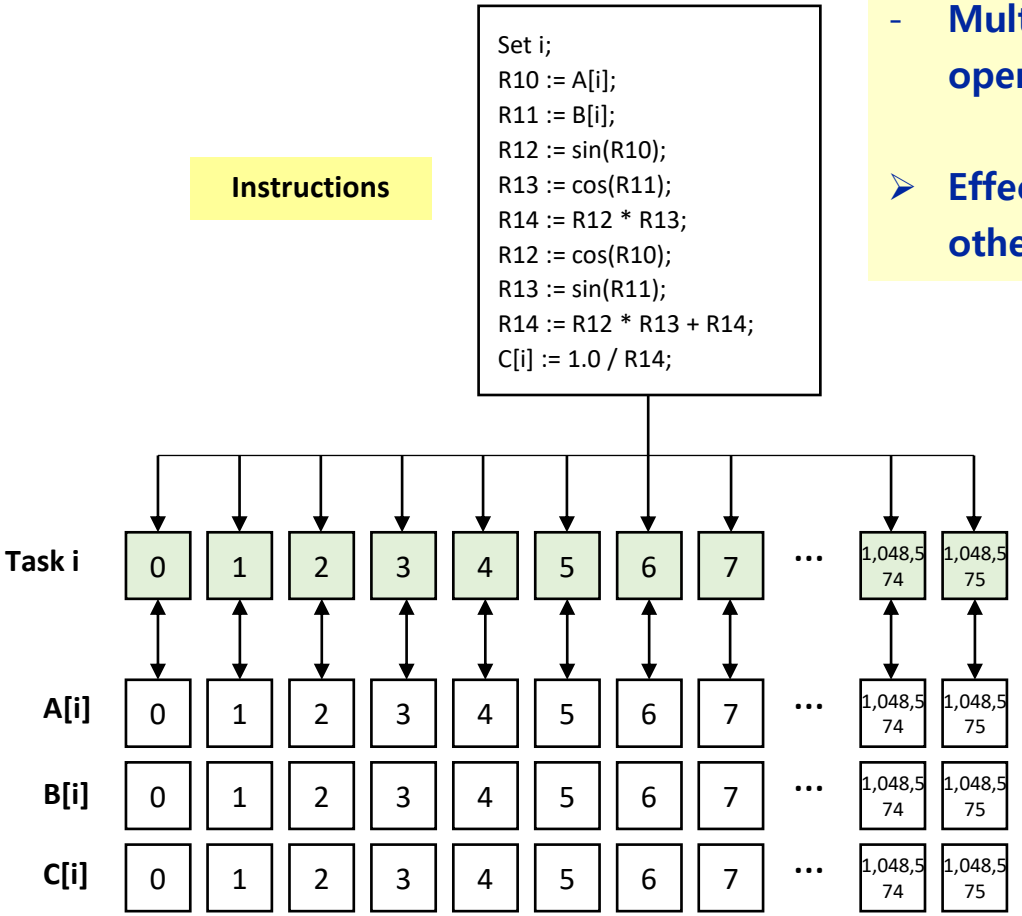
- 동일한 방식의 계산이 서로 다른 데이터에 대하여 반복이 되고 있음.
- 각 데이터에 대한 계산은 서로 독립적임.
- 만약 프로세서가 MAX_N_ELEMENTS개만큼 있다면, …



- 동일한 프로그램의 명령어들을 순차적으로 ← Single Instruction Stream
- 서로 다른 데이터 스트림에 대하여 ← Multiple Data Stream
- 동기화하면서 수행(run in lockstep)하면서 계산하면, …
- Multiple processing elements perform the same operation on multiple data points simultaneously.

➢ Effective graphics/multimedia computations and others.

