

Massively Parallel GPU Computing with CUDA: Introduction

Overview of CUDA memory hierarchy
Introduction to CUDA Deep Neural Network library (cuDNN)

Arnis Lektauers, Riga Technical University
arnis.lektauers@rtu.lv

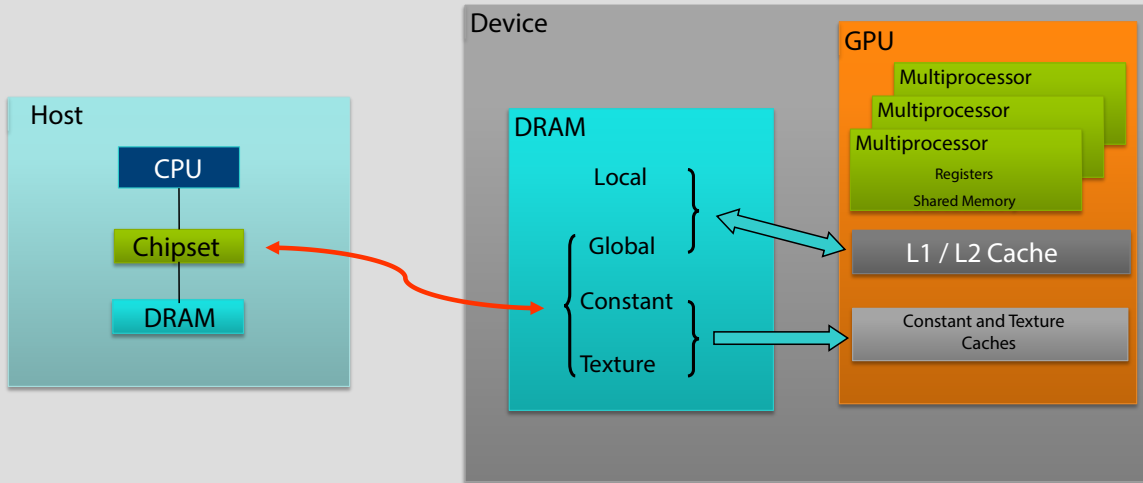
20.01.2022.

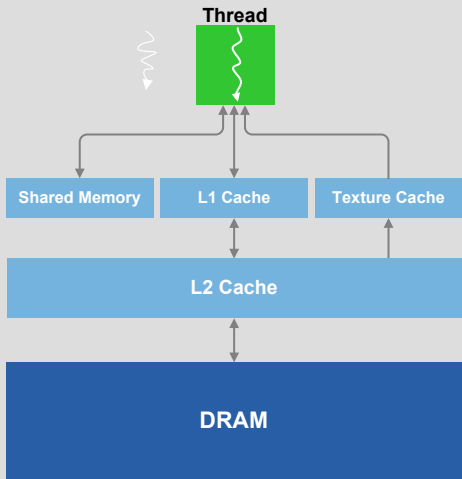
1. Overview of CUDA memory hierarchy:
 - An overview of memory levels
 - Global memory
 - Registers, constant memory, texture memory
 - Shared memory and synchronization
2. Introduction to CUDA Deep Neural Network library (cuDNN):
 - Using cuDNN for deep neural networks
 - Convolutional neural networks in cuDNN
 - Integration with other CUDA libraries (cuBLAS, cuSOLVER, cuRAND, cuTENSOR, TensorRT)
3. Exercises on CUDA techniques: neural network implementation
 - Implementation from scratch with C/C++
 - Implementation from scratch with Python and CuPy
 - Implementation using cuDNN

CUDA Memory Hierarchy



CUDA Memory Architecture





- L1 cache: 64 KB on-chip memory for each SM
- L2 cache:
 - 768 KB (*Fermi* GF100)
 - 1536 KB (*Kepler* GK110)
- Texture (Read-Only Data) cache:
 - 12 KB (*Fermi* GF100)
 - 48 KB (*Kepler* GK110)

CUDA device memory can be allocated and accessed in a variety of ways:

- **Global memory** may be allocated statically or dynamically and accessed via pointers in CUDA kernels, which translate to global load/store instructions
- **Constant memory** is read-only memory accessed via different instructions that cause the read requests to be serviced by a cache hierarchy optimized for broadcast to multiple threads
- **Local memory** contains the stack: local variables that cannot be held in registers, parameters, and return addresses for subroutines

