CMSC 5743

Efficient Computing of Deep Neural Networks

Implementation 05: CUDA

Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk

(Latest update: September 2, 2023)

2023 Fall

Overview

Introduction

The Benefits of Using GPUs

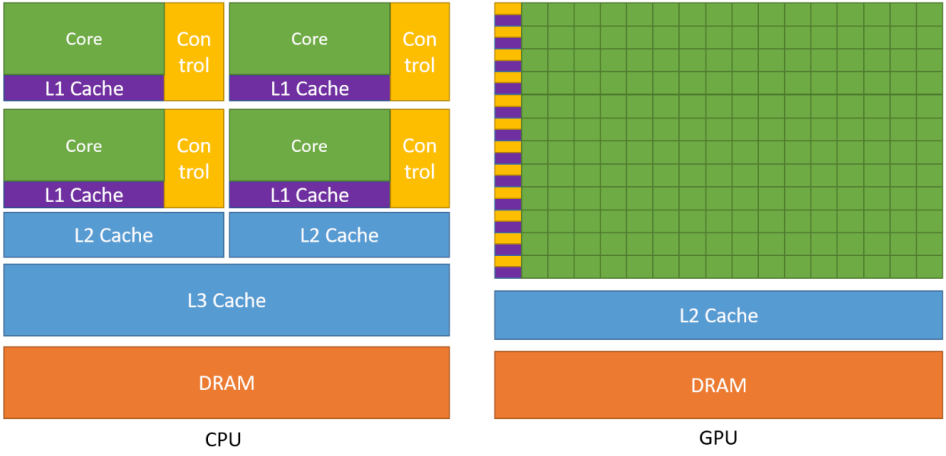
The Graphics Processing Unit (GPU)1 provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU (see GPU Applications). Other computing devices, like FPGAs, are also very energy efficient, but offer much less programming flexibility than GPUs.

This difference in capabilities between the GPU and the CPU exists because they are designed with different goals in mind. While the CPU is designed to excel at executing a sequence of operations, called a *thread*, as fast as possible and can execute a few tens of these threads in parallel, the GPU is designed to excel at executing thousands of them in parallel (amortizing the slower single-thread performance to achieve greater throughput).

The Benefits of Using GPUs

The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. The following figure shows an example distribution of chip resources for a CPU versus a GPU.

The Benefits of Using GPUs

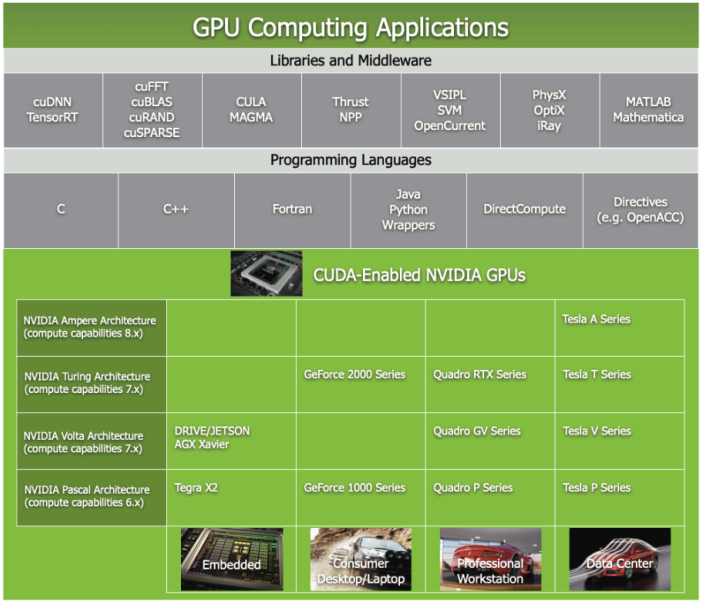


The Benefits of Using GPUs

Devoting more transistors to data processing, for example, floating-point computations, is beneficial for highly parallel computations; the GPU can hide memory access latencies with computation, instead of relying on large data caches and complex flow control to avoid long memory access latencies, both of which are expensive in terms of transistors.
In general, an application has a mix of parallel parts and sequential parts, so systems are designed with a mix of GPUs and CPUs in order to maximize overall performance. Applications with a high degree of parallelism can exploit this massively parallel nature of the GPU to achieve higher performance than on the CPU.

