GPU Programming (1)

M. Sato

RIKEN R-CCS
and University of Tsukuba

Outline

- **Why GPU is emerging?**
- **GPU programming environment (1)**
  - **CUDA**
  - **GPU by Libraries**
- **GPU programming environment (2)**
  - **OpenCL, SYCL**
  - **OpenMP/OpenACC**

GPU Computing

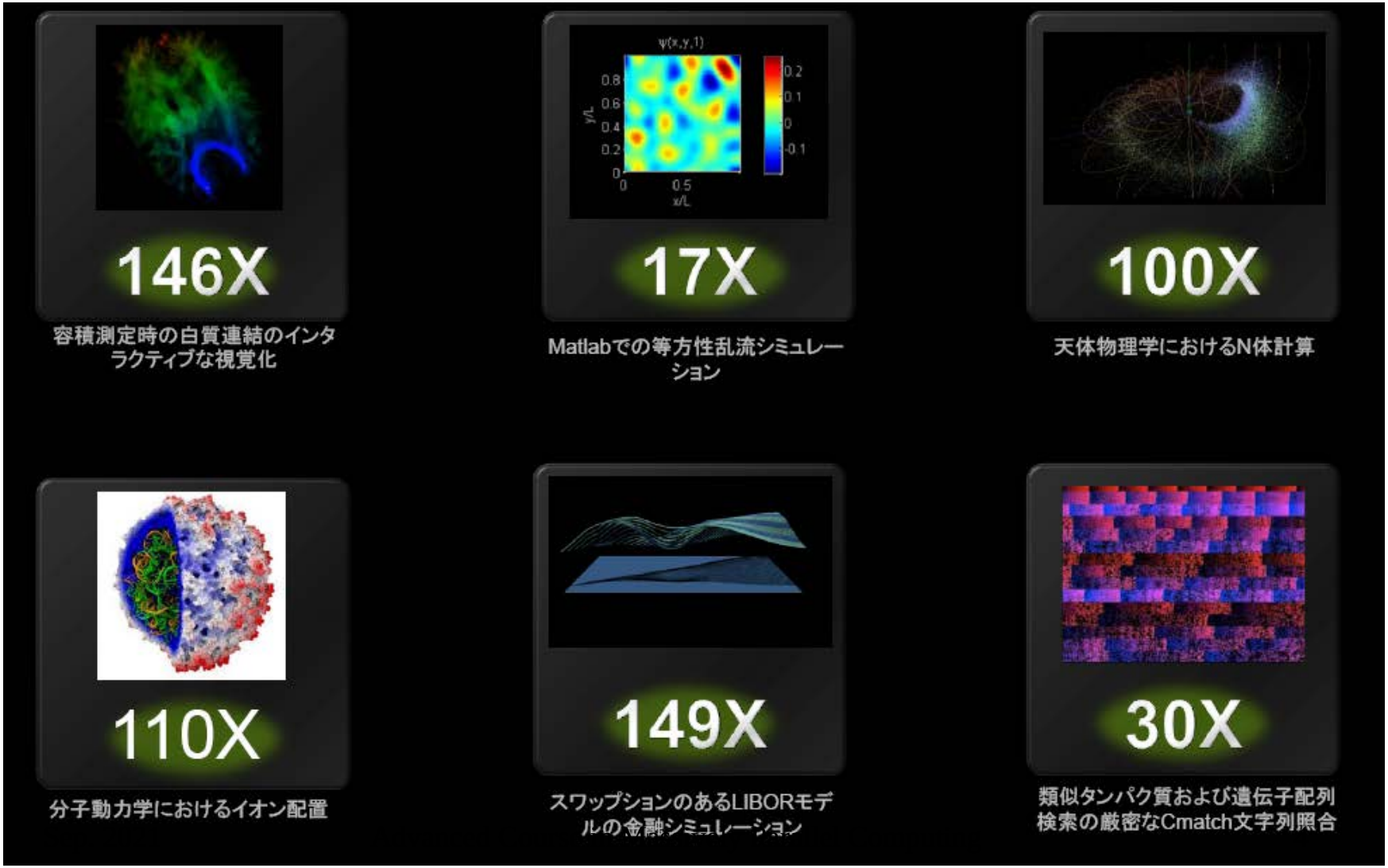
- **GPGPU - General-Purpose Graphic Processing Unit**
  - **A technology to make use of GPU for general-purpose computing (scientific applications)**

- **CUDA （Compute Unified Device Architecture）**
  - **Co-designed Hardware and Software to exploit computing power of NVIDIA GPU for GP computing.**
  - **(In other words), in order to obtain full performance of GPGPU, a program must be written in CUDA language.**

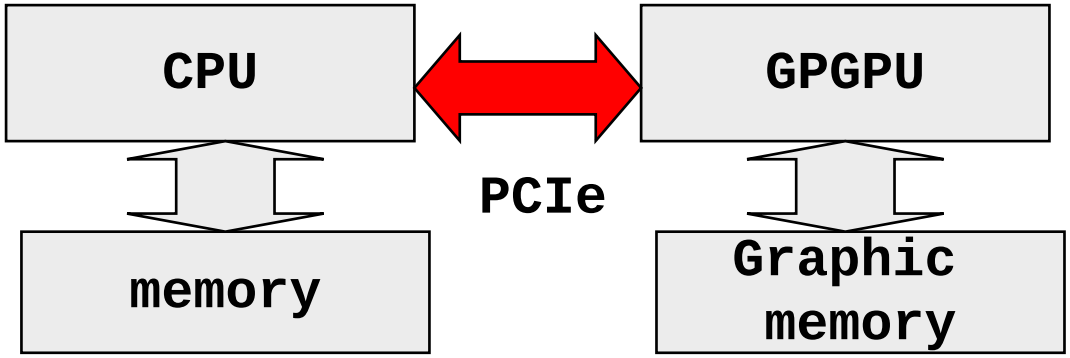
- **It is attracting many people's interest since GPU enables great performance much more than that of CPU (even multi-core) in some scientific fields.**