- **Texture memory** is accessed via texture and surface load/store instructions
Like constant memory, read requests from texture memory are serviced by a separate cache that is optimized for readonly access
- **Shared memory** is an important type of memory in CUDA that is not backed by device memory
Instead, it is an abstraction for an on-chip “scratchpad” memory that can be used for fast data interchange between threads within a block

Host memory refers to memory accessible to the CPU(s) in the system

- Host memory is managed with `malloc()/free()` and `new[]/delete[]`
- On all operating systems that run CUDA, host memory is *virtualized*

Pinned memory is a host memory that has been page-locked and mapped for access by the CUDA hardware

- In the context of operating system, the terms *page-locked* and *pinned* are synonymous
- Pinned host memory is allocated by CUDA with the functions `cuMemHostAlloc()` / `cudaHostAlloc()` and freed with `cuMemFreeHost()` / `cudaFreeHost()`

Global memory is the main abstraction by which CUDA kernels read or write device memory

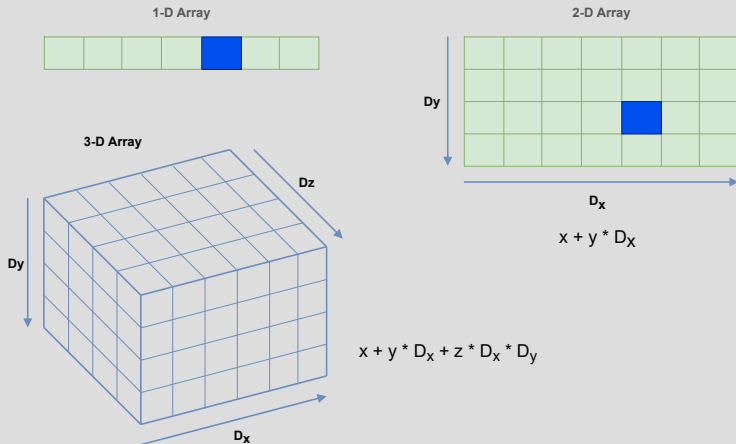
- Global memory is available to all blocks and all threads
- The device pointer base resides in the *Device Address Space*, separate from the CPU address space used by the host code in the CUDA program
- When using the CUDA runtime, device pointers and host pointers both are typed as `void *`
- Most global memory in CUDA is obtained through dynamic allocation using the functions

```
cudaError_t cudaMalloc(void **, size_t);  
cudaError_t cudaFree(void);
```

- It is possible to create *Pitched* memory allocations

Global Memory

Data Indexing



- 1D data array:

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
```

- 2D data array:

```
dim3 blksz (8, 8, 1); // Block size
// Number of blocks required for the whole array.
// Always round this number up
dim3 grdsz ((nx + blksz.x - 1) / blksz.x, (ny + blksz.y - 1) / blksz.y, 1);
my_kernel<<<grdsz, blksz>>>(...);
```

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
int idx = i + j * nx; // global index in the linear array
```

- 3D data array:

```
dim3 blksz (8, 8, 8); // Block size
// Number of blocks
dim3 grdsz ((nx + blksz.x - 1) / blksz.x,
            (ny + blksz.y - 1) / blksz.y,
            (nz + blksz.z - 1) / blksz.z);
```

