An Introduction to GPU Architecture and CUDA C/C++ Programming

Bin Chen

April 4, 2018

Research Computing Center

FLORIDA STATE UNIVERSITY
RESEARCH COMPUTING CENTER

Outline

- Introduction to GPU architecture
- Introduction to CUDA programming model
- Using the GPU resources at the FSU/RCC
- Deep learning accelerated by GPUs

GPU-Computing

- **GPU**: Graphics Processing Unit
- **GPGPU**: General Purpose GPU.
- **GPU-accelerated computing**: is the use of a GPU together with a CPU to accelerate scientific, analytics, engineering, consumer, and enterprise applications.
- CUDA: Compute Unified Device Architecture
- **Remark.** GPU does NOT work by itself. It is used as a device of a CPU.

GPU Market Shares (Q4-2017)

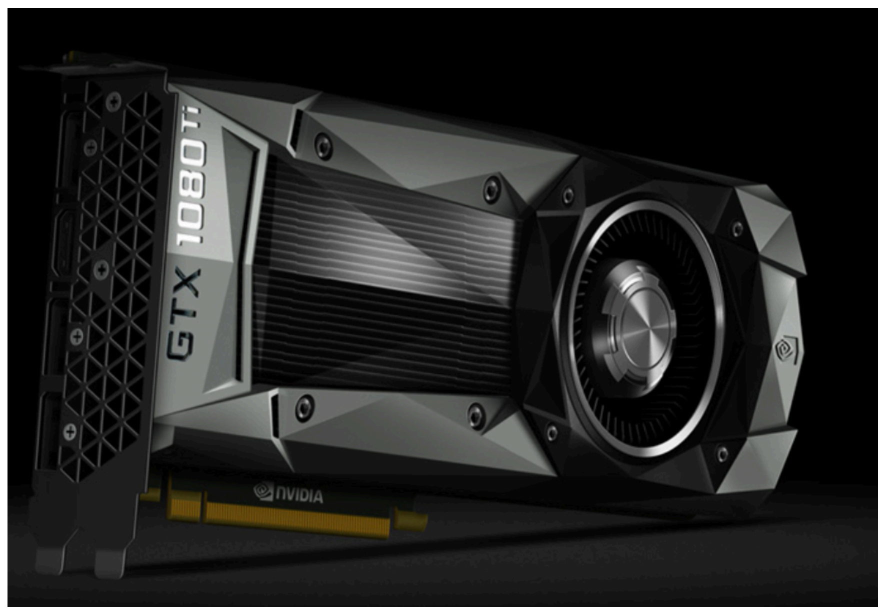
GPU Supplier	Market share this quarter	Market share last quarter	Market share last year.
AMD	33.7%	27.2%	29.5%
Nvidia	66.3%	72.8%	70.5%
Total	100%	100%	100%

Source: Jon Peddie Research

- We will be concentrate on NVIDIA GPU product
- The CUDA language was created by NVIDIA

NVIDIA GPU Product Families

- **Tegra**: mobile and embedded devices (e.g., phones)
- **GeForce**: consumer graphics (e.g., gaming)
- **Quadro**: professional visualization
- **Tesla:** high performance computing (Tesla M2050)



GTX 1080 Ti

Compute Capability (p1)

- GPU product family is classified using Compute Capability
- **Volta** class architecture has major version number 7
- **Pascal** class architecture has major version number 6
- **Maxwell** class architecture has major version number 5
- **Kepler** class architecture has major version number 3
- **Fermi** class architecture has major version number 2
- **Tesla** class architecture has major version number 1

Compute Capability: major.minor say, 6.1

Compute Capability (p2)

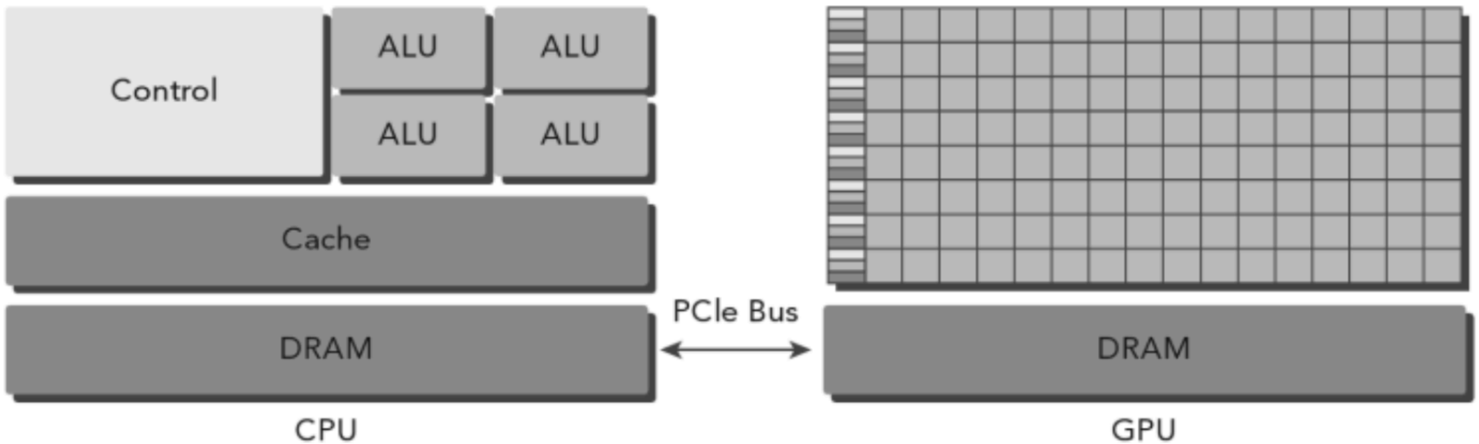
- **GTX1080Ti key data:**

Brand Name	GTX1080 Ti
Compute Capability	6.1
Micro-Architecture	Pascal
Number Stream Multi-Processors	28
Number of CUDA Cores	3584
Boost Clock	1600 MHZ
Memory Capacity	11 GB
Memory Bandwidth	~484GBs
FP32 TFLOPS	~11.4 TFLOPS

GPU core VS CPU core

- **CPU core:** relatively heavy-weight, designed for complex control logic, optimized for sequential programs.
- **GPU core**: relatively light-weight, designed with simple control logic, optimized for data-parallel tasks, focusing on throughput of parallel programs.
- **CPU+GPU**: heterogeneous architecture