- **Why GPGPU now?－－ price (cost-performance)!!!**

Applications（From NVIDIA’s slides, 2010?）

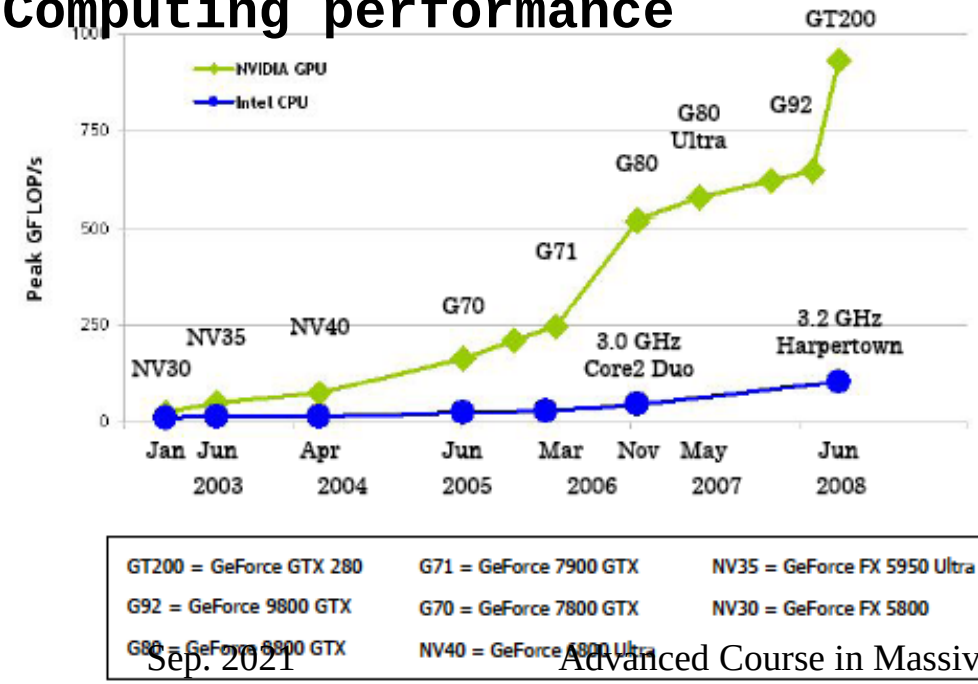


CPU vs. GPU

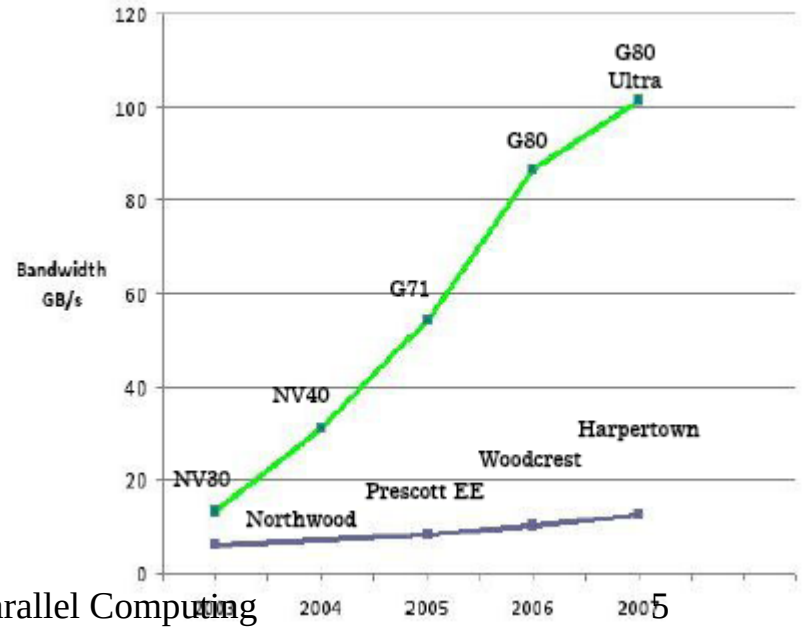


Connected via PCIexpress

Computing performance



Memory bandwidth

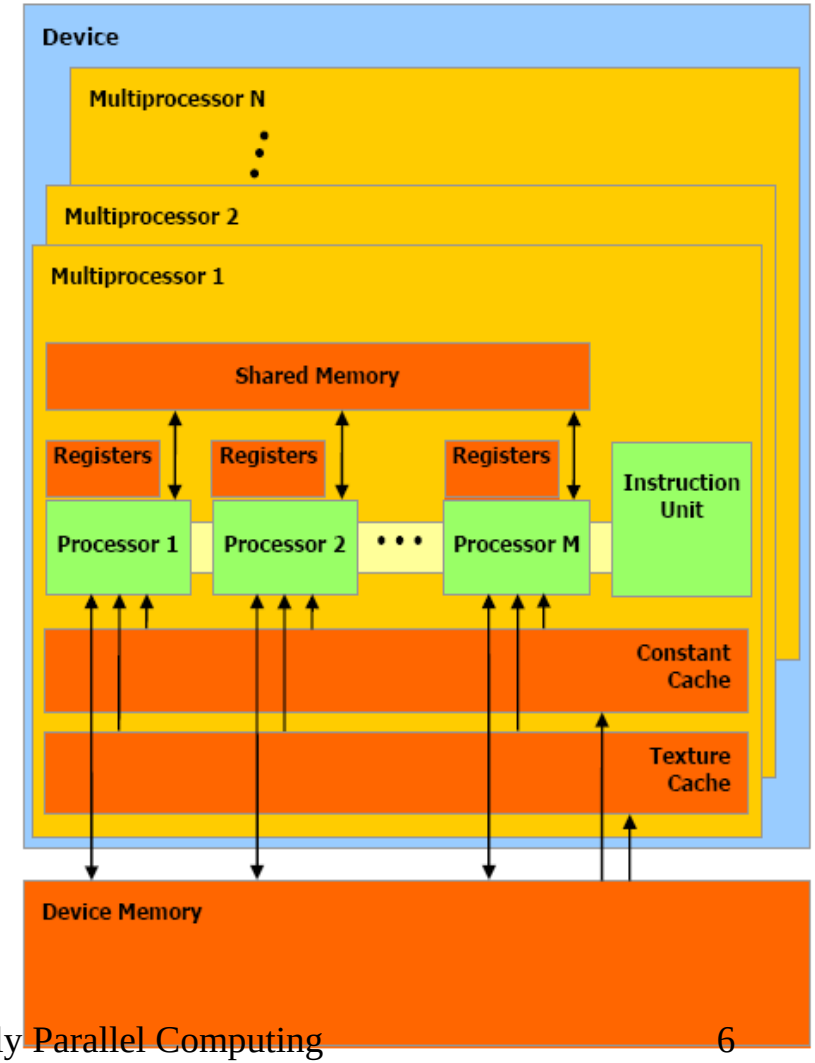


NVIDIA GPGPU's architecture

- **Many multiprocessor in a chip**
  - **eight Scalar Processor (SP) cores,**
  - **two special function units for transcendentals**
  - **a multithreaded instruction unit**
  - **on-chip shared Memory**

- **SIMT (single-instruction, multiple-thread).**
  - **The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state.**
  - **creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.**

- **Complex memory hierarchy**
  - **Device Memory (Global Memory)**
  - **Shared Memory**
  - **Constant Cache**
  - **Texture Cache**

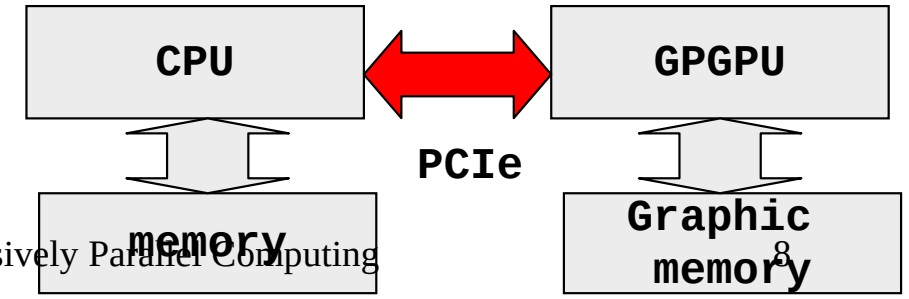


CUDA (Compute Unified Device Architecture)

- C programming language on GPUs
- Requires no knowledge of graphics APIs or GPU programming
- Access to native instructions and memory
- Easy to get started and to get real performance benefit
- Designed and developed by NVIDIA
- Requires an NVIDIA GPU (GeForce 8xxx/Tesla, ….)
- Stable, available (for free), documented and supported
- For both Windows and Linux

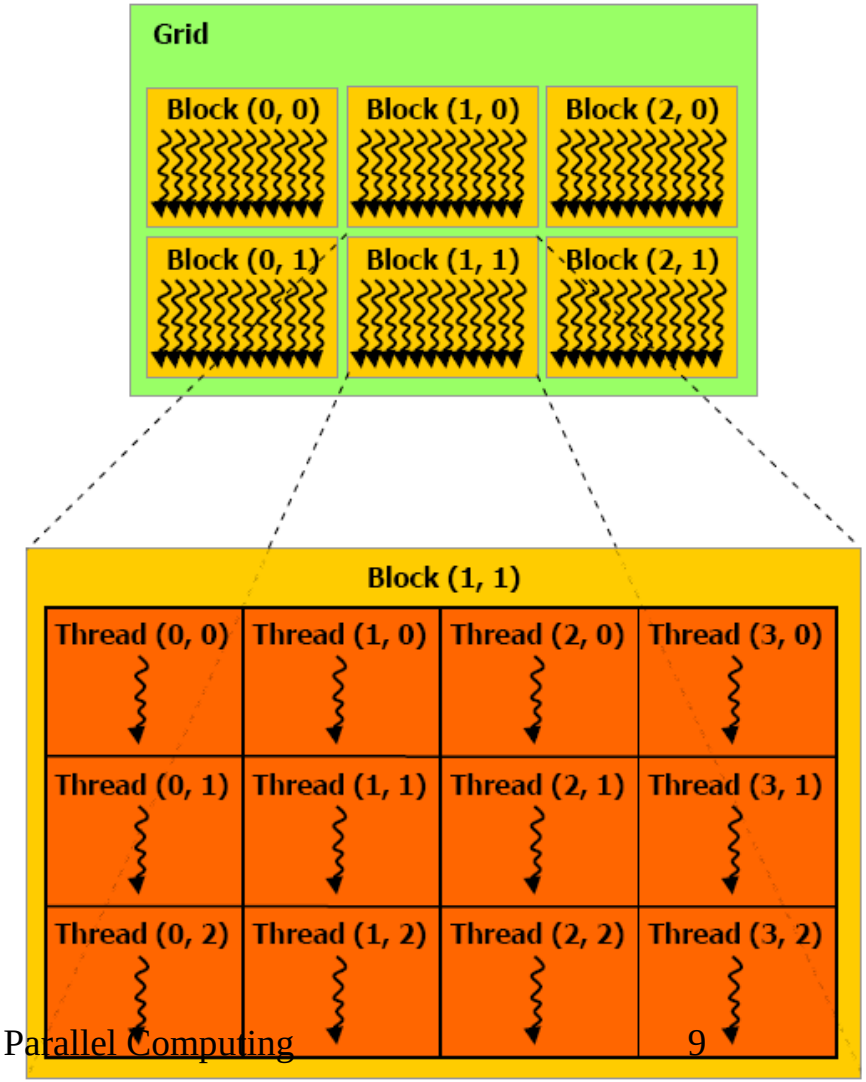
CUDA Programming model (1/2)

