Introduction to High Performance Machine Learning

Lecture 6 02/29/2024

Dr. Parijat Dube

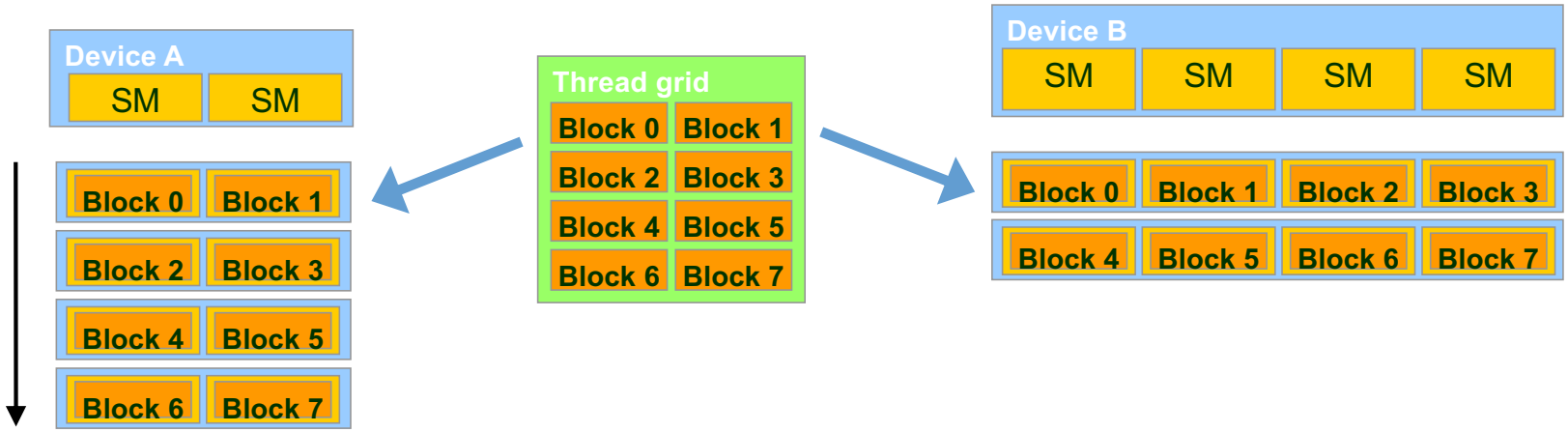
Dr. Kaoutar El Maghraoui

Lesson Key Points

- Heterogenous architectures motivations
- Hierarchy of Computations:
  - Threads
  - Blocks
  - Grids
- Corresponding Memory Spaces
  - Local
  - Shared
  - Global
- Synchronization Primitives
  - Implicit Barriers
  - Thread Synchronization
- NVIDIA GPUs and CUDA:
  - Compute capability
- CUDA Programming Model:
  - Grid, Block, Thread
  - UVM
- CUDA Warp Scheduling
- CUDA Context and Streams
- CUDA Pinned Memory
- CUDA Memory Access
- CUDA Compilation and Runtime:
  - CUDA Runtime, CUDA Driver, AoT and JIT compilation
- CUDA Hardware
- CUDA and PyTorch
- cuDNN and cuBLAS

CUDA Warp Scheduling

CUDA Transparent Scalability



- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales to any number of SMs

CUDA – Grids and Blocks maximum sizes

- Grid, and Block sizes have been the same for all Compute capabilities
- Newer GPUs can have more SMs

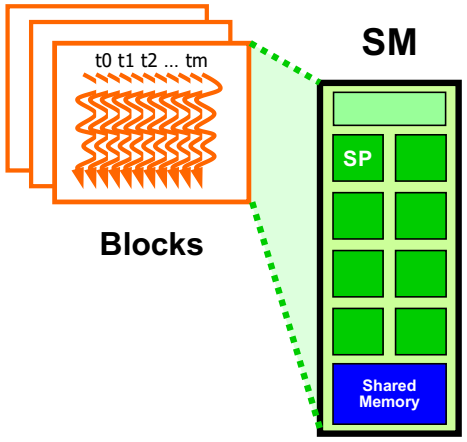
	Compute Capability										
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128
Maximum dimensionality of grid of thread blocks	3										
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$										
Maximum y- or z-dimension of a grid of thread blocks	65535										
Maximum dimensionality of thread block	3										
Maximum x- or y-dimension of a block	1024										
Maximum z-dimension of a block	64										
Maximum number of threads per block	1024										
Warp size	32										

From: NVIDIA

http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

Example: Executing Thread Blocks on Fermi

- Threads are assigned to Streaming Multiprocessors (SM) in block granularity
  - Fermi's compute capability:
    - See: https://en.wikipedia.org/wiki/CUDA
    - Up to **8** blocks to each SM as resources allow
  - Example: A Fermi SM can take up to **1536** threads
    - It could be 256 (threads/block) * 6 blocks
    - Or 512 (threads/block) * 3 blocks, etc.
  - Recent GPUs (Maxwell, Pascal, Volta)
    - Resident blocks per SM: 32 (the maximum number of blocks that can be executed simultaneously on the SM – CUDA hardware limit)
- SM maintains thread/block idx #s
- SM manages/schedules thread execution

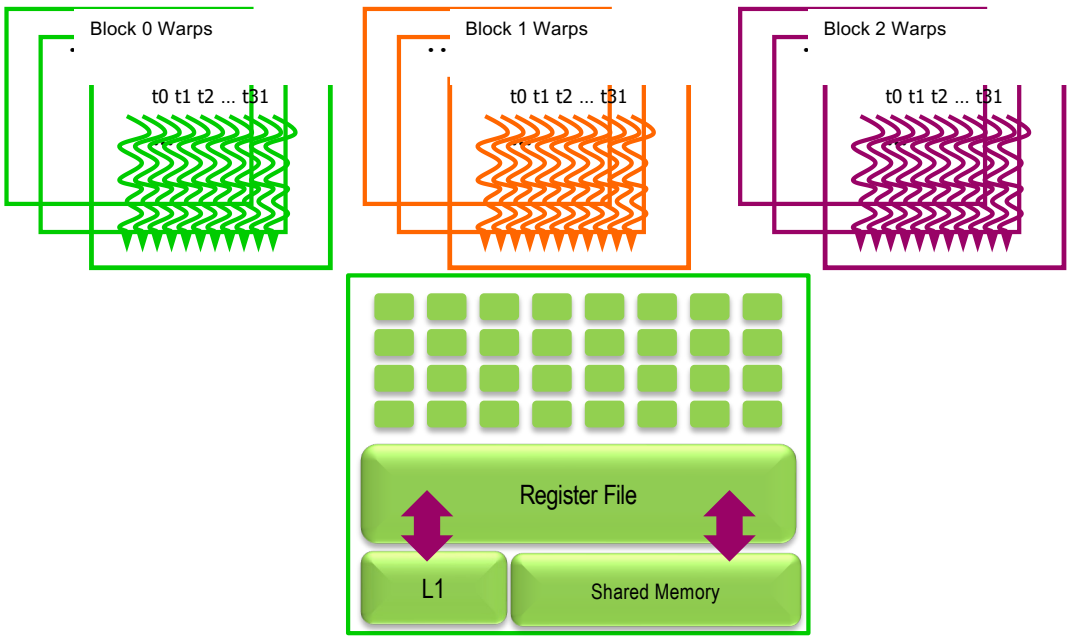


Warps as Scheduling Units

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in SIMD
  - Future GPUs may have different number of threads in each warp

Warp Example

- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
- Each Block is divided into 256/32 = 8 Warps
- There are 8 * 3 = 24 Warps



Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - Warps whose **next instruction has its operands ready** for consumption are eligible for execution
  - Eligible Warps are selected for execution based on an optimized scheduling policy
  - All threads in a warp execute the **same instruction** when selected

SM Occupancy for performance

- Always try to achieve maximum **SM occupancy:**
  - #Active-Warps / #Max-Active-Warps
  - Approach: increase block size until it can fit the SM
  - For Fermi:
    - up to **8** blocks to each SM are allowed
    - Each SM can take up to **1536** threads

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks for Fermi?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, which translates to **24 Blocks** (1536/64), **each SM can only take up to 8 Blocks**, and only 512 **(8*64**) threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks (1536/256) and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM.
- https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

CUDA Context and Streams

CUDA Context

- Each process has a unique context
- Only a single context can be active on a device at a time
- Multiple processes on a single GPU could not operate concurrently
- Best practice would be to create one CUDA context per device
- Device memory allocated in context A cannot be accessed by context B.
- A CUDA context is created when an application calls CUDA API, and it remains active until application releases it.

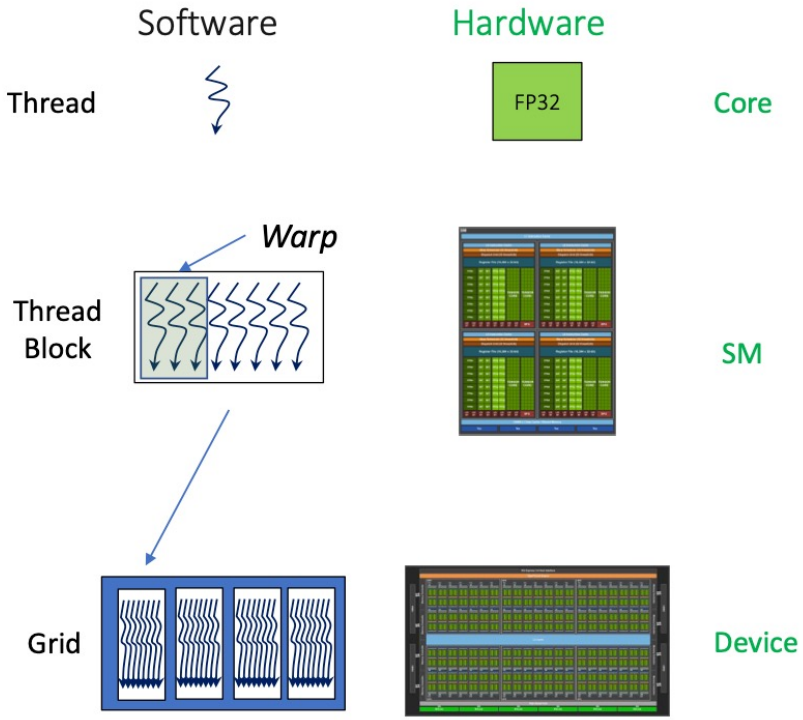
Recap – GPU Execution

We've looked at how code executes on GPUs, lets have a quick recap:

- For each warp, code execution is effectively synchronous within the warp.
- Different warps execute in an arbitrary overlapped fashion – use `__syncthreads()` if necessary to ensure correct behaviour.
- Different thread blocks execute in an arbitrary overlapped fashion.

So nothing new here.

Over the next few slides we will discuss streams – asynchronous execution and the implications for CPU and GPU execution.



Host code – blocking calls

Most CUDA calls are synchronous (often called “blocking”).

An example of a blocking call is `cudaMemcpy()`.

1. Host call starts the copy (HostToDevice / DeviceToHost).
2. Host **waits** until it the copy has finished.
3. Host continues with the next instruction in the host code once the copy has completed.

Why do this???

This mode of operation ensures correct execution.

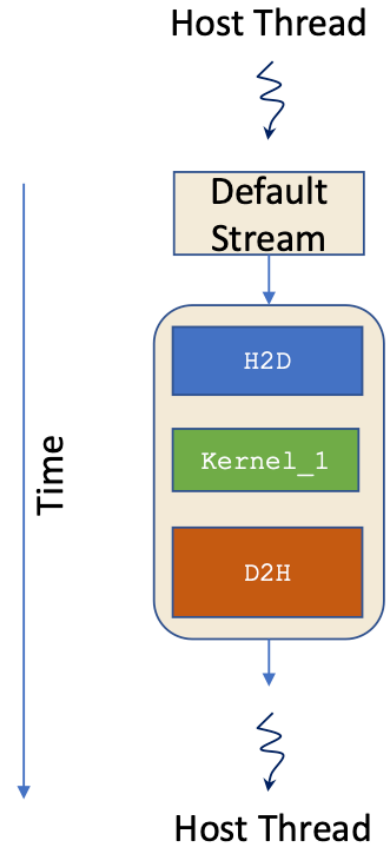
For example it ensures that data is present if the next instruction needs to read from the data that has been copied…

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

Host code – blocking calls

So the control flow for our code looks something like…

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

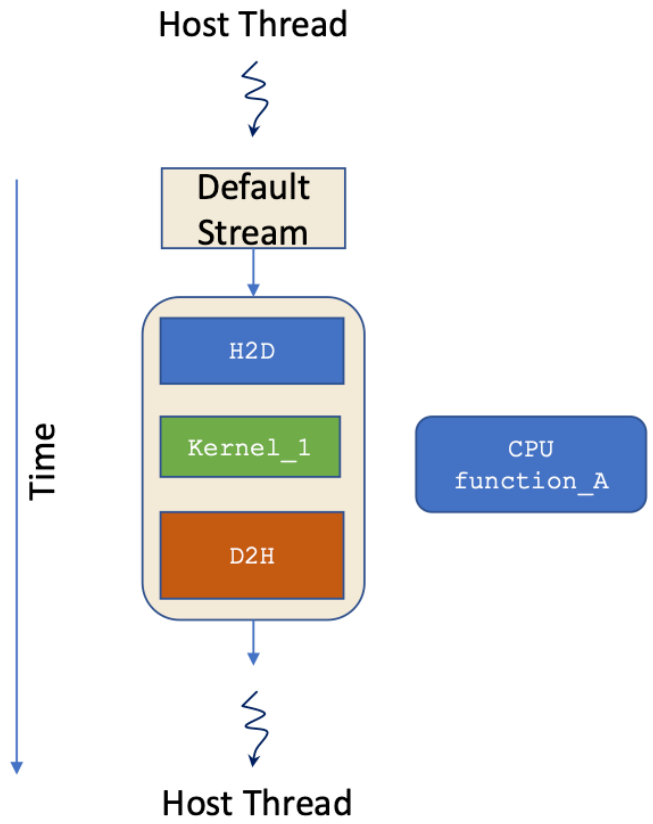


Host code – non-blocking calls

In CUDA, kernel launches are asynchronous (often called “non-blocking”).

An example of kernel execution from host perspective:

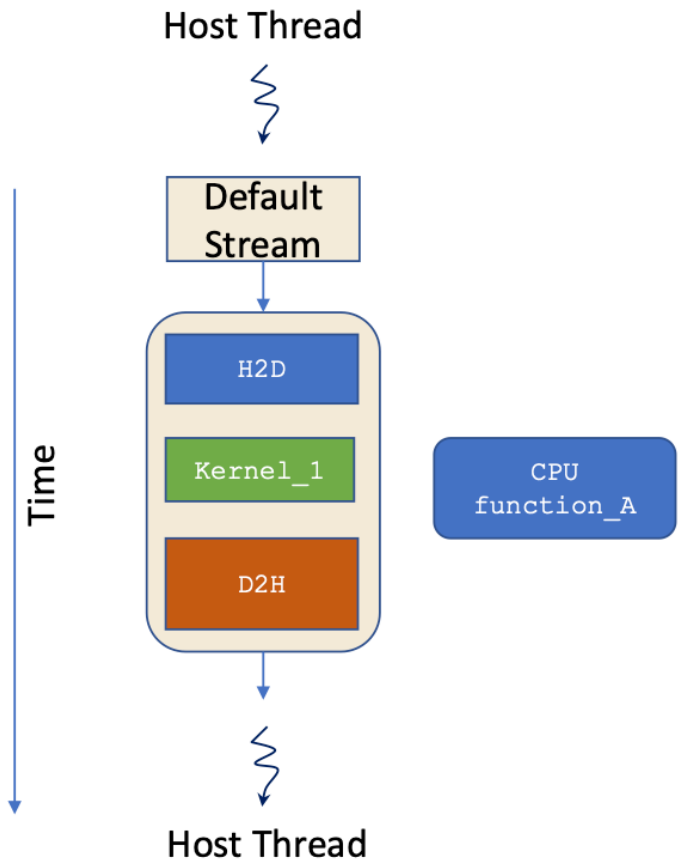
1. Host call starts the kernel execution.
2. Host does not wait for kernel execution to finish.
3. Host moves onto the next instruction.



Host code – non-blocking calls

Example Code

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
CPU_function_A( … );
cudaMemcpy ( …, D2H );

…
```



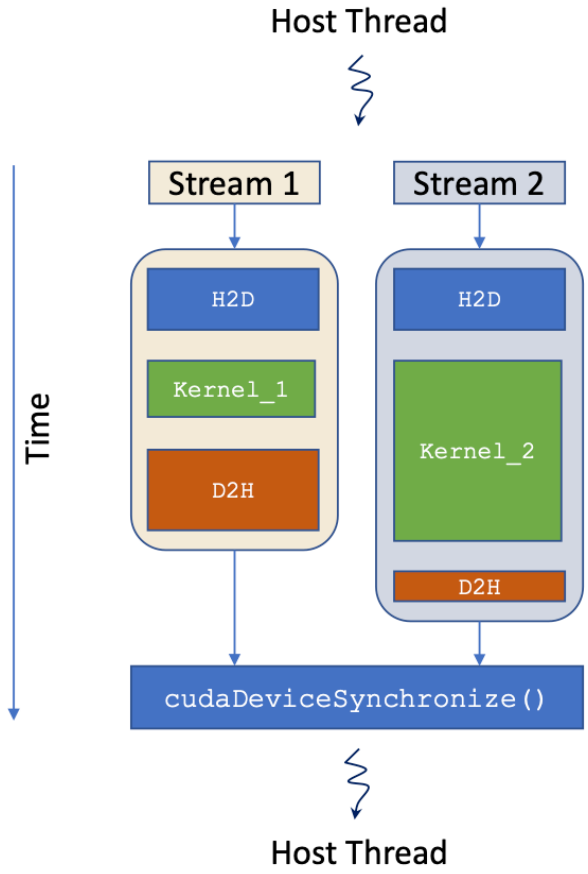
Asynchronous host code

When using asynchronous calls, things to watch out for, and things that can go wrong are:

- Kernel timing – need to make sure it's finished.
- Could be a problem if the host uses data which is read/written directly by kernel, or transferred by `cudaMemcpyAsync()`.
- `cudaDeviceSynchronize()` can be used to ensure correctness (similar to `syncthreads()` for kernel code).

CUDA Streams

- Streams are a feature of the CUDA APIs which allow for concurrent operations within a single GPU context.
- A stream is a queue of device work
  - The host places work in the queue and continues immediately
  - Device schedules work from streams when resources are free
- CUDA operations are placed within a stream
  - e.g., Kernel launches, memory copies
- Operations within the same stream are ordered (FIFO) and cannot overlap
- Operations in different streams are unordered and can overlap
- Streams are valid only in the context in which they were created
- By default, CUDA uses one primary stream, but more can be created.



http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf

Multiple CUDA Streams

When using streams in CUDA, you must supply a "stream" variable as an argument to:

- kernel launch
- `cudaMemcpyAsync()`

Which is created using `cudaStreamCreate();`

As shown over the last couple of slides:

- Operations within the same stream are ordered - (i.e. FIFO – first in, first out) – they cant overlap.
- Operations in different streams are unordered wrt each other and can overlap.

Use multiple streams to increase performance by overlapping memory communication with compute.