Heterogeneous Architecture



Multi-core CPU + Many-core GPU

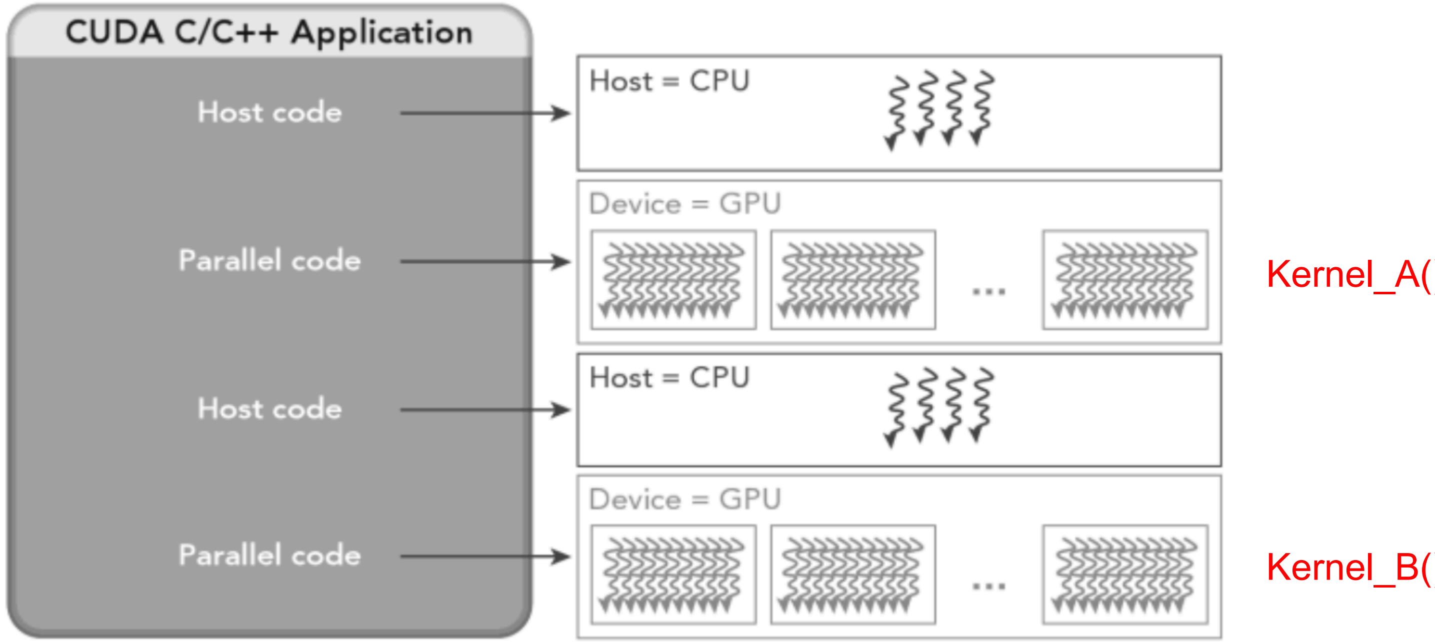
Remark: GPU has its own memory, connect to GPU via PCI-express bus

Remark: Differentiate Multi-Core from Many-Core (Intel Phi co-processor)

CUDA Programming Model (p1)

- Divide your code into Host (CPU) and Device (GPU) Code
- Processing flow of a CUDA program:
  a. Copy data from CPU memory to GPU memory
  b. Invoke kernel to run on the GPU.
  c. Copy data back from GPU to CPU memory
  d. Release the GPU memory and reset the GPU.
- CUDA code file name extension .cu
- CUDA compiler: nvcc (it compiles .c, .cpp too!)
- `$ nvcc -o a.out a.cu`

CUDA Programming Model (p2)



- The kernel function is run concurrently by many threads on the GPU.
- CPU might or might not wait for GPU depending on synchronization.
- Can have more than one kernel functions in your CUDA application.

2-Level Thread Hierarchy (p1)

- There are many threads, so they need be managed.
- Grid: All threads spawned by a single kernel.
- Grid is made up of many thread blocks.
- A thread block is a group of threads which can cooperate

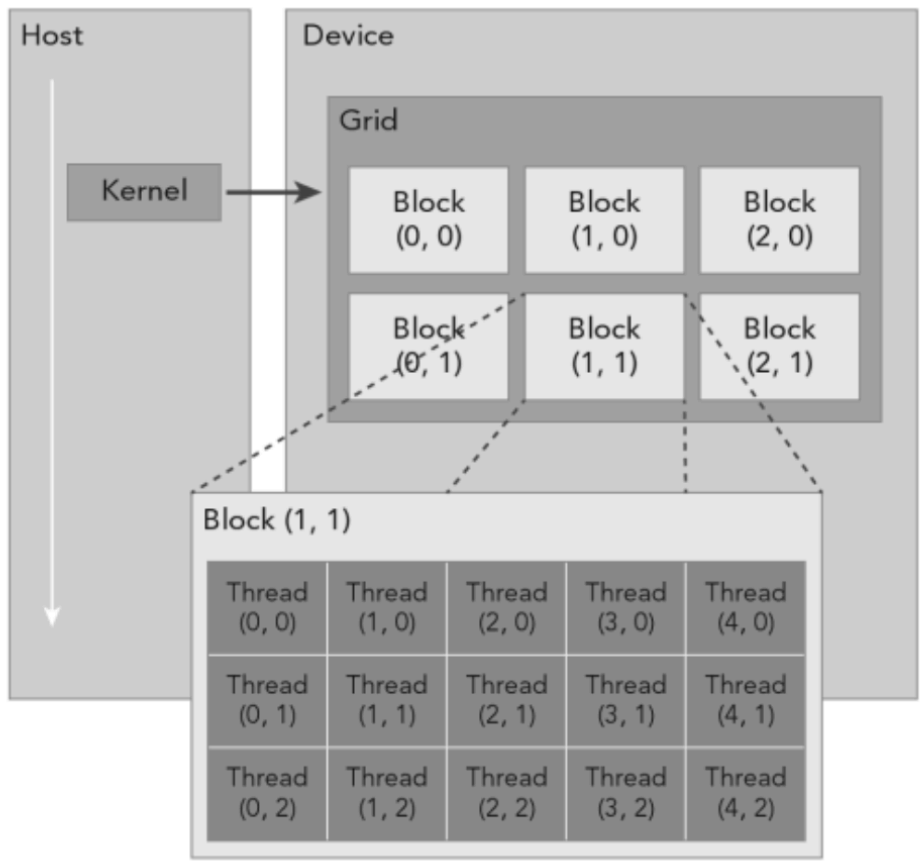
    Intra-block synchronization

    Shared memory within a block
- A thread finds its own unique id using two coordinates:

  blockIdx and threadIdx, for example (1D case):

    id = threadIdx.x + blockIdx.x*blockDim.x

2-Level Thread Hierarchy (p2)