- **GPU is programmed as a compute device working as co-processor from CPU(host).**
  - **Codes for data-parallel, compute intensive part are offloaded as functions to the device**
  - **Offload hot-spot in the program which is frequently executed on the same data**
    - **For example, data-parallel loop on the same data**
  - **Call "kernel" a code of the function compiled as a function for the device**
  - **Kernel is executed by multiple threads of device.**
    - **Only one kernel is executed on the device at a time.**
  - **Host (CPU) and device(GPU) has its owns memory, host memory and device memory**
  - **Data is copied between both memory.**



CUDA Programming model (2/2)

- Computational Grid is composed of multiple thread blocks
- Thread block includes multiple threads
- Each thread executes kernel
  - A function executed by each thread called “kernel”
  - Kernel can be thought as one iteration in parallel loop
- Computational Grid and block can have 1,2,3 dimension
- The reserved variable, blockID and threadID have ID of threads.



Example: Element-wise Matrix Add

```
void add_matrix
( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
  for ( int j = 0; j < N; ++j ) {
    index = i + j*N;
    c[index] = a[index] + b[index];
  }
}
int main() {
 add_matrix( a, b, c, N );
}
```

CPU program

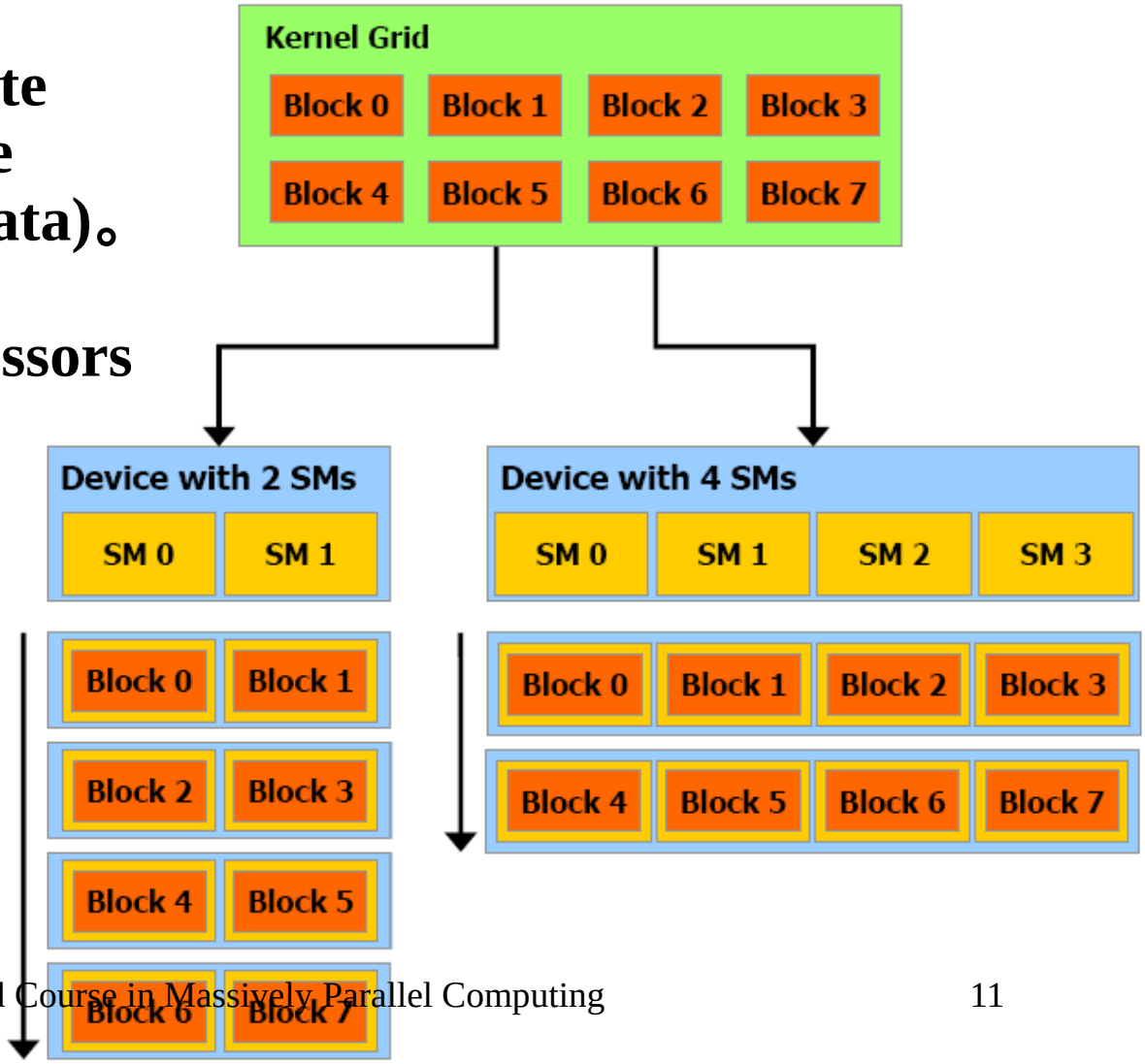
The nested for-loops are replaced with an implicit grid