```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
my_kernel_one<<<blocks,threads,0,stream1>>>(…);
cudaStreamDestroy(stream1);
```

An example of launching a kernel in a stream that isn't the "default stream".

Kernel_Name<<< GridSize, BlockSize, SMEMSize, Stream >>> (arguments,….);

Stream commands

- Some useful stream commands are:
  - `cudaStreamCreate(&stream)`
  - Creates a stream and returns an opaque “handle” – the “stream variable”.

  - `cudaStreamSynchronize(stream)`
  - Waits until all preceding commands have completed.

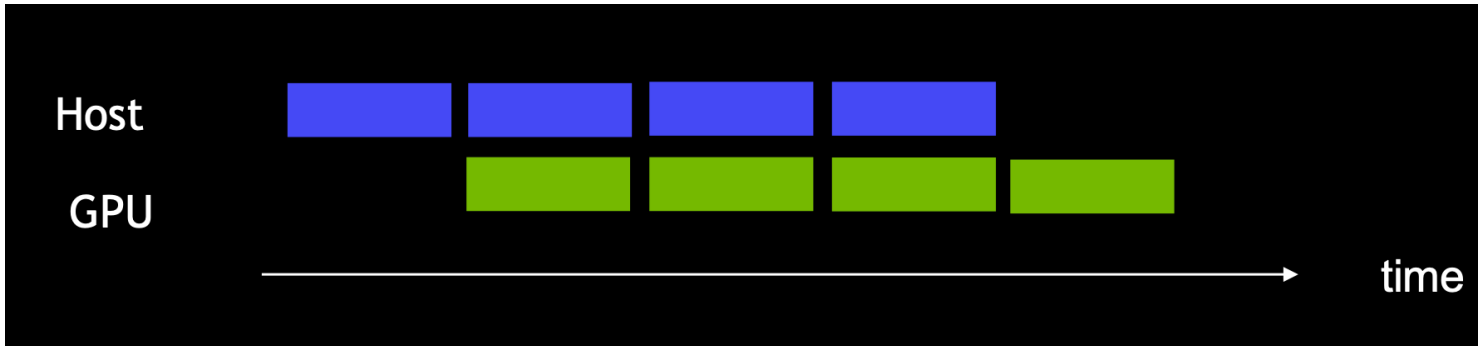
  - `cudaStreamQuery(stream)`
  - Checks whether all preceding commands have completed.

  - `cudaStreamAddCallback()`
  - Adds a callback function to be executed on the host once all preceding commands have completed.

- References:
  - http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf
  - https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

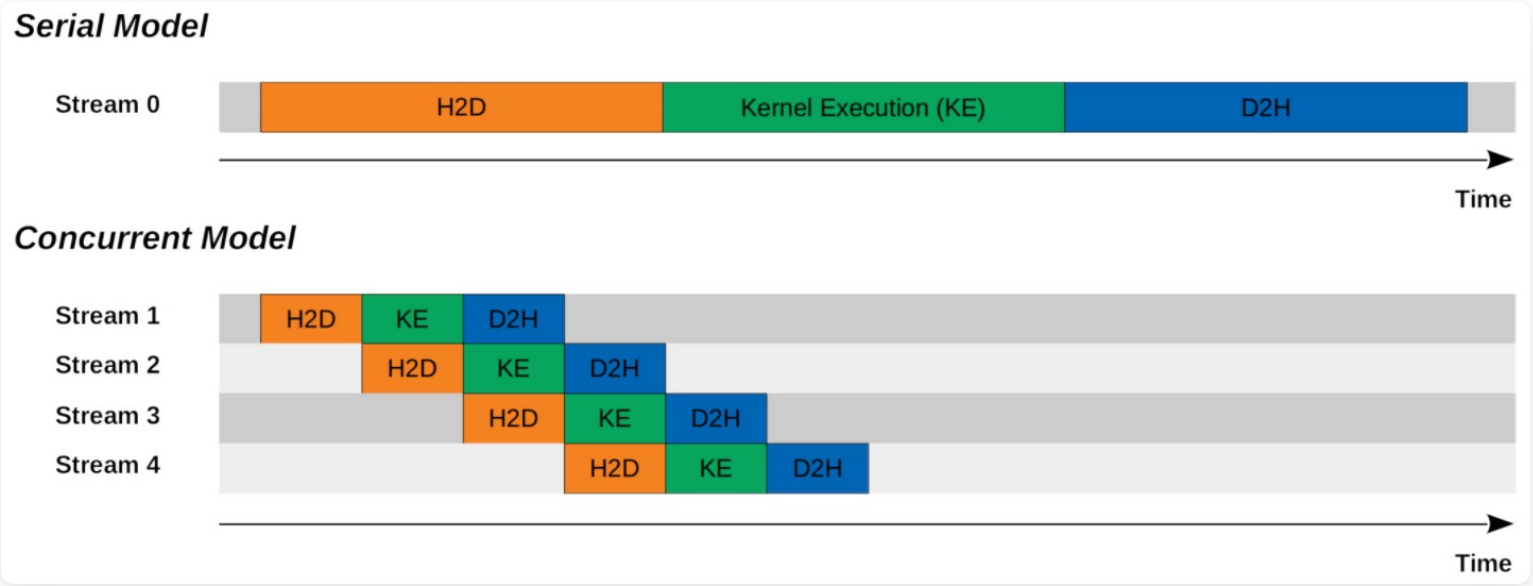
Synchronicity in CUDA

- All CUDA calls are either synchronous or asynchronous w.r.t the host
  - Synchronous: enqueue work and wait for completion
  - Asynchronous: enqueue work and return immediately
- Kernel Launches are asynchronous

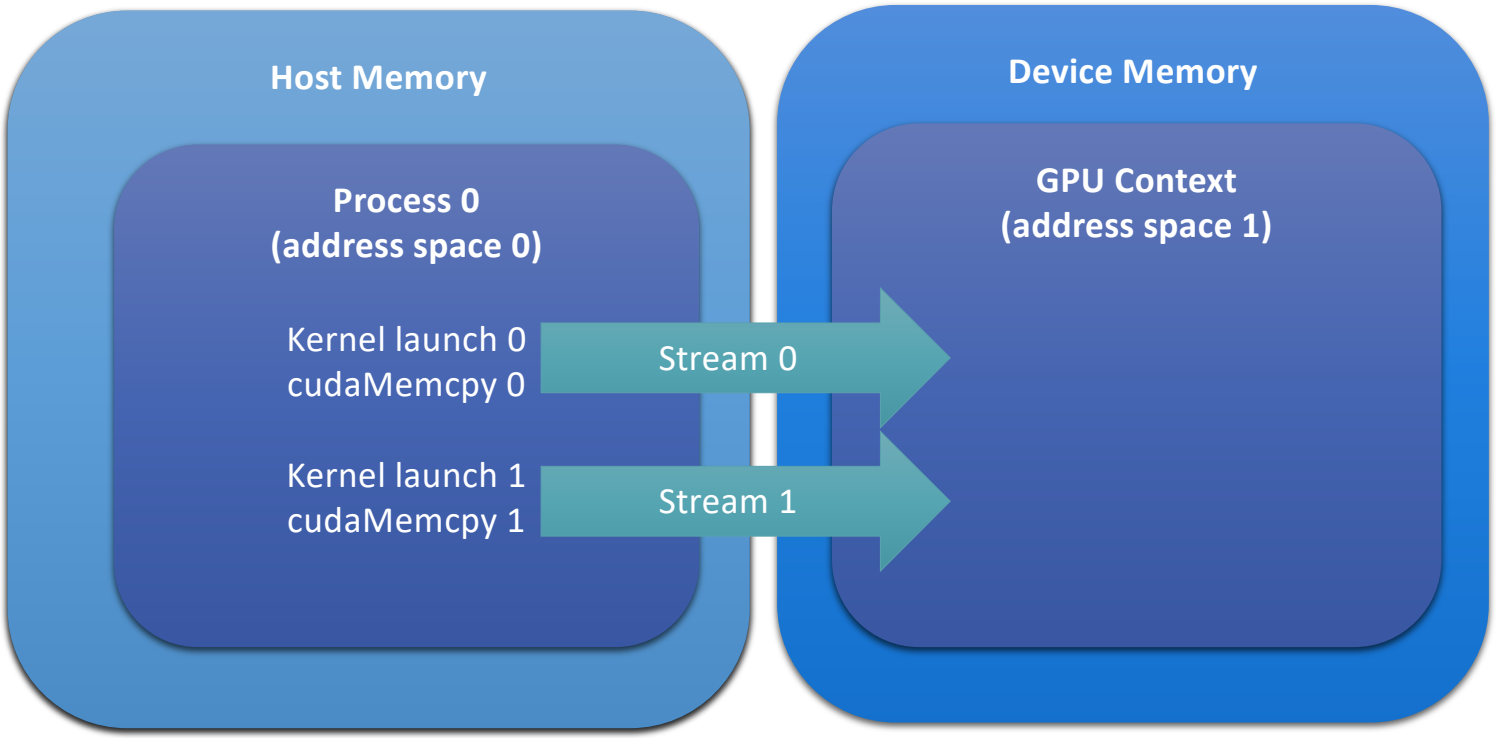


CUDA Streams: Best Practices and Common Pitfalls

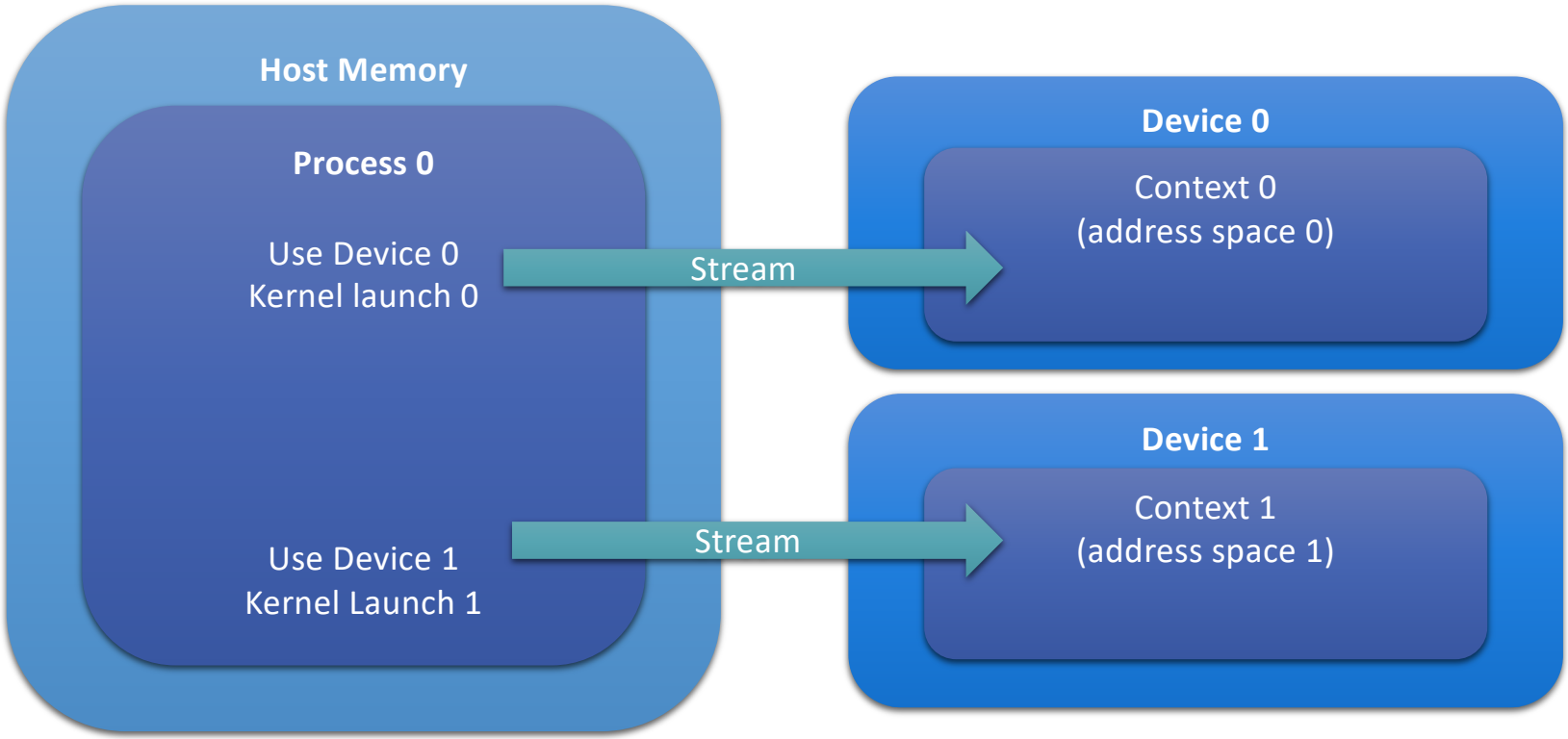
Serial vs concurrent execution with Stream



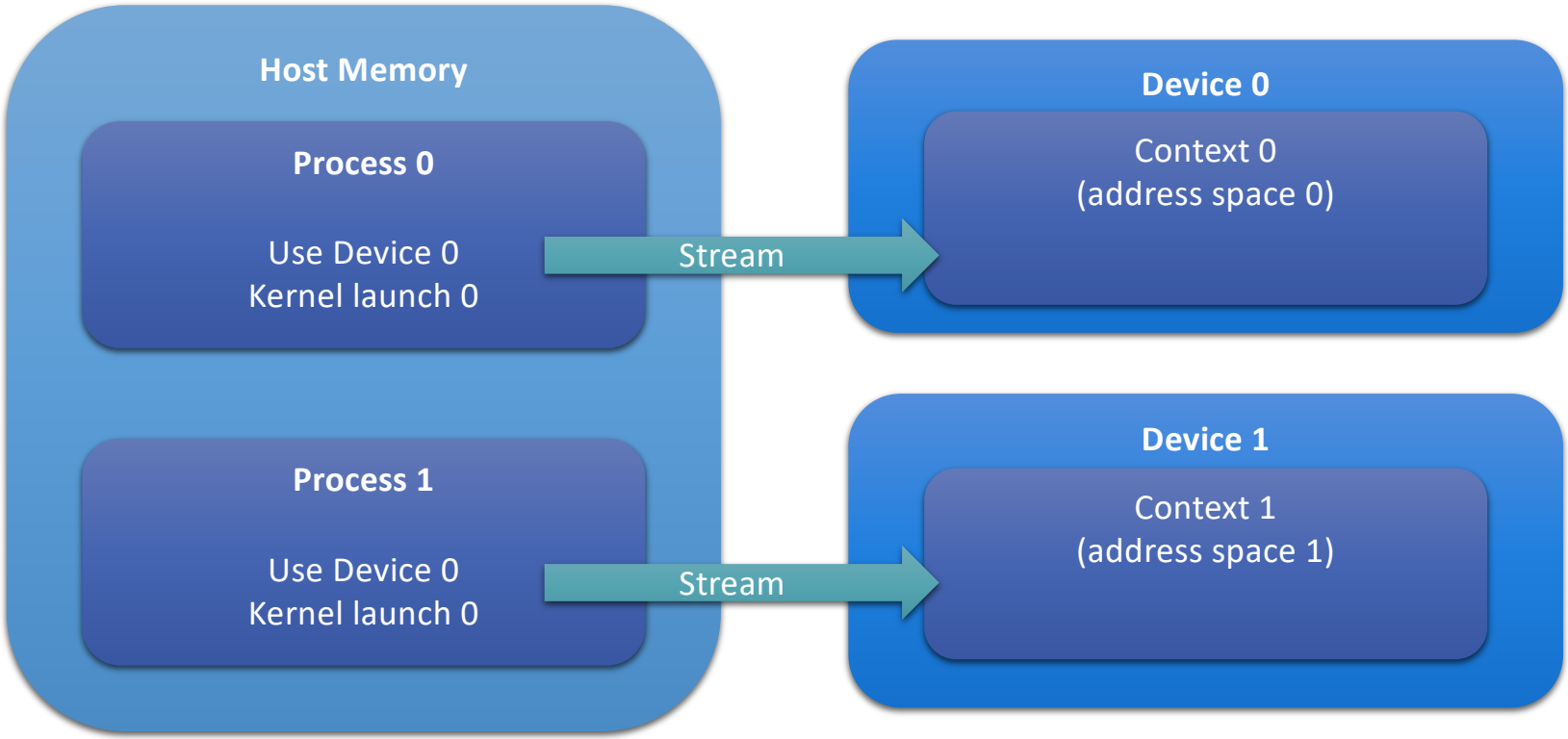
CUDA Context and Streams



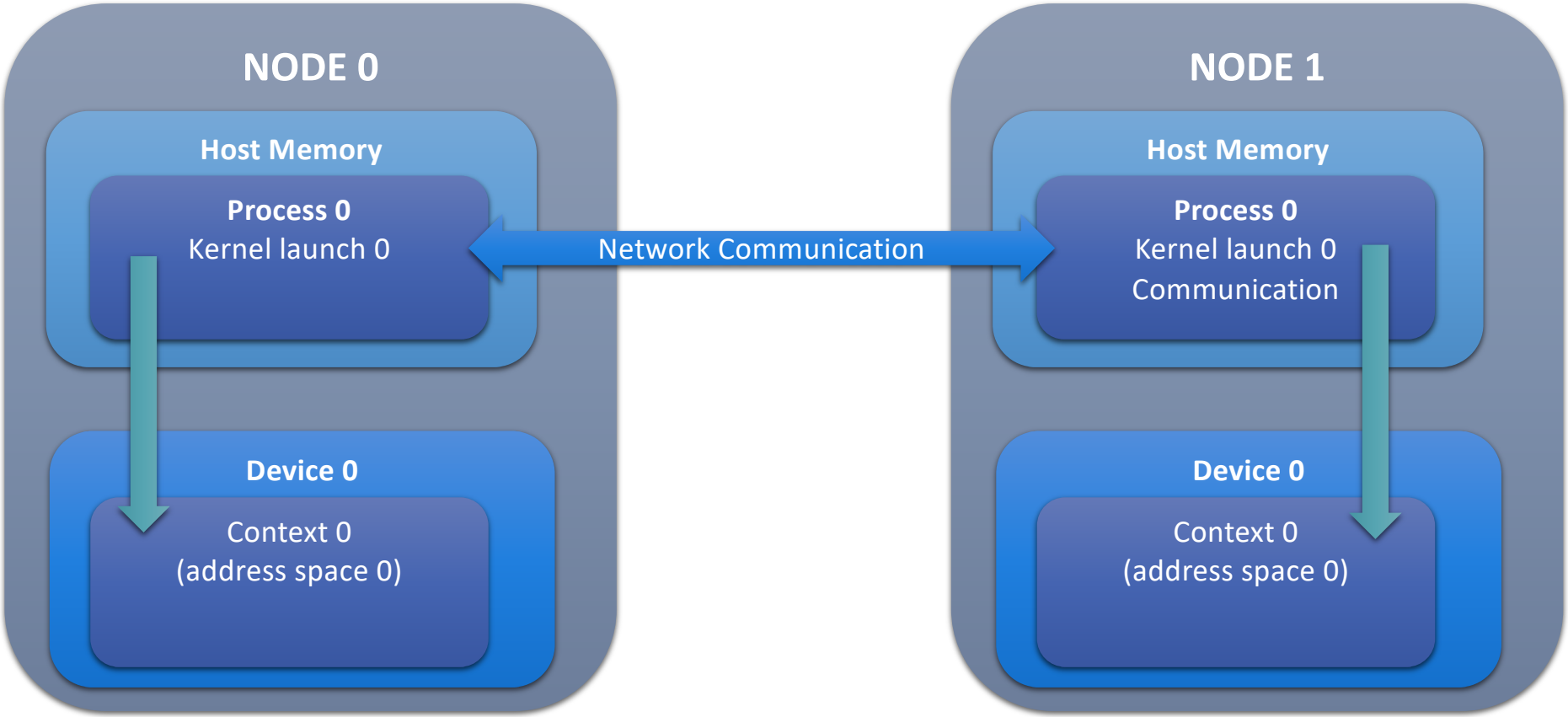
Multi-GPUs with 1 Process



Multi-GPUs with Multiple Processes



Multi-GPU – Multiple processes - Distributed



Timing with CUDA Event API