Example: 2D grid + 2D block

Hello World

```
#include <stdio.h>

__global__ void helloFromGPU() {
  printf("Hello World from GPU thread %d\n", threadIdx.x);
}

int main(void) {

  helloFromGPU<<<1,16>>>();
  cudaDeviceSynchronize();
  return 0;

}
```

CUDA Hello World

First CUDA Kernel: show my id

```
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void checkIdx() {

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int tz = threadIdx.z;

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;

    printf("threadIdx (%d,%d,%d), gridIdx (%d,%d,%d)\n",
            tx,ty,tz,bx,by,bz);

}
```

An example Kernel function checkIdx()

First CUDA Code Example: main()

```
int main(){

    int  nElem = 15;
    dim3 dimBlock(4,1,1);
    dim3 dimGrid( (nElem + dimBlock.x - 1)/dimBlock.x,  1, 1);

    printf("blockdim = (%d, %d, %d)\n", dimBlock.x, dimBlock.y, dimBlock.z);
    printf("griddim  = (%d, %d, %d)\n", dimGrid.x,  dimGrid.y,  dimGrid.z);

    checkIdx<<<dimGrid, dimBlock>>>();
    cudaDeviceReset();
    return 0;
}
```

Kernel function invocation: checkIdx<<<grid, block>>>();

GUDA Kernel Function

- Declaration Syntax:

```
__global__ void name(arg1, arg2, …) {
        function body;
}
```

- __global__ is a function type qualifier.
- Kernel function is invoked by CPU, but run on GPU in many copies (one thread per copy).
- Kernel invoking Syntax:

  name<<<grid, block>>>(arg1, arg2, …)

  both grid, block are of type dim3, e.g,

```
dim3  gridDim(4,1,1);   dim3  blockDim(16,1,1);
```

GUDA Function Type Qualifiers

- Declaration Syntax:

```
__global__  void name1(arg1, arg2, …){
        name2(arg1,arg2); // invoke device function
}
__device__  double name2(arg1, arg2, …){
        function body;
}
__host__  float name3(arg1, arg2, …){
        function body;
}
```

Host and device routines only run on CPU, and GPU respectively. Global declares kernel function, run on GPU, which can call device functions.

GUDA Kernel Function (again)

- What should be in the Kernel function?

```
for (i = 0; i < 1000; i++)
        C[i] = A[i] + B[i];
 }
```

- ```
  __global__  kernel (int* A, int* B, int* C) {
          id = threadIdx.x + blockIdx.x*blockDim.x;
          C[id] = A[id] +B[id];
  }
  ```

In essence, your for loop with for peeled off, but keep the things inside.

The key part is to map your data to threads (array indices).
  ```

GUDA Memory Operations

- How to move data between CPU and GPU?

  cudaMalloc((void**) &A_d, size_t n_bytes);

  cudaMemcpy(ptr_dest, ptr_src, n_bytes, direction);

- Where

  ptr_dest, ptr_src are destination/source pointers;

  direction can be

    cudaMemcpyHostToDevice

    cudaMemcpyDeviceToHost

    cudaMemcpyDeviceToDevice

- How to free Cuda Memory?

  cudaFree(A_d);

GUDA Memory Operations (2)

```
int     nElem   = 1024;
size_t nbytes = nElem*sizeof(float);
float *A_h, *B_h, *C_h;
float *A_d, *B_d, *C_d;
int    i;

A_h  = (float*) malloc(nbytes);
B_h  = (float*) malloc(nbytes);
C_h  = (float*) malloc(nbytes);
init_data(A_h, nElem);
init_data(B_h, nElem);

cudaMalloc( (void **) &A_d, nbytes);
cudaMalloc( (void **) &B_d, nbytes);
cudaMalloc( (void **) &C_d, nbytes);

cudaMemcpy(A_d, A_h, nbytes, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, nbytes, cudaMemcpyHostToDevice);

sum_1D<<<2, 512>>>(A_d, B_d, C_d, nElem);
cudaMemcpy(C_h, C_d, nbytes, cudaMemcpyDeviceToHost);
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
```

GUDA Memory Operations (3)

Example: Sum two 1D arrays assuming 1D block and 1D grid:
Here is the kernel routine sum_1D:

```
__global__ void sum_1D(float* A_d, float* B_d, float* C_d,
                       int size) {

    int tx = threadIdx.x;
    int bx = blockIdx.x;
    int id = tx + bx*blockDim.x;
    if (id < size) {
        C_d[id] = A_d[id] + B_d[id];
    }
    return;
}
```

The main function was already shown in the previous page

How to Compile CUDA Code?

Cuda nvcc compiler (cuda-9.0 at the RCC):

a. Pure C code: a.c

```
$ nvcc  -o a.out  a.c
```

b. Single Cuda code: a.cu

```
$ nvcc   -arch sm_61  -O3  -o a.out  a.cu
```

c. C and Cuda Mixed: a.cu and b.c

```
$ gcc    -o b.o   -c  b.c
$ nvcc  -o a.out   b.o    a.cu
```

Use GPU ON the HPC Cluster (p1)

Step 1: Load the cuda module

$ module load cuda

Step 2: Compile your cuda code

$ nvcc -o a.out a.cu

Step 3: Create a slurm job script

$ vi slurm.sub

Step 4: Submit your job.

$ sbatch slurm.sub

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -J "cuda-job"
#SBATCH -t 4:00:00
#SBATCH -p backfill2
#SBATCH --gres=gpu:1
#SBATCH --mail-type=ALL

echo $SLURM_JOB_NODELIST
module load cuda
srun -n 1 sum1d
```

Use GPU ON the HPC Cluster (p2)

- There is NO partition called gpu_q anymore!
- Both partition “backfill/backfill2” has a few nodes with GPUs
- Not all nodes of the above partitions have GPUs

```
#SBATCH  -p backfill
#SBATCH  --gres=gpu:[1-4]
```

- The clock limit is 4 hours.

```
#SBATCH -t  4:00:00
```

- At least one node in “genacc_q” has GPUs (14 days!)
- There is no GPU installed on the login node

```
srun -p backfill2  -t 30:00  -n 1 --gres=gpu:1 --pty /bin/bash
```

Querying GPU Devices

- How do I get information about the GPU on a node?
- To get number of GPU cards on a node:

```
cudaError_t  cudaGetDeviceCount(int * dev_count);
```

- To get device properties of a device:

```
cudaDeviceProp  devProp
cudaGetDeviceProperties(&devProp, dev_number);
printf("device name:  %s", devProp.name);
```

- See following page for an example

Querying GPU Devices

```
#include <stdio.h>
#include <cuda_runtime.h>

int main(int argc, char ** argv){

  printf("%s running...\n", argv[0]);
  int   devCount;
  cudaGetDeviceCount(&devCount);
  printf("number of devices: %d\n", devCount);
  cudaDeviceProp devProp;
  cudaGetDeviceProperties(&devProp, 0);
  printf("maxThreadsPerBlock = %d\n", devProp.maxThreadsPerBlock);
  printf("max block dimension (%d, %d, %d)\n", devProp.maxThreadsDim[0],
     devProp.maxThreadsDim[1], devProp.maxThreadsDim[2]);
  printf("max grid dimension (%d, %d, %d)\n", devProp.maxGridSize[0],
     devProp.maxGridSize[1], devProp.maxGridSize[2]);

    return 0;
}
```

Monitoring GPU Activities

Do we have a GPU utility similar to the tool top in linux?
Yes. nvidia-smi

```
bchen3@hpc-i36-5:~ $ nvidia-smi
Fri Mar 30 16:16:44 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.81                 Driver Version: 384.81                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 108...  Off  | 00000000:02:00.0 Off |                  N/A |
| 28%   32C    P2    76W / 250W |    239MiB / 11172MiB |     24%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 108...  Off  | 00000000:03:00.0 Off |                  N/A |
| 28%   39C    P2    64W / 250W |    339MiB / 11172MiB |     21%      Default |
+-------------------------------+----------------------+----------------------+
|   2  GeForce GTX 108...  Off  | 00000000:81:00.0 Off |                  N/A |
| 28%   37C    P2    62W / 250W |    281MiB / 11172MiB |      9%      Default |
+-------------------------------+----------------------+----------------------+
|   3  GeForce GTX 108...  Off  | 00000000:82:00.0 Off |                  N/A |
| 28%   29C    P0    56W / 250W |      0MiB / 11172MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     17213      C   ...c13m/execut/charmm_022018a_md5a_hpc_gpu   229MiB |
|    1     26698      C   .../home/ewa11/charmm_032318a_rmsn_hpc_gpu   329MiB |
|    2     20109      C   .../home/ewa11/charmm_032318a_rmsn_hpc_gpu   271MiB |
+-----------------------------------------------------------------------------+
```

Timing My Kernel Code?

We can time a CUDA kernel by building a CPU timer.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <cuda_runtime.h>

double cpuSecond( ) {

    double sec;
    struct timeval tp;
    gettimeofday(&tp, NULL);
    sec = (double) tp.tv_sec + (double) tp.tv_usec*1.e-6;
    return sec;
}
```

Timing My Kernel Code (2)?

Then we can time a kernel call by wrapping it by two cpuSecond() calls.

```
iStart = cpuSecond();
sumMatrixOnGPU<<<grid, block>>>(A_d, B_d, C_d, nx, ny);
error = cudaPeekAtLastError();
if (error != cudaSuccess)  {
   printf("GPU error: %s\n", cudaGetErrorString(error));
   exit(-1);
}
cudaDeviceSynchronize();
iElaps = cpuSecond() - iStart;
printf("GPU Matrix SUM takes %10.4f seconds\n", iElaps);
```

Timing the CUDA kernel sumMatrixOnGpu()

The above code snippet also deals with CUDA errors…

CUDA Error Handling?

Here are 3 functions which help you debugging your CUDA code:

a. cudaError_t cudaGetLastError(void);

b. cudaError_t cudaPeekAtLastError(void);

c. const char* cudaGetErrorString(cudaError_t error);

What do they do?

a. return the last error code, and reset it to cudaSuccess.
b. return the last error code, but do NOT reset it.
c. convert error code to a readable error string.

Synchronizing the Device

You probably have noticed this line in the previous example

```
cudaDeviceSynchronize();
```

a. Why we need this line?
b. What does it do?

Answers:

a. CUDA programming model is asynchronous between CPU and GPU.
b. The cudaDeviceSynchronize() force the CPU to wait for the kernel code to finish before moving on.
c. The CPU timer will fail if CPU does not wait for the kernel.

Thread Synchronization?

How about synchronization of all threads of a Kernel?

a. Threads within a block can be synchronized

__syncthreads(); (see example near the end)

b. Threads of different blocks CAN NOT be synchronized.

They should NOT be (dead lock!).

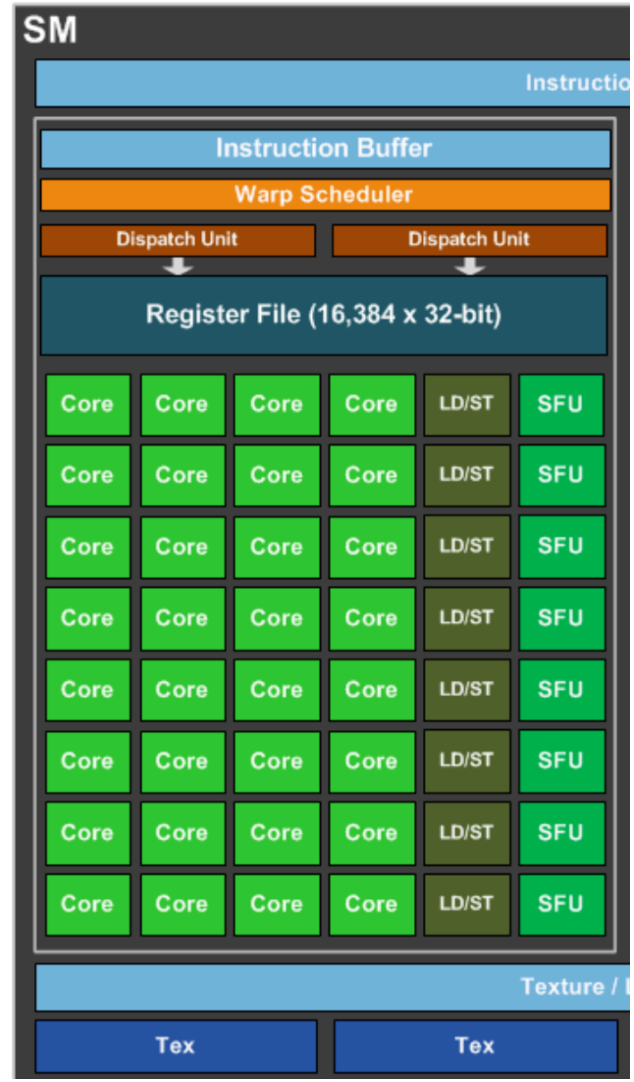
c. Different blocks can be scheduled to start at different time by the GPU.

Thread Organization—Hardware view

- **Software view:**

  grid of blocks,

  blocks of threads
- **Hardware view**?
- **Streaming Multi-Processor (SM) see the right**
- A GTX1080 Ti has 28 SMs.
- Each SM has 128 cores.
- 28*128 = 3584 cores
- A warp = 32 consecutive threads



A Quarter of a Pascal SM

Thread Organization—Hardware view

- A thread block can be assigned to only one steaming multi-processor.
- One multi-processor can have many blocks assigned to it.
- Threads within a block are grouped into warps, each warp has 32 consecutive threads.
- Comment: number of threads in a block should be a multiple of 32 (the warp size).
- Question: How about a block with say, 8 or 16 threads?
- Question: A SM of GTX1080 has 128 cores, why a block can have thousands of threads?

Thread/Warp Divergence

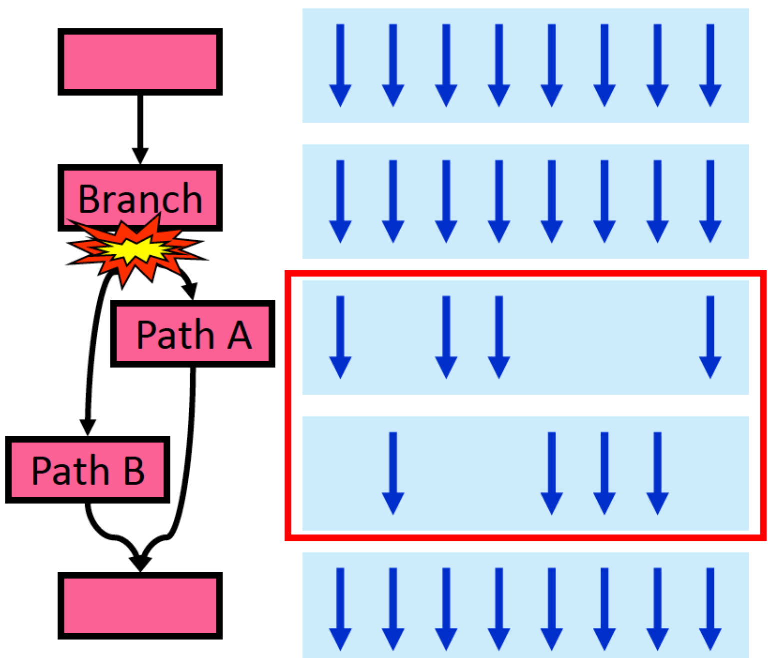
- Threads of the same warp work in the SIMT mode
- SIMT: single instruction multiple threads
- Only one instruction can be executed at one time
- Warp divergence: when threads in the same warp are executing different instructions. For example,

```
id = threadIdx.x;
if ( id < 16 )  {
     printf("I take branch one");
} else {
     printf("I take branch two");
}
```

Performance will be degraded because of warp divergence.

Thread/Warp Divergence

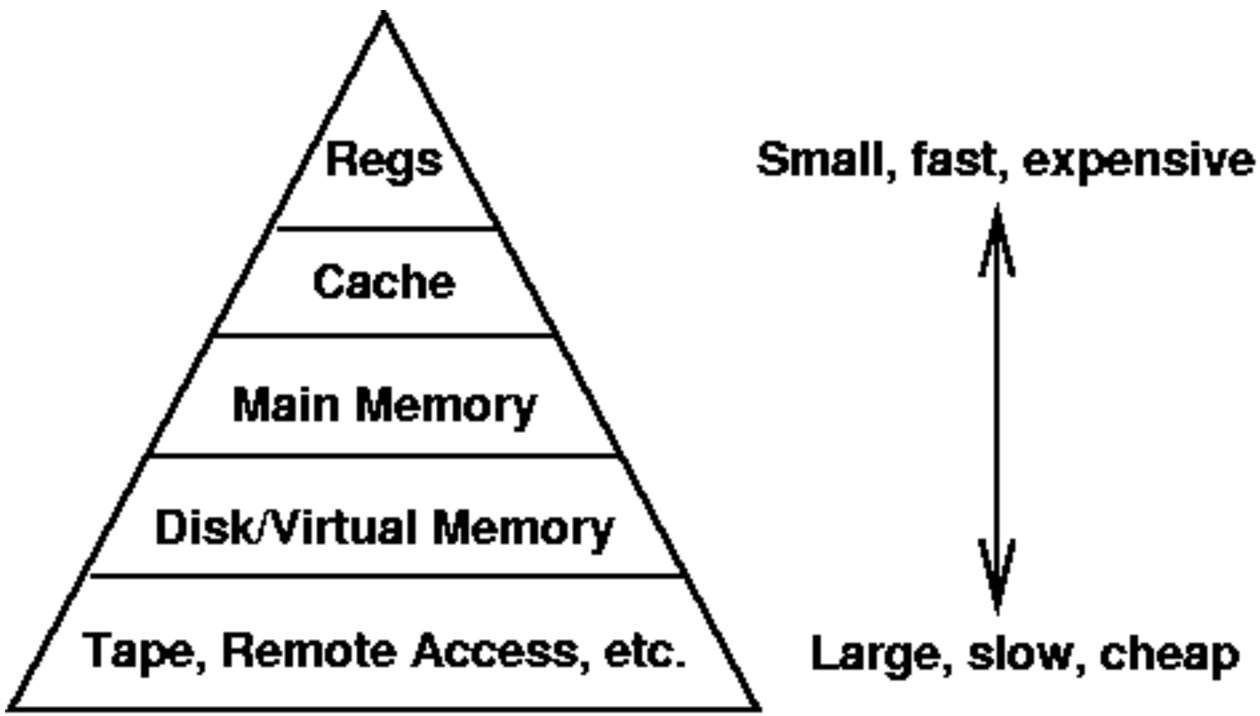
- **Performance will be degraded because of warp divergence.**



50% Performance Loss

CUDA Memory Model

- Similar to thread hierarchy, GPU has a memory hierarchy, and CUDA expose a lot of this hierarchy to you.



CPU Memory Hierarchy

CUDA Memory Model

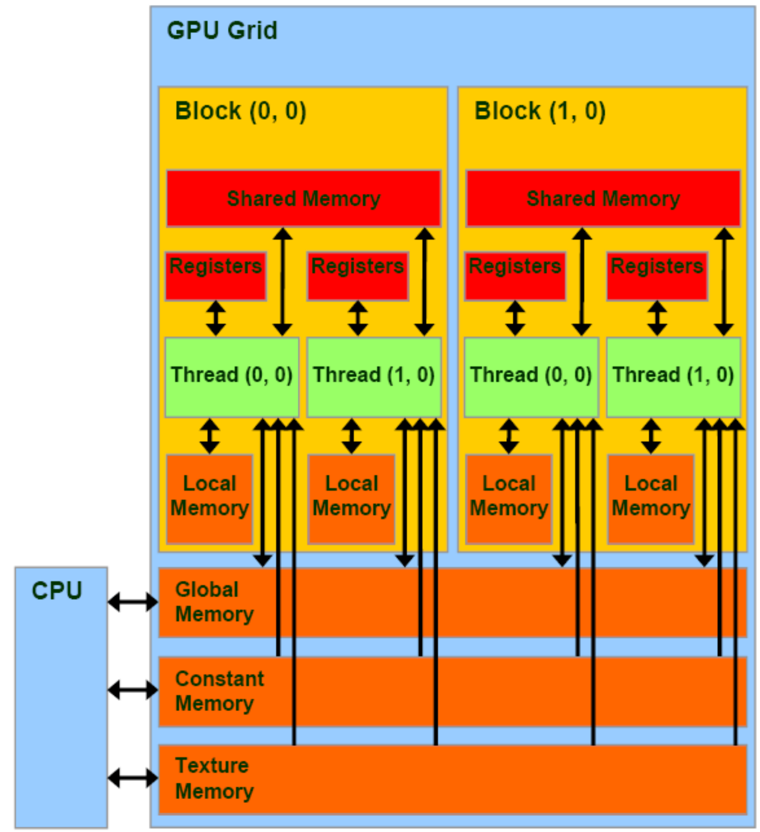
- Registers (threads)
- Shared Memory (block)
- Local Memory (threads)
- Constant Memory (Application)
- Global Memory (Application)
- Texture Memory (Application)

Fermi: 63 registers per thread

Kepler: 255 register per thread

Each SM has a L1 cache

Each device has a L2 cache



GPU Memory Hierarchy

CUDA Memory Model

- Shared Memory + L1 cache = 64KB per SM (precious)
- Each Fermi GPU have 768KB L2 cache (precious)
- Local memory is off chip, and is on the device memory
- Access of local memory is sped up by L1/L2 cache.
- Question: Which one is faster, shared or local memory?
- Answer: shared memory

CUDA Variable Type Qualifiers

- Shared Memory (sit in the SM)

  `__shared__ double a`

- Global Memory (sit in Device memory)

  `__device__ double a`

- Constant Memory (sit in Device memory)

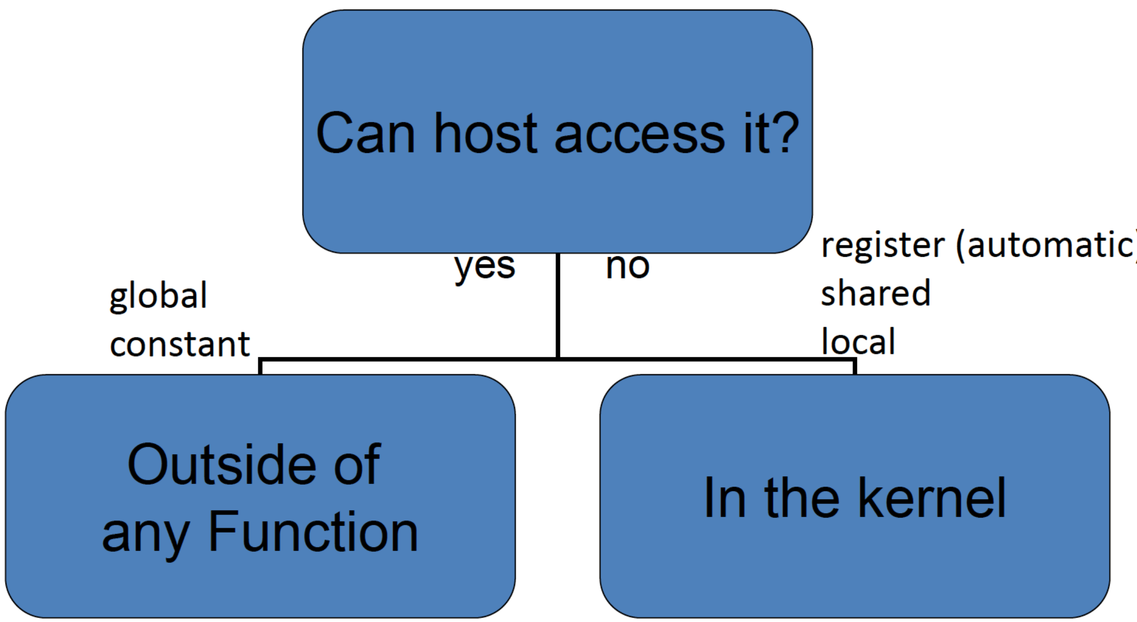
  `__constant__ double a`

- Registers (automatic)
- Local variables (automatic)

CUDA Variable **Scope**

Where to declare these many different types of variables?

a. __global__ and __constant__ outside of any function
b. registers/local/shared variables are declared in the kernel



CUDA Memory Model

GPU Variable Type Qualifiers:

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__, __shared__, int SharedVar;`	Shared	Block	Kernel
`__device__, int GlobalVar;`	Global	Grid	Application
`__device__, __constant__, int ConstVar;`	Constant	Grid	Application

Remark: Local memory does not physically exist, it is put in the global memory by the compiler.

Local/Shared/Registers Example

A GPU local variable example (localVariable.cu):