CUDA program

```
__global__ add_matrix
( float* a, float* b, float* c, int N ) {
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 int j = blockIdx.y * blockDim.y + threadIdx.y;
 int index = i + j*N;
 if ( i < N && j < N )
  c[index] = a[index] + b[index];
}
int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

How Threads are executed

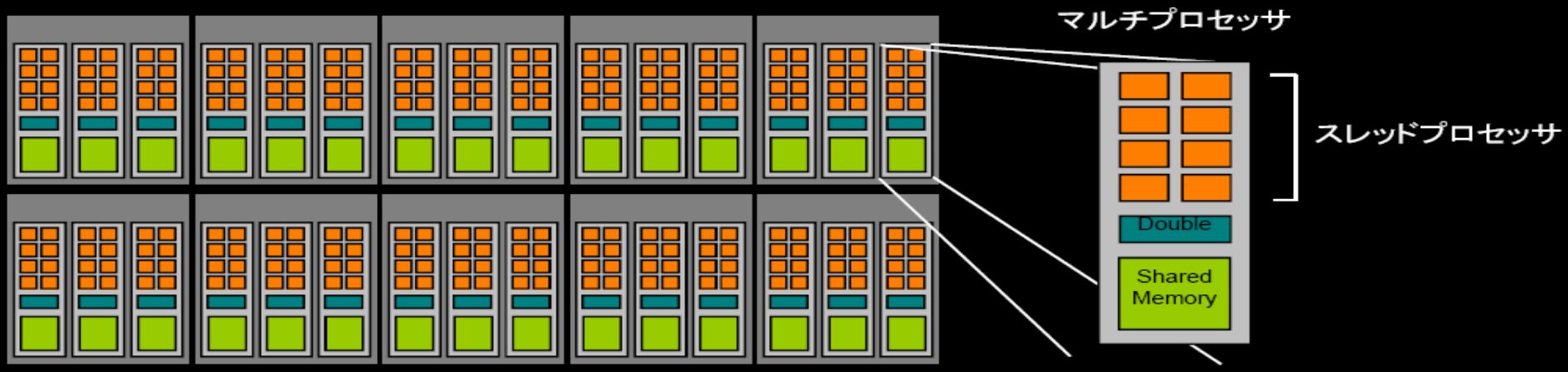
- SM (Streaming Multiprocessor) execute blocks in SIMD (single instruction/multiple data)。
- SM consists of 8 processors



An example of GPGPU configuration

10シリーズアーキテクチャ

- 240個のスレッドプロセッサがカーネルスレッドを処理
- 30個のマルチプロセッサ、それぞれが次のユニットを内蔵
  - 8個のスレッドプロセッサ
  - 1個の倍精度ユニット
  - スレッド協調のための共有メモリ



	Number of Multiprocessors (1 Multiprocessor = 8 Processors)	Compute Capability
GeForce GTX 295	2x30	1.3
GeForce GTX 285, GTX 280	30	1.3
GeForce GTX 260	24	
GeForce 9800 GX2	2x16	
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512	16	
GeForce 8800 Ultra, 8800 GTX	16	
GeForce 9800 GT, 8800 GT, GTX 280M, 9800M GTX	14	
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTX 260M, 9800M GT	12	
Tesla S1070	4x30	1.3
Tesla C1060	30	1.3
Tesla S870	4x16	1.0
Tesla D870	2x16	1.0
Tesla C870	16	1.0
Quadro Plex 2200 D2	2x30	1.3
Quadro Plex 2100 D4	4x14	1.1
Quadro Plex 2100 Model S4	4x16	1.0

2008!

Tesla C1060

```
#core: 240 cores
Clock Frequency: 1.3GHz
memory capacity: 4GB
Performance (SP): 933GFlops (peak)
Performance (DP): 78GFlops (peak)
Memory Bandwidth: 102GB/sec
Standard Power Consumption: 187.8W
Floating Point Format: IEEE 754 SP/DP
Host Intf: PCI Express x16 (PCIe 2.0)
```

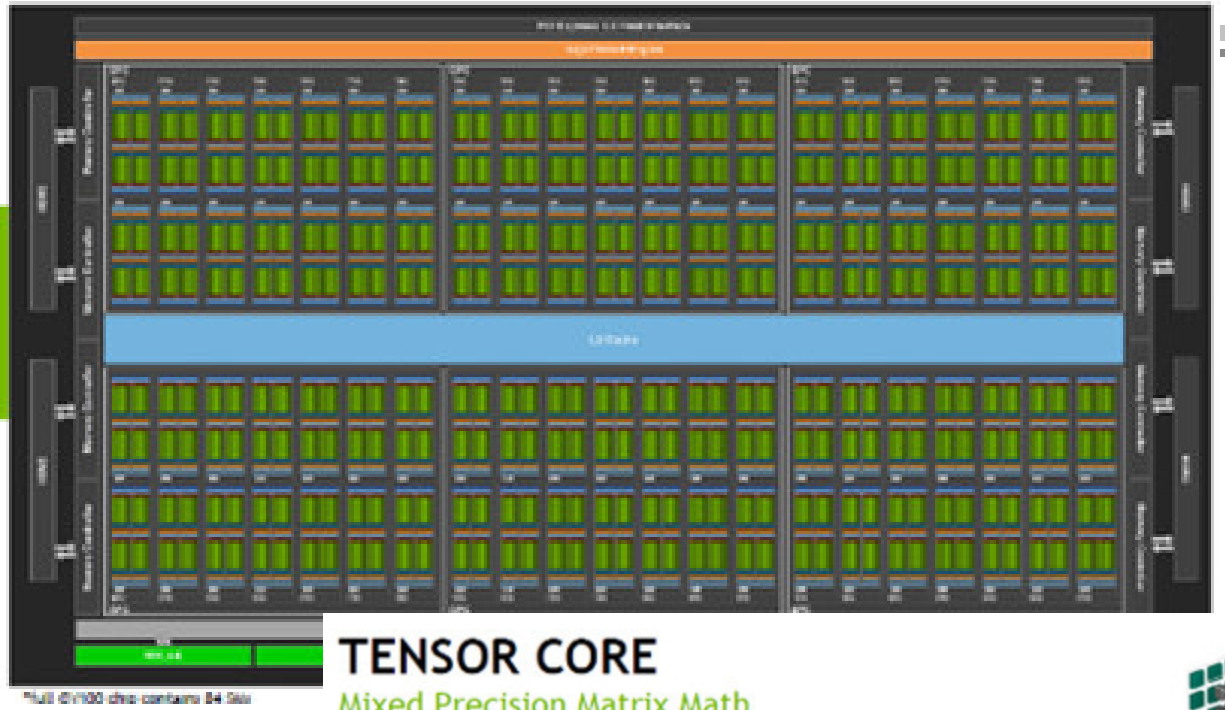
2018

TESLA V100

21B transistors
815 mm²

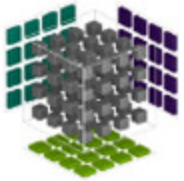
80 SM
5120 CUDA Cores
640 Tensor Cores

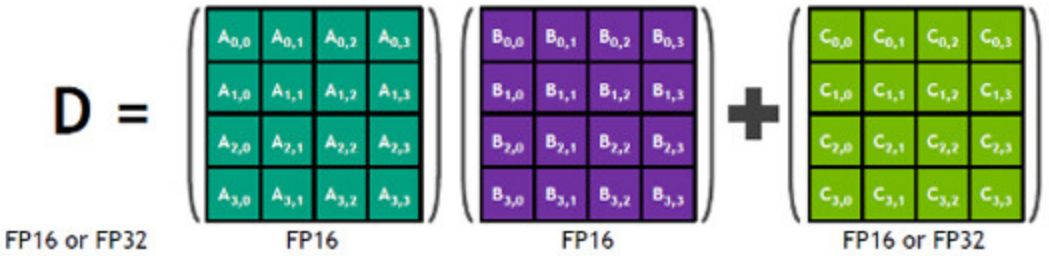
16 GB HBM2
900 GB/s HBM2
300 GB/s NVLink



TENSOR CORE

Mixed Precision Matrix Math
4x4 matrices





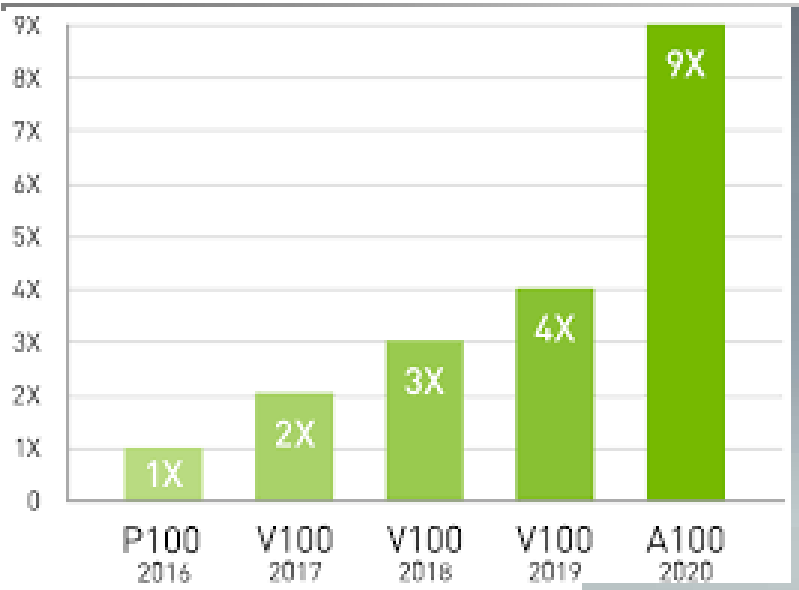
Volta v100

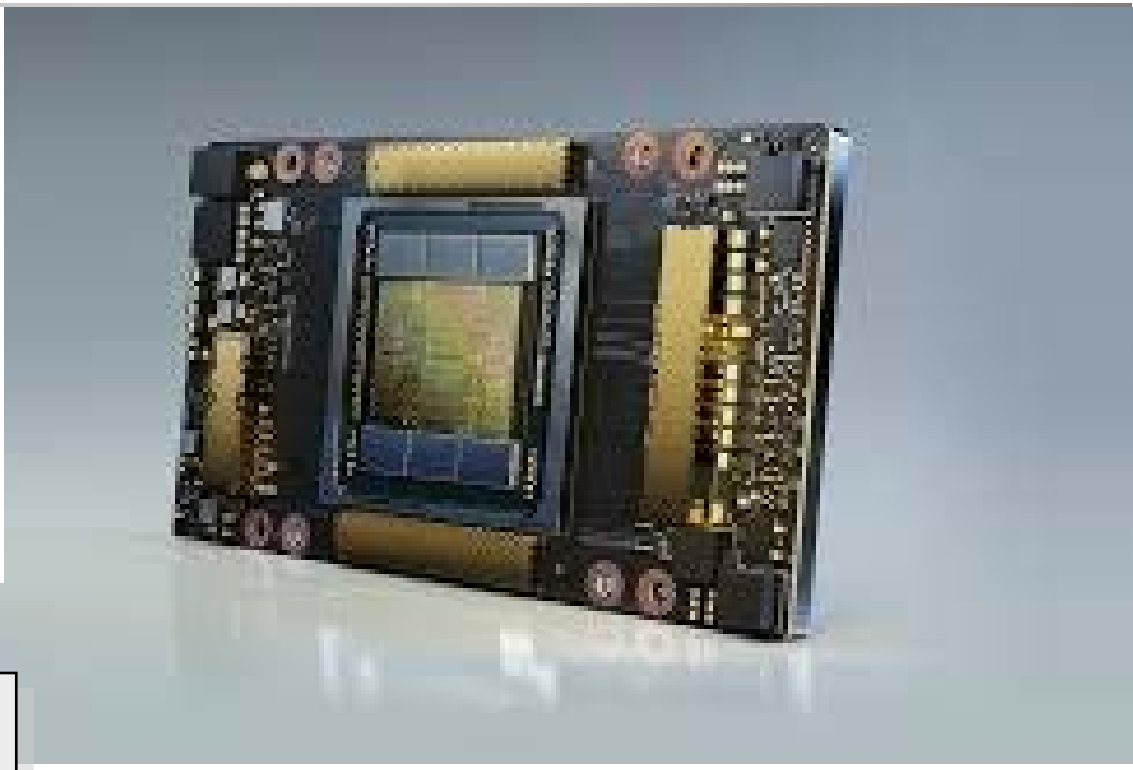
```
#core: 5129 cores (80SM)
Clock Frequency: 1.455GHz (boost)
memory capacity: 16GB
Performance (SP): 15TFlops (peak)
Performance (DP): 7.5TFlops (peak)
Memory Bandwidth: 1TB/sec ???
Standard Power Consumption: ??? W
Floating Point Format: IEEE 754 SP/DP
Host Interface: PCI Express x16 (PCIe 3.0)
```

GPU PERFORMANCE COMPARISON

	P100	V100	Ratio
DL Training	10 TFLOPS	120 TFLOPS	12x
DL Inferencing	21 TFLOPS	120 TFLOPS	6x
FP64/FP32	5/10 TFLOPS	7.5/15 TFLOPS	1.5x
HBM2 Bandwidth	720 GB/s	900 GB/s	1.2x
STREAM Triad Perf	557 GB/s	855 GB/s	1.5x
NVLink Bandwidth	160 GB/s	300 GB/s	1.9x
L2 Cache	4 MB	6 MB	1.5x
L1 Caches	1.3 MB	10 MB	7.7x

Ampere 100





Ampere A100