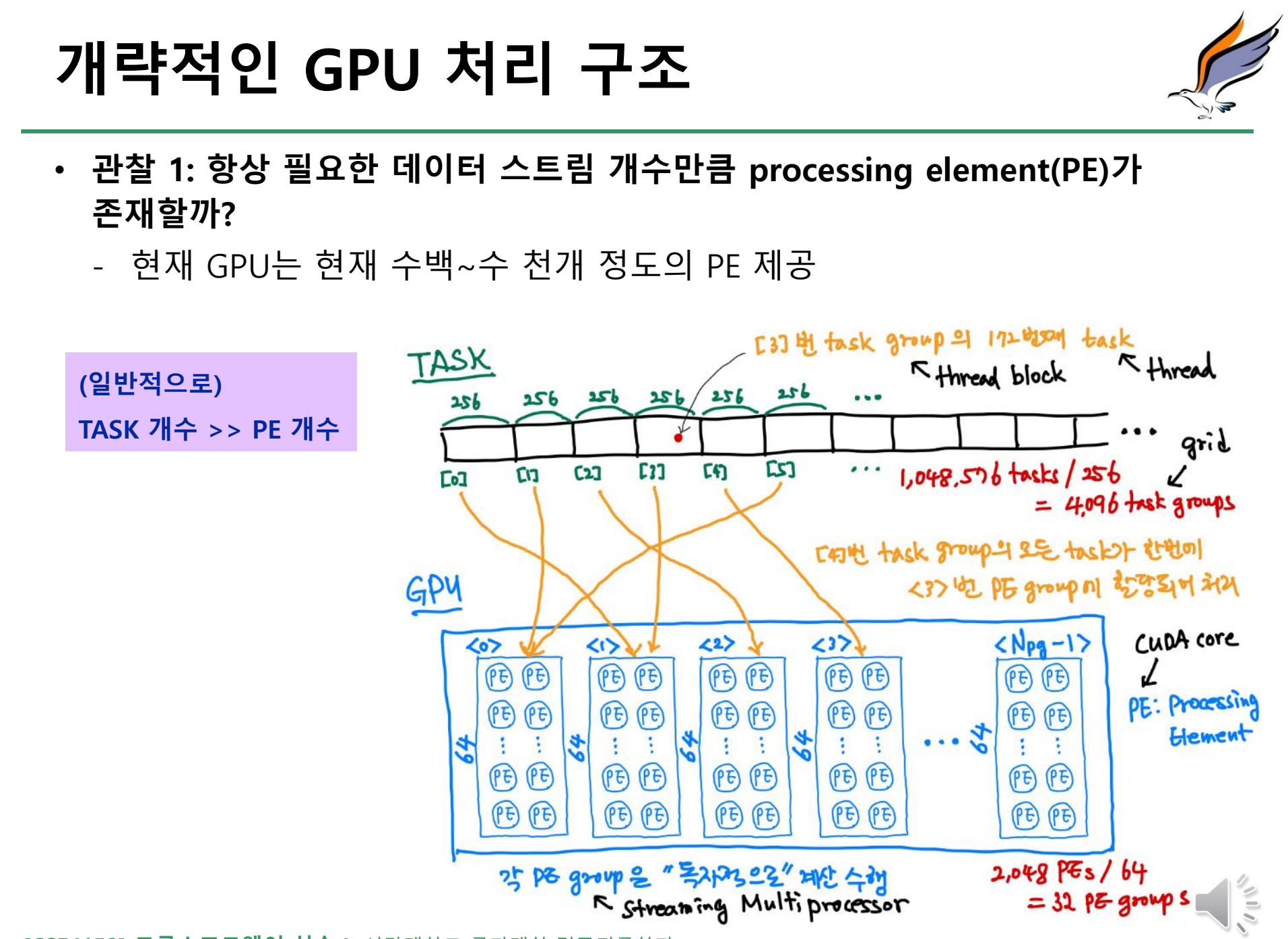
```
z[i] = 1.0f/(sin(x[i])*cos(y[i]) + cos(x[i])*sin(y[i]));
```



SIMD Model

An execution model in parallel computing

개략적인 GPU 처리 구조



- **관찰 1: 항상 필요한 데이터 스트림 개수만큼 processing element(PE)가 존재할까?**
  - 현재 GPU는 현재 수백~수 천개 정도의 PE 제공

(일반적으로)

TASK 개수 >> PE 개수

```
z[i] = 1.0f/(sin(x[i])*cos(y[i]) + cos(x[i])*sin(y[i]));
```

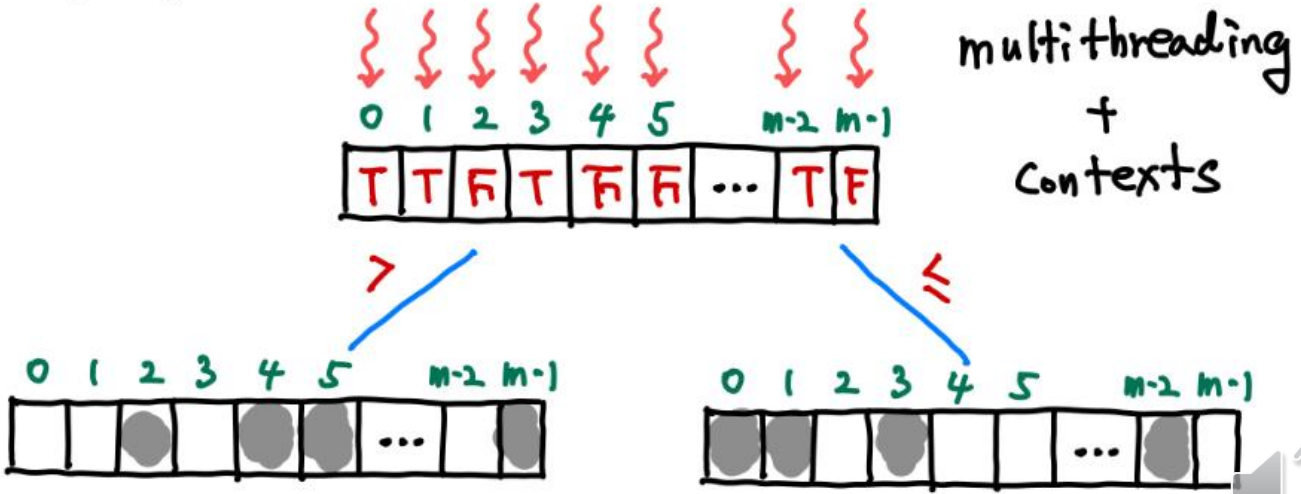
- **관찰 2: 항상 프로그램이 단순한 control flow만으로 구성되어 있을까?**
  - 만약 if-문장이 있다면,

```
if (x[i] > 0) {
    z[i] = x[i] - y[i];
}
else {
    z[i] = 2*x[i] + y[i];
}
```