```
__global__ void kernel() {
    double  a = 2.71828;     //register variables, automatic
    double  c[100];          //local variable, automatic
    __shared__ double b;     //shared variable
    int  tx  = threadIdx.x; //register variable
    if (tx == 0) {
        b = 3.1415926f;
    }
    __syncthreads();         // run with/without this line
    printf("id = %d, a=%7.5f, b=%9.7f\n", tx, a, b);
}

int main() {
    kernel<<<1,8>>>();
    cudaDeviceReset();
    return 0;
}
```

To do: compile and run with/without __syncthreads() line.

Global Variable Example

A GPU global variable example (globleVariable.cu):

```
#include <stdio.h>
#include <cuda_runtime.h>

__device__ float devData;

__global__ void checkGlobal() {
    printf("Device: devData = %f\n", devData);
    devData *= 2;
}

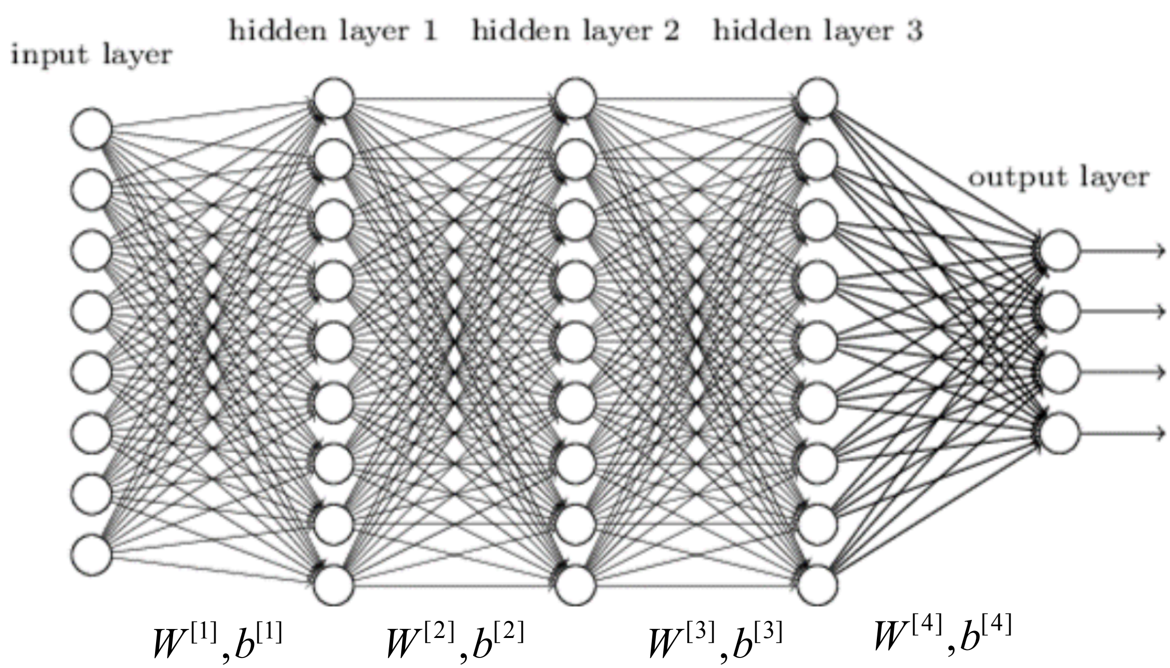
int main() {
    float value = 3.1415926f;
    cudaMemcpyToSymbol(devData, &value, sizeof(float));
    checkGlobal<<<1,1>>>();
    cudaMemcpyFromSymbol(&value, devData, sizeof(float));
    cudaDeviceReset();
    printf("CPU: now the value is %f\n", value);
    return 0;
}
```

Summary

- Heterogeneous programming model
- Thread hierarchy (blocks, grids; warps)
- Memory hierarchy (not enough details today)