```
int main ()
{
      cudaEvent_t start, stop;
      float time;

      cudaEventCreate (&start);
      cudaEventCreate (&stop);

      cudaEventRecord (start, 0);

      someKernel <<<grids, blocks, 0, 0>>> (...);

      cudaEventRecord (stop, 0);
      cudaEventSynchronize (stop);

      cudaEventElapsedTime (&time, start, stop);

      cudaEventDestroy (start);
      cudaEventDestroy (stop);

      printf ("Elapsed time %f sec\n", time*.001);

      return 1;
}
```

CUDA Event API Timer are,

- OS independent
- High resolution
- Useful for timing asynchronous calls

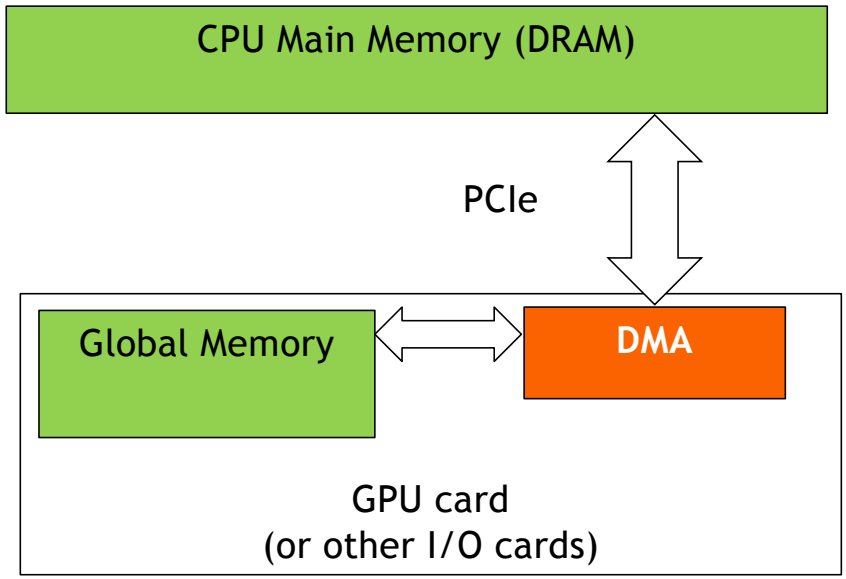
Ensures kernel execution has completed

Standard CPU timers will not measure the timing information of the device.

CUDA Pinned (page-locked) Memory

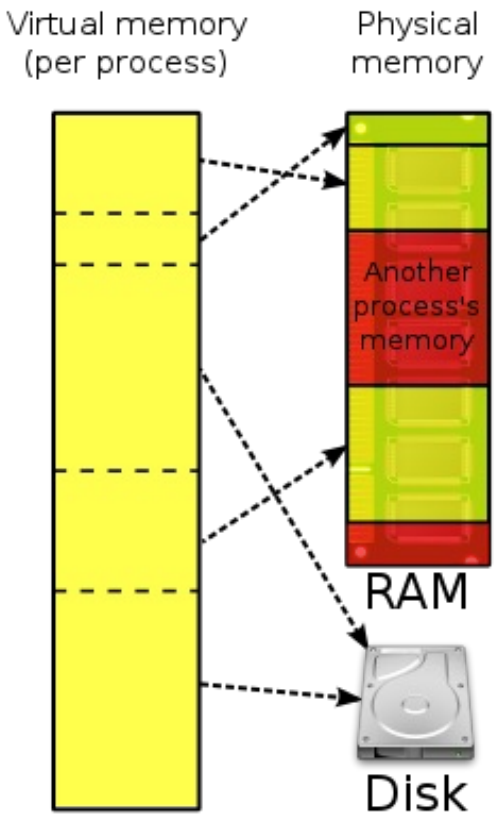
CPU-GPU Data Transfer using DMA

- *cudaMemcpy()* uses DMA (Direct Memory Access) for better performance:
  - Frees CPU for other tasks
  - Hardware unit specialized to transfer a number of bytes requested by OS
  - Between physical memory address space regions (some can be mapped I/O memory locations)
  - Uses system interconnect, typically PCIe in today's systems



Virtual Memory Management

- Virtual memory management:
  - Many virtual memory spaces mapped into a single physical memory
  - Virtual addresses (pointer values) are translated into physical addresses
- Not all variables and data structures are always in the physical memory
  - Each virtual address space is divided into pages that are mapped into and out of the physical memory
  - Virtual memory pages can be mapped out of the physical memory (page-out) to make room
  - Whether or not a variable is in the physical memory is checked at address translation time



Data Transfer and Virtual Memory

- DMA uses physical addresses
  - When `cudaMemcpy()` copies an array, it is implemented as one or more DMA transfers
  - Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
  - No address translation for the rest of the same DMA transfer so that high efficiency can be achieved

- The OS could accidentally page-out the data that is being read or written by a DMA and page-in another virtual page into the same physical location

Pinned (page-locked) Memory and DMA Data Transfer

- **Pinned memory** uses virtual memory pages that are specially marked so that they **cannot be paged out**
- Allocated with a special system API function call
- a.k.a. Page Locked Memory, Locked Pages, etc.
- CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

CUDA data transfer uses pinned memory.

- The DMA used by *cudaMemcpy()* requires that any source or destination in the host memory is allocated as pinned memory
- If a source or destination of a *cudaMemcpy()* in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory – extra overhead
- *cudaMemcpy()* is faster if the host memory source or destination is allocated in pinned memory since no extra copy is needed

Allocate/Free Pinned Memory

- *cudaHostAlloc(),* three parameters
  - Address of pointer to the allocated memory
  - Size of the allocated memory in bytes
  - Option – use cudaHostAllocDefault for now

- cudaFreeHost(), one parameter
  - Pointer to the memory to be freed

Using Pinned Memory in CUDA

- Use the allocated pinned memory and its pointer the same way as those returned by malloc()
- The only difference is that the allocated memory cannot be paged by the OS
- The cudaMemcpy() function should be about 2X faster with pinned memory
- Pinned memory is a limited resource
  - over-subscription can have serious consequences

Pinned memory allocation Example

```
int main()
{
   float *h_A, *h_B, *h_C;
…
   // Allocate pinned memory
   cudaHostAlloc((void **) &h_A, N* sizeof(float), cudaHostAllocDefault);
   cudaHostAlloc((void **) &h_B, N* sizeof(float), cudaHostAllocDefault);
   cudaHostAlloc((void **) &h_C, N* sizeof(float), cudaHostAllocDefault);
…
   // cudaMemcpy() runs faster
}
```

CUDA Memory Access (Part2)

Memory Alignment

Coalescing

Tiling

Recap: Thread Memory Correspondence

Threads ⇔ Local Memory (and Registers)

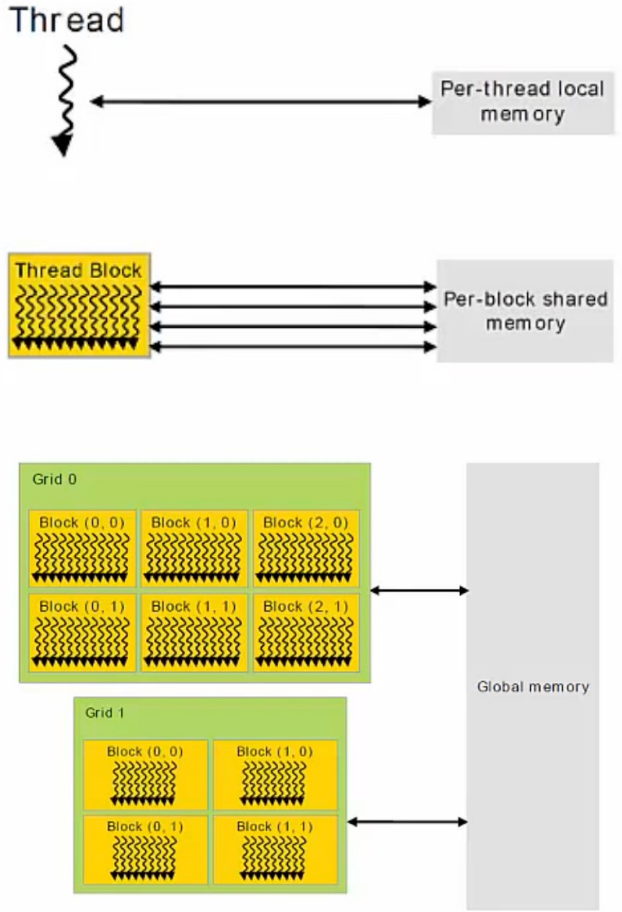
- Scope: Private to its corresponding Thread
- Lifetime: Thread

Blocks ⇔ Shared Memory

- Scope: Every Thread in the Block has access
- Lifetime: Block

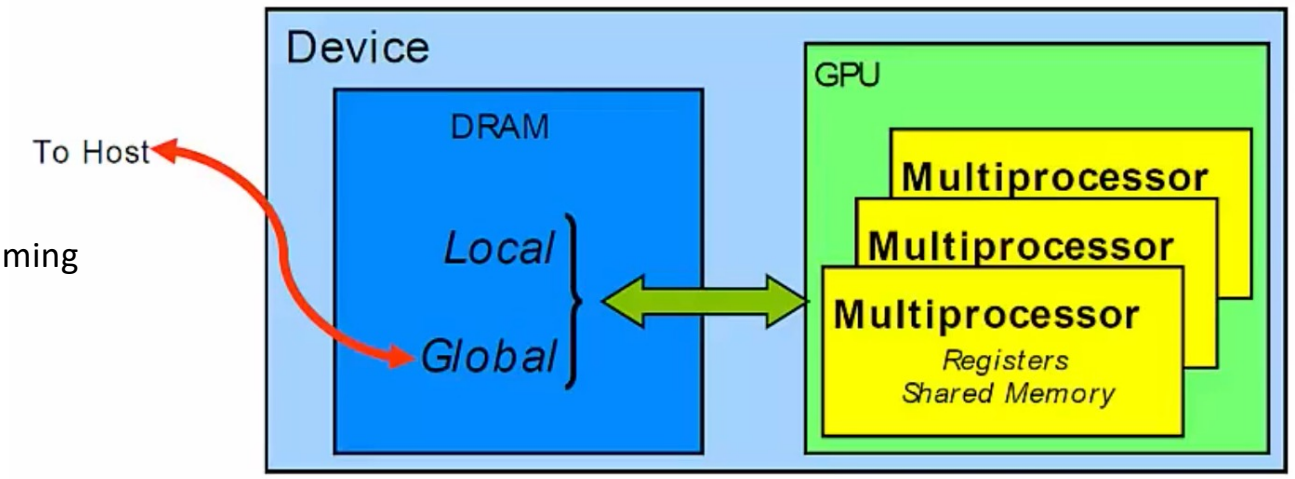
Grids ⇔ Global Memory

- Scope: Every Thread in all Grids have access
- Lifetime: Entire program in Host code – `main()`

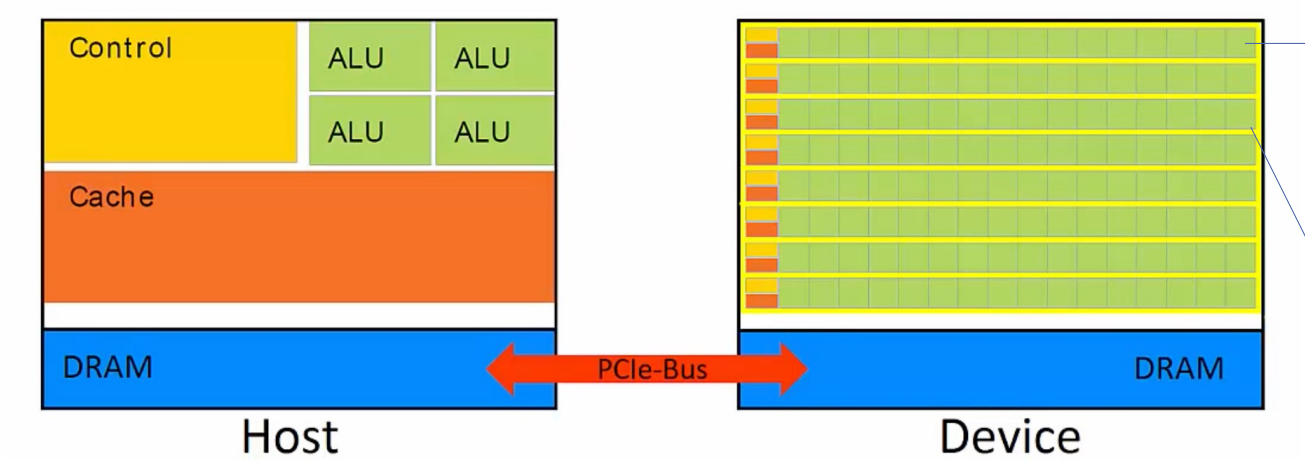


Recap: Memory Model

- Yellow rectangles represent SM: Streaming Multiprocessors
- Memory located in SMs is called:
  - **"On-chip"** Device memory
- Memory not in SM is called
  - **"Off-chip"** Device memory



Term **local** refers to the scope of lifetime and not physical location



CUDA Memory Types

- **Global Memory:**
  - DRAM or HBM
  - Lower Bandwidth than other device memories (but higher than host memory BW)
  - Higher latency than other device memories
- **Shared Memory**
  - **User-managed cache**
  - High Bandwidth
  - Low latency
- **Caches**
  - L1 and L2
  - Texture cache
  - Constant cache

Global memory – Memory Alignment

- The device (GPU) can read **1b, 2b, 4b, 8b, or 16b** aligned words from global memory into registers in a single instruction if the memory access is aligned to the size of the type (ex: char, float, etc.)
- Aligned access – will compile in a single instruction:

```
double array[16];
array[5] = 0.0; // access aligned to its type size (8 bytes)
```

- Misaligned access – will NOT compile in a single instruction:

```
struct {
        char a; // 1 byte
        double b;  // 8 bytes
} misaligned;
misaligned.b = 0.0 // access not aligned to its type size (8 bytes)
```

- See http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses

Global memory – Memory Alignment

- Built-in types like float2 or float4 are aligned automatically:

Data Type	Size	Alignment
float2	8 byte	8
float3	12 byte	4
float4	16 byte	16

- Data structures can be explicitly aligned with the *__align__(x)* keyword:

```
struct __align__(8){
float a;
float b;
};
```

```
struct __align__(16){
float a;
float b;
float c;
};
```

```
struct __align__(16) {
float a;
float b;
float c;
float d;
float e;
};
```

Global memory – Memory Coalescing

- Global memory bandwidth is used most efficiently by simultaneous memory accesses by many threads (32 is a warp)
- Accesses by fewer threads is not enough to tolerate access latencies
- Example of contiguous accesses:
  - 64 bytes accesses: each thread reads a word (2 bytes): int, float, etc.
  - 128 bytes accesses: each thread reads a double-word (4 bytes): int2, float2, etc
  - 256 bytes accesses: each thread reads a quad-word (8 bytes): int4, float4, etc.

Shared Memory

- Shared memory:
  - Shared accessed by **all threads in a block**
  - Faster access than global memory: high bandwidth / low-latency
- Usage:
  - *__shared__* keyword for arrays (pointers)
- Static shared memory:

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Dynamic Shared Memory:

```
extern __shared__ int s[];
int *integerData = s;                        // nI ints
float *floatData = (float*)&integerData[nI]; // nF floats
char *charData = (char*)&floatData[nF];      // nC chars

myKernel<<<gridSize, blockSize,
nI*sizeof(int)+nF*sizeof(float)+nC*sizeof(char)>>>(...);
```

Example

```
#include

__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}

__global__ void dynamicReverse(int *d, int n)
{
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

```
int main(void)
{
  const int n = 64;
  int a[n], r[n], d[n];

  for (int i = 0; i < n; i++) {
    a[i] = i;
    r[i] = n-i-1;
    d[i] = 0;
  }

  int *d_d;
  cudaMalloc(&d_d, n * sizeof(int));

  // run version with static shared memory
  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
  staticReverse<<<1,n>>>(d_d, n);
  cudaMemcpy(d, d_d, n*sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);

  // run dynamic shared memory version
  cudaMemcpy(d_d, a, n*sizeof(int), cudaMemcpyHostToDevice);
  dynamicReverse<<<1,n,n*sizeof(int)>>>(d_d, n);
  cudaMemcpy(d, d_d, n * sizeof(int), cudaMemcpyDeviceToHost);
  for (int i = 0; i < n; i++)
    if (d[i] != r[i]) printf("Error: d[%d]!=r[%d] (%d, %d)n", i, i, d[i], r[i]);
}
```

Static Shared Memory

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

- If the shared memory array size is known at compile time, then we can explicitly declare an array of that size, as we do with the array s.