CUDA, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.
CUDA comes with a software environment that allows developers to use C++ as a high-level programming language. As illustrated by Figure, other languages, application programming interfaces, or directives-based approaches are supported, such as FORTRAN, DirectCompute, OpenACC.

CUDA: A General-Purpose Parallel Computing Platform and



A Scalable Programming Model

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.
The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.
At its core are three key abstractions — a hierarchy of thread groups, shared memories, and barrier synchronization — that are simply exposed to the programmer as a minimal set of language extensions.

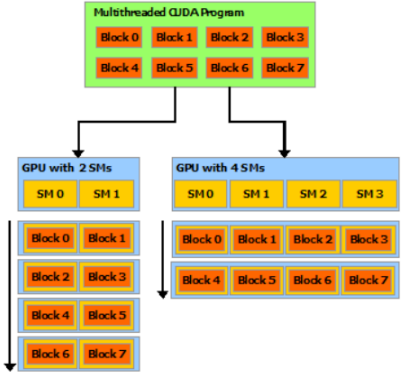
A Scalable Programming Model

At its core are three key abstractions — a hierarchy of thread groups, shared memories, and barrier synchronization — that are simply exposed to the programmer as a minimal set of language extensions.
These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors, and only the runtime system needs to know the physical multiprocessor count.

A Scalable Programming Model

This scalable programming model allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions.



Thread Hierarchy

For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (Dx, Dy), the thread ID of a thread of index (x, y) is $(x + yD_x)$; for a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x + yD_x + zD_xD_y)$.

As an example, the following code adds two matrices A and B of size NxN and stores the result into matrix C:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Programming Model

This chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C++.
An extensive description of CUDA C++ is given in **Programming Interface**.
Full code for the vector addition example used in this lecture.

Kernels

CUDA C++ extends C++ by allowing the programmer to define C++ functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C++ functions.
A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<< ... >>> execution configuration syntax (see C++ Language Extensions). Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through built-in variables.

Kernels

As an illustration, the following sample code, using the built-in variable threadIdx, adds two vectors A and B of size N and stores the result into vector C:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Here, each of the N threads that execute VecAdd() performs one pair-wise addition.

Thread Hierarchy

For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (Dx, Dy), the thread ID of a thread of index (x, y) is $(x + yD_x)$; for a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x + yD_x + zD_xD_y)$.

As an example, the following code adds two matrices A and B of size NxN and stores the result into matrix C:

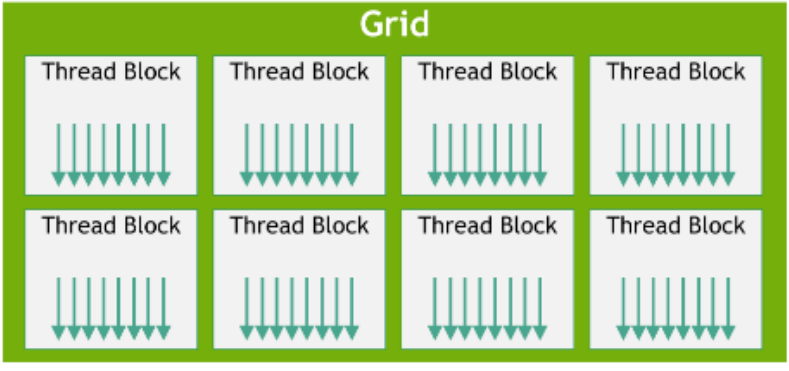
```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same streaming multiprocessor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.
However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

Thread Hierarchy

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks. The number of thread blocks in a grid is usually dictated by the size of the data being processed, which typically exceeds the number of processors in the system.



Thread Hierarchy

The number of threads per block and the number of blocks per grid specified in the <<< ... >>> syntax can be of type int or dim3. Two-dimensional blocks or grids can be specified as in the example above.
Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional unique index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.

Extending the previous MatAdd() example to handle multiple blocks, the code becomes as follows.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

A thread block size of 16×16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.
Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores.

Thread Hierarchy

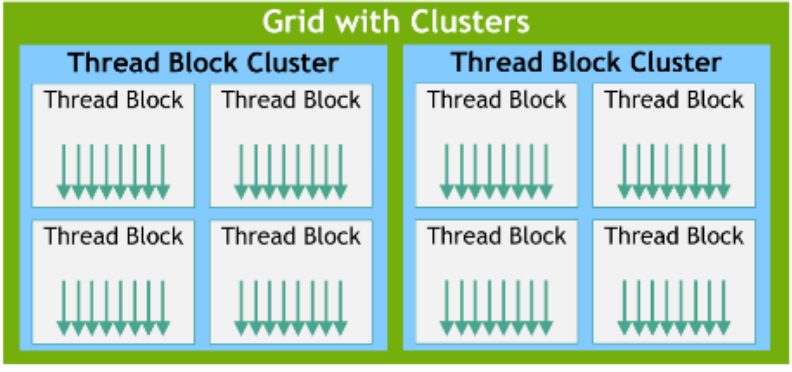
Threads within a block can cooperate by sharing data through some *shared memory* and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the *__syncthreads()* intrinsic function; *__syncthreads()* acts as a barrier at which all threads in the block must wait before any is allowed to proceed. In addition to *__syncthreads()*, the **Cooperative Groups API** provides a rich set of thread-synchronization primitives.
For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and *__syncthreads()* is expected to be lightweight.

Thread Block Clusters

With the introduction of NVIDIA Compute Capability 9.0, the CUDA programming model introduces an optional level of hierarchy called Thread Block Clusters that are made up of thread blocks. Similar to how threads in a thread block are guaranteed to be co-scheduled on a streaming multiprocessor, thread blocks in a cluster are also guaranteed to be co-scheduled on a GPU Processing Cluster (GPC) in the GPU.

Thread Block Clusters

Similar to thread blocks, clusters are also organized into a one-dimension, two-dimension, or three-dimension as illustrated by the following Figure. The number of thread blocks in a cluster can be user-defined, and a maximum of 8 thread blocks in a cluster is supported as a portable cluster size in CUDA. Note that on GPU hardware or MIG configurations which are too small to support 8 multiprocessors the maximum cluster size will be reduced accordingly.



Thread Block Clusters

A thread block cluster can be enabled in a kernel either using a compiler time kernel attribute using __cluster_dims__(X,Y,Z) or using the CUDA kernel launch API cudaLaunchKernelEx. The example below shows how to launch a cluster using compiler time kernel attribute. The cluster size using kernel attribute is fixed at compile time and then the kernel can be launched using the classical <<< ... >>>. If a kernel uses compile-time cluster size, the cluster size cannot be modified when launching the kernel.

```
// Kernel definition
// Compile time cluster size 2 in X-dimension and 1 in Y and Z dimension
__global__ void __cluster_dims__(2, 1, 1) cluster_kernel(float *input, float* output)
{

}

int main()
{
    float *input, *output;
    // Kernel invocation with compile time cluster size
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);

    // The grid dimension is not affected by cluster launch, and is still enumerated
    // using number of blocks.
    // The grid dimension must be a multiple of cluster size.
    cluster_kernel<<<numBlocks, threadsPerBlock>>>(input, output);
}
```

A thread block cluster size can also be set at runtime and the kernel can be launched using the CUDA kernel launch API cudaLaunchKernelEx. The code example below shows how to launch a cluster kernel using the extensible API.

Thread Block Clusters

```cpp
// Kernel definition
// No compile time attribute attached to the kernel
__global__ void cluster_kernel(float *input, float* output)
{

}

int main()
{
    float *input, *output;
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    cluster_kernel<<<numBlocks, threadsPerBlock>>>();
    // Kernel invocation with runtime cluster size
    {
        cudaLaunchConfig_t config = {0};
        // The grid dimension is not affected by cluster launch, and is still enumerated
        // using number of blocks.
        // The grid dimension should be a multiple of cluster size.
        config.gridDim = numBlocks;
        config.blockDim = threadsPerBlock;

        cudaLaunchAttribute attribute[1];
        attribute[0].id = cudaLaunchAttributeClusterDimension;
        attribute[0].val.clusterDim.x = 2; // Cluster size in X-dimension
        attribute[0].val.clusterDim.y = 1;
        attribute[0].val.clusterDim.z = 1;
        config.attrs = attribute;
        config.numAttrs = 1;

        cudaLaunchKernelEx(&config, cluster_kernel, input, output);
    }
}
```

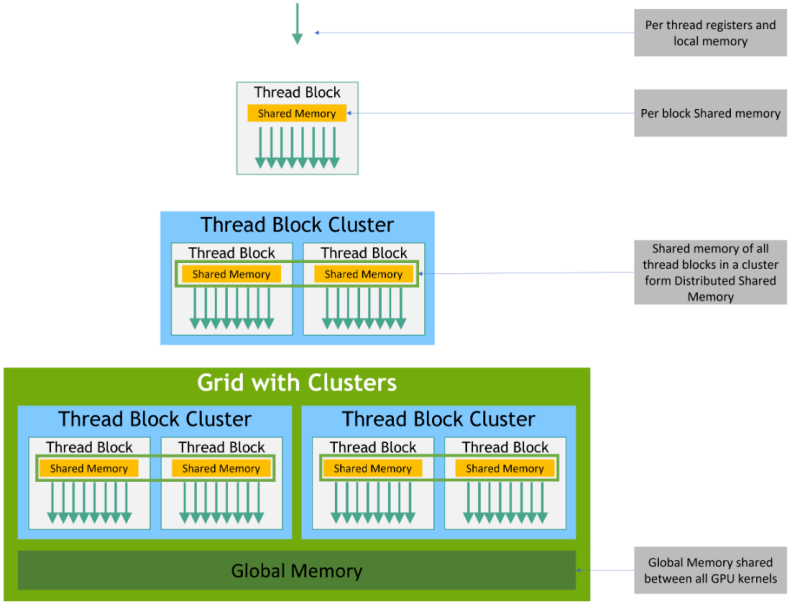
Thread Block Clusters

In GPUs with compute capability 9.0, all the thread blocks in the cluster are guaranteed to be co-scheduled on a single GPU Processing Cluster (GPC) and allow thread blocks in the cluster to perform hardware-supported synchronization using the **Cluster Group** API cluster.sync(). Cluster group also provides member functions to query cluster group size in terms of number of threads or number of blocks using num_threads() and num_blocks() API respectively. The rank of a thread or block in the cluster group can be queried using dim_threads() and dim_blocks() API respectively.
Thread blocks that belong to a cluster have access to the Distributed Shared Memory. Thread blocks in a cluster have the ability to read, write, and perform atomics to any address in the distributed shared memory. Distributed Shared Memory gives an example of performing histograms in distributed shared memory.

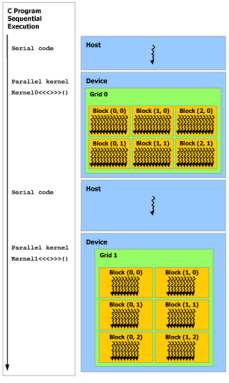
Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. Thread blocks in a thread block cluster can perform read, write, and atomics operations on each other's shared memory. All threads have access to the same global memory.
There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats.

Memory Hierarchy



As illustrated by the following figure, the CUDA programming model assumes that the CUDA threads execute on a physically separate *device* that operates as a coprocessor to the *host* running the C++ program. This is the case, for example, when the kernels execute on a GPU and the rest of the C++ program executes on a CPU.



The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory.
Unified Memory provides managed memory to bridge the host and device memory spaces. Managed memory is accessible from all CPUs and GPUs in the system as a single, coherent memory image with a common address space. This capability enables oversubscription of device memory and can greatly simplify the task of porting applications by eliminating the need to explicitly mirror data on host and device. See Unified Memory Programming for an introduction to Unified Memory.

Asynchronous SIMT Programming Model

In the CUDA programming model a thread is the lowest level of abstraction for doing a computation or a memory operation. Starting with devices based on the NVIDIA Ampere GPU architecture, the CUDA programming model provides acceleration to memory operations via the asynchronous programming model. The asynchronous programming model defines the behavior of asynchronous operations with respect to CUDA threads.
The asynchronous programming model defines the behavior of Asynchronous Barrier for synchronization between CUDA threads. The model also explains and defines how cuda::memcpy_async can be used to move data asynchronously from global memory while computing in the GPU.

Asynchronous Operations

An asynchronous operation is defined as an operation that is initiated by a CUDA thread and is executed asynchronously as-if by another thread. In a well formed program one or more CUDA threads synchronize with the asynchronous operation. The CUDA thread that initiated the asynchronous operation is not required to be among the synchronizing threads.
Such an asynchronous thread (an as-if thread) is always associated with the CUDA thread that initiated the asynchronous operation. An asynchronous operation uses a synchronization object to synchronize the completion of the operation. Such a synchronization object can be explicitly managed by a user (e.g., cuda::memcpy_async) or implicitly managed within a library (e.g., (cooperative_groups::memcpy_async).

These synchronization objects can be used at different thread scopes. A scope defines the set of threads that may use the synchronization object to synchronize with the asynchronous operation. The following table defines the thread scopes available in CUDA C++ and the threads that can be synchronized with each.

Thread Scope	Description
cuda::thread_scope::thread_scope_thread	Only the CUDA thread which initiated asynchronous operations synchronizes.
cuda::thread_scope::thread_scope_block	All or any CUDA threads within the same thread block as the initiating thread synchronizes.
cuda::thread_scope::thread_scope_device	All or any CUDA threads in the same GPU device as the initiating thread synchronizes.
cuda::thread_scope::thread_scope_system	All or any CUDA or CPU threads in the same system as the initiating thread synchronizes.

These thread scopes are implemented as extensions to standard C++ in the CUDA Standard C++ library.

Programming Interface

Programming Interface

CUDA C++ provides a simple path for users familiar with the C++ programming language to easily write programs for execution by the device.

It consists of a minimal set of extensions to the C++ language and a runtime library.
The core language extensions have been introduced in **Programming Model**. They allow programmers to define a kernel as a C++ function and use some new syntax to specify the grid and block dimension each time the function is called.

The runtime is introduced in CUDA Runtime. It provides C and C++ functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc. A complete description of the runtime can be found in the CUDA reference manual.

Compilation with NVCC Compiler

Kernels can be written using the CUDA instruction set architecture, called PTX, which is described in the PTX reference manual. It is however usually more effective to use a high-level programming language such as C++. In both cases, kernels must be compiled into binary code by nvcc to execute on the device.
nvcc is a compiler driver that simplifies the process of compiling C++ or PTX code: It provides simple and familiar command line options and executes them by invoking the collection of tools that implement the different compilation stages. This section gives an overview of nvcc workflow and command options. A complete description can be found in the nvcc user manual.

Compilation Workflow: Offline Compilation

Source files compiled with nvcc can include a mix of host code (i.e., code that executes on the host) and device code (i.e., code that executes on the device). The basic workflow consists in separating device code from host code and then:

- compiling the device code into an assembly form (PTX code) and/or binary form (cubin object),
- modifying the host code by replacing the <<< ... >>> syntax introduced in Kernels by the necessary CUDA runtime function calls to load and launch each compiled kernel from the PTX code and/or cubin object.

The modified host code is output either as C++ code that is left to be compiled using another tool or as object code directly by letting nvcc invoke the host compiler during the last compilation stage.

Compilation Workflow: Just-in-Time Compilation

Any PTX code loaded by an application at runtime is compiled further to binary code by the device driver. This is called just-in-time compilation. Just-in-time compilation increases application load time, but allows the application to benefit from any new compiler improvements coming with each new device driver. It is also the only way for applications to run on devices that did not exist at the time the application was compiled.

Compilation Workflow: Just-in-Time Compilation

When the device driver just-in-time compiles some PTX code for some application, it automatically caches a copy of the generated binary code in order to avoid repeating the compilation in subsequent invocations of the application. The cache - referred to as compute cache - is automatically invalidated when the device driver is upgraded, so that applications can benefit from the improvements in the new just-in-time compiler built into the device driver.

As an alternative to using nvcc to compile CUDA C++ device code, NVRTC can be used to compile CUDA C++ device code to PTX at runtime. NVRTC is a runtime compilation library for CUDA C++; more information can be found in the NVRTC User guide.

PTX Compatibility

Some PTX instructions are only supported on devices of higher compute capabilities. For example, Warp Shuffle Functions are only supported on devices of compute capability 5.0 and above. The -arch compiler option specifies the compute capability that is assumed when compiling C++ to PTX code. So, code that contains warp shuffle, for example, must be compiled with -arch=compute_30 (or higher).

Which PTX and binary code gets embedded in a CUDA C++ application is controlled by the -arch and -code compiler options or the -gencode compiler option as detailed in the nvcc user manual. For example,

```
nvcc x.cu
        -gencode arch=compute_50,code=sm_50
        -gencode arch=compute_60,code=sm_60
        -gencode arch=compute_70,code=\"compute_70,sm_70\"
```

PTX Compatibility

The Volta architecture introduces Independent Thread Scheduling which changes the way threads are scheduled on the GPU. For code relying on specific behavior of **SIMT scheduling** in previous architectures, Independent Thread Scheduling may alter the set of participating threads, leading to incorrect results. To aid migration while implementing the corrective actions detailed in **Independent Thread Scheduling**, Volta developers can opt-in to Pascal's thread scheduling with the compiler option combination -arch=compute_60 -code=sm_70.

The nvcc user manual lists various shorthands for the -arch, -code, and -gencode compiler options. For example, -arch=sm_70 is a shorthand for -arch=compute_70 -code=compute_70,sm_70.

C++ and 64-bit Compatibility

The front end of the compiler processes CUDA source files according to C++ syntax rules. Full C++ is supported for the host code.

The 64-bit version of nvcc compiles device code in 64-bit mode (i.e., pointers are 64-bit). Device code compiled in 64-bit mode is only supported with host code compiled in 64-bit mode.

The runtime is implemented in the cudart library, which is linked to the application, either statically via cudart.lib or libcudart.a, or dynamically via cudart.dll or libcudart.so. Applications that require cudart.dll and/or cudart.so for dynamic linking typically include them as part of the application installation package. It is only safe to pass the address of CUDA runtime symbols between components that link to the same instance of the CUDA runtime.

All its entry points are prefixed with cuda.

Initialization of Runtime

There is no explicit initialization function for the runtime; it initializes the first time a runtime function is called (more specifically any function other than functions from the error handling and version management sections of the reference manual). One needs to keep this in mind when timing runtime function calls and when interpreting the error code from the first call into the runtime.

The runtime creates a CUDA context for each device in the system. This context is the primary context for this device and is initialized at the first runtime function which requires an active context on this device. It is shared among all the host threads of the application. As part of this context creation, the device code is just-in-time compiled if necessary and loaded into device memory. This all happens transparently. If needed, for example, for driver API interoperability, the primary context of a device can be accessed from the driver API.

Initialization of Runtime

When a host thread calls cudaDeviceReset(), this destroys the primary context of the device the host thread currently operates on. The next runtime function call made by any host thread that has this device as current will create a new primary context for this device.

- The CUDA interfaces use global state that is initialized during host program initiation and destroyed during host program termination. The CUDA runtime and driver cannot detect if this state is invalid, so using any of these interfaces (implicitly or explicitly) during program initiation or termination after main) will result in undefined behavior.
- As of CUDA 12.0, cudaSetDevice() will now explicitly initialize the runtime after changing the current device for the host thread. Previous versions of CUDA delayed runtime initialization on the new device until the first runtime call was made after cudaSetDevice(). This change means that it is now very important to check the return value of cudaSetDevice() for initialization errors.

Device Memory

As mentioned in Heterogeneous Programming, the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory.

Device memory can be allocated either as linear memory or as CUDA arrays.

CUDA arrays are opaque memory layouts optimized for texture fetching.

Linear memory is allocated in a single unified address space, which means that separately allocated entities can reference one another via pointers, for example, in a binary tree or linked list. The size of the address space depends on the host system (CPU) and the compute capability of the used GPU.

Device Memory

Linear memory is typically allocated using cudaMalloc() and freed using cudaFree() and data transfer between host memory and device memory are typically done using cudaMemcpy(). In the vector addition code sample of Kernels, the vectors need to be copied from host memory to device memory:

Device Memory