- Racing Conditions/Atomic Operations (not covered)
- Tune CUDA code performance (not covered)

Deep Learning Neural Network

- A neural network with at least 2 hidden layers
- The hidden layers can be very wide (millions of hidden units)
- The width (# of units) varies from layer to layer.

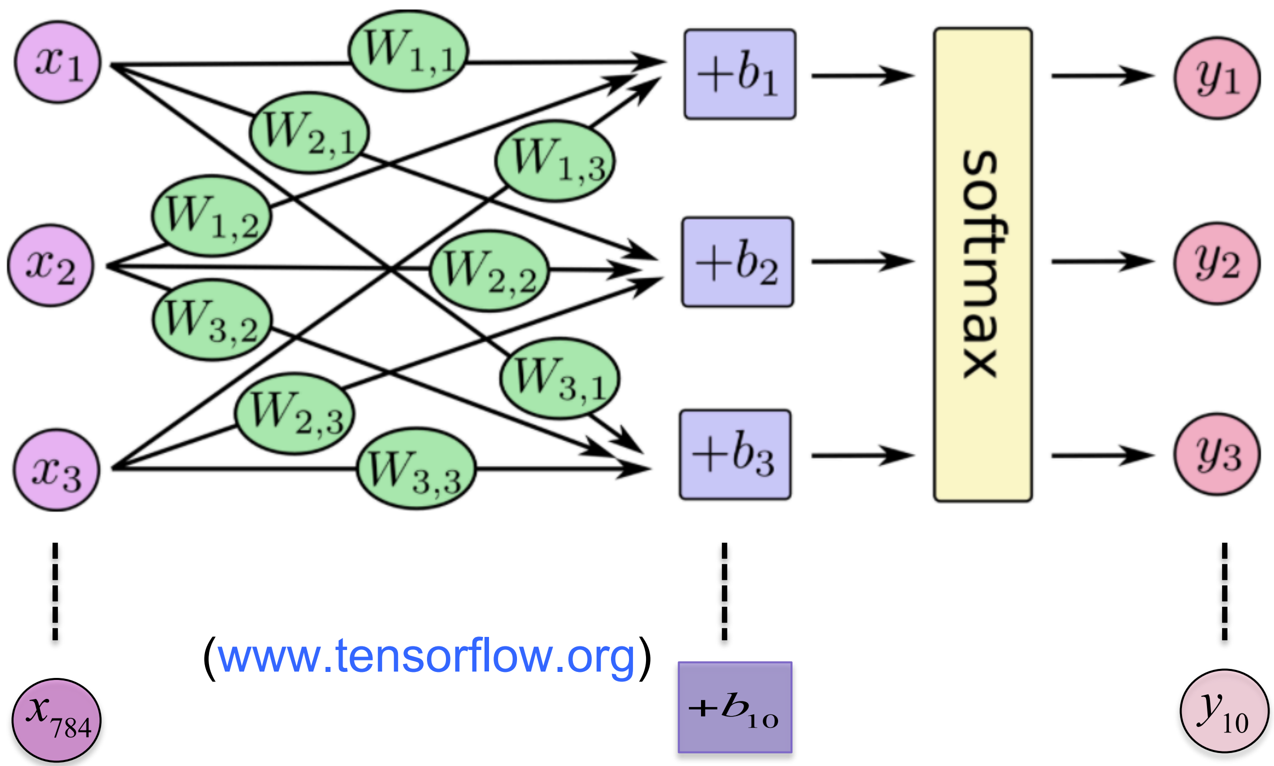


A 4-layer deep neural network

Example: digit recognition

- Model: simple 1-layer neural network.
- Activation function:

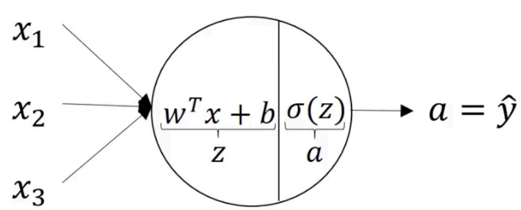
$$\mathrm{softmax}(x)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$



(www.tensorflow.org)

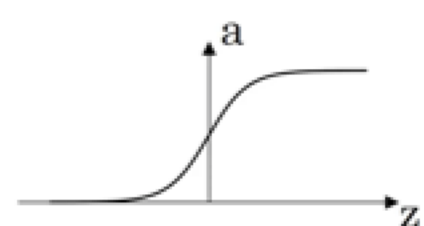
Why IS GPU Ideal for Deep Learning?

- Simple floating point calculation (e.g, matrix operation)
- Special function unit (exponential function)
- A huge amount of brute force calculation
- Cuda library such as cuDNN (libcudnn.so)
- Framework such as Tensorflow (Python/C++), Keras, etc.



$z = w^T x + b$

$a = \sigma(z)$



sigmoid: $a = \dfrac{1}{1 + e^{-z}}$

Why IS GPU Ideal for Deep Learning? (p2)

- NVIDIA Volta GPUs dedicated for deep learning.

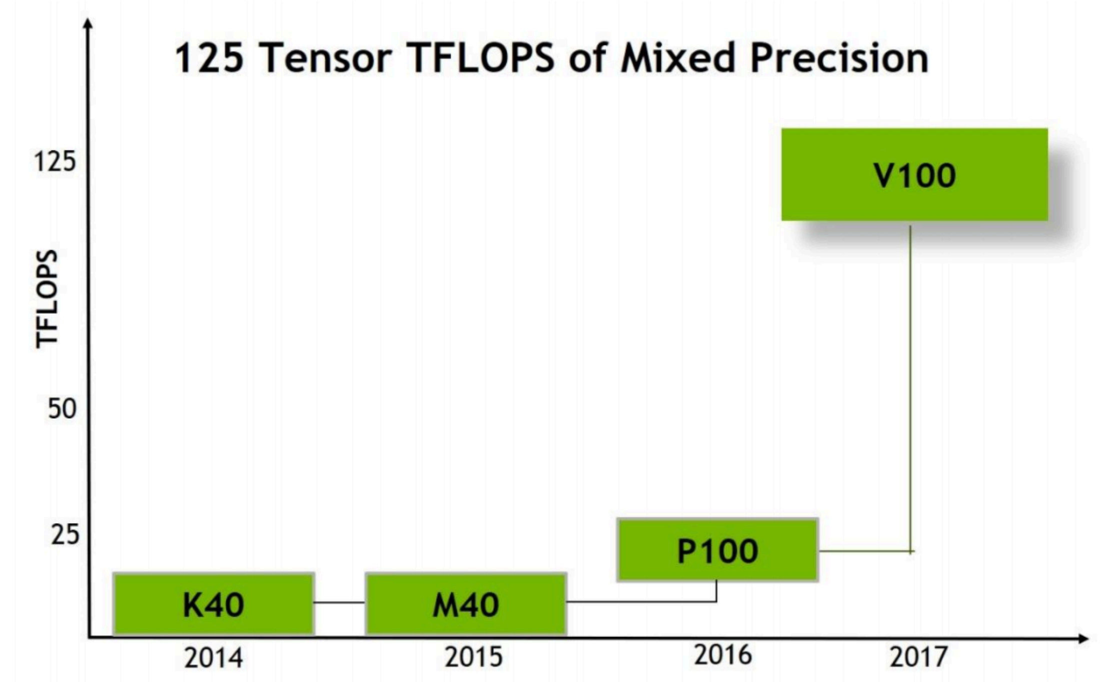


Figure 3. Tesla V100 Provides a Major Leap in Deep Learning Performance with New Tensor Cores

(picture from Volta User Guide)