- In this kernel, t and tr are the two indices representing the original and reverse order, respectively.
- Threads copy the data from global memory to shared memory with the statement s[t] = d[t], and the reversal is done two lines later with the statement d[t] = s[tr].
- need to make sure all threads have completed the loads to shared memory, by calling __syncthreads().

Dynamic Memory

```
__global__ void dynamicReverse(int *d, int n)
{
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

- This example uses dynamically allocated shared memory, which can be used when the amount of shared memory is not known at compile time.

```
dynamicReverse<<<1, n, n*sizeof(int)>>>(d_d, n);
```

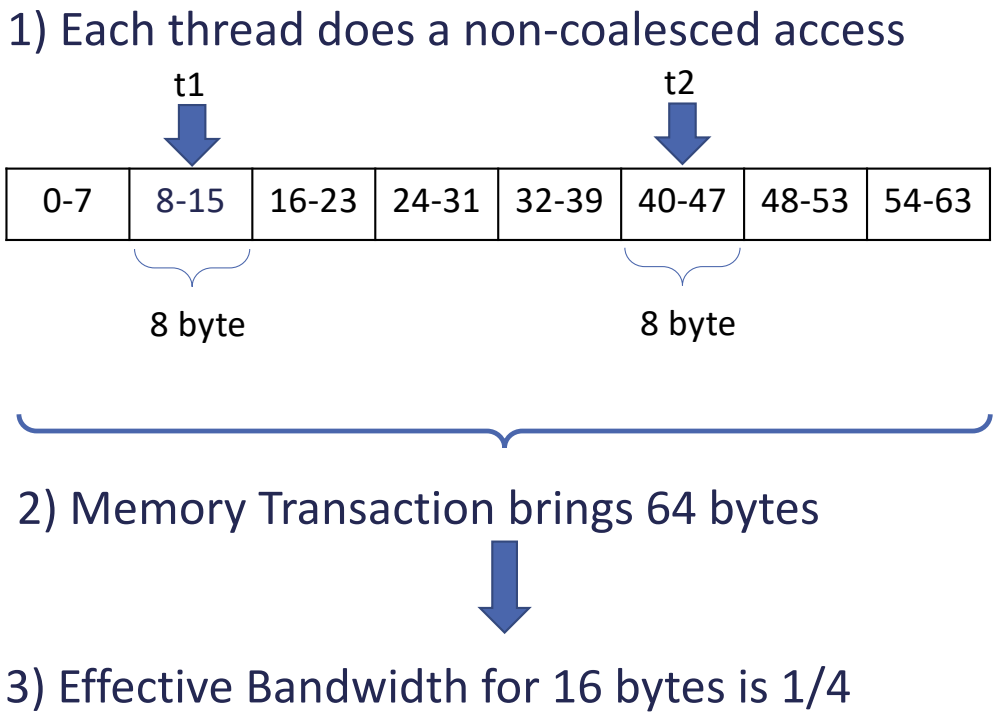
- In this case the shared memory allocation size per thread block must be specified (in bytes) using an optional third execution configuration parameter, as in the following excerpt.
- The dynamic shared memory kernel, dynamicReverse(), declares the shared memory array using an unsized extern array syntax, extern **shared** int s[] (note the empty brackets and use of the extern specifier).
- The size is implicitly determined from the third execution configuration parameter when the kernel is launched. The remainder of the kernel code is identical to the staticReverse() kernel.

Global memory – Memory Coalescing

- Current CUDA devices employ a technique that allows programmers to achieve high global memory access efficiency by organizing thread memory accesses into favorable patterns.
- This technique takes advantage of the fact that threads in a warp execute the same instruction at any given time.
- The most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations.
- When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations. If that's the case, the hardware combines (coalesces) all these accesses into consolidated access to consecutive DRAM locations.
- For example, for a given warp load instruction, if thread 0 accesses global memory location N2, thread 1 location N+1, thread 2 location N+2, and so on, all these accesses will be coalesced into a single request for consecutive locations when accessing the DRAMs.
- Such coalesced access allows the DRAMs to deliver data as a burst.

Global memory – Memory Coalescing

- Accesses to memory only happen for 32b, 64b, 128b transaction granularity
- Accesses must be coalesced for best performance
- **Example:**
  - Two threads (*t1* and *t2*) do non-coalesced accesses for a total of 16 bytes
  - Need to bring in 64 bytes



Unfavorable data access pattern

- The figure illustrates the data access pattern of the M array
- Threads in a warp read adjacent rows
- During iteration 0, threads in a warp read **element 0** of rows 0 through 31.
- During iteration 1, these same threads read **element 1** of rows 0 through 31.
- **None** of the accesses will be coalesced.

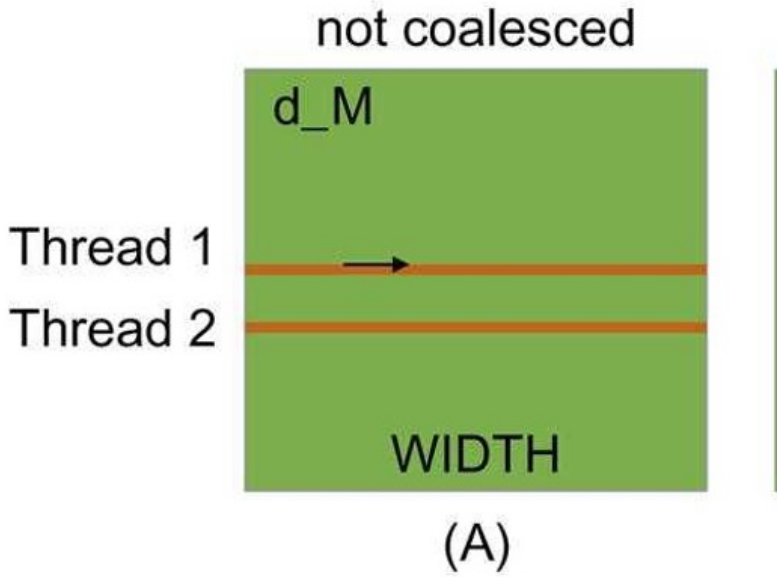
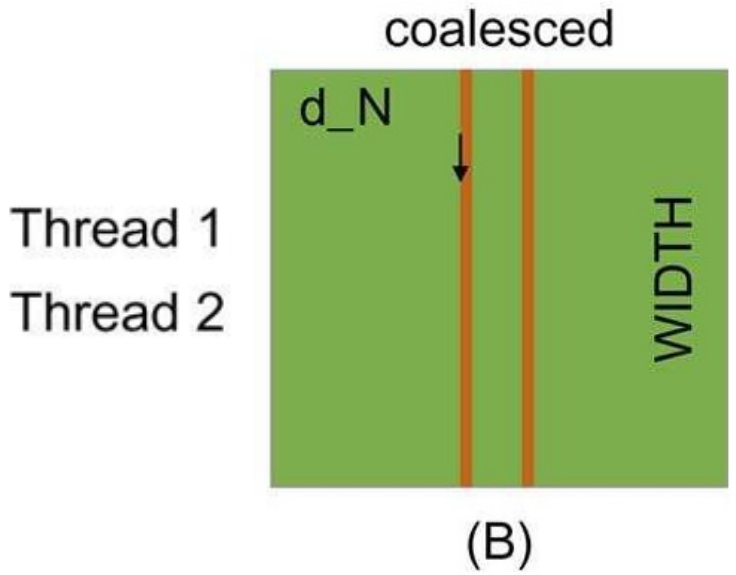


Image source: https://nichijou.co/cuda5-coalesce/

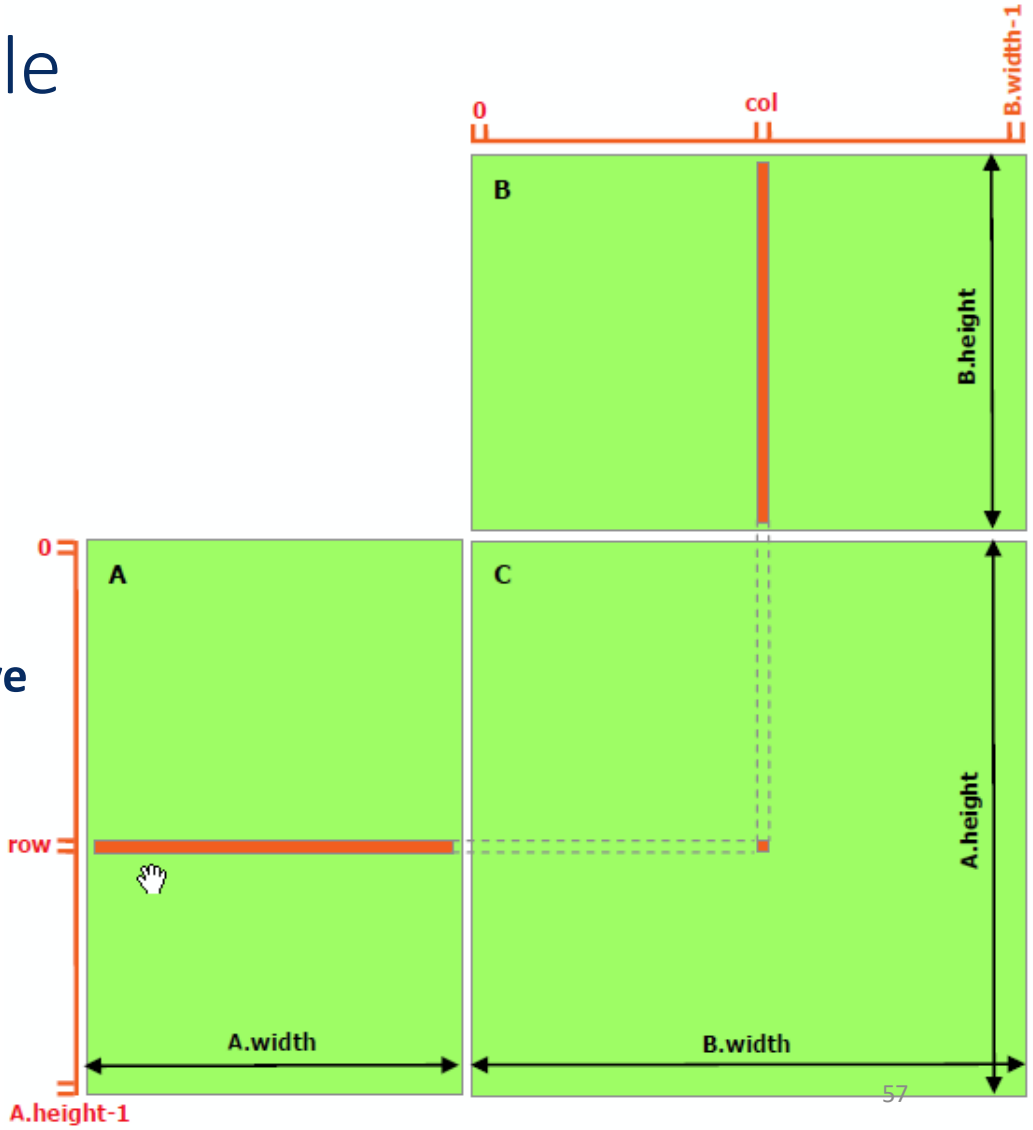
Favorable data access pattern

- The figure illustrates the data access pattern of the N array
- Each thread reads a column of N
- During iteration 0, threads i read **element 1** of columns through 31.
- All these accesses will be coalesced.



Matrix Multiplication Example

- Computing matrix multiplication
  - C = A × B
- Each thread compute one (or more) elements of C:
  - Dot product: *row **dot** column*
- Each thread needs to access global memory for the row and the column
  - Column accesses are in **non-consecutive** addresses



Recap: Dimensions and thread indexing

- ***dim3*** type:
  - an **integer vector type** based on uint3 that is used to specify dimensions. When defining a variable of type dim3, any component left unspecified is initialized to 1

sometimes, 2D maps better

- 1D thread indexing example:

```
dim3 dimBlock(512); //  512 threads in 1D
dim3 dimGrid(1024); // 1024 blocks  in 1D
mykernel<<dimGrid, dimBlock>>();

// Equivalent to
mykernel<<1024,512>>();
```

- 2D thread indexing example:

```
dim3 dimBlock(16,32); //  512 threads in 2D
dim3 dimGrid(32,32);  // 1024 blocks  in 2D
mykernel<<dimGrid, dimBlock>>();
```

Matrix Multiplication Example (1)

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
```

- Matrix in row-major order
- Computing matrix multiplication C = A × B

Matrix Multiplication Example (2)

- Matrix sizes assumed to be multiple of BLOCK_SIZE
- Allocating A,B and C
- Is block size optimal for every GPU?

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    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);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
```

Matrix Multiplication Example (3)

- Using 2D index
- Copy to device
- Call kernel
- Copy from device
- Free memory

```
// Second part of
// void MatMul(const Matrix A, const Matrix B, Matrix C)
// {
//

// Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

Matrix Multiplication Example (4)

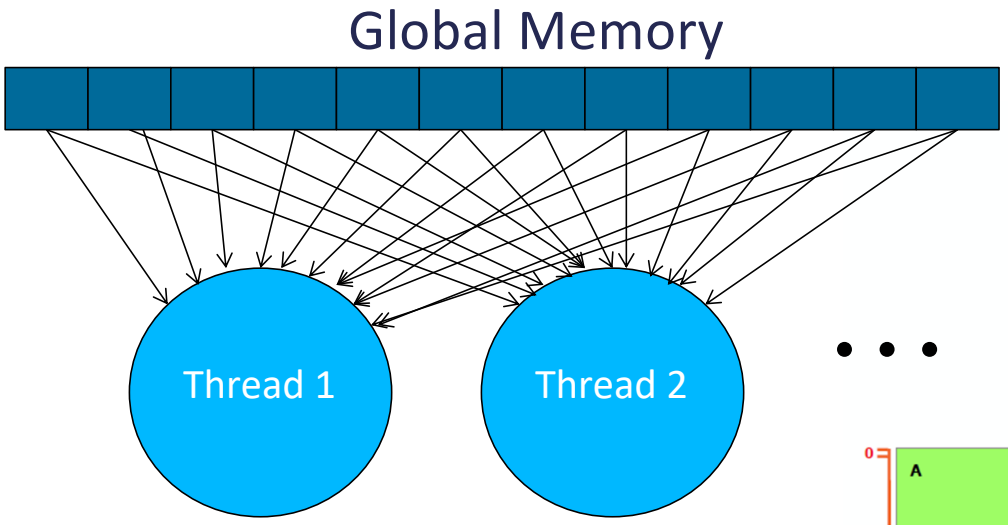
- Using 2D index
- Copying to/from device
- Free memory