```
#core: 6,912 cores
Clock Frequency: ????
memory capacity: 40 GB
Performance (SP): 19.5 TFlops (peak)
Performance (DP): 9.7 TFlops (peak)
Memory Bandwidth: 1.6 TB/sec
Standard Power Consumption: 400 W
Floating Point Format: IEEE 754 SP/DP
Host Interface: PCI Express x ?? (PCIe 4.0)
```

Invoke (Launching) Kernel

- **Host processor invoke the execution of kernel in this form similar to function call:**

```
kernel<<<dim3 grid, dim3 block, shmem_size>>>(…)
```

- **Execution Configuation (“<<< >>>”)**
  - **Dimension of computational grid : x and y**
  - **Dimension of thread block: x、y、z**

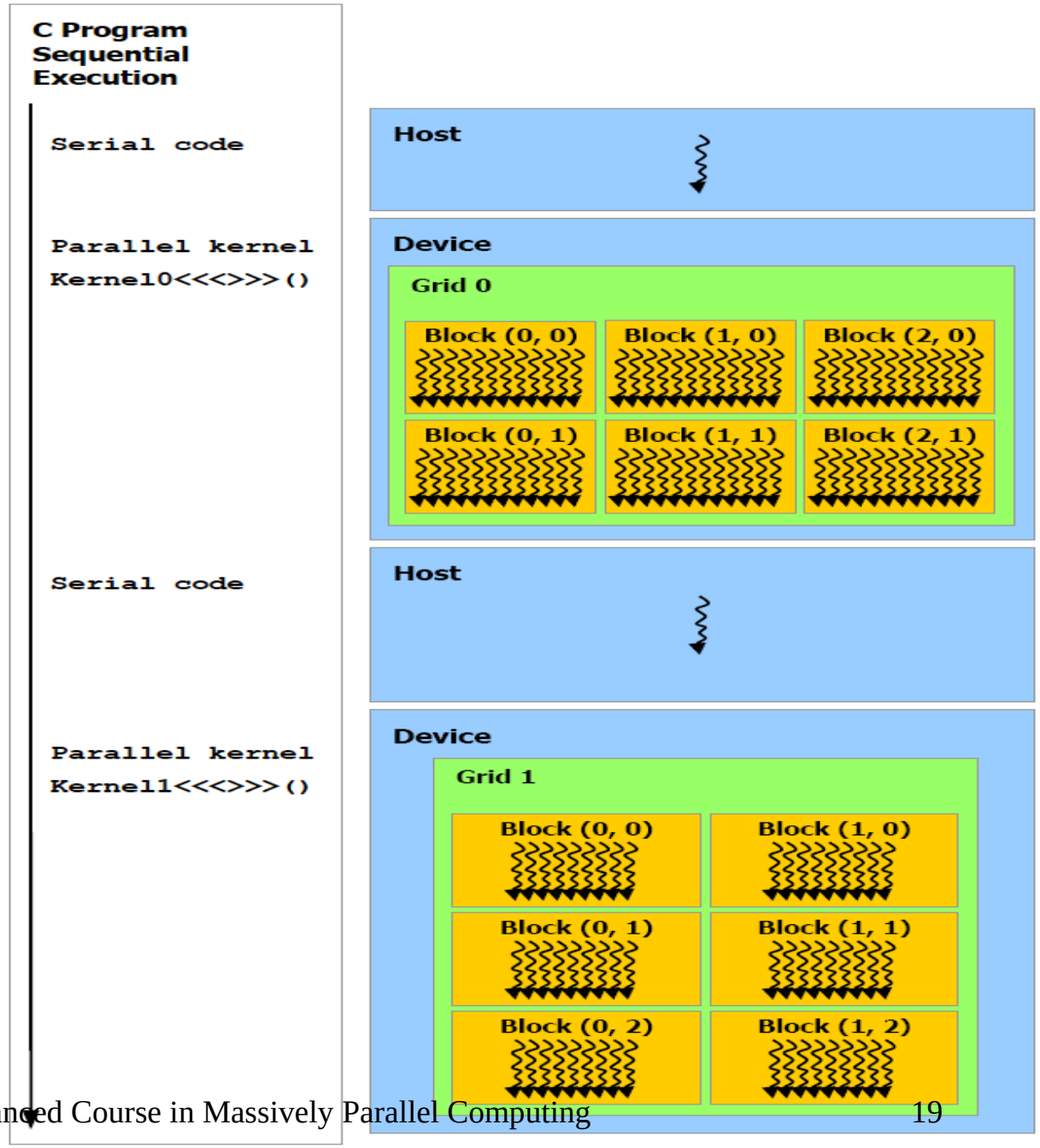
```
dim3 grid(16 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

CUDA kernel and thread

- **Parallel part of applications are executed as a kernel of CUDA on the device**
  - **One kernel is executed at a time**
  - **Many threads execute kernel function in parallel.**

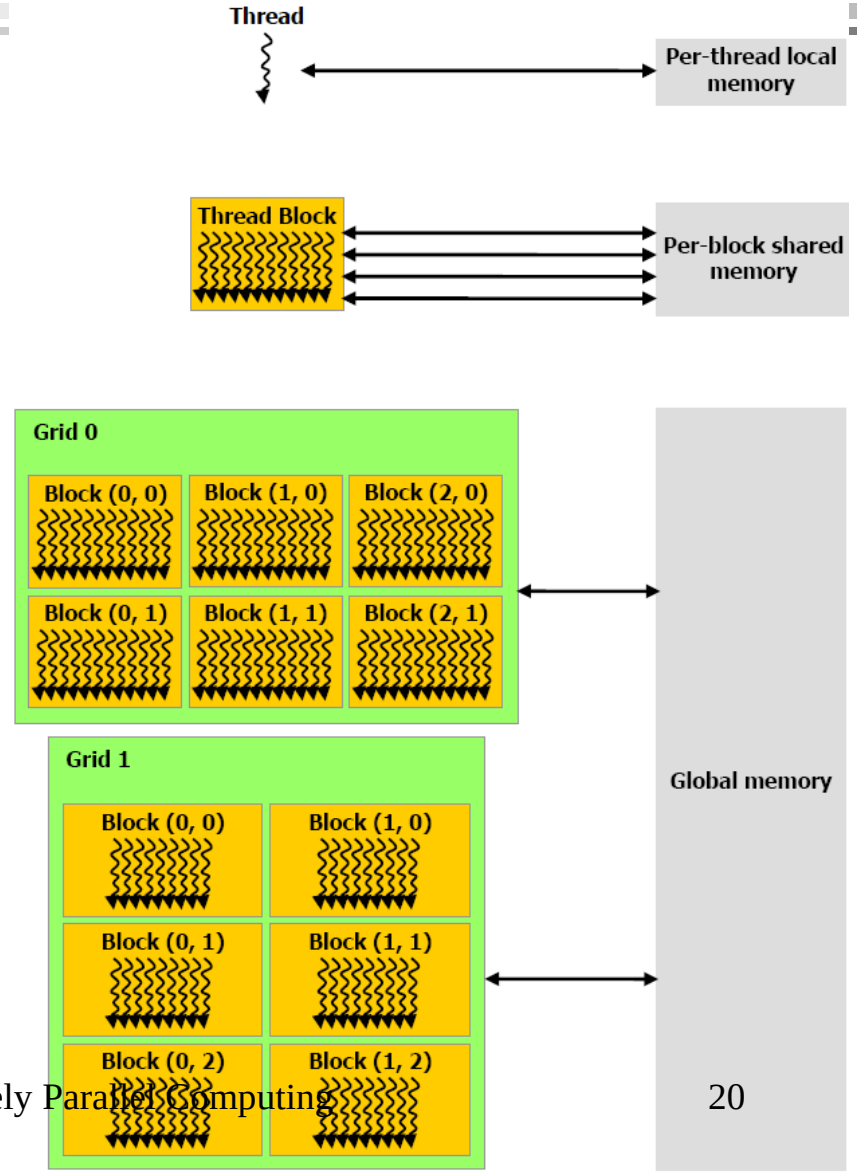
- **Difference between CUDA thread and CPU thread**
  - **CUDA thread is a very light-weight thread**
    - **Overhead of thread creation is very small**
    - **Thread switching is also very fast since it is supported by hardware.**
  - **CUDA exploit its performance and efficient execution by a thousands of threads.**
    - **Conventional Multicore supports only a few threads (by software)**

Execution of CPU Code and Kernel code by Device



Grid, Block, thread and Memory hierarchy

- Thread can access local memory (per-thread)
- Thread can access “shared memory” on chip, which is attached for each thread block (SM).
- Thread in Computational Grid access and share a global memory.



Memory management (1/2)

- CPU and GPU have different memory space.
- Hosts（CPU）manages device (GPU）memory

- Allocation and Deallocation of GPU memory
  - `cudaMalloc(void ** pointer, size_t nbytes)`
  - `cudaMemset(void * pointer, int value, size_t count)`
  - `cudaFree(void* pointer)`

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
cudaMalloc( (void**)&d_a nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Memory management (2/2)

- **Data copy operation between CPU and device**
  - `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
    - **Direction specifies how to copy from src to dst , see below**
    - **Block a caller of CPU thread (execution) until the memory transfer completes.**
    - **Copy operation starts after previous CUDA calls.**
  - `enum cudaMemcpyKind`
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost`
    - `cudaMemcpyDeviceToDevice`

Executing Code on the GPU

- **Kernels are C functions with some restrictions**
  - **Can only access GPU memory**
  - **Must have void return type**
  - **No variable number of arguments (“varargs”)**
  - **Not recursive**
  - **No static variables**
  - **Function arguments**

- **Function arguments automatically copied from CPU to GPU memory**

Function Qualifiers

- **`__global__` : invoked from within host (CPU) code,**
  cannot be called from device (GPU) code must return void
- **`__device__` : called from other GPU functions,**
  cannot be called from host (CPU) code
- **`__host__` : can only be executed by CPU, called from host**

- **`__host__` and `__device__` can be combined.**
  - **Sample use: overloading operators**
  - **Compiler will generate both CPU and GPU code**

CUDA Built-in Device Variables

- **`__global__` and `__device__` functions have access to these automatically defined variables**
  - **`dim3 gridDim;`**
    - **Dimensions of the grid in blocks (at most 2D)**
  - **`dim3 blockDim;`**
    - **Dimensions of the block in threads**
  - **`dim3 blockIdx;`**
    - **Block index within the grid**
  - **`dim3 threadIdx;`**
    - **Thread index within the block**

A simple example

```
__global__ void minimal( int* d_a)
{
      *d_a = 13;
}
```

```
__global__ void assign( int* d_a, int value)
{
  int idx = blockDim.x * blockIdx.x + threadIdx.x;
  d_a[idx] = value;
}
```

A simple example

```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
  int iy = blockDim.y * blockIdx.y + threadIdx.y;
  int ix = blockDim.x * blockIdx.x + threadIdx.x;
  int idx = iy * w + ix;
  d_a[idx] = value;
}
...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```

Example code to increment array elements

CPU code

```
void inc_cpu(int*a, intN)
{
  int idx;
  for (idx =0;idx<N;idx++)
    a[idx]=a[idx] + 1;
}

voidmain()
{
 ...
 inc_cpu(a, N);
}
```

CUDA codes

```
__global__ void
  inc_gpu(int*a_d, intN){
  int idx = blockIdx.x* blockDim.x
            +threadIdx.x;
  if (idx < N)
    a_d[idx] = a_d[idx] + 1;
}
void main()
{
   …
   dim3dimBlock (blocksize);
   dim3dimGrid(ceil(N/
               (float)blocksize));
   inc_gpu<<<dimGrid,
       dimBlock>>>(a_d, N);
}
```

Example (host-side program)

```
// allocate host memory
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);

// Execute kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

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

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

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
   a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;
  return EXIT_SUCCESS;
}
```

CUDA Qualifiers for variable

- `__device__`
  - **Allocated in device global memory（Large, high-latency, no cache）**
  - **Allocated by cudaMallocで（__device__ is default）**
  - **Access by every thread.**
  - **extent: during execution of application**

- `__shared__`
  - **Stored in on-chip “shared memory” (SRAM, low latency)**
  - **Allocated by execution configuration or at compile time**
  - **Accessible by all threads in the same thread block**

- **Unqualified variables**
  - **Scalars and built-in vector types are stored in registers**
  - **Arrays may be in registers or local memory *(registers are not addressable)***

How to use/specify shared memory

Compile time