T or F

SIMT Model

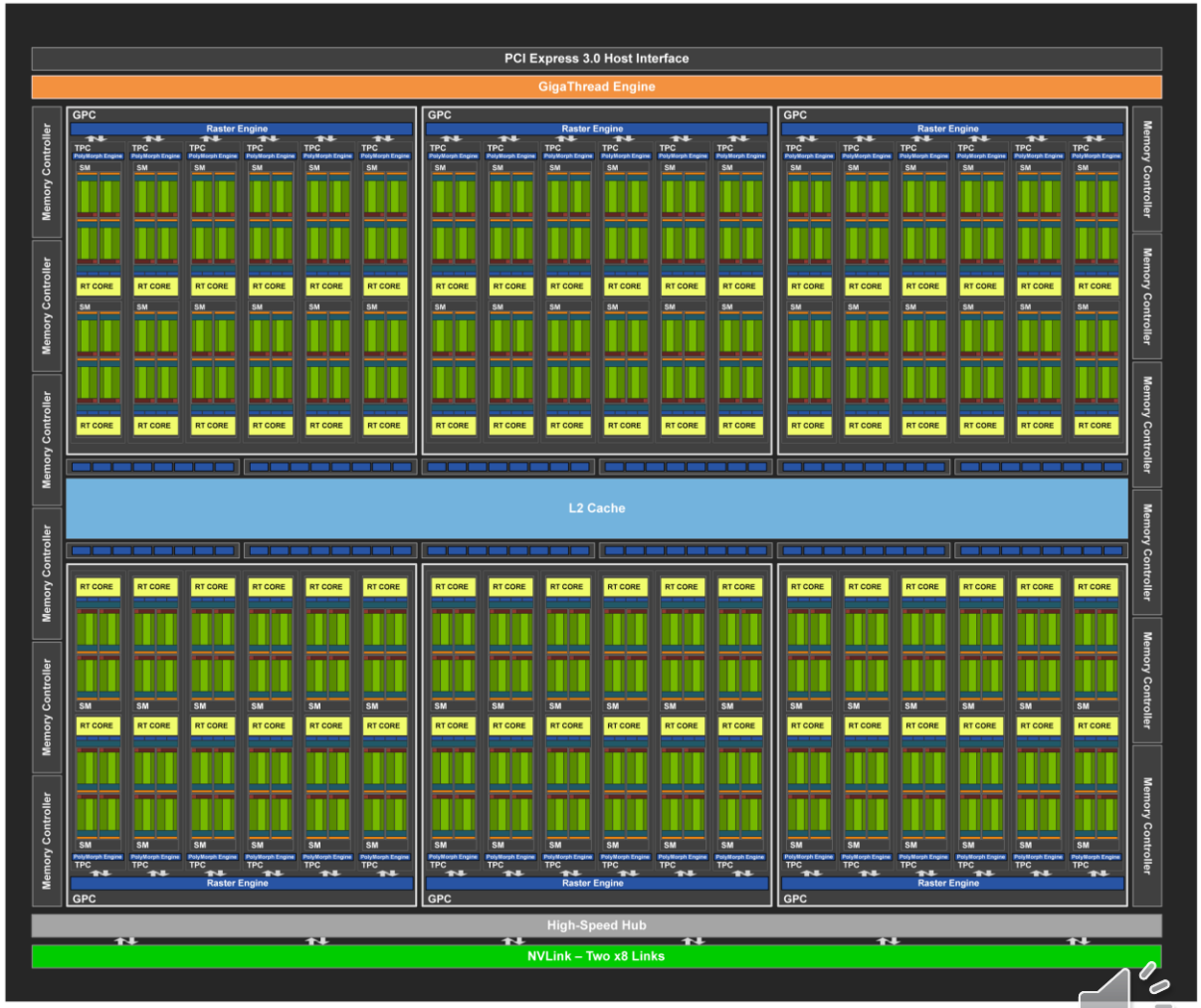
SIMD + multithreading



NVIDIA Turing TU102 GPU

- ✓ 6 **Graphics Processing Clusters**(GPCs)
- ✓ 6 **Texture Processing Clusters**(TPCs) / GPC
- ✓ → 36 TPCs
- ✓ 2 **Streaming Multiprocessors**(SMs) / TPC → **72 SMs**
- ✓ 64 **CUDA Cores** / SM → 4,608 CUDA Cores
- ✓ 1 **RT Core** / SM → 72 RT Cores
- ✓ 8 **Tensor Cores** / SM → 576 Tensor Cores
- ✓ 4 **texture units** / SM → 288 texture units
- ✓ 12 **memory controllers**
- ✓ 8 **ROP units** / MC → 96 ROP units
- ✓ 512 KB of **L2 cache** / MC → 6144 KB of L2 cache

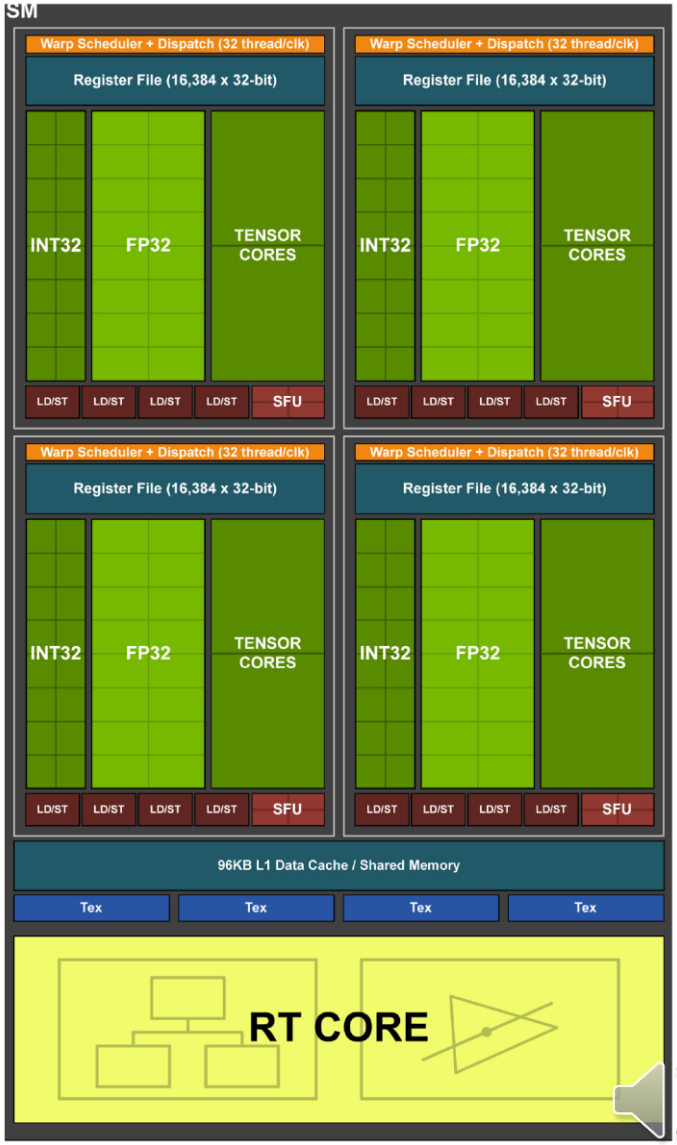
NVIDIA Turing GPU Architecture: Graphics Reinvented (2018)



TU102/TU104/TU106 Streaming Multiprocessor (SM)

NVIDIA Turing GPU Architecture: Graphics Reinvented (2018)

- **Each SM includes**
  - 64 FP32 Cores / 64 INT32 Cores / 2 FP64 Cores
    - Supports concurrent execution of FP32 and INT32 operations.
  - 8 mixed-precision Tensor Cores / 1 RT core
- **Each SM is partitioned into** four processing blocks **each with**,
  - 16 FP32 Cores / 16 INT32 Cores (2 cycles per warp)
  - 2 Tensor Cores
  - 1 warp scheduler
  - 1 L0 instruction cache
  - 1 **64 KB register file: 16,384 x 32-bit**
- The four blocks shares **a combined 96 KB L1 data cache/shared memory**:
  - 64 KB of graphics shader RAM + 32 KB for texture cache
  - **32 KB shared memory + 64 KB L1 cache**
  - **64 KB shared memory + 32 KB L1 cache**



APIs for GPU Programming

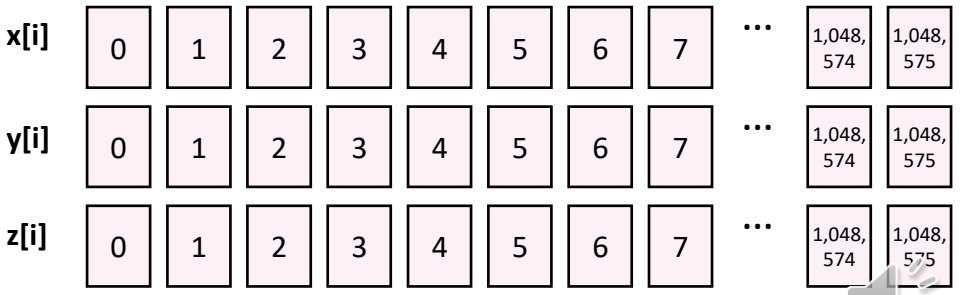
- **OpenGL (Open Graphics Library): Vertex/Geometry/Fragment/Compute Shaders**
  - **OpenGL ES (Open Graphics Library for Embedded System)**
  - ★ http://www.khronos.org
- **Vulkan / Direct X**
  - ★ http://www.khronos.org / http://www.microsoft.com/windows/directx/
- **Metal**
  - ★ https://developer.apple.com/documentation/metal

- **CUDA (Compute Unified Device Architecture)**
  - ★ **http://www.nvidia.com/object/cuda_home.html**
- **OpenCL (Open Computing Language)**
  - ★ **http://www.khronos.org/opencl/**

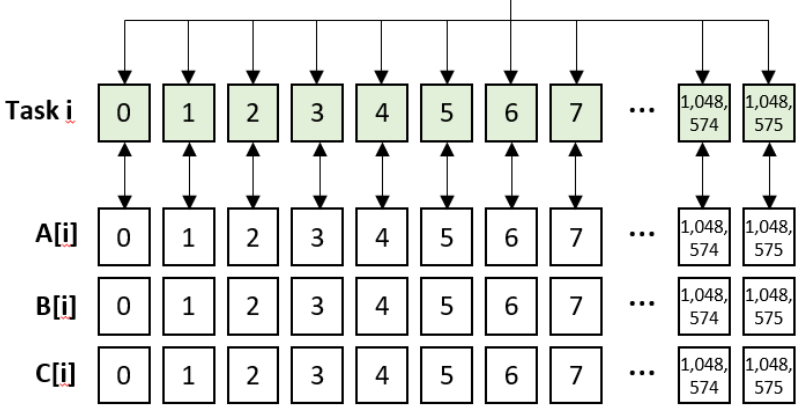
CUDA Implementation of Example Code

```
#define MAX_N_ELEMENTS 1048576 // pow(2,20)

void combine_two_arrays(float *x, float *y, float *z, int n) {
  int i;
  for (i = 0; i < n; i++) {
    z[i] = 1.0f/(sin(x[i])*cos(y[i]) + cos(x[i])*sin(y[i]));
  }
}

void main(void) {
  int n_elements;
  float A[MAX_N_ELEMENTS], B[MAX_N_ELEMENTS], C[MAX_N_ELEMENTS];
  n_elements = MAX_N_ELEMENTS;
          :
  combine_two_arrays(A, B, C, n_elements);
          :
}
```

- 동일한 방식의 계산이 서로 다른 데이터에 대하여 반복이 되고 있음.
- 각 데이터에 대한 계산은 서로 독립적임.
- 만약 프로세서가 MAX_N_ELEMENTS개만큼 있다면, …

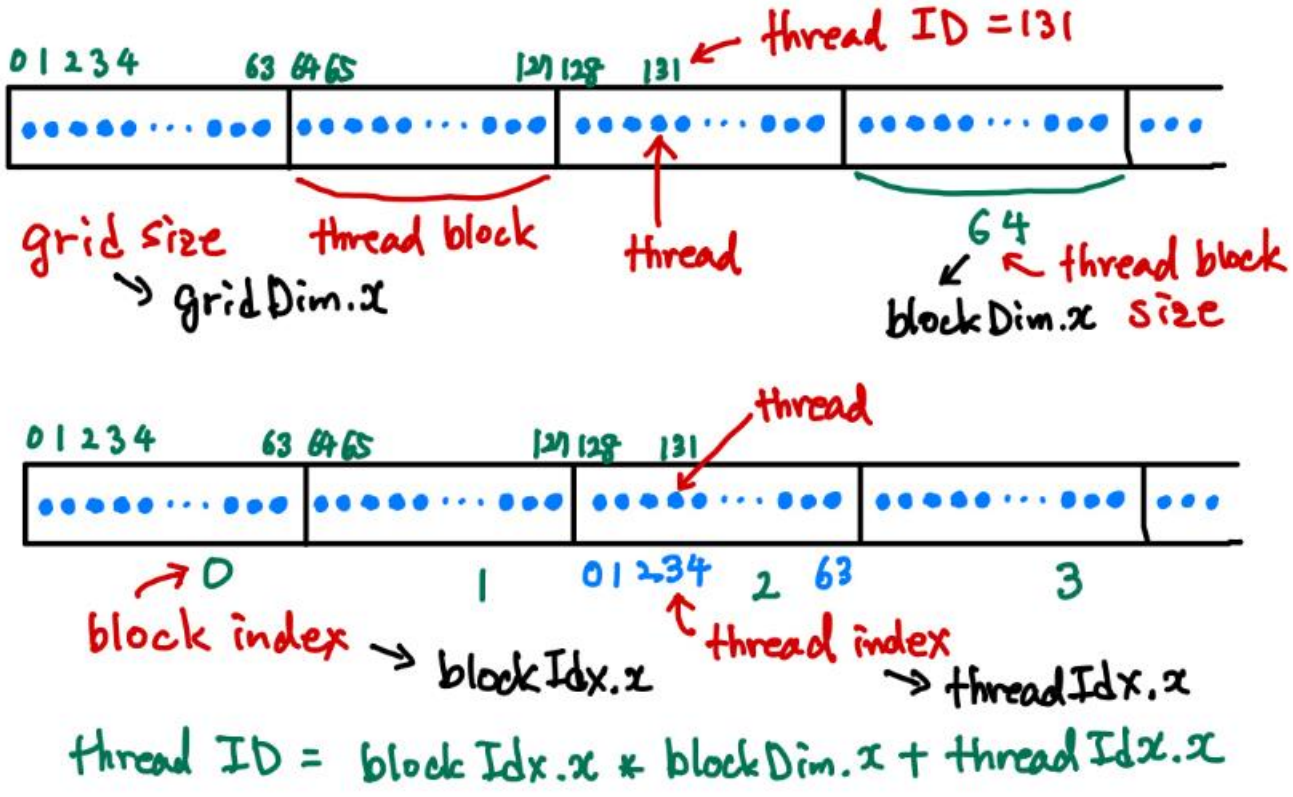


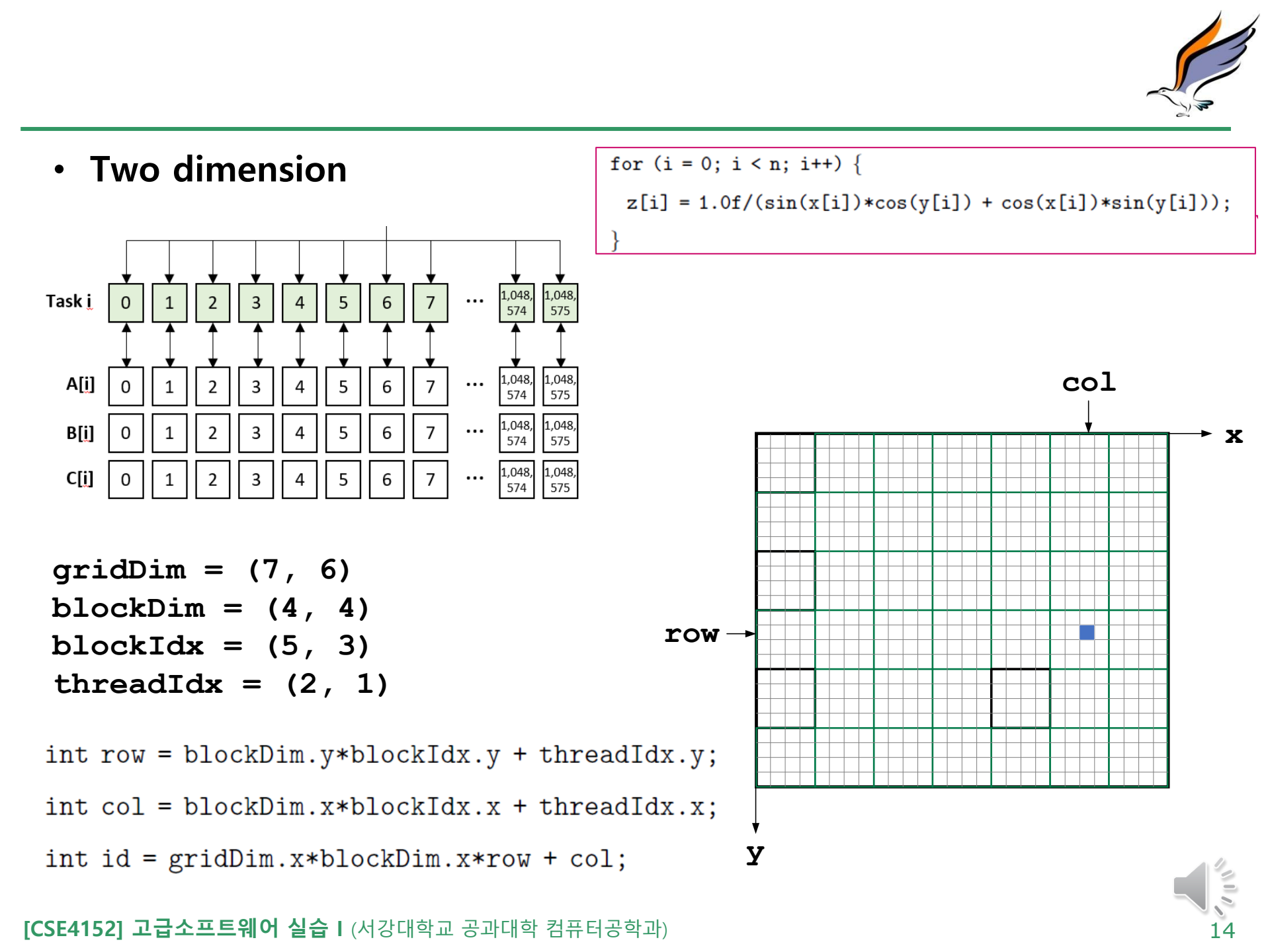
Tasks and CUDA Threads



- **One dimension**

```
for (i = 0; i < n; i++) {
  z[i] = 1.0f/(sin(x[i])*cos(y[i]) + cos(x[i])*sin(y[i]));
}
```





- **Two dimension**

```
for (i = 0; i < n; i++) {
  z[i] = 1.0f/(sin(x[i])*cos(y[i]) + cos(x[i])*sin(y[i]));
}
```



```
gridDim = (7, 6)
blockDim = (4, 4)
blockIdx = (5, 3)
threadIdx = (2, 1)
```



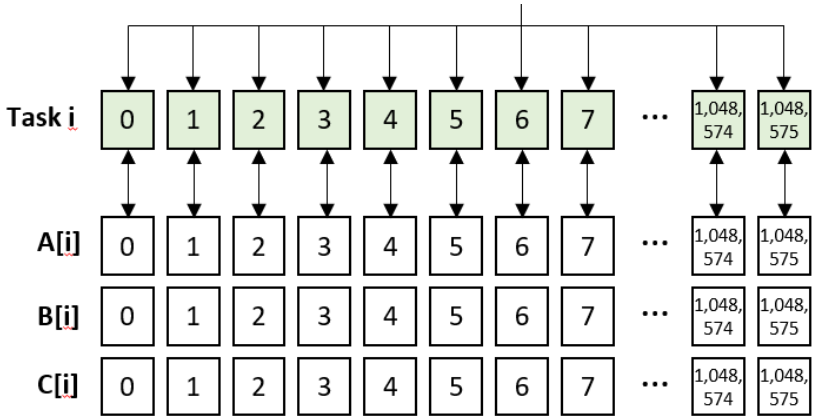
```
int row = blockDim.y*blockIdx.y + threadIdx.y;
int col = blockDim.x*blockIdx.x + threadIdx.x;
int id = gridDim.x*blockDim.x*row + col;
```

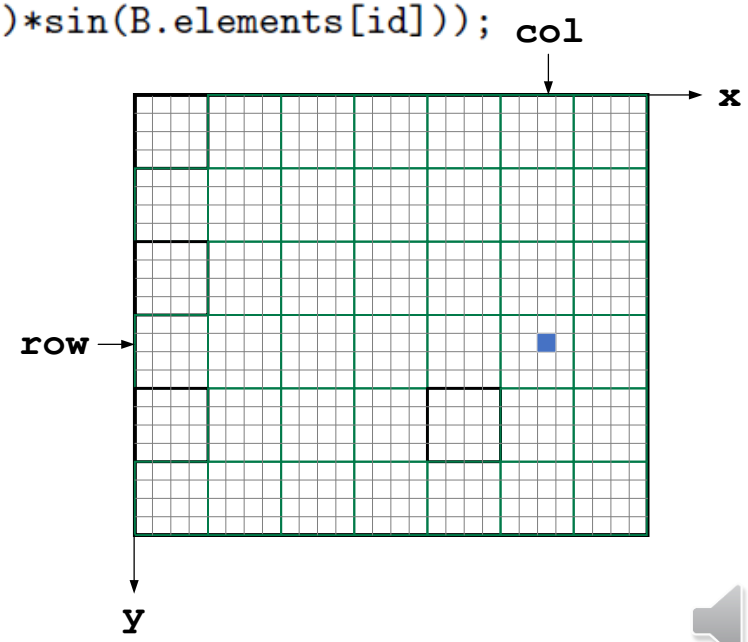
CUDA Kernel

```
for (i = 0; i < n; i++) {
  z[i] = 1.0f/(sin(x[i])*cos(y[i]) + cos(x[i])*sin(y[i]));
}
```

```
__global__ void CombineTwoArrraysKernel(Array A, Array B, Array C) {
  int row = blockDim.y*blockIdx.y + threadIdx.y;
  int col = blockDim.x*blockIdx.x + threadIdx.x;
  int id = gridDim.x*blockDim.x*row + col;

  C.elements[id] = 1.0f / (sin(A.elements[id])*cos(B.elements[id])
                          + cos(A.elements[id])*sin(B.elements[id]));
}
```

Host: CPU
Device: GPU
Kernel: the function that runs on the device





CUDA Host Code

```
int main(void) {
    ⋮
  Array A, B, C;

  // Assume that MAX_N_ELEMENTS is a multiple of 1,024.
  A.width = B.width = C.width = 1024;
  A.height = B.height = C.height = MAX_N_ELEMENTS/1024;
  BLOCK_SIZE = 16;

  A.elements = (float *) malloc(sizeof(float)*MAX_N_ELEMENTS);
  B.elements = (float *) malloc(sizeof(float)*MAX_N_ELEMENTS);
  C.elements = (float *) malloc(sizeof(float)*MAX_N_ELEMENTS);

  generate_random_float_array(A.elements, MAX_N_ELEMENTS);
  generate_random_float_array(B.elements, MAX_N_ELEMENTS);

  combine_two_arrays(A, B, C);
    ⋮
}
```

```
#define MAX_N_ELEMENTS 1048576 // pow(2, 20)
    ⋮
int BLOCK_SIZE;

typedef struct {
    int width;
    int height;
    float *elements;
} Array;
```

```
void combine_two_arrays(const Array A, const Array B, Array C) {
  Array d_A, d_B, d_C;
  size_t size;

  d_A.width = A.width; d_A.height = A.height;
  size = A.width * A.height * sizeof(float);
  cudaMalloc(&d_A.elements, size);
  cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);

  d_B.width = B.width; d_B.height = B.height;
  size = B.width * B.height * sizeof(float);
  cudaMalloc(&d_B.elements, size);
  cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);

  d_C.width = C.width; d_C.height = C.height;
  size = C.width * C.height * sizeof(float);
  cudaMalloc(&d_C.elements, size);
```

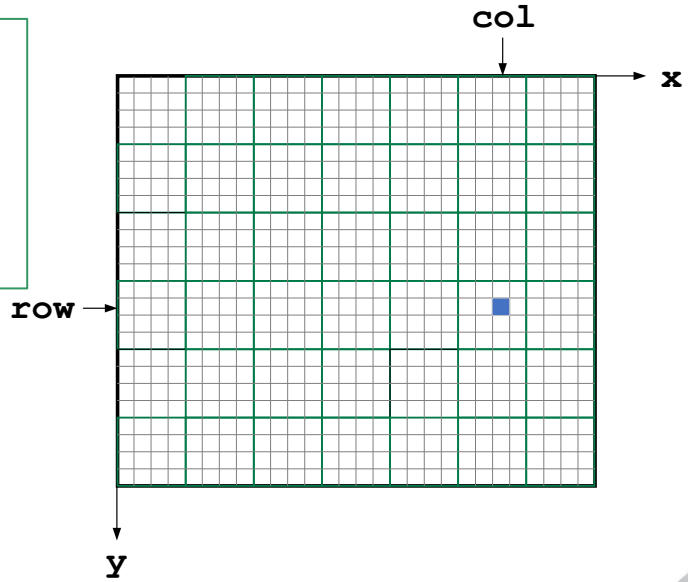
Memory allocation on the device memory

Memory transfer from the host memory to the device memory

```
// Assume that width and height are multiples of BLOCK_SIZE.
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(A.width/dimBlock.x, A.height/dimBlock.y);
CombineTwoArrraysKernel <<< dimGrid, dimBlock >>> (d_A, d_B, d_C);

cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
}
```

Execution of the kernel

```
gridDim = (7, 6)
blockDim = (4, 4)
blockIdx = (5, 3)
threadIdx = (2, 1)
```




```
// Assume that MAX_N_ELEMENTS is a multiple of 1,024.
A.width = B.width = C.width = 1024;
A.height = B.height = C.height = MAX_N_ELEMENTS/1024;
BLOCK_SIZE = 16;
```