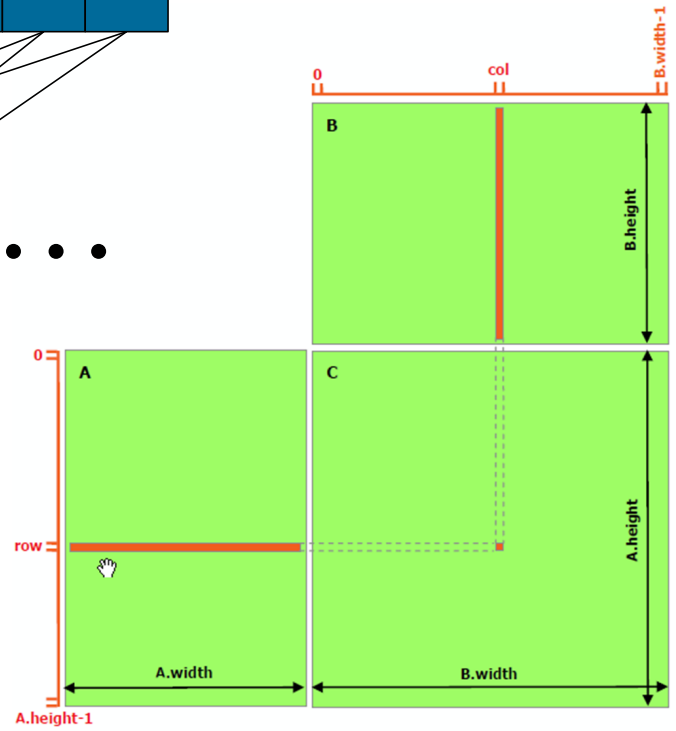
```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

not good, still accessing from global memory

Global memory access without Tiling

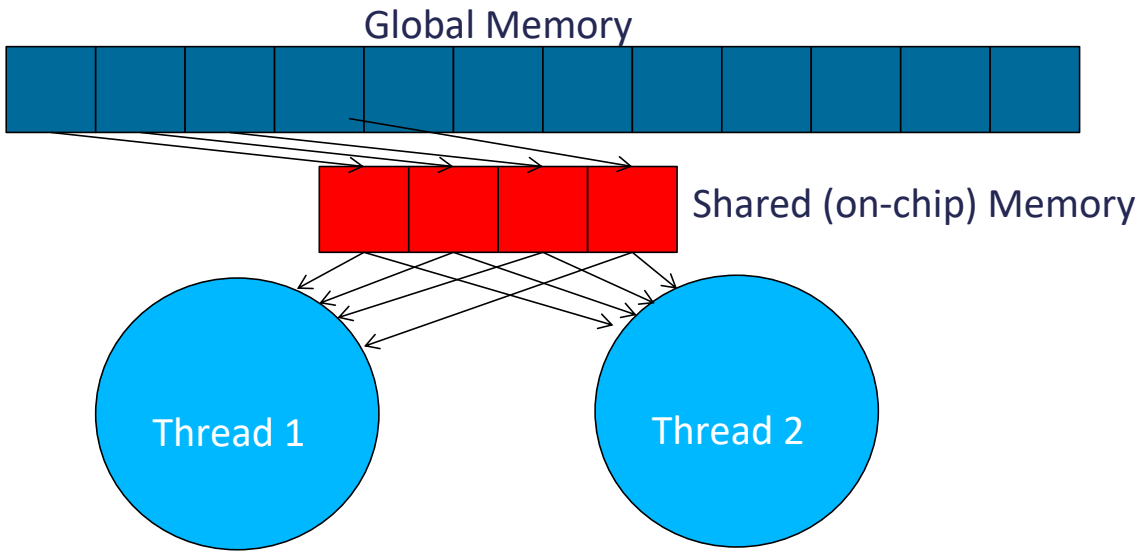
- Every thread access a specific address
- No reuse: accesses happen in parallel on addresses that are far apart (stride)





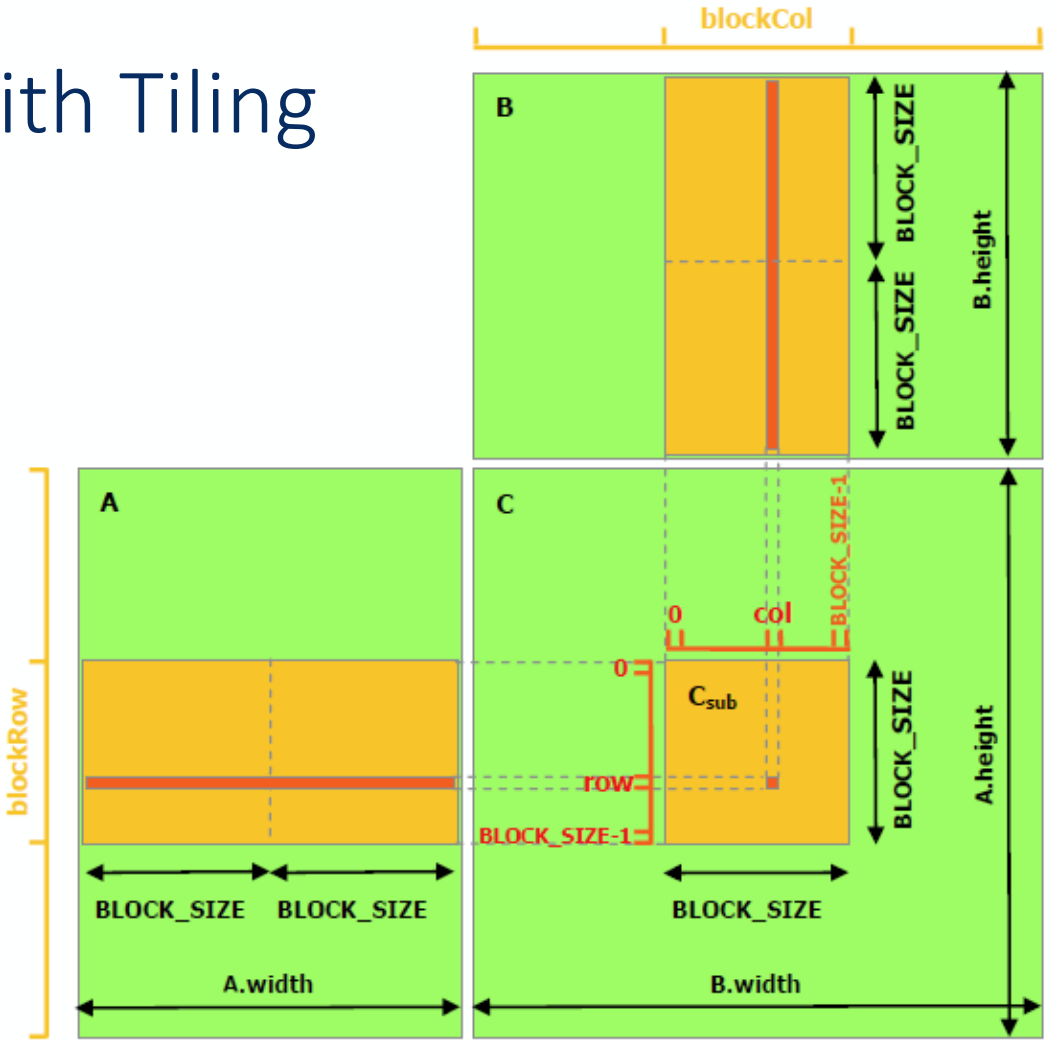
Global memory access with Tiling technique

1. Load in a tile (block)
2. **shared memory** Work on it with lower access latency
3. Store it back into global memory



Matrix Multiplication with Tiling

- Computing matrix multiplication
  - C = A × B
- Use shared memory to reduce global memory access
- Threads in a **block** work on a **tile:**
  1. Load tile in shared memory
  2. Then load elements from shared memory with lower latency
  3. Write tile back to global memory
- Each thread compute one (or more) elements of C:
  - Dot product: *row **dot** column*



Tiling Illustration

- https://nichijou.co/cuda7-tiling

Matrix Multiplication w. Tiling Example (1)

- Elements in row-major order
- Implement getter/setter for easy elements access

```
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value)
{
    A.elements[row * A.stride + col] = value;
}
```

Matrix Multiplication w. Tiling Example (2)

- Function to obtain the sub-matrix

```
// Get the BLOCK_SIZE x BLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices
// down from the upper-left corner of A
 __device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                         + BLOCK_SIZE * col];
    return Asub;
}

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix,
Matrix);
```

Matrix Multiplication w. Tiling Example (3)

- Load A, B into device memory
- Allocate C into device memory

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = 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);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
```

Matrix Multiplication w. Tiling Example (4)

- Invoke the kernel
- Copy C to the host
- Free device memory

```
// Second part of host code function
// void MatMul(const Matrix A, const Matrix B, Matrix C)
// {

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
```

Matrix Multiplication w. Tiling Example (5)

- Device Code first part
- Compute dimensions

```
// Matrix multiplication kernel called by MatMul()
 __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // All threads in a block compute one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;
```

Matrix Multiplication w. Tiling Example (6)

- Device code second part:

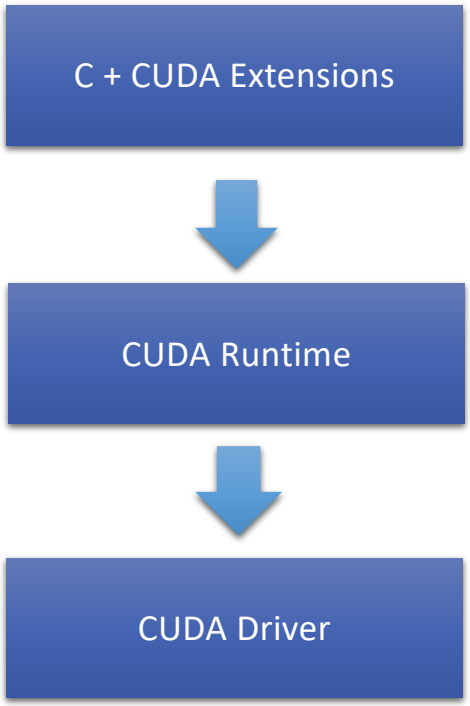
1. Loop over all the sub-matrices of A and B
2. Multiply each pair of sub-matrices together and accumulate the results
3. Write back

```
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);

        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load Asub and Bsub from device memory to shared memory
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        __syncthreads(); // Synch after A and B loaded

        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];
       __syncthreads(); // Synch to make computation is done
    }
    SetElement(Csub, row, col, Cvalue); // Write Csub to device mem
}
```

CUDA Compilation and Runtime

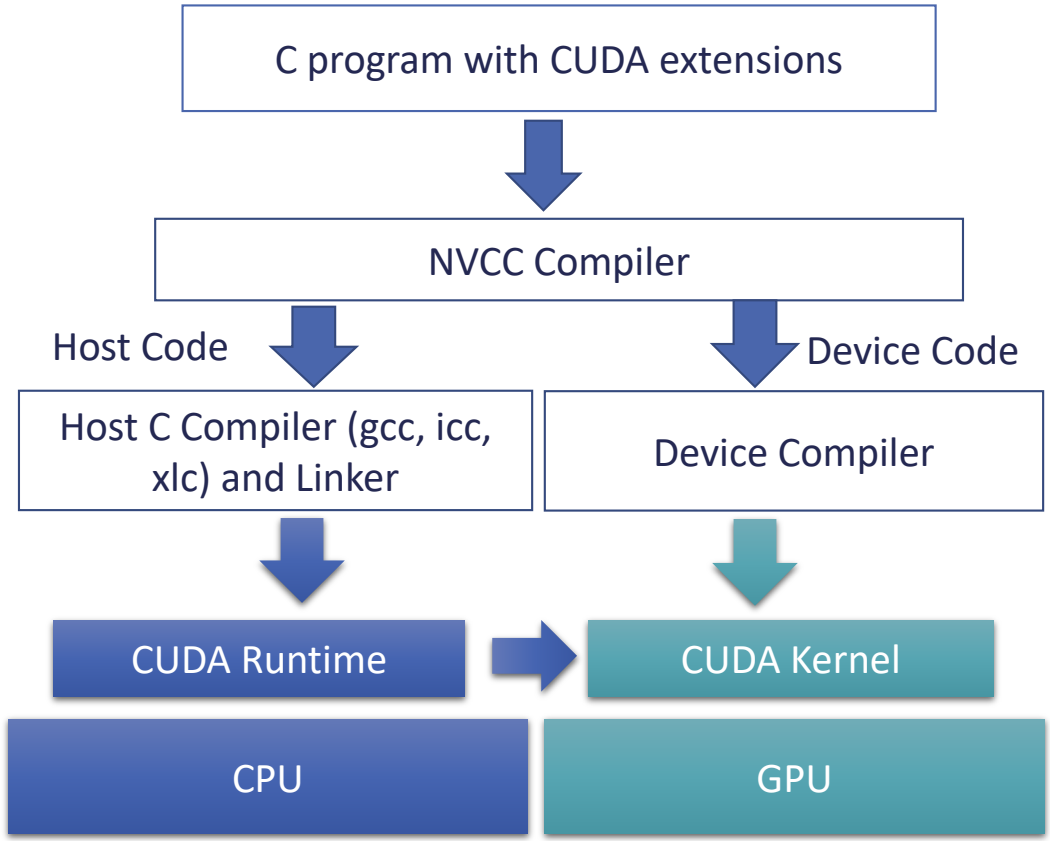
CUDA C Extension Runtime and Driver

- There are three layers for a CUDA program:
  1. CUDA C extension:
     - Developers write code using this extension
     - Ex. The <<<>>> kernel construct or the threadIdx variable
  2. CUDA Runtime API
     - NVCC compiler produces code that calls the runtime
     - Functions preceded by *cudaXXX*
     - Library: *cudart.dll or cudart.so*
  3. CUDA Driver API
     - The runtime uses the CUDA driver APIs
     - Library: *cuda.dll or cuda.so*
     - Functions preceded by cuXXX
     - Uses PTX code to execute on the GPU
- Parallel Thread Execution (PTX or NVPTX) is a low-level parallel thread execution virtual machine and instruction set architecture used in Nvidia's CUDA programming environment.
  - PTX is stable across multiple GPU generations
- The runtime API is a wrapper/helper of the driver API.



CUDA Compilation and Runtime

- NVCC compiler divides code in two parts:
  - Host C/C++ compiler
  - Device compiler
- Host C/C++ code is passed to the host compiler
- Host C code:
  - Contains calls to **CUDA runtime** and pass the **kernel code** as argument
- Kernel code is executed on the GPU



Virtual vs. real architecture when compiling

- A virtual GPU is defined entirely by the set of capabilities, or features, that it provides to the application
- PTX code (text format) can be considered as the assembly for a virtual GPU architecture
- nvcc compilation command always uses two architectures: a compute architecture to specify the virtual intermediate architecture, plus a real GPU architecture to specify the intended processor to execute on

```
nvcc x.cu --gpu-architecture=compute_50 --gpu-code=sm_50,sm_52
```

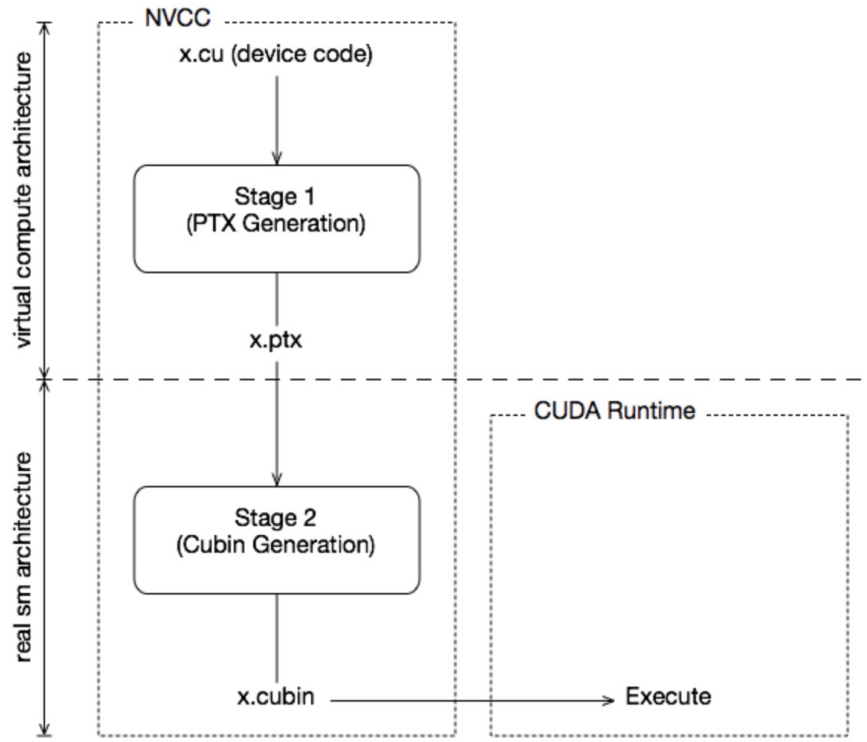
real architecture

- The real architecture must be an implementation (someway or another) of the virtual architecture

What architecture to specify when compiling?

- Specify the lowest virtual architecture that has a sufficient feature set to enable the program to be executed on the widest range of physical architectures
- Chosen virtual architecture is more of a statement on the GPU capabilities that the application requires
- Using a smallest virtual architecture still allows a widest range of actual architectures for the second nvcc stage (PTX→cubin)
- Specifying a virtual architecture that provides features unused by the application unnecessarily restricts the set of possible GPUs that can be specified in the second nvcc stage.

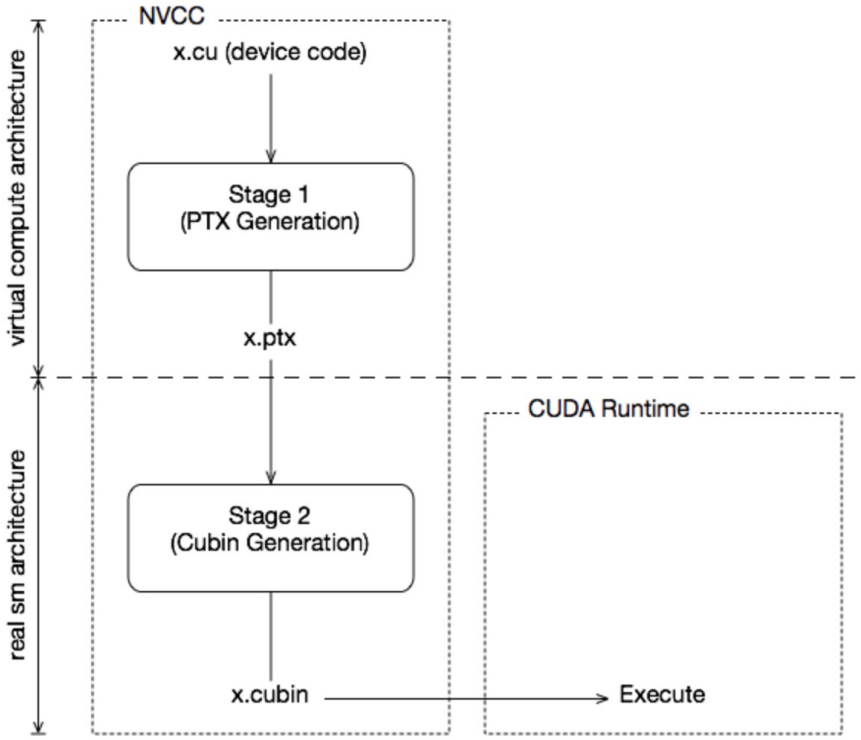
Two staged compilation with Virtual and Real architectures



- nvcc compiler driver (similar to gcc)
- Off-line Compilation
  - Host Code
    - Modified to run kernels
    - Compiled to (x86) binary
  - Device Code
    - Compiled to PTX (Parallel Thread Execution)
- Just-in-Time Compilation
  - Compile PTX into native GPU instructions
  - Allows for wards compatibility

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-architecture-feature-list

Two staged compilation with Virtual and Real architectures



TIPS:
Maximize portability of code: Choose *virtual* arch as *low* as possible when compiling

Get the optimal performance on a GPU: r*eal* arch should be chosen as *high* as possible

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-architecture-feature-list

NVIDIA Virtual Architectures Feature List

`compute_35`, and `compute_37`	Kepler support Unified memory programming Dynamic parallelism support
`compute_50`, `compute_52`, and `compute_53`	+ Maxwell support
`compute_60`, `compute_61`, and `compute_62`	+ Pascal support
`compute_70` and `compute_72`	+ Volta support
`compute_75`	+ Turing support
`compute_80`, `compute_86` and `compute_87`	+ NVIDIA Ampere GPU architecture support
`compute_89`	+ Ada support
`compute_90`	+ Hopper support