```
__global__ void kernel(…)
{
…
__shared__ float sData[256];
…
}
int main(void)
{
…
kernel<<<nBlocks,blockSize>>>(…);
}
```

Invocation time

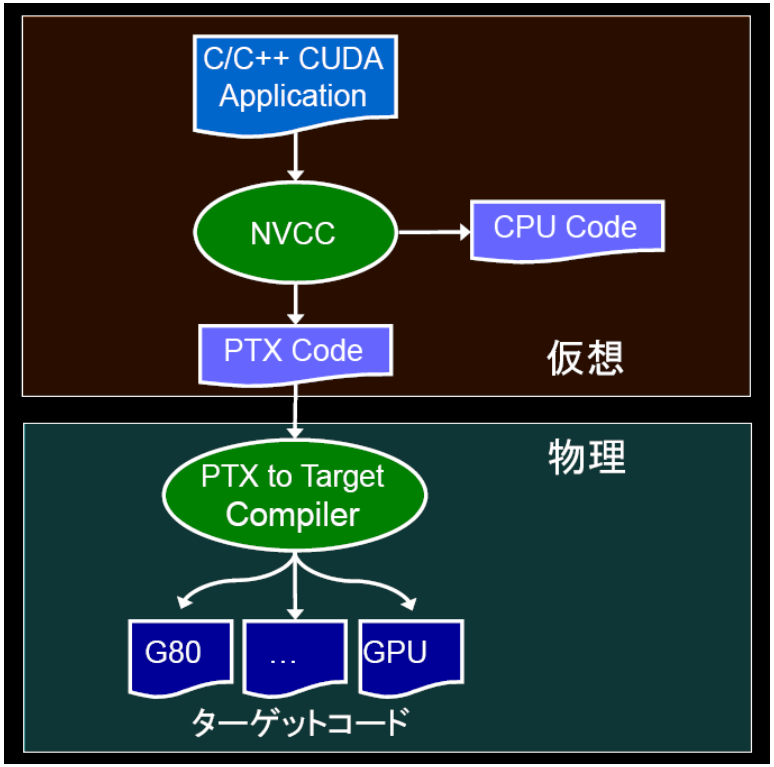
```
__global__ void kernel(…)
{
  …
  extern __shared__ float sData[];
  …
}

int main(void)
{
  …
  smBytes =
  blockSize*sizeof(float);
  kernel<<<nBlocks, blockSize,
                smBytes>>>(…);
  …
}
```

GPU Thread Synchronization

- `void __syncthreads();`
  - Synchronizes all threads in a block
  - Generates barrier synchronization instruction
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Allowed in conditional code only if the conditional is uniform across the entire thread block
- Synchronization between blocks is not supported
  - Done by host-side

Compiler

- C Source program with CUDA is compiled by nvcc.
- Nvcc is a comile-driver:
  - Execute required tools and cudacc、g++、cl
- Nvcc generates following codes:
  - C object code（CPU code）
  - PTX code for GPU
  - Glue code to call GPU from CPU
- Objects required to execute CUDA program
  - CUDA core library（cuda）
  - CUDA runtime library（cudart）



Optimization of GPU Programming

- **Maximize parallel using GPGPU**
- **Optimize/ avoid memory access to global memory**
  - **Rather than storing data, re-computation may be cheaper in some cases**
  - **Coalescing memory access**
  - **Use cache in recent NVIDIA GPGPU**
- **Optimize/avoid communication between CPU(host) and GPU (Device)**
  - **Communication through PCI Express is expensive**
  - **Re-computing (redundant computing) may be cheaper than communications.**

Optimization of Memory access

- **Coalescing global memory access**
  - **Combine memory access to contiguous area**
- **Make use of shared memory**
  - **Much faster than global memory (several x 100 times faster)**
    - **On-chip Memory**
    - **Low latency**
  - **Threads in block share the memory.**
  - **All threads can share the data computed by other threads.**
  - **To load shared memory from global memory, coalesce the memory and use them**
- **Use cache (shared memory) as in conventional CPU**
  - **Recent GPGPU has a cache at the same level of shared memory**

Optimization of Host-device communication

- **The bandwidth between host and device is very narrow compared with the bandwidth of device memory.**
  - **Peak bandwidth 4GB/s （PCIe x16 1.0） vs. 76 GB/s （Tesla C870）**
- **Minimize the communication between host-device**
  - **Intermediate results must be kept in device memory to avoid communications**
- **Grouping communication**
  - **Large chunk of communication is more efficient than several small chunk of communications**
- **Asynchronous communication**
  - **Make use of stream**
    - **cudaMemcpyAsync(dst, src, size, direction, 0);**

Host Synchronization

- **All kernel launches are *asynchronous***
  - **control returns to CPU immediately**
  - **kernel executes after all previous CUDA calls have completed**

- **`cudaMemcpy()` is *synchronous***
  - **control returns to CPU after copy complete**
  - **copy starts after all previous CUDA calls have completed**

- **`cudaThreadSynchronize()`**
  - **blocks until all previous CUDA calls complete**

GPU by Libraries

- **NVIDIA Library**
  - **CuFFT Fast Fourier transform**
  - **CuBLAS Basic Liner Algebra Lib (Dense matrix)**
  - **CuSPARSE Sparse Matrix lib**
  - **CuSOLVER Matrix Solvers (Dense and Sparse)**
  - **CuDNN Deep neural network**
  - **CuRAND random number generator**

APIs for GPU Libs

- 1. Make handle
- 2. Allocate device memory
- 3. Transfer data to device from host
- 4. Convert input data format in GPU
- 5. Execute functions
- 6. Convert output data format for host
- 7. Transfer data from device to host
- 8. Deallocate device memory
- 9. Remove handle

```
float *hstA,*hstB,*hstC;
float *devA,*devB,*devC;

// 行列演算 C=αAB+βC のパラメータ
float alpha = 1.0f;
float beta = 0.0f;
// szie A B C
int num = 8192;
int n2 = num*num;
size_t memSz = n2 * sizeof(float);
// allocate host memory
hstA=(float*)malloc(msmSz);
hstB=(float*)malloc(msmSz);
hstC=(float*)malloc(msmSz);

// Initialize hstA,hstB
// allocate device memory
cudaMalloc((void **)&devA,memSz);
cudaMalloc((void **)&devB,memSz);
cudaMalloc((void **)&devC,memSz);

//device memcpy
cublasSetVector(n2, sizeof(float), hstA, 1, devA, 1);
cublasSetVector(n2, sizeof(float), hstB, 1, devB, 1);