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
int k = blockIdx.z * blockDim.z + threadIdx.z;
int idx = i + j * nx + k * nx * ny; // global index in the linear array
```

Constant memory is optimized for read-only broadcast to multiple threads

- The compiler for constants has 64K of memory available to use
- Constant memory resides in device memory but is accessed using different instructions that cause the GPU to access it using a special *"Constant Cache"*
- CUDA constants are declared with the `__constant__` keyword
- `__constant__` memory can be changed:
 - By memory copies
 - By querying the pointer to `__constant__` memory and writing to it with a kernel
- CUDA runtime applications can copy to and from `__constant__` memory using functions `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`
- The pointer to `__constant__` memory can be queried with `cudaGetSymbolAddress()`

Local memory contains the stack for every thread in a CUDA kernel

- Supported by the L1 cache
- Configurable size (functions `cudaFuncSetCacheConfig()` or `cudaDeviceSetCacheconfig()`):
 - *Fermi*: 16; 48 KB
 - *Kepler*: 16; 32; 48 KB

It is used as follows:

- To implement the *Application Binary Interface* (ABI) - that is, the CUDA calling convention
- To spill data out of registers
- Local to each thread in the block
- Very fast

The concept of CUDA *Texture Memory* is realized in two parts:

- *CUDA Array* that contains the physical memory allocation
- *Texture Reference* or *Surface Reference*

CUDA Arrays are allocated from the same pool of physical memory as device memory, but they have an opaque layout that is optimized for 2D and 3D locality

Texture References are objects that CUDA uses to set up the texturing hardware to “interpret” the contents of underlying memory

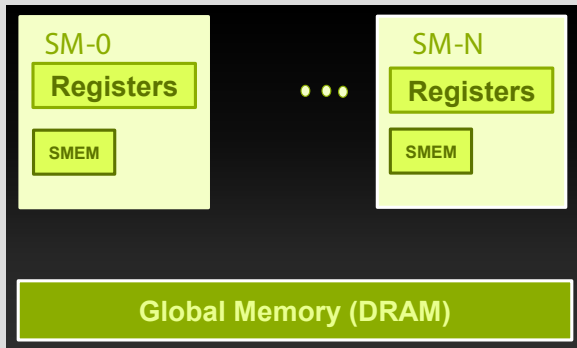
Texture memory features:

- You can read data from here fast!
- But cannot write data directly
- Available to all blocks and all threads

- Texture reference is declared by invoking a template called *texture*
`texture<Type, Dimension, ReadMode> Name`
- The texture reference must be *bound* to underlying memory before it can be used:
 - 1D device memory: `cudaBindTexture()`
 - 2D device memory: `cudaBindTexture2D()`
- Once the texture reference is bound to underlying memory, CUDA kernels may read the memory by invoking:
 - 1D: `tex1D(float x, float y)`
 - 2D: `tex2D(float x, float y)`

Shared memory is used to exchange data between CUDA threads within a block

- Local to the block
- Difficult to use correctly – but very powerful
- Accessible by all threads in a block
- Fast compared to global memory
 - Low access latency
 - High bandwidth (10-150x faster than global memory)
- Common uses:
 - Reducing multiple loads of device data
 - Data layout conversion



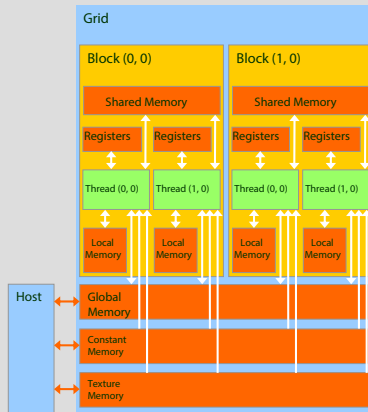
Kernels that use shared memory typically are written in three phases:

1. Load shared memory and `__syncthreads()`
2. Process shared memory and `__syncthreads()`
3. Write results

L1 Cache Sizing:

- Shared memory and L1 use the same 64KB program-configurable split:
 - Fermi: *48:16, 16:48*
 - Kepler: *48:16, 16:48, 32:32*
 - CUDA API for specifying the preferred cache configuration:
`cudaDeviceSetCacheConfig()`, `cudaFuncSetCacheConfig()`
 - Large L1 can improve performance when:
 - Spilling registers (more lines in the cache -> fewer evictions)

- Each thread can:
 - Read/write per-thread registers
 - Read/write per-block shared memory
 - Read/write per-grid global memory
 - Most important, commonly used
- Each thread can also:
 - Read/write per-thread local memory
 - Read only per-grid constant memory
 - Read only per-grid texture memory
 - Used for convenience/performance
- The host can read/write global, constant, and texture memory (stored in DRAM)



CPU Code

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    data = (char *)malloc(N);  
  
    fread(data, 1, N, fp);  
  
    qsort(data, N, 1, compare);  
  
    use_data(data);  
  
    free(data);  
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {  
    char *data;  
    cudaMallocManaged(&data, N);  
  
    fread(data, 1, N, fp);  
  
    qsort<<<...>>>(data, N, 1, compare);  
    cudaDeviceSynchronize();  
  
    use_data(data);  
  
    cudaFree(data);  
}
```

CUDA Libraries

CUDA Toolkit includes several libraries, for example:

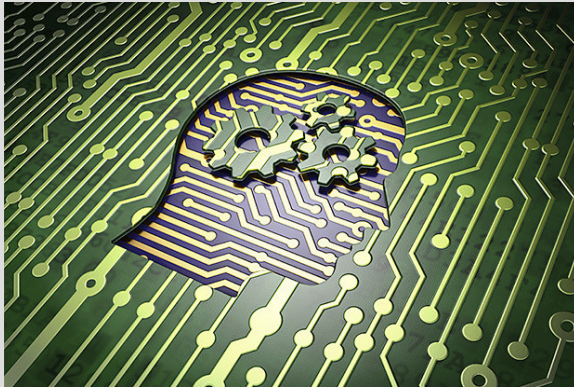
- *CUBLAS* - Complete BLAS Library
- *CUSPARSE* - Sparse Matrix Library
- *CUFFT* - Fast Fourier Transforms Library
- *CURAND* - Random Number Generation (RNG) Library
- *Thrust* - Templated Parallel Algorithms & Data Structures
- *NPP* - Performance Primitives for Image & Video Processing

- **CUBLAS** - implementation of BLAS (*Basic Linear Algebra Subprograms*)
- Self-contained at the API level
- Supports all the BLAS functions
 - Level1 (vector, vector): $O(N)$
 - AXPY : $y = \alpha \cdot x + y$
 - DOT : $\text{dot} = x \cdot y$
 - Level 2 (matrix, vector): $O(N^2)$
 - Vector multiplication by a General Matrix : GEMV
 - Triangular solver : TRSV
 - Level 3 (matrix, matrix): $O(N^3)$
 - General Matrix Multiplication : GEMM
 - Triangular Solver : TRSM
- Following BLAS convention, CUBLAS uses column-major storage

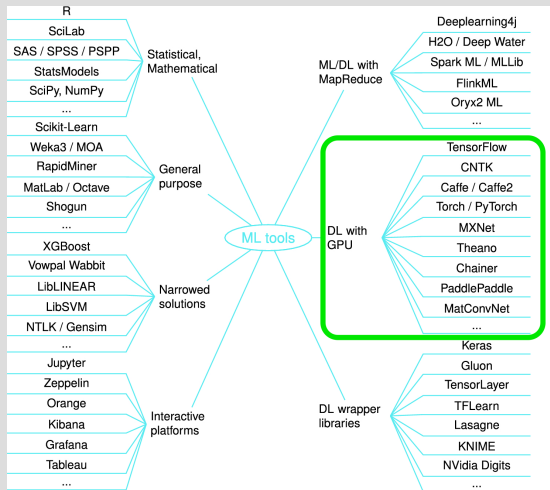
- Support of 4 types :
 - Float, Double, Complex, Double Complex
 - Respective Prefixes : S, D, C, Z
 - Contains 152 routines : S(37), D(37), C(41), Z(41)
 - Function naming convention: cublas + BLAS name
 - Example: cublasSGEMM
 - S: single precision (float)
 - GE: general
 - M: multiplication
 - M: matrix

- Interface to CUBLAS library is in *cublas.h*
- Function naming convention
 - cublas + BLAS name
 - E.g., **cublasSGEMM**
- Error handling
 - CUBLAS core functions do not return error
 - CUBLAS provides function to retrieve last error recorded
 - CUBLAS helper functions do return error
- Helper functions:
 - Memory allocation, data transfer

Introduction to CUDA Deep Neural Network library (cuDNN)



Machine Learning Frameworks and Libraries



- Starting with the Volta architecture, the cuDNN and cuBLAS libraries support GPU **Tensor Cores**
- GPU tensor cores work at hardware level with 4×4 matrices

$$\mathbf{D} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

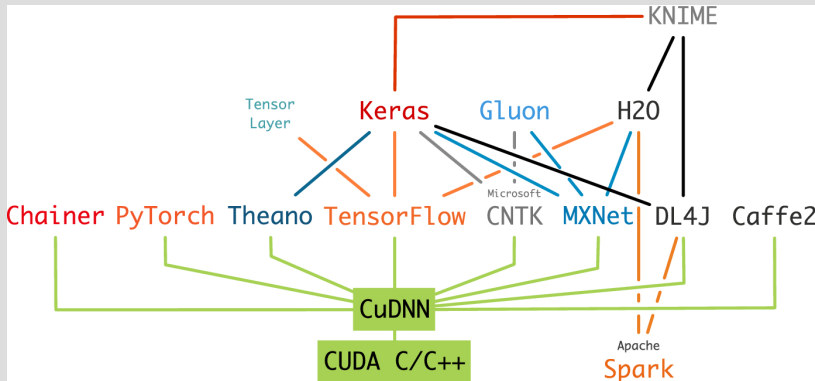
FP16 or FP32 FP16 FP16 FP16 or FP32

- For the programmer, the only operation allowed when using tensor cores is the *Matrix-Multiply-Accumulate* (MMA) operation
- CUDA programming model exposes the MMA operation in terms of dimensions $m \times n \times k$: $D_{m \times k} = A_{m \times n} \times B_{n \times k} + C_{m \times k}$, where $m \times n$, $n \times k$, $m \times k$ cannot exceed 256 elements