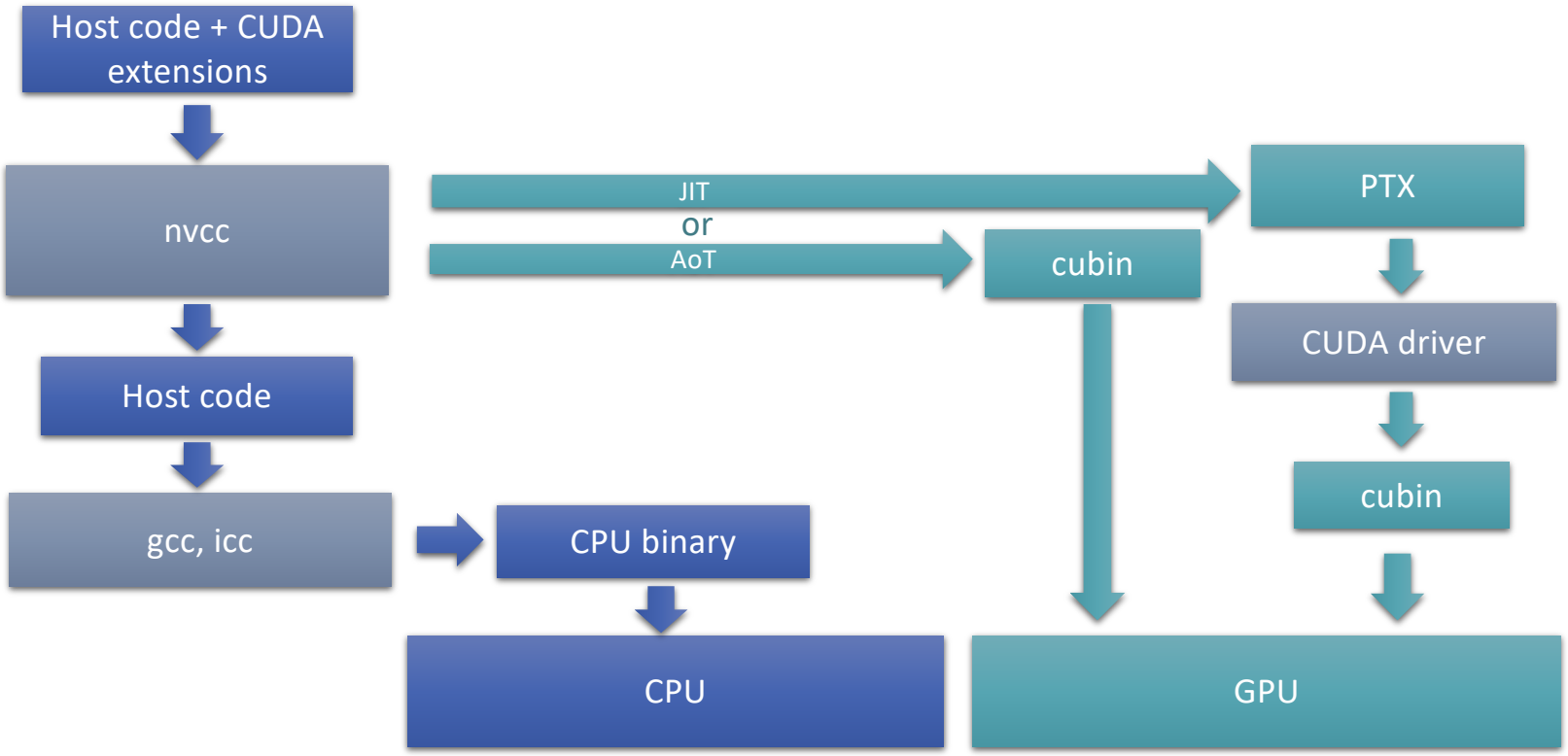
https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-architecture-feature-list

CUDA Just-in-time compilation

- Two approaches to compiling CUDA
- CUDA Ahead of Time/Offline Compilation:
  - Compilation flag code (example: --code=sm_60 for compute capability 6.0)
  - Directly compile into CUDA binary: **cubin**
- CUDA JIT:
  - Compilation flag arch (example: --arch=compute_60 for compute capability 6.0)
  1. Compile kernels in **PTX** (CUDA assembly)
  2. PTX code is compiled by the **CUDA Driver**

What is the advantage of JIT for CUDA applications like games?

CUDA Compilation Schema



http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compilation-with-nvcc

Lesson Key Points

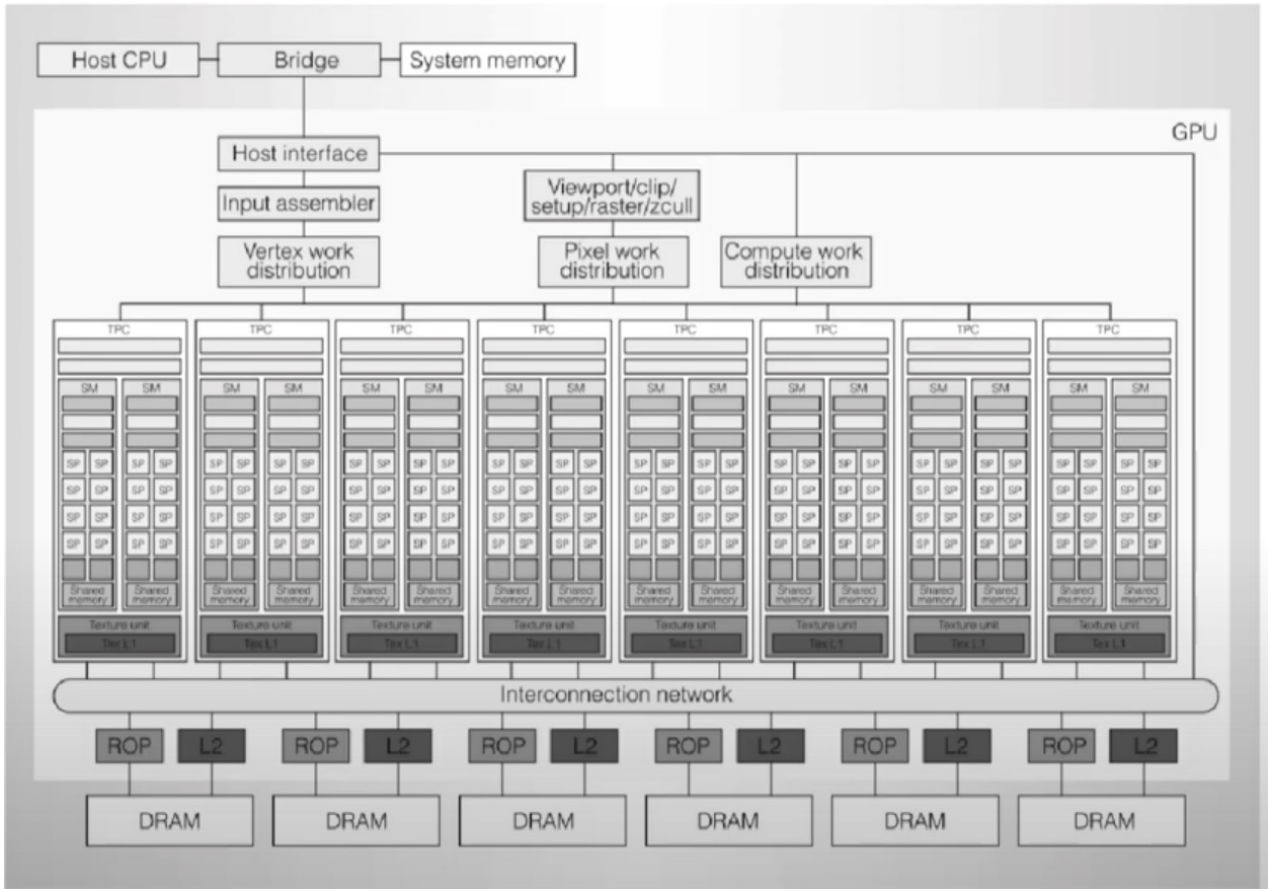
- Heterogenous architectures motivations
- Hierarchy of Computations:
  - Threads
  - Blocks
  - Grids
- Corresponding Memory Spaces
  - Local
  - Shared
  - Global
- Synchronization Primitives
  - Implicit Barriers
  - Thread Synchronization
- NVIDIA GPUs and CUDA:
  - Compute capability
- CUDA Programming Model:
  - Grid, Block, Thread
  - UVM
- CUDA Warp Scheduling
- CUDA Context and Streams
- CUDA Memory Alignment and Coalescing
- CUDA Compilation and Runtime:
  - CUDA Runtime, CUDA Driver, AoT and JIT compilation
- CUDA Hardware
- CUDA Profiling and Debugging
- cuDNN and cuBLAS

CUDA Hardware

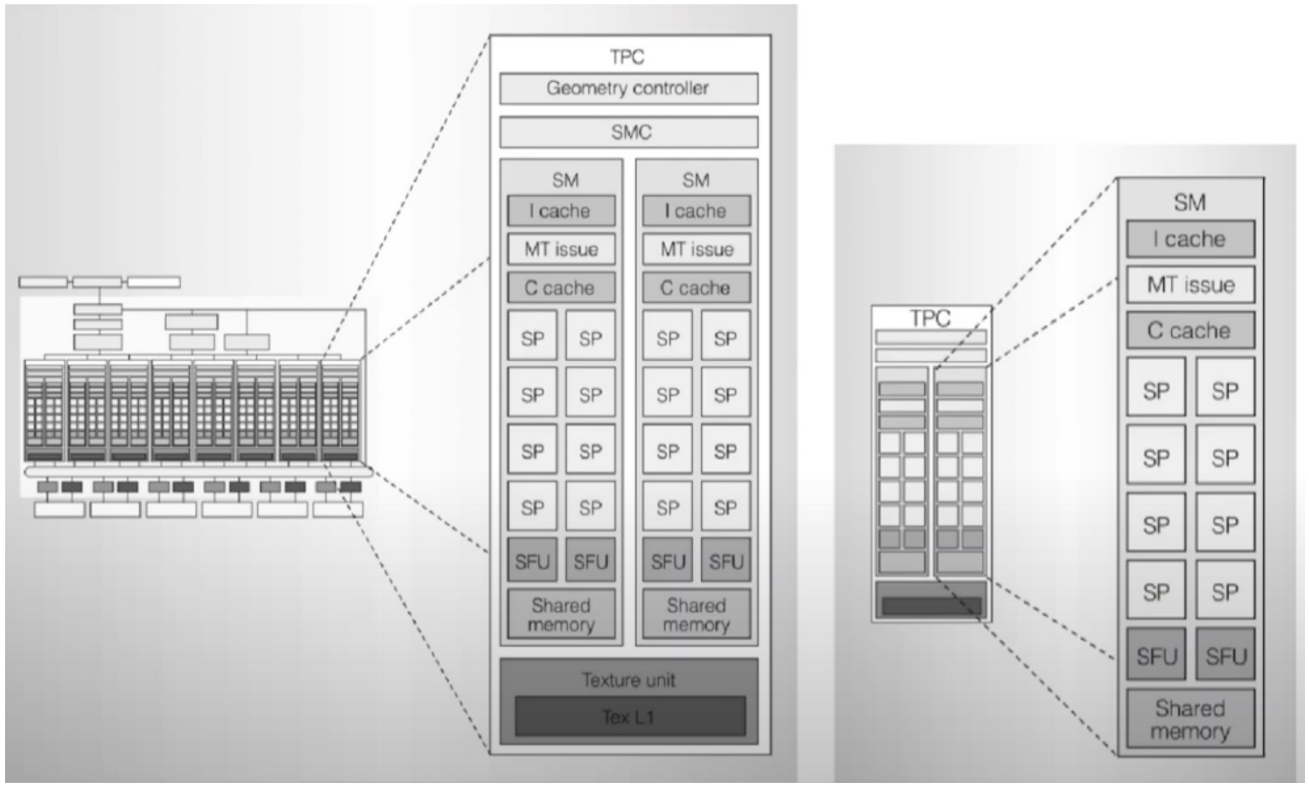
Terminology

Term	Definition
SM	Streaming Multiprocessor
SP	Streaming Processor
TPC	Texture/Processor Cluster
GPC	Graphics Processing Cluster
SP	Single-Precision (32-bit)
DP	Double-Precision (64-bit)

GeForce 8800 (2008)



GeForce 8800 (2008) SM



I Cache: Instruction Cache

C Cache: Constant Cache

SF: Special Function Unit

Pascal GPU (2016)

- 6 GPC
- 10 SM/GPC
- 60 SM
- 64 SP/SM
- 3840 SP

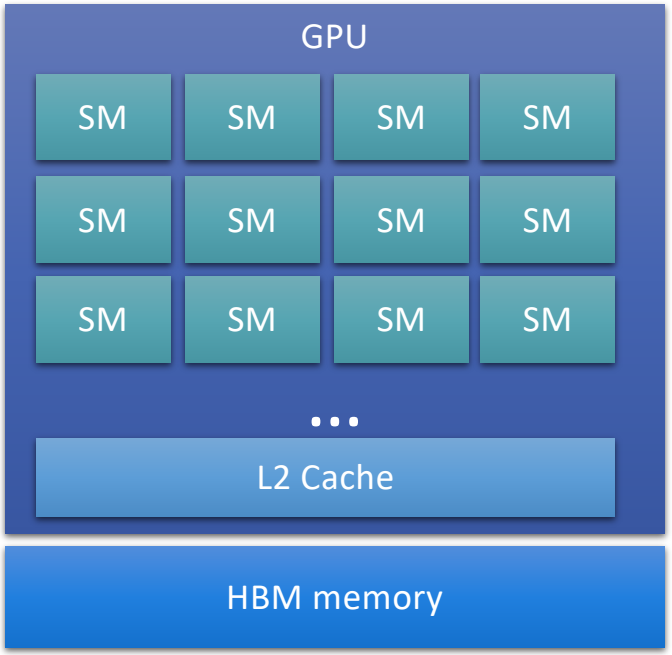
Pascal GPU (2016) – SM Closeup

Volta GPU (2017)

- 6 GPC
- 14 SM/GPC
- 84 SM
- 64 SP Float Cores/SM
- 64 SP Int Cores/SM
- 32 DP Float Cores/SM
- 5376 SP Float Cores
- 5376 SP Int Cores
- 2688 DP Float Cores

NVIDIA Volta Architecture (Tesla V100)

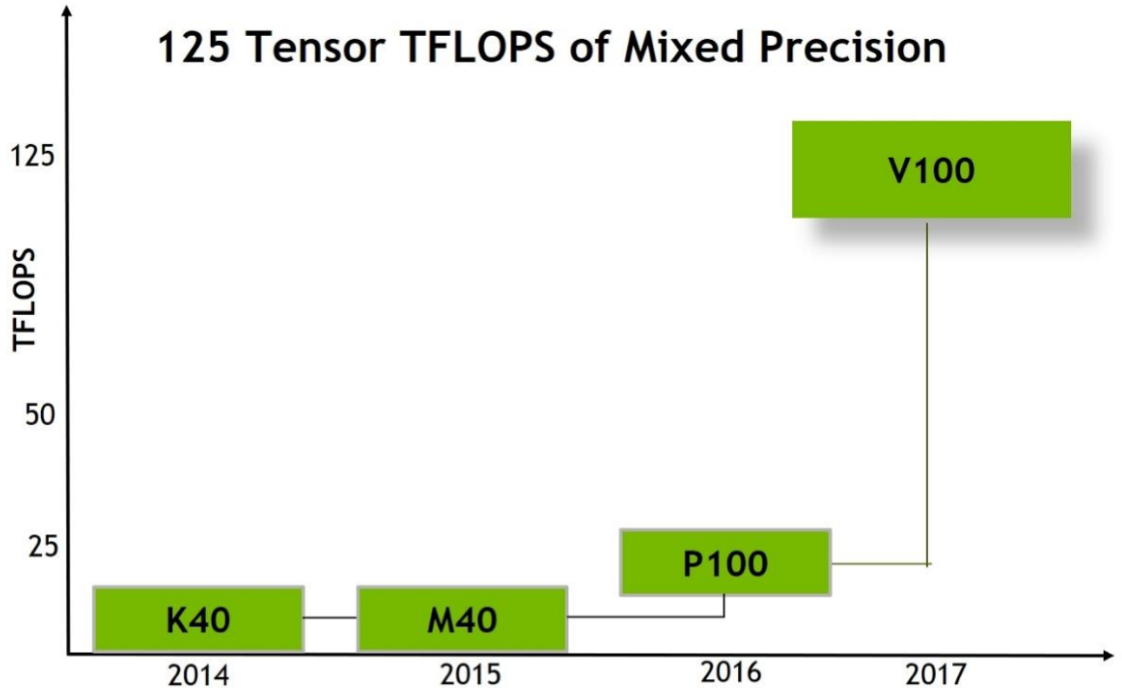
- Streaming Multiprocessors
  - 32 hardware threads Double Precision (DP: 64 bits)
  - or 64 hardware threads Single Precision (SP: 32 bits)
- DP FLOPS: 7,000 GFLOPS
- L2 size: 6MB
- High Bandwidth Memory
  - Size: 16 GB
  - Bandwidth: 900 GB/s



Volta GPU (2017) – SM Closeup

- Introduction of Tensor core: dedicated hardware for Deep Learning.
- Performs basic calculations for training and evaluation of neural networks
- Provide enormous speedups for AI neural network training and inferencing

Impact of Tensor Cores



Tesla V100 provides a major leap in Deep Learning Performance with New Tensor Cores

Ampere GPU (2020) – Current Architecture

- SM
- 7 GPC
- 12 SM/GPC
- 84 SM
- 128 CUDA Cores/SM
- 28 Tensor Cores/SM
- 10752 CUDA Cores
- 336 Tensor Cores

NVIDIA A100 (2020)

108 cores on the A100 https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

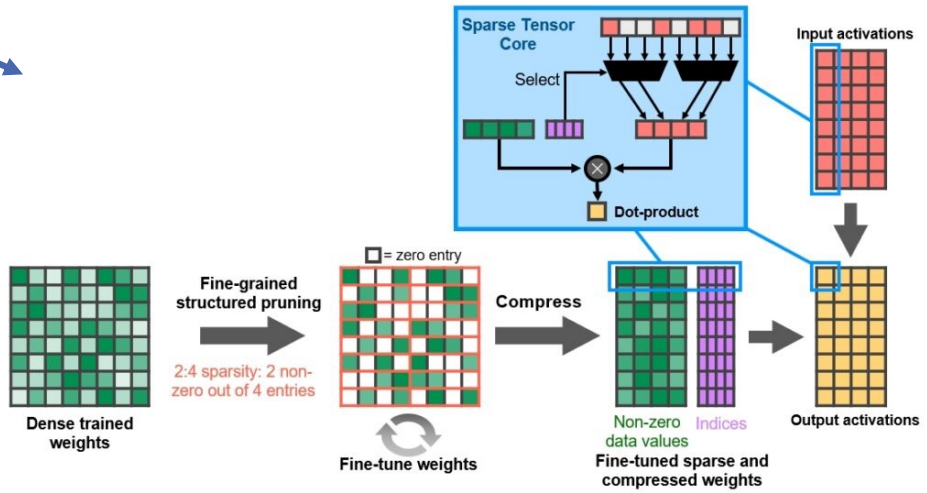
(Up to 128 cores in the full-blown chip)

40MB L2 cache

NVIDIA A100 Core

GPU compute throughput:

- 19.5 TFLOPS Single Precision
- 9.7 TFLOPS Double Precision
- 312 TFLOPS for Deep Learning (Tensor cores)



https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

NVIDIA H100 Block Diagram

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

144 cores on the full GH100

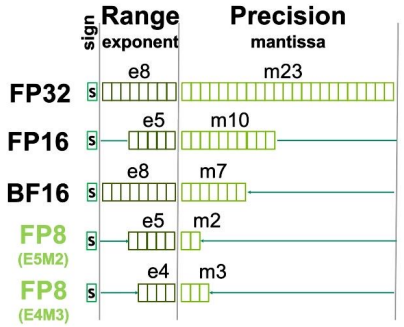
60MB L2 cache

NVIDIA H100 Core

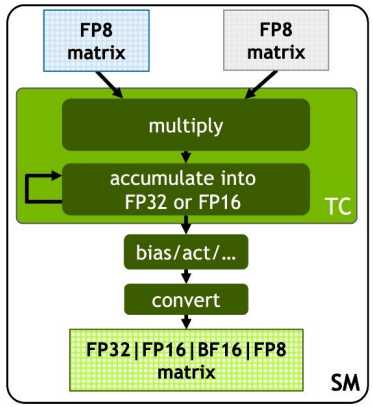
48 TFLOPS Single Precision*

24 TFLOPS Double Precision*

800 TFLOPS (FP16, Tensor Cores)*



Allocate 1 bit to either range or precision



Support for multiple accumulator and output types

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

* Preliminary performance estimates

NVIDIA A100 vs. H100

Nvidia Datacenter GPU	Nvidia A100 SXM	Nvidia H100 SXM	Nvidia H100 PCIe
GPU codename	GA100	GH100	GH100
GPU architecture	Ampere	Hopper	Hopper
GPU board form factor	SXM4	SXM5	PCIe Gen5
Launch date	May 2020	March 2022	March 2022
GPU process	TSMC 7nm N7	custom TSMC 4N	custom TSMC 4N
Die size	826mm2	814 mm2	814 mm2
Transistor Count	54 billion	80 billion	80 billion
FP64 CUDA cores	3,456	8,448	7,296
FP32 CUDA cores	6,912	16,896	14,592
Tensor Cores	432	528	456
Streaming Multiprocessors	108	132	114
Peak FP64	9.7 teraflops	30 teraflops	24 teraflops
Peak FP64 Tensor Core	19.5 teraflops	60 teraflops	48 teraflops
Peak FP32	19.5 teraflops	60 teraflops	48 teraflops
Peak FP32 Tensor Core	156 teraflops | 312 teraflops*	500 teraflops | 1,000 teraflops*	400 teraflops | 800 teraflops*
Peak BFLOAT16 Tensor Core	312 teraflops | 624 teraflops*	1,000 teraflops | 2,000 teraflops*	800 teraflops | 1,600 teraflops*
Peak FP16 Tensor Core	312 teraflops | 624 teraflops*	1,000 teraflops | 2,000 teraflops*	800 teraflops | 1,600 teraflops*
Peak FP8 Tensor Core	-	2,000 teraflops | 4,000 teraflops*	1,600 teraflops | 3,200 teraflops*
Peak INT8 Tensor Core	624 TOPS | 1,248 TOPS*	2,000 TOPS | 4,000 TOPS*	1,600 TOPS | 3,200 TOPS*
Peak INT4 Tensor Core	1,248 TOPS | 2,496 TOPS*	-	-
Interconnect	NVLink: 600GB/s PCI Gen4: 64GB/s	NVLink: 900GB/s PCI Gen5: 128GB/s	NVLink: 600GB/s PCI Gen5: 128GB/s
Max TDP	400 watts	700 watts	350 watts

*Effective TFLOPS or FLOPS using the Sparsity feature

Source: https://www.hpcwire.com/2022/03/22/nvidia-launches-hopper-h100-gpu-new-dgxs-and-grace-megachips/

- From: NVIDIA

Table 1. Comparison of NVIDIA Tesla GPUs

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN
[1] Peak TFLOPS rates are based on GPU Boost Clock				

Summary/Comparisons

Year	μArch	SMs	SPs	Note
2008	Tesla	16	128	GeForce 8800
		30	240	GeForce 280
2010	Fermi	16	512	
2012	Kepler	15	2880	
			384	Pseudo Lab (Quadro K620)
2014	Maxwell	16	2048	
2016	Pascal	60	3840	
2017	Volta	84	5376	
2019	Turing	72	4608	
2020	Ampere	84	10752	

PyTorch and CUDA

https://pytorch.org/docs/stable/notes/cuda.html

Overview

- **CUDA Basics**
  - Integrated with NVIDIA CUDA toolkit.
  - Enables GPU-accelerated tensor operations and model training.
  - Automatically utilizes CUDA libraries (cuDNN, cuBLAS) for optimized performance.
- **Key Features of torch.cuda**
  - **Device Management**: Easily switch between multiple GPUs with torch.cuda.device().
  - **Memory Management**: Allocate and free GPU memory seamlessly.
  - **Performance Optimization**: Use CUDA streams for asynchronous execution and overlapping computation with communication.
  - **Utilities and Helpers**: Functions for debugging and profiling, such as memory usage tracking and timing operations.
- torch.cuda is used to set up and run CUDA operations. It keeps track of the currently selected GPU, and all CUDA tensors you allocate will, by default, be created on that device.
- Once a tensor is allocated, operations can be performed on it regardless of the selected device, and the results will always be placed on the same device as the tensor.
- Cross-GPU operations are not allowed by default, except copy_() and other methods with copy-like functionality such as to() and cuda().

Initialization and Configuration

1. **CUDA Availability Check**: PyTorch checks for the availability of CUDA on the system with torch.cuda.is_available(). Writing device-agnostic code that runs on both CPU and GPU is essential.
2. **Device Management**: PyTorch allows you to select a device with torch.device("cuda:0") where "cuda:0" refers to the first GPU. If multiple GPUs are available, you can select which one to use with "cuda:1", "cuda:2", etc.
3. **Automatic Memory Management**: When a tensor is moved to a CUDA device, PyTorch handles the memory allocation on the GPU. It abstracts away the details of memory pinning and transfer, simplifying the development process.

Tensor Operations

1. **Tensor to GPU**: PyTorch tensors can be moved onto any device by calling .to(device) or .cuda(). Once a tensor is on the GPU, all operations on it are performed by the GPU.
2. **CUDA Tensors**: A tensor on the GPU is often called a CUDA tensor. PyTorch operations automatically use optimized CUDA kernels when available, providing acceleration without requiring manual intervention from the developer.
3. **Tensor Types**: PyTorch maintains the same dtype (data type) when moving tensors between devices, ensuring consistent behavior of mathematical operations.

```
# Check for CUDA availability
cuda_available = torch.cuda.is_available()
print("CUDA Available:", cuda_available)

# Select a CUDA device
device = torch.device("cuda" if cuda_available else "cpu")
print("Selected Device:", device)
```

```
# Create a tensor on CPU
tensor_cpu = torch.randn(3, 3)

# Move the tensor to GPU
tensor_gpu = tensor_cpu.to(device)  # If CUDA is available, tensor_gpu will be on GPU
```

Computation Graph and Autograd

1. **Computation Graph on GPU**: PyTorch builds a dynamic computation graph (the graph that tracks operations on tensors) even when the tensors are on the GPU. This allows for automatic differentiation using PyTorch's autograd system.
2. **Backpropagation**: During backpropagation, gradients are calculated on the GPU if the forward pass was computed there, leading to faster training cycles for neural networks.

```
# Dummy input tensor
input_tensor = torch.randn(1, 10).to(device)

# Forward pass
output = net(input_tensor)

# Dummy target for loss computation
target = torch.randn(1, 5).to(device)

# Loss function
criterion = nn.MSELoss()

# Compute loss
loss = criterion(output, target)

# Zero gradients before backward pass
net.zero_grad()

# Backward pass
loss.backward()
```

Neural Networks Module

1. **Model to GPU**: Entire neural network models can be moved to the GPU with .to(device) or .cuda(). This transfers all model parameters and buffers to the GPU.
2. **Data Parallelism**: PyTorch supports data parallelism with torch.nn.DataParallel, which can automatically distribute batches of data across multiple GPUs and aggregate the results.

CUDA Streams

1. **Asynchronous Execution**: PyTorch leverages CUDA streams to manage asynchronous execution of tensor operations. This can help overlap computation with memory transfers to improve performance.
2. **Stream Management**: Advanced users can create and manage their own CUDA streams to further control the parallelism of tensor operations.

Example 1: Operations inside each stream are serialized in the order they are created, but operations from different streams can execute concurrently in any relative order.

```
cuda = torch.device('cuda')
s = torch.cuda.Stream()  # Create a new stream.
A = torch.empty((100, 100), device=cuda).normal_(0.0, 1.0)
with torch.cuda.stream(s):
    # sum() may start execution before normal_() finishes!
    B = torch.sum(A)
```

Example 1: There are two new additions.
The torch.cuda.Stream.wait_stream() call ensures that the **normal_()** execution has finished before we start running **sum(A)** on a side stream.

```
cuda = torch.device('cuda')
s = torch.cuda.Stream()  # Create a new stream.
A = torch.empty((100, 100), device=cuda).normal_(0.0, 1.0)
s.wait_stream(torch.cuda.default_stream(cuda))  # NEW!
with torch.cuda.stream(s):
    B = torch.sum(A)
A.record_stream(s)  # NEW!
```

Error Handling

1. **CUDA Errors**: PyTorch captures and reports CUDA errors, making it easier to debug issues that occur on the GPU.
2. **Memory Management**: PyTorch provides functions like torch.cuda.memory_allocated() to monitor and debug GPU memory usage.

```
# Print current memory allocation (in bytes) for a specific device
print(torch.cuda.memory_allocated(device))

# Clear cache to release unreferenced memory
torch.cuda.empty_cache()
```

Interoperability

1. **CUDA Extensions**: Developers can write custom CUDA kernels and integrate them into PyTorch using the torch.utils.cpp_extension module.
2. **Third-party Libraries**: PyTorch integrates with third-party CUDA libraries, like cuDNN for deep learning-specific optimizations and cuBLAS for basic linear algebra operations.

Cleanup and Shutdown

1. **Memory Cleanup**: PyTorch automatically deallocates CUDA tensor memory when the tensors go out of scope. Developers can also manually.
   - free memory using torch.cuda.empty_cache() to clear unused memory from the cache.

2. **Device Reset**: Although PyTorch manages device contexts automatically, it also allows manual resetting of the current device to a clean state using torch.cuda.reset_max_memory_allocated() and other similar functions.

```
# Assuming you have CUDA tensors a, b, c
del a
del b
del c
torch.cuda.empty_cache()  # Clear unused memory from the cache
```

PyTorch CUDA Ecosystem

1. **Extensions and Tools**: The PyTorch ecosystem includes several tools and libraries that extend its CUDA capabilities, such as TorchVision for image-related operations and TorchText for natural language processing, both of which can leverage GPU acceleration.
2. **Distributed Training**: PyTorch's torch.distributed package supports distributed model training across multiple GPUs and nodes, enabling large-scale training tasks. It uses collective communication algorithms optimized for CUDA-enabled devices.
3. **Mixed-Precision Training**: PyTorch supports mixed-precision training using NVIDIA's Apex library, which can significantly speed up training with reduced precision while maintaining model accuracy.
4. **Profiler**: PyTorch provides a profiler that can track both CPU and GPU activity, giving developers insights into which operations are taking the most time and how to optimize them.

CUDA and PyTorch Versions Compatibility

1. **Version Matching**: It's essential to have compatible versions of PyTorch and CUDA installed. PyTorch releases are typically linked to specific versions of CUDA, and using the wrong combination can lead to errors.
2. **Installation**: When installing PyTorch, users need to select the CUDA version that matches their system's CUDA installation. PyTorch binaries are precompiled with CUDA support, simplifying installation and setup.

cuDNN and cuBLAS

Reading Assignment and References

NVIDIA Deep Learning SDK

- **Deep Learning Primitives (CUDA® Deep Neural Network library™ (cuDNN))**
  - High-performance building blocks for deep neural network applications including convolutions, activation functions, and tensor transformations.
- **Deep Learning Inference Engine (TensorRT™)**
  - High-performance deep learning inference runtime for production deployment.
- **Deep Learning for Video Analytics (NVIDIA DeepStream™ SDK)**
  - High-level C++ API and runtime for GPU-accelerated transcoding and deep learning inference.
- **Linear Algebra (CUDA® Basic Linear Algebra Subroutines library™ (cuBLAS))**
  - GPU-accelerated BLAS functionality that delivers 6x to 17x faster performance than CPU-only BLAS libraries,
- **Sparse Matrix Operations (NVIDIA CUDA® Sparse Matrix library™ (cuSPARSE))**
  - GPU-accelerated linear algebra subroutines for sparse matrices that deliver up to 8x faster performance than CPU BLAS (MKL), ideal for applications such as natural language processing.
- **Multi-GPU Communication (NVIDIA® Collective Communications Library ™ (NCCL))**
  - Collective communication routines, such as all-gather, reduce, and broadcast that accelerate multi-GPU deep learning training on up to eight GPUs.

cuDNN

- Nvidia's library of low-level primitives for deep learning
  - Closed source
  - Highly optimized
- Provides
  - Tensor manipulation
  - Convolution
  - Pooling
  - Softmax
  - Activation functions
    - Sigmoid, ReLU, Tanh, Clipped ReLu, ELU, identity

cuDNN context

- cuDNN API are executed on the **host** (and not on the device!)
- A *cudnnHandle_t* stores the state of the library and must be used in any subsequent library call
- The cuDNN library context must be created using cudnnCreate() and the returned handle must be passed to all subsequent library function calls. The context should be destroyed at the end using cudnnDestroy().
- Usually we need a single handle per device
- Multiple devices => multiple handles

```
#include <cudnn.h>

    /* … */
    cudnnHandle_t cudnn;
    cudnnCreate(&cudnn);
    /* … */
    CudnnDestroy(cudnn);
```

Error checking

- Calls return a *cudnnStatus_t* value:
  - an enumerated type used for function status returns
  - CUDNN_STATUS_SUCCESS denotes... success
  - *cudnnGetErrorString(value)* returns a description of the error
- Generally used as:

```
#define CUDNN_CALL(x) do {                                          \
    cudnnStatus_t ___s = (x);                                       \
    if (___s != CUDNN_STATUS_SUCCESS) {                             \
        fprintf(stderr, "%s:%d ERROR: %s\n", __FILE__,              \
                __LINE__, cudnnGetErrorString(___s));           \
        exit(-1);                                                   \
    }                                                               \
} while (0)

/* … */
CUDNN_CALL(cudnnCreate(&handle));
```

Tensors

- Tensors are represented in memory as a contiguous sequence of elements
- For 4D tensors, the {batch, channel, height, width} coordinates of a data point are linearized in one of three ways (specified by cudnnTensorFormat_t):
  - *CUDNN_TENSOR_NCHW*: batch, channel, height, width
  - *CUDNN_TENSOR_NHWC*: batch, height, width, channel
  - *CUDNN_TENSOR_NCHW_VECT_C*: same as NCHW, but elements are vectors of features
- An arbitrary number of dimensions is supported (up to CUDNN_DIM_MAX)
- Multiple data types are supported (specified by cudnnDataType_t):
  - *CUDNN_DATA_{HALF, FLOAT, DOUBLE}*
  - *CUDNN_DATA_INT{8, 32, 8x4}*
  - *CUDNN_DATA_UINT{8, 8x4}*

Reproducibility (Determinism)

- Most cuDNN routines produce the same result across runs when:
  - executed on GPUs with the same architecture (compute capability) and the same number of SMs
- Bit-wise reproducibility is not guaranteed across cuDNN versions
- Across different architectures, no cuDNN routines guarantee bit-wise reproducibility.
- Some of them use atomic operations so results **can be different at every run on the same GPU and cuDNN version**.
  - Random floating point rounding errors
- See https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html#reproducibility

Tensor cores arithmetic – Starting From Volta

- The arithmetic used on tensors is specified by *cudnnMathType_t*
  - *CUDNN_DEFAULT_MATH*: standard floating point behavior
  - *CUDNN_TENSOR_OP_MATH*: allows the use of **Tensor Cores**
- **Tensor Cores** provide improved performance at the expense of reduced precision
- Only supported in a subset of primitives and with specific parameters:
  - *cudnnConvolution{Forward,BackwardData,BackwardFilter}*: only specific algorithms
  - *cudnnRNN{ForwardInference,ForwardTraining,BackwardData,BackwardWeights}*: only specific algorithms and batch sizes

API and Descriptors

- Most entities in cuDNN are associated with a descriptor
  - The data lives in **GPU memory**
  - The **descriptor, in CPU memory**, contains information on how to access the data
- Some of the descriptors we'll encounter:
  - *cudnnTensorDescriptor_t*
  - *cudnnFilterDescriptor_t*
  - *cudnnConvolutionDescriptor_t*
  - *cudnnActivationDescriptor_t*
  - *cudnnPoolingDescriptor_t*