// デバイス側ハンドル作成
```

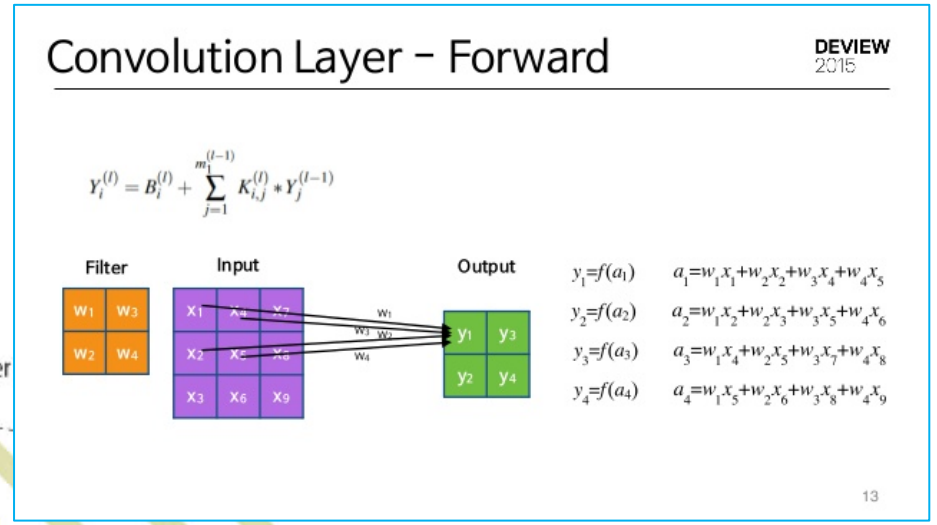
APIs for GPU Libs

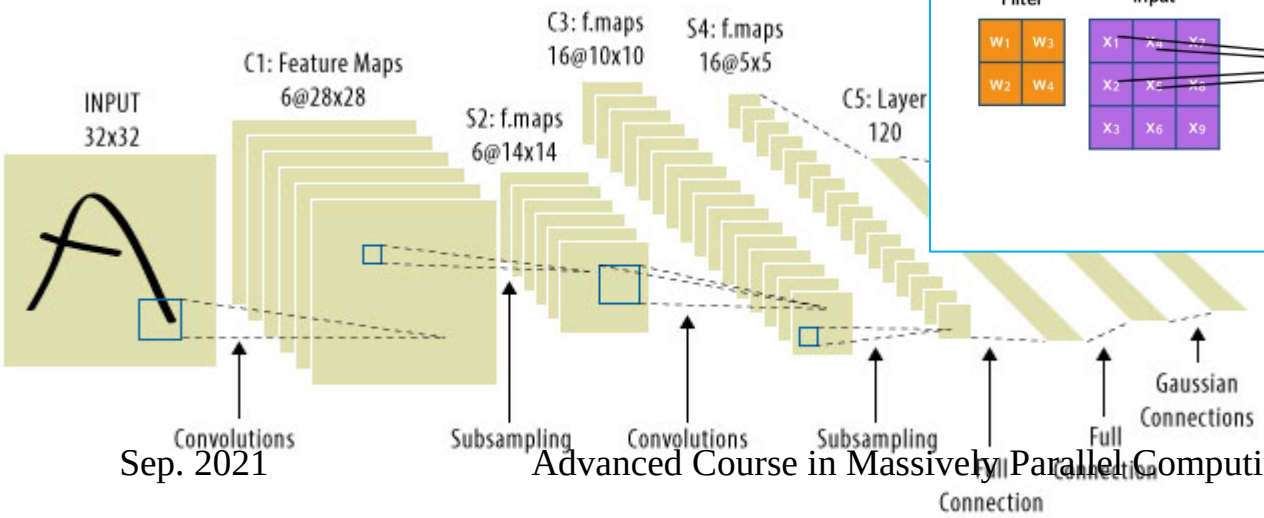
- 1. Make handle
- 2. Allocate device memory
- 3. Transfer data to device from host
- 4. Convert input data format in GPU
- 5. Execute functions
- 6. Convert output data format for host
- 7. Transfer data from device to host
- 8. Deallocate device memory
- 9. Remove handle

```
// make handle
cublasHandle_t handle;
cublasCreate(&handle);
// call blas
cublasSgemm(
        handle,
        CUBLAS_OP_N, //行列A 転置有無
        CUBLAS_OP_N, //行列B 転置有無
        num,    // #col of A
        num,    // #row of B
        num,    // #row of A (== #col of B)
        &alpha, //
        devA,   // A
        num,    // #col of A
        devB,   // B
        num,    // #col of B
        &beta,  //
        devC,   // C
        num // #col of C
);
status = cublasDestroy(handle);
// get result
cublasGetVector(n2, sizeof(float), devC, 1, hstC, 1)
// free
cudaFree(devA);
cudaFree(devB);
cudaFree(devC);
```

GPU for DL/AI

- **cuDNN - GPU Library for Deep Learning**
  - **Mainly for training**
  - **Fast convolution (2D, 3D) for CNN (Convolution Neural Network)**
  - **Used by Caffe, PyTorch, …**

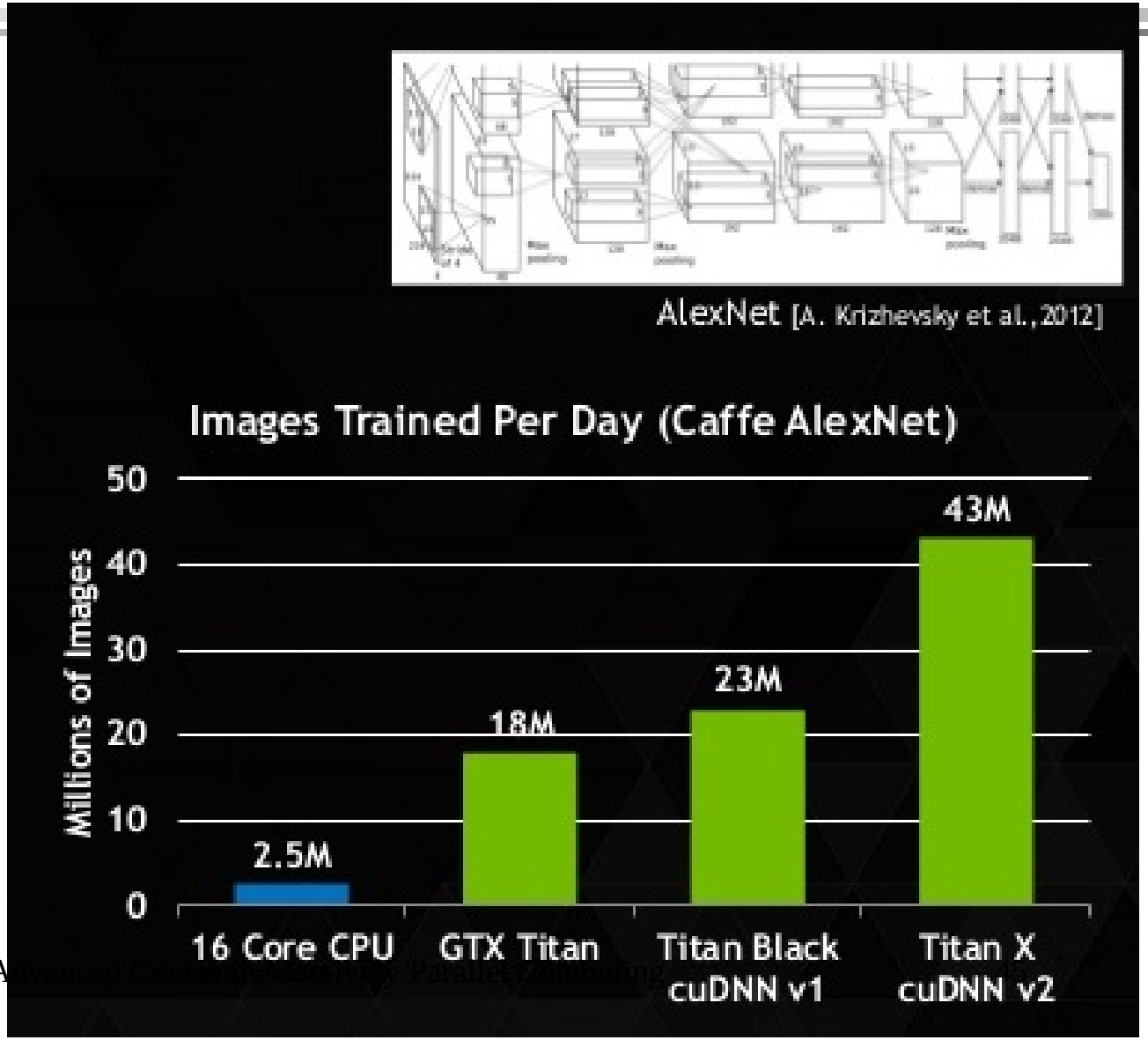




convolution

Performance of cuDNN

- **https://www.slideshare.net/NVIDIAJapan/1072-cuda**



Final remarks

- **GPGPU is a good solution for apps which can be parallelized for GPU.**
  - **It can be very good esp. when the app fits into one GPU.**
  - **If the apps needs more than one GPU, the cost of communication may kill performance.**

- **Programming in CUDA is still difficult ...**
  - **Performance tuning, memory layout ...**
  - **OpenACC / OpenMP will help you!**