```
// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Linear memory can also be allocated through cudaMallocPitch() and cudaMalloc3D(). These functions are recommended for allocations of 2D or 3D arrays as it makes sure that the allocation is appropriately padded to meet the alignment requirements described in Device Memory Accesses, therefore ensuring best performance when accessing the row addresses or performing copies between 2D arrays and other regions of device memory (using the cudaMemcpy2D() and cudaMemcpy3D() functions). The returned pitch (or stride) must be used to access array elements. The following code sample allocates a **width** $\times$ **height** 2D array of floating-point values and shows how to loop over the array elements in device code:

Device Memory

```
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch,
                width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code
__global__ void MyKernel(float* devPtr,
                         size_t pitch, int width, int height)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c) {
            float element = row[c];
        }
    }
}
```

Device Memory

The following code sample allocates a **width** $\times$ **height** $\times$ **depth** 3D array of floating-point values and shows how to loop over the array elements in device code:

```
// Host code
int width = 64, height = 64, depth = 64;
cudaExtent extent = make_cudaExtent(width * sizeof(float),
                                    height, depth);
cudaPitchedPtr devPitchedPtr;
cudaMalloc3D(&devPitchedPtr, extent);
MyKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);

// Device code
__global__ void MyKernel(cudaPitchedPtr devPitchedPtr,
                         int width, int height, int depth)
{
    char* devPtr = devPitchedPtr.ptr;
    size_t pitch = devPitchedPtr.pitch;
    size_t slicePitch = pitch * height;
    for (int z = 0; z < depth; ++z) {
        char* slice = devPtr + z * slicePitch;
        for (int y = 0; y < height; ++y) {
            float* row = (float*)(slice + y * pitch);
            for (int x = 0; x < width; ++x) {
                float element = row[x];
            }
        }
    }
}
```

Device Memory

The reference manual lists all the various functions used to copy memory between linear memory allocated with cudaMalloc(), linear memory allocated with cudaMallocPitch() or cudaMalloc3D(), CUDA arrays, and memory allocated for variables declared in global or constant memory space. The following code sample illustrates various ways of accessing global variables via the runtime API:

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
cudaMemcpyFromSymbol(data, constData, sizeof(data));

__device__ float devData;
float value = 3.14f;
cudaMemcpyToSymbol(devData, &value, sizeof(float));

__device__ float* devPointer;
float* ptr;
cudaMalloc(&ptr, 256 * sizeof(float));
cudaMemcpyToSymbol(devPointer, &ptr, sizeof(ptr));
```

cudaGetSymbolAddress() is used to retrieve the address pointing to the memory allocated for a variable declared in global memory space. The size of the allocated memory is obtained through cudaGetSymbolSize().