Descriptor lifecycle

- Descriptors in general are used according to the following pattern:

    1. Creation: with one of the cudnnCreate*${type}*Descriptor()
    2. Configuration: with one of the cudnnSet*${type}*Descriptor()
    3. Use
    4. Destruction: with one of the cudnnDestroy*${type}*Descriptor()

- Except for tensors, we're omitting the creation and destruction stages

Creating a 4D tensor

- The data has to be allocated separately, e.g., with cudaMalloc()
- The descriptor can be initialized in multiple ways, e.g.:

```
cudnnTensorDescriptor_t in_desc;
CUDNN_CALL(cudnnCreateTensorDescriptor(&in_desc));
CUDNN_CALL(cudnnSetTensor4dDescriptor(in_desc,CUDNN_TENSOR_NCHW,
                                      CUDNN_DATA_FLOAT, N, C, H, W));
```

https://docs.nvidia.com/deeplearning/cudnn/api/index.html#cudnnSetTensor4dDescriptor

Creating a N dimensional tensor

- The same descriptor could have been set by using cudnnSetTensorNdDescriptor and arrays of size 4 for *dimA* and *strideA*:

```
cudnnStatus_t cudnnSetTensorNdDescriptor(cudnnTensorDescriptor_t tensorDesc,
                                         cudnnDataType_t dataType, int nbDims,
                                         const int dimA[], const int strideA[])
```

  - Tensors are restricted to having at least 4 dimensions, and at most CUDNN_DIM_MAX dimensions (defined in cudnn.h)
  - Return is either CUDNN_STATUS_SUCCESS or CUDNN_STATUS_BAD_PARAM

https://docs.nvidia.com/deeplearning/cudnn/api/index.html#cudnnSetTensorNdDescriptor

More on tensor descriptors (1)

- cudnnSetTensor4dDescriptorEx() allows custom strides with the simplified interface for 4D tensors
  - Initializes a previously created generic tensor descriptor object into a 4D tensor, similarly to cudnnSetTensor4dDescriptor() but with the strides explicitly passed as parameters.
  - Strides are necessary when padding is present in memory at the end of each dimension
  - In a 4D tensor with { N, C, H, W } coordinates the default strides (with no padding present) would be { C*H*W, H*W, W, 1 }

https://docs.nvidia.com/deeplearning/cudnn/api/index.html#cudnnSetTensor4dDescriptorEx

More on tensor descriptors (2)

- Information on a tensor descriptor can be retrieved with:

```
cudnnStatus_t cudnnGetTensorNdDescriptor(const cudnnTensorDescriptor_t tensorDesc,
                                         int nbDimsRequested, cudnnDataType_t *dataType,
                                         int *nbDims, int dimA[], int strideA[])
```

- The tensor's size in bytes can be retrieved with:

```
cudnnStatus_t cudnnGetTensorSizeInBytes(const cudnnTensorDescriptor_t tensorDesc, size_t *size)
```

More on tensor descriptors

- Remember to destroy a tensor descriptor (or any other descriptor) when not needed anymore:

```
cudnnStatus_t
cudnnDestroyTensorDescriptor(cudnnTensorDescriptor_t
tensorDesc)
```

- Also, remember that none of the resources associated with a descriptor will be freed for you
  - If tensor data was allocated with cudaMalloc() it is the programmer's responsibility to call cudaFree() on it

Filter descriptors

- Convolution coefficients are stored in a filter:
  - A tensor-like object with its own descriptor type
  - The layout is still specified in terms of { N, C, H, W }, but N refers to the output maps, and C to the input maps
- cudnnFilterDescriptor_t is a pointer to an opaque structure holding the description of a filter dataset.
- Initialize 4D filter with:

```
cudnnStatus_t cudnnSetFilter4dDescriptor(cudnnFilterDescriptor_t filterDesc,
                                         cudnnDataType_t dataType,
                                         cudnnTensorFormat_t format, int k, int c, int h, int w)
```

- Or the generic N-dim form:

```
cudnnStatus_t cudnnSetFilterNdDescriptor(cudnnFilterDescriptor_t filterDesc,
                                         cudnnDataType_t dataType,
                                         cudnnTensorFormat_t format, int nbDims, const int filterDimA[])
```

Convolution descriptors

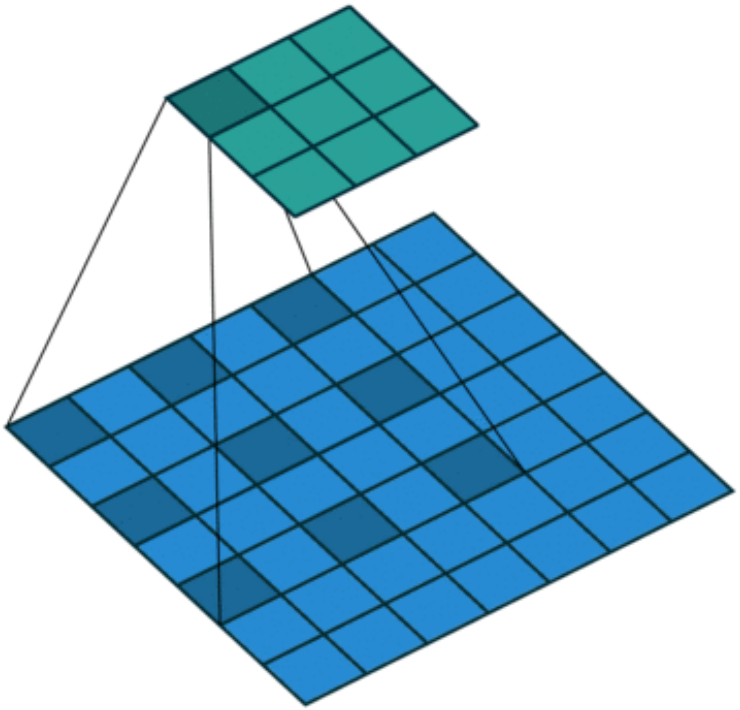
- Specify the desired behavior for a convolution
- Initialize 2D Convolution with:

```
cudnnStatus_t cudnnSetConvolution2dDescriptor(cudnnConvolutionDescriptor_t convDesc,
                                              int pad_h, int pad_w, int u, int v,
                                              int dilation_h, int dilation_w,
                                              cudnnConvolutionMode_t mode,
                                              cudnnDataType_t computeType)
```

- Or the generic N-dim form:

```
cudnnStatus_t cudnnSetConvolutionNdDescriptor(cudnnConvolutionDescriptor_t convDesc,
                                              int arrayLength, const int padA[],
                                              const int filterStrideA[],
                                              const int dilationA[],
                                              cudnnConvolutionMode_t mode,
                                              cudnnDataType_t dataType)
```

Dilated convolution



Convolution parameters

- Padding: **zeros implicitly added around the input data** on each dimension
  - The most common case for images is usually half the size of the kernel (e.g., 1 for a 3x3 kernel)
- Stride: increment in each dimension between two consecutive filtering windows
  - To produce one output for each input value, use all ones
- Dilation: used for dilated convolutions
  - All ones, unless dilated convolution is required
- Mode: either CUDNN_CONVOLUTION or CUDNN_CROSS_CORRELATION
- Data type: the precision required for the convolution

Forward convolution

- The generic form for a N-dim convolution is:

```
cudnnStatus_t cudnnConvolutionForward(cudnnHandle_t handle, const void *alpha,
                                      const cudnnTensorDescriptor_t xDesc, const void *x,
                                      const cudnnFilterDescriptor_t wDesc, const void *w,
                                      const cudnnConvolutionDescriptor_t convDesc,
                                      cudnnConvolutionFwdAlgo_t algo,
                                      void *workSpace, size_t workSpaceSizeInBytes,
                                      const void *beta,
                                      const cudnnTensorDescriptor_t yDesc, void *y)
```

- Executes convolutions or cross-correlations over x using filters specified with w, returning results in y
- x is the input tensor, and xDesc its descriptor, y the output
- w points to the weights of the filter and wDesc to the filter descriptor
- convDesc is the convolution descriptor
- algo is the desired algorithm (more about it in a moment)
- workSpace is a pointer to a work area of size workSpaceSizeInBytes
- **alpha** and **beta** are scaling factors for input and output (for **ResNets**):
  - dstValue = alpha[0]*result + beta[0]*priorDstValue
  - If not ResNet: alpha = 1, beta = 0

https://docs.nvidia.com/deeplearning/cudnn/api/index.html#cudnnConvolutionForward

Convolution algorithms

- Multiple algorithms are available
  - A subset of the possible combinations (tensor formats, data type, dilation) is supported for different algorithms
  - Predicting performance of any specific parameter set is not trivial
  - Always refer to the documentation for the specific details of a cuDNN release
- Main algorithm families:
  - **GEMM-based:** transform the computation to the application of matrix multiplications to use GPUs more efficiently
  - **FFT-based:** use the FFT to reduce the computational complexity of convolution, by operating in the transformed domain
  - **Winograd:** use minimal filtering algorithms based on polynomial factorization to reduce the number of operations needed

Selecting an algorithm

- The library provides a primitive to find best algorithm based on preference:

```
cudnnStatus_t cudnnGetConvolutionForwardAlgorithm(cudnnHandle_t handle,
                                        const cudnnTensorDescriptor_t xDesc,
                                        const cudnnFilterDescriptor_t wDesc,
                                        const cudnnConvolutionDescriptor_t convDesc,
                                        const cudnnTensorDescriptor_t yDesc,
                                        cudnnConvolutionFwdPreference_t preference,
                                        size_t memoryLimitInBytes,
                                        cudnnConvolutionFwdAlgo_t *algo)
```

- Where:
  - xDesc, wDesc, convDesc and yDesc specify what kind of convolution we want to perform
  - **preference** and memoryLimitInBytes specify the constraints:
    - CUDNN_CONVOLUTION_FWD_{NO_WORKSPACE,PREFER_FASTEST,SPECIFY_WORKSPACE_LIMIT}
    - The memory limit applies in the latter case

Selection based on benchmarking

- **PyTorch** and other frameworks offer the option of benchmarking multiple algorithms for any combination of input parameters to a convolution
  - Check findAlgorithm() in aten/src/ATen/native/cudnn/Conv.cpp
  - Every time a convolution is attempted with a new input size:
    - try multiple algorithms to find the best
    - reuse best algorithm
  - Cache the results for subsequent invocations
  - Enabled with torch.backends.cudnn.benchmark=True
  - **Not convenient if convolution size changes continuously**

Benchmarking through cuDNN

- Further info for algorithm selection is available through cuDNN via:

```
cudnnStatus_t cudnnFindConvolutionForwardAlgorithm(
    cudnnHandle_t                       handle,
    const cudnnTensorDescriptor_t       xDesc,
    const cudnnFilterDescriptor_t       wDesc,
    const cudnnConvolutionDescriptor_t convDesc,
    const cudnnTensorDescriptor_t       yDesc,
    const int                           requestedAlgoCount,
    int                                *returnedAlgoCount,
    cudnnConvolutionFwdAlgoPerf_t      *perfResults)
```

- **This runs multiple algorithms and returns their performance in a list**
- The caller requests at most requestedAlgoCount (making sure there is enough room in perfResults) and the count of the returned performance samples is stored in returnedAlgoCount
- A cudnnFindConvolutionForwardAlgorithmEx() is available, and it allows specifying a custom temporary workspace

Algorithm performance

- The **benchmarking results** for each algorithm are returned in a cudnnConvolutionFwdAlgo_t
- The fields are:
  - **algo**: the algorithm (enum)
  - **status**: an error code, or the return status of the convolution call
  - **time**: execution time (sorting key)
  - **memory**: workspace size
  - **determinism**: a flag specifying if the algorithm is deterministic
  - **mathType**: whether the algorithm uses TensorCores

Workspace size

- Convolution algorithms **may require GPU memory for temporary storage**
  - Unavailability of enough memory can **limit algorithm selection**
- To determine the memory required by al algorithm on a certain configuration:

```
cudnnStatus_t cudnnGetConvolutionForwardWorkspaceSize(cudnnHandle_t handle,
                                        const cudnnTensorDescriptor_t xDesc,
                                        const cudnnFilterDescriptor_t wDesc,
                                        const cudnnConvolutionDescriptor_t convDesc,
                                        const cudnnTensorDescriptor_t yDesc,
                                        cudnnConvolutionFwdAlgo_t algo, size_t *sizeInBytes)
```

Activation layers

- The operation is described by cudnnActivationDescriptor_t

```
cudnnStatus_t cudnnSetActivationDescriptor(cudnnActivationDescriptor_t activationDesc,
                                           cudnnActivationMode_t mode,
                                           cudnnNanPropagation_t reluNanOpt,double coef)
```

- Where:
  - mode is one of: CUDNN_ACTIVATION_{SIGMOID,RELU,TANH,CLIPPED_RELU,ELU,IDENTITY}
  - reluNanOpt specifies how NaN values are propagated
  - coef specifies the clipping for CLIPPED_RELU or the alpha coefficient for ELU

Activation layers

- The output of an activation layer is calculated with:

```
cudnnStatus_t cudnnActivationForward(cudnnHandle_t handle,
                                     cudnnActivationDescriptor_t activationDesc,
                                     const void *alpha,
                                     const cudnnTensorDescriptor_t xDesc,
                                     const void *x, const void *beta,
                                     const cudnnTensorDescriptor_t yDesc, void *y)
```

- Where:
  - activationDesc specifies the layer parameters
  - x and y are the input and output
  - xDesc and yDesc their descriptors
  - Alpha and beta are scaling factors to combine the previous value of y with the result, if desired

Pooling layers

- The pooling operations are described by a cudnnPoolingDescriptor_t object
- Configured through:

```
cudnnStatus_t cudnnSetPooling2dDescriptor(cudnnPoolingDescriptor_t poolingDesc,
                                          cudnnPoolingMode_t mode,
                                          cudnnNanPropagation_t maxpoolingNanOpt,
                                          int windowHeight, int windowWidth,
                                          int verticalPadding, int horizontalPadding,
                                          int verticalStride, int horizontalStride)
```

- Or:

```
cudnnStatus_t cudnnSetPoolingNdDescriptor(cudnnPoolingDescriptor_t poolingDesc,
                                          int nbDims, const int windowDimA[],
                                          const int paddingA[], const int strideA[])
```

Pooling parameters

- The pooling mode:
  - One of CUDNN_POOLING_{MAX,AVERAGE_COUNT_INCLUDE_PADDING, AVERAGE_COUNT_EXCLUDE_PADDING,MAX_DETERMINISTIC}
  - The average can include or exclude the zero padding from the count
  - The output of MAX can be non-deterministic if atomics are used
- The size of the window being subject to pooling
- The padding applied to the input in each dimension
- The stride for consecutive windows in each dimension

Dropout layers

- Described by a cudnnDropoutDescriptor_t, configured by:

```
cudnnStatus_t cudnnSetDropoutDescriptor(cudnnDropoutDescriptor_t dropoutDesc,
                                        cudnnHandle_t handle, float dropout,
                                        void *states, size_t stateSizeInBytes,
                                        unsigned long long seed)
```

- Where:
  - dropout specifies the probability of zeroing an input element
  - states points to GPU memory to be used to store the PRNG state
  - stateSizeInBytes is the state size
    - Use cudnnGetDropoutStateSize() to determine
  - seed the PRNG seed

Calculating dropout layers

- Performed by:

```
cudnnStatus_t cudnnDropoutForward(cudnnHandle_t handle,
                                  const cudnnDropoutDescriptor_t dropoutDesc,
                                  const cudnnTensorDescriptor_t xdesc, const void *x,
                                  const cudnnTensorDescriptor_t ydesc, void *y,
                                  void *reserveSpace, size_t reserveSpaceSizeInBytes)
```

- Where:
  - ReserveSpace points to a GPU memory area used to exchange information between forward and backward calls to this function
  - ReserveSpaceSizeInBytes is the size of the area
    - Determined by cudnnDropoutGetReserveSpaceSize()
  - Other parameters follow the usual conventions

Applying pooling

- Pooling is calculated with:

```
cudnnStatus_t cudnnPoolingForward(cudnnHandle_t handle,
                                  const cudnnPoolingDescriptor_t poolingDesc,
                                  const void *alpha, const cudnnTensorDescriptor_t xDesc,
                                  const void *x, const void *beta,
                                  const cudnnTensorDescriptor_t yDesc, void *y)
```

SoftMax layers

- Don't require a descriptor
- Calculated by:

```
cudnnStatus_t cudnnSoftmaxForward(cudnnHandle_t handle,
                                  cudnnSoftmaxAlgorithm_t algorithm,
                                  cudnnSoftmaxMode_t mode, const void *alpha,
                                  const cudnnTensorDescriptor_t xDesc, const void *x,
                                  const void *beta,
                                  const cudnnTensorDescriptor_t yDesc, void *y)
```

- Where:
  - The algorithm is one of CUDNN_SOFTMAX_{FAST,ACCURATE,LOG}, with FAST being the straightforward algorithm, ACCURATE an implementation avoiding overflows and LOG the logarithmic implementation
  - The mode is one of CUDNN_SOFTMAX_MODE_{INSTANCE,CHANNEL}, with INSTANCE aggregating by N over C, H, W, and CHANNEL aggregating by N, H, W over C
  - The other parameters follow the usual conventions

Lesson Key Points

- Heterogenous architectures motivations
- Hierarchy of Computations:
  - Threads
  - Blocks
  - Grids
- Corresponding Memory Spaces
  - Local
  - Shared
  - Global
- Synchronization Primitives
  - Implicit Barriers
  - Thread Synchronization
- NVIDIA GPUs and CUDA:
  - Compute capability
- CUDA Programming Model:
  - Grid, Block, Thread
  - UVM
- CUDA Warp Scheduling
- CUDA Context and Streams
- CUDA Pinned Memory
- CUDA Memory Access
- CUDA Compilation and Runtime:
  - CUDA Runtime, CUDA Driver, AoT and JIT compilation
- CUDA Hardware
- CUDA Profiling and Debugging
- cuDNN and cuBLAS

- Part of this material has been adapted from the “The GPU Teaching Kit” that is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License
- Part of this material has been adapted from the video “How do Convolutional Neural Networks work?” under the Creative Commons License