- **cuDNN** (*CUDA Deep Neural Network*) - a low-level library that provides optimised GPU implementations of neural network primitives (convolutions, activations, etc.)
<https://developer.nvidia.com/cudnn>
- The main features of cuDNN:
 - Convolution forward and backward, including cross-correlation
 - Matrix multiplication
 - Pooling forward and backward
 - Softmax forward and backward
 - Neuron activations forward and backward: relu, tanh, sigmoid, elu, gelu, softplus, swish
 - Arithmetic, mathematical, relational and logical pointwise operations
 - Tensor transformation functions
 - LRN, LCN and batch normalization forward and backward

cuDNN (1)

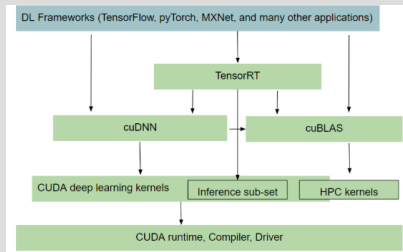
cuDNN is used in the background by most popular high-level neural network libraries, including PyTorch and TensorFlow



TensorRT - C++/Python SDK for high-performance deep learning inference:

<https://developer.nvidia.com/tensorrt>

- Built on the basis of cuDNN and cuBLAS
- Can optimize neural network models trained in all major frameworks, calibrate for lower precision with high accuracy, and deploy to hyperscale data centers, embedded, or automotive product platforms



- The cuDNN library exposes a host API but assumes that for operations using the GPU, the necessary data is directly accessible from the device
- An application using cuDNN must initialize a handle to the library context by calling `cudaCreate(cudaHandle_t *handle)`
- Once the application finishes using cuDNN, it can release the resources associated with the library handle using `cudaDestroy(cudaHandle_t handle)`
- Almost every function we will talk about today returns a `cudaStatus_t` (an enum saying whether a cuDNN call was successful or how it failed)

- The cuDNN library describes data holding images, videos and any other data with contents with a generic n-D tensor defined with the following parameters:
 - Dimension **nbDims** from 3 to 8
 - Data type (32-bit floating-point, 64 bit-floating point, 16-bit floating-point...)
 - **dimA** integer array defining the size of each dimension
 - **strideA** integer array defining the stride of each dimension (for example, the number of elements to add to reach the next element from the same dimension)

- **WXYZ tensor** descriptor: the tensor descriptor format is identified by acronyms, and each letter refers to the corresponding dimension
- **4-D tensor** descriptor: define the format of 4-letter batch 2D images. N , C , H , W represent batch size, number of feature maps, height and width respectively
 - Commonly used 4-D tensor formats:
 - NCHW
 - NHWC
 - CHWN
- **5-D tensor** descriptor: contains 5 letters: N , C , D , H , W represent batch size, number of feature maps, depth, height and width
 - Commonly used 5-dimensional tensor formats:
 - NCDHW
 - NDHWC
 - CDHWN

Consider a batch of images in 4D with the following dimensions: $N = 1$ (batch size); $C = 64$ (number of feature maps (i.e., number of channels)); $H = 5$ (image height); $W = 4$ (image width)

EXAMPLE

$N = 1$

$C = 64$

$H = 5$

$W = 4$

$c = 0$

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

$c = 1$

20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39

$c = 2$

40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59

...

$c = 30$

600	601	602	603
604	605	606	607
608	609	610	611
612	613	614	615
616	617	618	619

$c = 31$

620	621	622	623
624	625	626	627
628	629	630	631
632	633	634	635
636	637	638	639

$c = 32$

640	641	642	643
644	645	646	647
648	649	650	651
652	653	654	655
656	657	658	659

...

$c = 62$

1240	1241	1242	1243
1244	1245	1246	1247
1248	1249	1250	1251
1252	1253	1254	1255
1256	1257	1258	1259

$c = 63$

1260	1261	1262	1263
1264	1265	1266	1267
1268	1269	1270	1271
1272	1273	1274	1275
1276	1277	1278	1279

...

drive. enable. innovate.



The CoE RAISE project has received funding from the European Union's Horizon 2020 – Research and Innovation Framework Programme H2020-INFRAEDI-2019-1 under grant agreement no. 951733

Follow us:

