2023 NSF-sponsored Workshop on Deep Learning Systems in Advanced GPU Cyberinfrastructure

NVIDIA cuBLAS / NVIDIA cuDNN / NVIDIA NCCL
NVIDIA Nsight Systems / NVIDIA Nsight Compute

Dr. Iraklis Anagnostopoulos

July 25, 2023

Contents

Graphics Processing Unit (GPUs)

1.1 Traditional Computing

Traditional single-core computing represents the earliest form of computer processors. As the name implies, single-core CPUs have one core, essentially a processing unit, which conducts all of the machine's computations. It can execute one thread at a time, following the basic Von Neumann architecture and the principle of sequential processing, also known as "serial" computing.

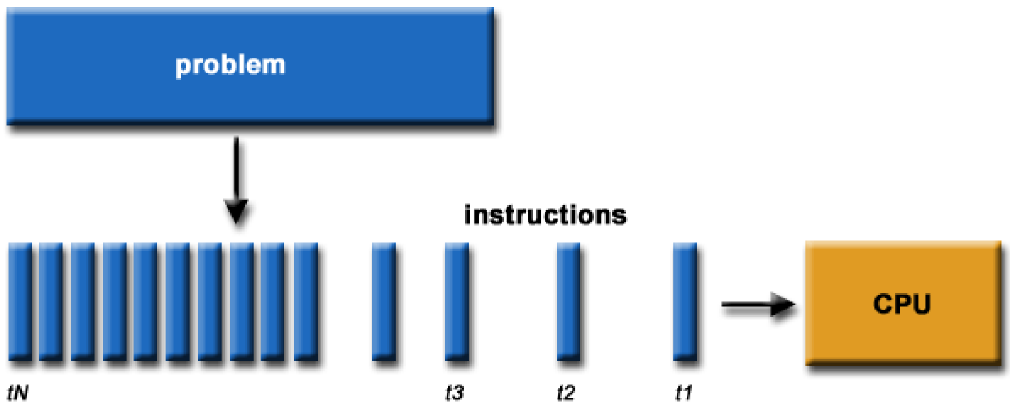


Figure 1.1: Execution instruction on traditional computing

This type of computing relies heavily on the "fetch-decode-execute" cycle. The CPU fetches an instruction from the primary memory, decodes it to ascertain the required course of action, and then executes it. Throughout this process, it keeps track of the next instruction in line with the aid of a program counter.

However, the fetch-decode-execute loop carries out these steps one at a time for each thread of execution. When an application is broken down into threads, a single-core CPU would process these threads one by one, sequentially. The CPU can switch quickly between these threads, giving an illusion of parallelism, a process known as context switching. But it results in overhead because resources must be saved and loaded each time the CPU switches between processes.

Another integral part of single-core CPUs is the use of various levels of cache memory—L1, L2, and sometimes L3. These storage areas on the CPU help speed up processes by storing values frequently used or reducing the time it takes for the CPU to access primary memory.

Considered the norm up to the early 2000s, the single-core CPU hit a wall known as the ”power wall” as manufacturers tried to increase clock speeds and transistor counts to enhance performance. This resulted in excessive heat production and high energy consumption. These issues, along with Moore’s law - which observes that the number of transistors on an affordable CPU would double every two years - drove the industry to develop multi-core processors and leverage parallel computing.

Despite the change, the importance and relevance of single-core computing is still evident today. While they might not perform well with multitasking, single-core CPUs still function well with individual tasks and applications that are not designed for parallel processing. They are also useful in lower power devices and simpler electronic systems where multi-core processing may not be needed or cost-effective.

In conclusion, single-core computing signifies a crucial era in computer history when the focus was on increasing clock speed and improving sequential execution performance. It provided the technological foundation that has led to more advanced processing units and paved the way to a new era: The era of parallel computing and multi-core processors.

1.2 Parallel Computing

Movement from single-core computing to parallel computing elevates a revolutionary change in the computing world. The transition was influenced by the need for enhanced computing speed, power, and efficiency. While single-core processors performed tasks serially, parallel computing accommodates execution of multiple instructions simultaneously, leading to faster and more efficient processing.

The early years of digital computing were characterized by relatively large, expensive, and slow single-core CPUs. A single-core CPU could only process one task at a time. To increase speed and efficiency, computer engineers and manufacturers began exploring ways to fit multiple cores into a single CPU, enabling multiple tasks to be processed concurrently.

The first dual-core processor, IBM’s POWER4, emerged in 2001. This technology boosted computer performance by almost twofold compared to single-core CPUs. AMD introduced their version in 2005, and Intel followed a year later with its Core 2 Duo. These companies later developed more advanced architectures and processors with 4, 6, 8, and even 16 cores, dramatically enhancing application performance and power efficiency.

The move towards parallel computing allowed various innovations in architecture. Multi-core processors, which include two or more independent cores in one chip, came into the picture. They provided a significant boost in processing speed without increasing clock rates, thus reducing power consumption. The evolution also saw the development of Graphic Processing Units (GPUs) as additional processors. The matrix-style computing of GPUs enables multiple calculations to run at the same time, improving the speed and efficiency of calculations tremendously, which revealed another side of parallel processing.

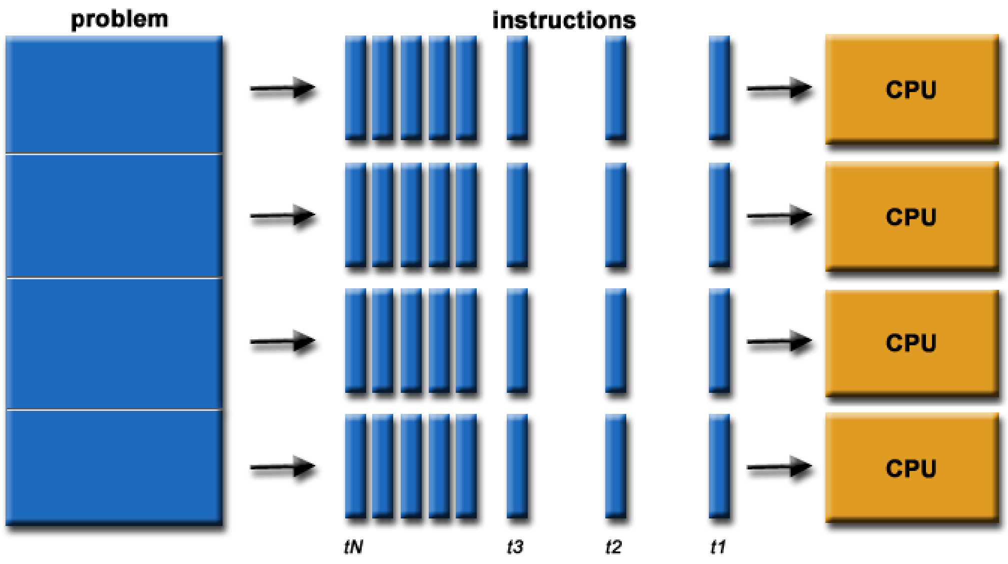


Figure 1.2: Execution instruction on traditional computing

One other innovation was the development of multithreading technology. Multithreading improved the way cores process and execute instructions, enabling a single core to manage multiple threads and allowing the CPU to accomplish more work faster, boosting overall system performance.

1.2.1 Advantages of Parallel Computing

Parallel computing provides several benefits over single-core computing. First, it dramatically increases program speed. Since many calculations can be performed simultaneously, the time to reach a solution is significantly reduced. This increase in speed is critical for big data analysis, scientific simulations, artificial intelligence applications, and other complex computation tasks.

Parallel computing also enhances efficiency and lowers energy consumption. Multiple core processors can run at lower clock speeds and still outperform a high-speed single-core CPU, saving power and producing less heat. This aspect is particularly vital for modern data centers that house thousands of servers.

Utilizing parallel computing also allows for problem-solving that was previously impossible or impractical. Problems that require high computational requirements or real-time processing are now feasible with the advent of parallel processing.

1.2.2 Challenges of Parallel Computing

Despite its numerous benefits, parallel computing does pose several challenges. The primary challenge lies in programming. Writing programs that can effectively utilize multiple cores simultaneously is more complex than writing for a single-core system. It requires managing tasks, synchronizing processes, and ensuring data integrity, all of which can dramatically increase the complexity of a program.

Another challenge is posed by Amdahl's Law, which states that the maximum improvement in performance gained from parallel processing is limited by the proportion of the program that can't be parallelized. That means, despite increasing the number of processors, there will always be a limit to the speedup in performance.

Hardware compatibility is another significant issue. Not all software applications can take advantage of multiple cores, and not all computer systems have the necessary memory or architecture to support parallel processing efficiently.

Inefficient parallel computing can also result in race conditions, where two or more operations must execute in a specific sequence, but the program's coded instructions don't enforce this order, resulting in erratic behavior.

1.2.3 Conclusion

The evolution from single-core to parallel computing represents a significant step forward in the pursuit of speed, efficiency, and power in computer processing. Innovations in architecture have played a pivotal role in this transition, enabling the handling of simultaneous tasks.

While parallel computing comes with substantial advantages, it introduces a new set of challenges, primarily in programming and hardware compatibility. However, the continuing evolution of technology, along with growing expertise in the field, promises to help overcome these challenges, catalyzing the full potential of parallel computing.

1.3 GPUs

The era of ubiquitous computing has led to an increasing demand for high-power processors. While Central Processing Units (CPUs) have traditionally been the heart of computing devices, Graphic Processing Units (GPUs) have surfaced as powerful computational devices. The fundamental difference in the architecture and processing capabilities between CPUs and GPUs has reshaped the computational landscape, making GPUs a dominant force in modern systems.

1.3.1 The Boom of GPU Computing

With the advent of compute-intensive applications such as machine learning, artificial intelligence, and high-performance computing, GPUs have gained prominence for their ability to deliver high threading capabilities and outstanding computational speeds. The complex nature of these applications necessitates significant parallel computing power alongside massive data manipulation, a capability inherent within GPUs.

GPUs have been designed initially for rendering high-quality graphics quickly. Over time, developers realized that the inherent parallel processing capabilities of GPUs could be harnessed for non-graphic related tasks. This realization gave birth to the field of General Purpose computing on GPUs (GPGPU), opening new avenues for multi-dimensional computing.

1.3.2 Difference between CPUs and GPUs

The principal distinction between CPUs and GPUs lies in their computational structures and processing styles. CPU architecture is designed to excel in tasks that require high clock speed and complex computations. CPUs are adept at executing single-threaded tasks and can switch tasks with minimal latency, thanks to their larger memory cache. CPUs are optimized to have high "Instructions Per Cycle" (IPC), providing faster processing speeds for tasks that do not require heavy parallelization.

Taking a starkly contrasting approach, GPUs are designed for significant data parallelism. Given their origin in graphics processing, they are built to handle hundreds of threads simultaneously, optimal for computations that can be executed in parallel. While a CPU might consist of 4, 8, or 16 cores, a typical GPU consists of thousands of smaller cores working together, enabling GPUs to handle larger datasets and perform complex calculations far more quickly than CPUs.

Furthermore, the memory architecture also varies between CPUs and GPUs. CPUs have hierarchical, segmented memory with a comparatively larger cache, optimizing for low latency data access. Conversely, GPUs possess a large, global memory space with a much smaller cache, focusing on the high throughput of bandwidth-intensive computations.

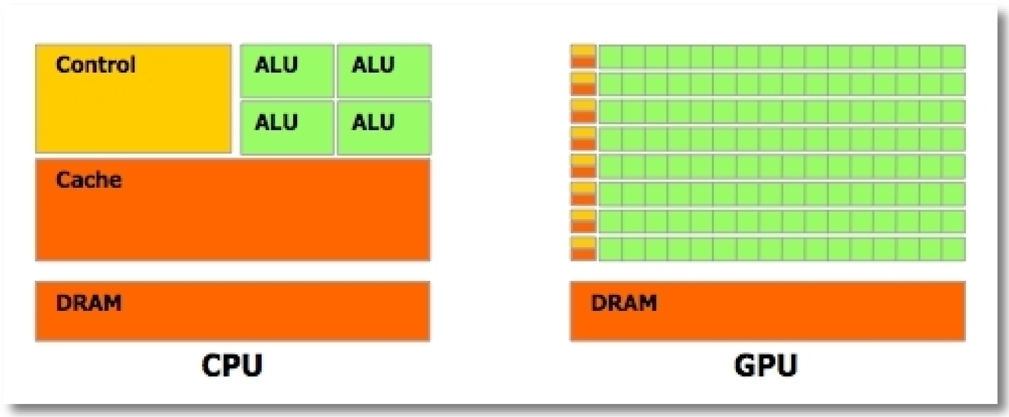


Figure 1.3: Qualitative differences between CPUs and GPUs

1.3.3 The Dominance of GPUs in Modern Systems

The dominance of GPUs in today's systems can primarily be attributed to their superior computation capabilities, particularly related to parallel processing.

In the realm of machine learning and artificial intelligence, computational demands surpass the speed and parallelization capabilities provided by the typical CPU. Machine learning algorithms involve intricate mathematical models, often requiring billions of calculations during the training phase. GPUs, with their many-core architecture, offer a significant edge here, allowing parallelizable tasks to be distributed across multiple cores, thus, expediting the overall computation process.

Hardware-wise, a higher number of transistors in GPUs means they can handle more information, making them far more suitable for applications such as deep learning, which require manipulation of large datasets in real-time.

Real-world applications have also seen a tremendous benefit from GPUs. Advanced gaming systems, for example, require high-quality, real-time graphic-rendering capabilities that are best served by GPUs. Scientific simulations that often involve performing a massive number of similar calculations simultaneously have also found a valuable ally in GPUs. Further, GPUs are increasingly utilized in cloud services, providing shared, scalable compute resources to multitudes of users.

Although GPUs have made significant headway, it does not imply that CPUs are no longer necessary. While the former is specialized for parallelism, the latter excels in executing serial computations quickly. Essential tasks, like running the operating system and other system-related tasks that do not lend themselves well to parallel processing, still rely heavily on CPUs.

1.3.4 Challenges and Future Prospects

Despite their advantages, dominant use of GPUs comes with its challenges. A core problem lies in programming GPUs for general-purpose computing. While there have been advances in programming languages and tools that bridge this gap (like CUDA and OpenCL), harnessing their full potential still requires a deep understanding of the underlying hardware and architecture.

There are also bottlenecks related to data transfer speeds between CPU and GPU and in the overall memory bandwidth, which can drastically constrain GPU performance. Further, while GPUs are power-efficient, they still generate significant heat which poses challenges related to heat dissipation and cooling.

Despite the challenges, the evolution and adoption of GPUs for general-purpose computing will no doubt continue to progress. Developments in GPU architectures focusing on variable precision matrix multiplication, enhanced memory allocation, GPU-direct storage are likely to mitigate the current challenges and make it easier for developers to integrate them into applications.

1.3.5 Conclusion

The emergence and dominance of GPUs in the computational world signify a transformative change, particularly significant for advanced computational applications. While they function differently from CPUs, the complementary nature of their capabilities

CUDA

NVIDIA CUDA, short for Compute Unified Device Architecture, is a parallel computing platform and application programming interface (API) model developed by NVIDIA. Introduced in 2007, CUDA was strategically created to leverage the parallel computing power of NVIDIA's Graphics Processing Units (GPUs). The fundamental goal was to allow developers to use a CUDA-enabled GPU to perform general computation tasks traditionally conducted by the Central Processing Unit (CPU).

Acting as the bridge between the software and the GPU hardware, CUDA provides a standard interface for developers to extract the GPU's high-performance computing capabilities. It works with many high-level programming languages like C, C++, and Fortran, which means it can be used conveniently by a wide variety of programmers.

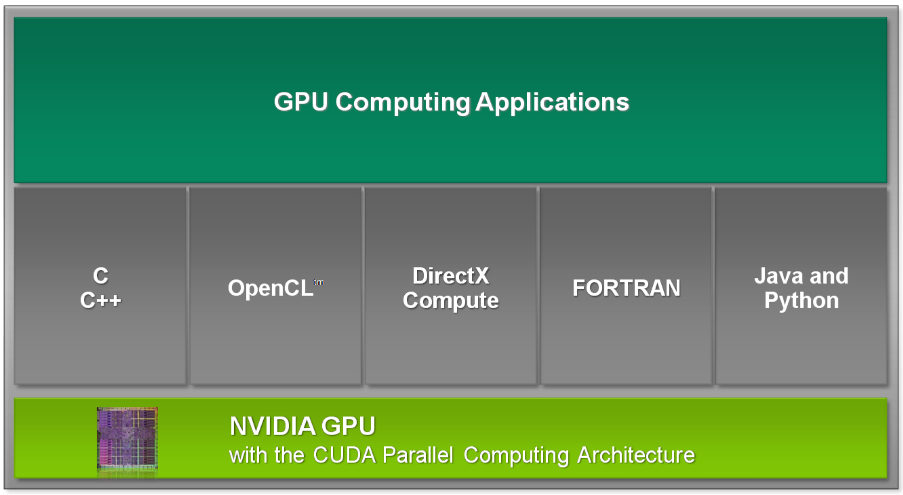

GPU Computing Applications
C
C++
OpenCL™
DirectX Compute
FORTRAN
Java and Python
NVIDIA GPU
with the CUDA Parallel Computing Architecture

Figure 2.1: NVIDIA CUDA

Essentially, CUDA allows the transformation of the GPU from solely a rendering device to a full-fledged, general-purpose processor capable of handling complex computational problems. This flexibility is particularly useful for applications requiring significant processing power, such as Artificial Intelligence (AI), machine learning, high-definition video rendering, scientific simulations, and 3D modeling.

The structure of CUDA involves a hierarchy of thread groups, shared memories, and barrier synchronization, providing developers with a familiar, flexible, and easier-to-handle environment. The CUDA software suite includes libraries, compilers, and tools designed to help developers get the most from their GPU hardware.

In the CUDA architecture, the individual processors in the GPU are referred to as cores, each capable of executing multiple threads simultaneously. This architecture makes CUDA exceptionally effective for problems that can be solved using parallel processing, where multiple calculations are performed at once. It ensures maximum usage of the GPU's processing power, resulting in an unparalleled performance boost.

One of the key benefits of CUDA is its scalability. It works smoothly with various NVIDIA GPUs with the code executing on various CUDA-enabled GPUs, irrespective of the model. CUDA technology is also compatible across devices; it can be used on laptops, workstations, supercomputers, or clusters with one or more GPUs.

However, using CUDA and exploiting the power of GPU parallel computing does not come without challenges. Implementing CUDA requires significant alterations to the conventional programming perspective as the execution flow is inherently parallel as opposed to sequential.

Yet, with the increased demand for high-performance computing in a digital-driven world, the utilization of CUDA technology remains vital. As a painless and cost-effective way to maximize both performance and efficiency, CUDA technology signifies a colossal stride towards a new era of parallel computing with GPUs.

2.1 Processing Flow

CUDA processing flow centers around the host and the device cooperating to solve complex computational problems. The CPU essentially manages the I/O, user interface, and resource allocation, while the GPU provides the computational horsepower needed for hefty calculations. Following are the key steps:

1. CPU initiates by executing a series of functions, called "kernels," and delegating computationally intense data-parallel portions of applications to the GPU.
2. Data for processing is transferred from the host memory to the device memory.
3. The kernel code is loaded into each GPU thread for parallel execution.
4. The GPU concurrently executes the threads, operating on different data.
5. Upon execution, the processed data is transferred back to host memory from the device memory.

This workflow enables the entire system to function in a concurrent manner. While the CPU undertakes tasks best suited for sequential processing, the GPU

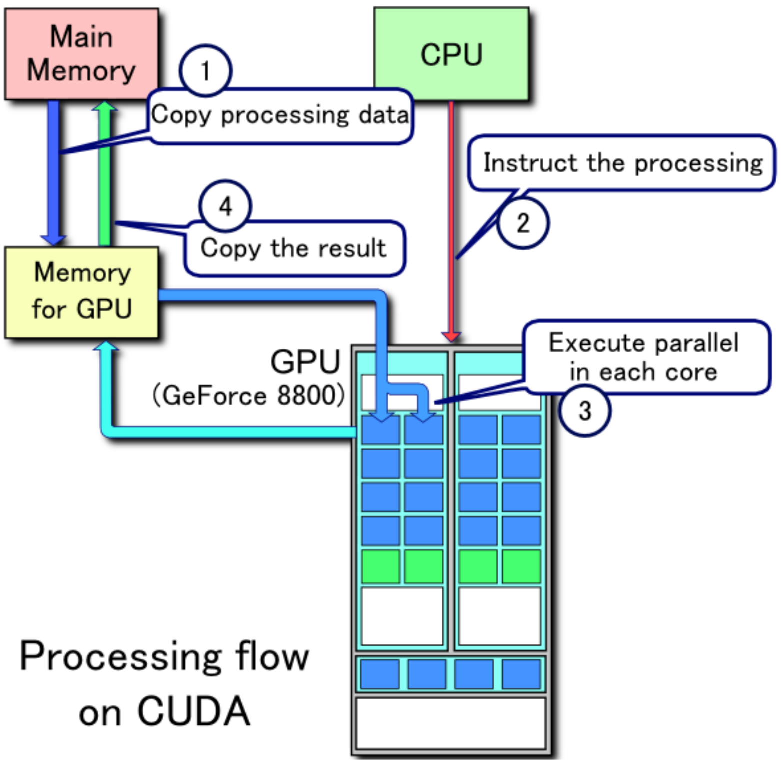


Figure 2.2: CUDA processing flow

performs those suitable for parallel execution.

2.2 Programming Model

CUDA's programming model reflects its underlying hardware's parallelism, hierarchy of the memory system, and thread execution model. The model introduces a hierarchy of thread groups, shared memories, and barrier synchronization, which is simply an automatic way of efficiently managing parallel execution.

In the CUDA model, the computation is performed by thousands of threads - light-weight, independent entities that execute the same kernel code concurrently. Threads are grouped into structures named "blocks" and "grids." These two entities define the three-layer CUDA programming model hierarchy:

- Thread: A thread represents the smallest execution unit and processes one data element.
- Block: A block is a group of threads that can cooperate with each other by sharing data through some shared memory.
- Grid: A grid contains an array of blocks that are executed independently.

CUDA kernels are written in a language similar to C/C++ without the class or template features. The CUDA framework offers built-in variables so that each

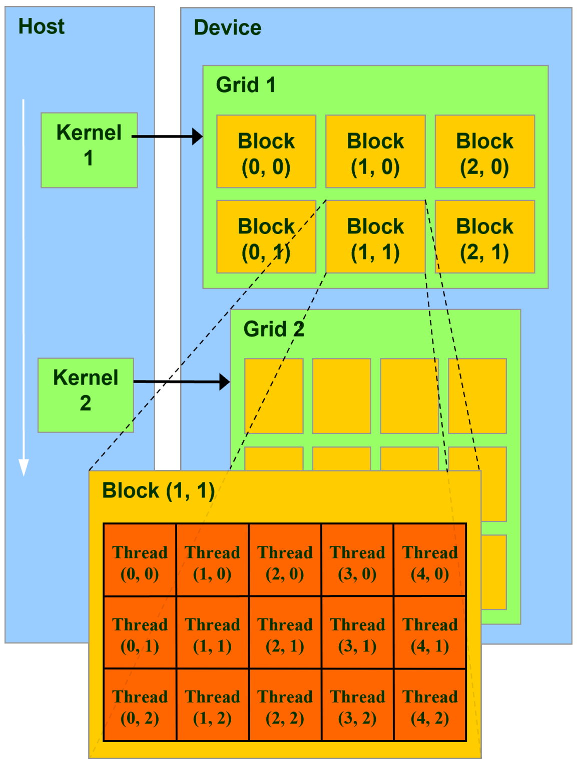


Figure 2.3: CUDA programming model

thread can identify its unique thread/block location within the grid, enabling it to fetch and operate on the appropriate data.

To recap, the CUDA processing flow and programming model consciously leverage the benefits of both CPUs and GPUs, bringing together the best of both worlds for efficient system-level computation. By co-opting the CPU's versatility and the GPU's raw power, CUDA delivers an effective single, heterogeneous computing environment.

2.3 CUDA version and GPUs

To check the CUDA version and nvcc (CUDA compiler) version installed on your system, you can follow the instructions mentioned below. These instructions are tailor-made for Linux-based systems:

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Tue_May__3_18:49:52_PDT_2022
Cuda compilation tools, release 11.7, V11.7.64
Build cuda_11.7.r11.7/compiler.31294372_0
$ nvidia-smi
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 530.41.03                 Driver Version: 530.41.03     CUDA Version: 12.1     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                                        |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce GTX 1050 Ti      Off| 00000000:01:00.0  On |                  N/A |
| 46%   37C    P0              N/A /  75W|    729MiB /  4096MiB |      5%      Default |
|                                        |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+
```

To compile a CUDA program, you would typically use the `nvcc` compiler, and the general syntax for the compilation is as below:

```
$ nvcc options -o outputfile inputfile.cu
```

Here, "options" represent any compiler options you want to include, "outputfile" is the name of the output file that will be produced, and "inputfile.cu" is the CUDA program file you are trying to compile. Check the provided `cuda_simple.cu`. The code calculates the square of an array in CPU and GPU.

Code 2.3.1: Calculation of the square of an array in CPU and GPU.

```
/*
This program first allocates memory for a large array
of integers on both the CPU and GPU. It then initializes
the array with some values. The square_cpu function squares
each element of the array on the CPU, and the square_gpu
kernel does the same on the GPU. The program measures and prints
the execution times for both the CPU and GPU computations
*/

#include <stdio.h>
#include <cuda.h>
#include <time.h>

#define N 100000000
#define BLOCK_SIZE 1024

// CPU function
void square_cpu(int *a, int n) {
    for (int i = 0; i < n; i++) {
        a[i] = a[i] * a[i];
    }
}

// GPU kernel
__global__ void square_gpu(int *a, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        a[idx] = a[idx] * a[idx];
    }
}

```

```
int main() {
    int *a_cpu, *a_gpu;
    int size = N * sizeof(int);

    // Allocate memory
    a_cpu = (int*)malloc(size);
    cudaMalloc((void**)&a_gpu, size);

    // Initialize array
    for (int i = 0; i < N; i++) {
        a_cpu[i] = i;
    }
    cudaMemcpy(a_gpu, a_cpu, size, cudaMemcpyHostToDevice);

    // Print CUDA version
    printf("CUDA version: %d.%d\n", CUDA_VERSION / 1000, (CUDA_VERSION % 100) / 10);

    // CPU execution
    clock_t start_cpu = clock();
    square_cpu(a_cpu, N);
    clock_t end_cpu = clock();

    // GPU execution
    clock_t start_gpu = clock();
    square_gpu<<<N/BLOCK_SIZE, BLOCK_SIZE>>>(a_gpu, N);
    cudaDeviceSynchronize();
    clock_t end_gpu = clock();

    // Print execution times
    double time_cpu = ((double) (end_cpu - start_cpu)) / CLOCKS_PER_SEC;
    double time_gpu = ((double) (end_gpu - start_gpu)) / CLOCKS_PER_SEC;
    printf("CPU execution time: %f seconds\n", time_cpu);
    printf("GPU execution time: %f seconds\n", time_gpu);

    // Free memory
    free(a_cpu);
    cudaFree(a_gpu);

    return 0;
}
```

Compiling and executing the program gives us the following output. Compare and discuss about it:

```
$ nvcc cuda_simple.cu -o cuda_simple
$ ./cuda_simple
CUDA version: 11.7
CPU execution time: 0.306754 seconds
GPU execution time: 0.009643 seconds
```

3

cuBLAS

3.1 BLAS

Before presenting and analyzing cuBLAS, we have to see what is Basic Linear Algebra Subprograms (BLAS). Basic Linear Algebra Subprograms (BLAS) is a specification that prescribes a set of low-level routines for performing basic operations in linear algebra, such as vector addition, scalar multiplication, dot products, linear combinations of vectors, and matrix multiplication. It was established due to the necessity for standardized building blocks for the implementation of linear algebra operations to encourage code reuse and portability.

BLAS is structured in the form of an application programming interface (API), offering a collection of standard routines that can be used in software applications to perform these operations. By utilizing BLAS routines, application designers can build programs that are portable and optimized for different hardware architectures. BLAS is divided into three levels:

1. Level 1 BLAS performs scalar, vector, and vector-vector operations. Examples of Level 1 BLAS operations include vector scaling and dot-product of two vectors.
2. Level 2 BLAS conducts matrix-vector operations, including matrix-vector multiplication and solving triangular systems.
3. Level 3 BLAS carries out matrix-matrix operations. A typical Level 3 BLAS operation is matrix multiplication.

It's essential to note that the BLAS does not implement these operations itself. Instead, it is a standardized interface to which optimized implementations can be written. There are several well-known implementations of the BLAS, including ATLAS, OpenBLAS, and Intel MKL, each providing highly optimized routines for different hardware configurations.

Many higher-level libraries and applications in numerical and scientific computing, such as LAPACK and SciPy, heavily use BLAS to perform linear algebra operations expeditiously. With BLAS, developers and researchers can write programs that scale effectively with various hardware configurations without needing to be experts in the subtle nuances of each hardware system, ultimately increasing the efficiency and portability of their software.

3.2 cuBLAS

cuBLAS is a Graphics Processing Unit (GPU)-accelerated version of the complete standard Basic Linear Algebra Subprograms (BLAS) library provided by NVIDIA. NVIDIA initially developed cuBLAS to harness the incredible computational power of its Graphics Processing Units (GPUs), and ever since, it has become a vital tool for developers and scientists working with high-performance computing.

cuBLAS leverages the parallel processing capabilities of NVIDIA GPUs to accelerate linear algebra operations, such as scalar, vector, and matrix operations. Additionally, cuBLAS is implemented on top of the NVIDIA CUDA (Compute Unified Device Architecture), which provides an interface to perform GPU-accelerated computing.

3.2.1 cuBLAS benefits

cuBLAS is characterized by the following benefits:

- Speed: cuBLAS functions are highly optimized to deliver significantly improved performance over CPU-only BLAS implementations. Linear algebra functions using cuBLAS will run faster because they take advantage of the massive parallelism in GPUs.
- Ease of Use: Incorporating cuBLAS into existing applications is straightforward. Developers can easily make their software applications run faster by replacing CPU BLAS calls with GPU-accelerated cuBLAS calls.
- Integration: cuBLAS is fully compatible with CUDA, and it works seamlessly with other CUDA libraries, allowing developers to create complex, high-performance applications.

3.2.2 What is cuBLAS good for?

cuBLAS is most beneficial when used for large-scale problems, where the tremendous parallelism of GPUs can be fully exploited, and the time required for transferring data between CPU and GPU memory is a small percentage of the overall computation time. Here are several cases where cuBLAS can be very beneficial:

- High-Performance Computing: In scenarios requiring substantial computational power, such as scientific computing, big data analytics, or complex simulations, cuBLAS can substantially speed up computation by harnessing the power of GPUs.
- Machine Learning and Deep Learning: Both these fields involve lots of matrix and vector manipulations (like in neural networks). Using cuBLAS can significantly accelerate these computations, making model training and inference faster.

- Computer Graphics: Many graphic algorithms require performing matrix operations on large datasets. cuBLAS can speed up these operations, leading to quicker rendering and improved performance.
- Real-time Analytics: Real-time analytics often involve processing large datasets instantly or in near-real-time. Through its parallel processing capabilities, cuBLAS can help achieve this.

In conclusion, cuBLAS can significantly improve performance when dealing with extensive computations involving vectors and matrices. By leveraging the power of NVIDIA's GPUs, cuBLAS offers a potent toolset to developers working with high-performance computing.

3.3 cuBLAS API

The function names in cuBLAS are similar to their BLAS counterparts, but with a prefix 'cublas'. Additionally, the functions are designed to operate on different data types, such as single-precision and double-precision floating-point numbers, as well as complex numbers:

- S, s : single precision (32 bit) real float
- D, d : double precision (64 bit) real float
- C, c : single precision (32 bit) complex float (implemented as a float2)
- Z, z : double precision (64 bit) complex float
- H, h : half precision (16 bit) real float

For example:

`cublasIsamax` → cublas "I," s, amax

- cublas : the cuBLAS prefix
- I : stands for index. Cuda naming left over from Fortran
- s : this is the single precision float variant
- amax : finds a maximum
- Returns smallest Index "i" of the earliest max element

`cublasSgemm` → cublas S gemm

- cublas : the prefix
- S : single precision real float
- gemm : general matrix-matrix multiplication

Note that for each function, a 'handle' to the cuBLAS library needs to be created using `cublasCreate()` before they can be used. After you are done with these functions, the handle must be destroyed with `cublasDestroy()`. It is also important to note that cuBLAS uses column-major storage, unlike C, which uses row-major storage.

3.3.1 cuBLAS handle

The cuBLAS library API employs a concept known as "handle" to manage the resources associated with each cuBLAS library context. A cuBLAS handle is a complete context in which all the cuBLAS routines are executed. This handle essentially encapsulates the data structures that cuBLAS uses to set up and manage resources required for each cuBLAS context.

The cuBLAS handle is of the type `cublasHandle_t`, and it maintains the cuBLAS library context. It is created to manage resources such as the hardware and software configurations, memory, CUDA context, and properties like matrix and vector parameters for the cuBLAS routines.

You begin by creating a handle using the `cublasCreate()` function which initializes the cuBLAS library and allocates hardware resources:

```
cublasHandle_t handle;
cublasCreate(&handle);
```

Then, you use this handle as the first argument for all subsequent calls to cuBLAS routines. This enables the cuBLAS library to manage various operations and resources within that specific context.

When you're done with your operations, you should destroy the cuBLAS handle using the `cublasDestroy()` function to safely deallocate the resources associated with it:

```
cublasDestroy(handle);
```

By using a handle to encapsulate the resources and context necessary for using the cuBLAS functions, the cuBLAS library allows better management of these resources and provides a straightforward interface for developers to maintain and regulate multiple contexts concurrently.

3.3.2 cuBLAS functions levels

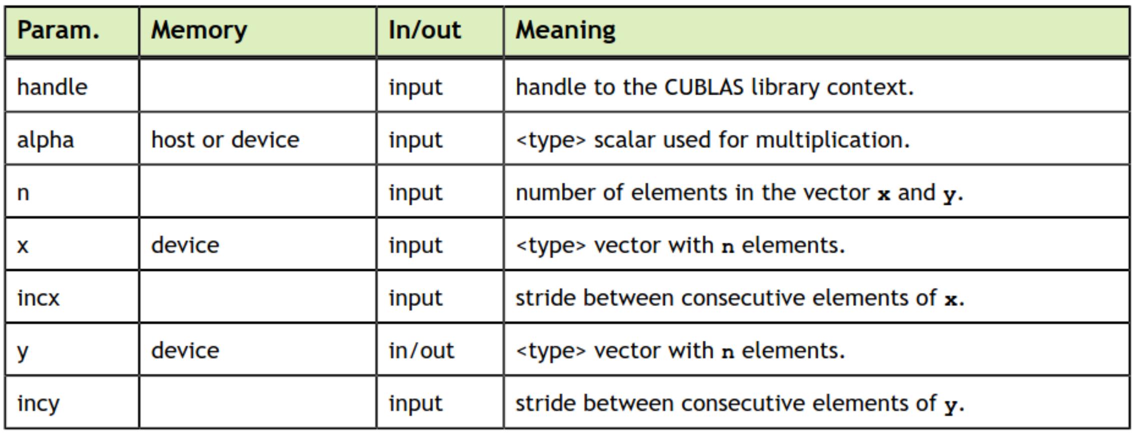
Similar to BLAS, cuBLAS is categorized into three distinct levels, each dedicated to a specific type of operation. Below is a description of each of these function levels. All functions in cuBLAS can be found here NVIDIA cuBLAS

Level 1 Functions These involve scalar, vector, and vector-vector operations. For example, `cublasSaxpy` is a Level 1 function that performs a vector operation defined as

$$y = \alpha * x + y$$

, where y and x are vectors, and α is a scalar. Other Level 1 functions include routines for dot products, vector scaling, and vector copies.

```
cublasStatus_t cublasSaxpy(cublasHandle_t handle, int n,
                           const float *alpha,
                           const float *x, int incx,
                           float *y, int incy)
```



Param.	Memory	In/out	Meaning
handle		input	handle to the CUBLAS library context.
alpha	host or device	input	<type> scalar used for multiplication.
n		input	number of elements in the vector `x` and `y`.
x	device	input	<type> vector with `n` elements.
incx		input	stride between consecutive elements of `x`.
y	device	in/out	<type> vector with `n` elements.
incy		input	stride between consecutive elements of `y`.

Figure 3.1: `cublasSaxpy` argument analysis

Level 2 Functions These consist of matrix-vector operations. A classic example is `cublasSgemv`, a Level 2 function that performs the matrix-vector operation

$$y = \alpha * op(A) * x + beta * y$$

, where A is a matrix and x, y are vectors. Other Level 2 functions include routines for solving triangular systems.

```
cublasStatus_t cublasSgemv(cublasHandle_t handle, cublasOperation_t trans,
                           int m, int n,
                           const float *alpha,
                           const float *A, int lda,
                           const float *x, int incx,
                           const float *beta,
                           float *y, int incy)
```

Level 3 Functions These involve matrix-matrix operations, and they offer the highest level of performance among the cuBLAS function levels due to their high computational intensity. `cublasSgemm` is a widely used Level 3 function. It performs the matrix-matrix multiplication operation $C = \alpha AB + \beta C$, where A, B, and C are matrices.

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
                        cublasOperation_t transa, cublasOperation_t transb,
                        int m, int n, int k,
                        const float *alpha,
                        const float *A, int lda,
                        const float *B, int ldb,
                        const float *beta,
                        float *C, int ldc)
```

In all the function names mentioned above, 'S' stands for 'single precision'. For double-precision versions, 'S' is replaced by 'D'. For single precision complex numbers, 'C' is used, and 'Z' for double precision complex numbers.

It is important to remember that all cuBLAS operations are performed on the GPU, and the data transferred from the CPU to the GPU (device memory). Hence, to avoid any performance hit due to excessive data transfers between the host and device, the operations to be performed need to be of high computational intensity. This is why Level 3 cuBLAS functions that operate on large matrices can deliver the most performance benefit.

3.4 cuBLAS example

Check the provided file `matmul.cu`. It performs matrix multiplication using the CPU, CUDA, and cuBLAS, and prints the execution time for each method

Code 3.4.1: Matrix multiplication using the CPU, CUDA, and cuBLAS

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cuda.h>
#include <cublas_v2.h>

#define N 1500
#define BLOCK_SIZE 16

// CPU function
void matmul_cpu(float *A, float *B, float *C) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0;
            for (int k = 0; k < N; k++) {
                sum += A[i * N + k] * B[k * N + j];
            }
            C[i * N + j] = sum;
        }
    }
}

// CUDA kernel
__global__ void matmul_gpu(float *A, float *B, float *C) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0;
    for (int k = 0; k < N; k++) {
        sum += A[row * N + k] * B[k * N + col];
    }
    C[row * N + col] = sum;
}

int main() {
    float *A, *B, *C, *D, *E;
    float alpha = 1.0f;
    float beta = 0.0f;
    cublasHandle_t handle;
    clock_t start, end;

    // Allocate memory
    A = (float*)malloc(N * N * sizeof(float));
    B = (float*)malloc(N * N * sizeof(float));
    C = (float*)malloc(N * N * sizeof(float));
    cudaMalloc((void**)&D, N * N * sizeof(float));
    cudaMalloc((void**)&E, N * N * sizeof(float));

    // Initialize A and B... using values between 1 and 10:
    srand(time(0));  // seed random number generator
    for (int i = 0; i < N * N; i++) {
        A[i] = 1 + static_cast <float> (rand()) /( static_cast <float> (RAND_MAX/(10-1)));
        B[i] = 1 + static_cast <float> (rand()) /( static_cast <float> (RAND_MAX/(10-1)));
    }

    // CPU execution
    start = clock();
    matmul_cpu(A, B, C);
    end = clock();
    printf("CPU execution time:\t %f seconds\n", ((double) (end - start)) / CLOCKS_PER_SEC);

    // CUDA execution
    cudaMemcpy(D, A, N * N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(E, B, N * N * sizeof(float), cudaMemcpyHostToDevice);
    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid(N / threads.x, N / threads.y);
    start = clock();
    matmul_gpu<<<grid, threads>>>(D, E, D);
    cudaDeviceSynchronize();
    end = clock();
    printf("CUDA execution time:\t %f seconds\n", ((double) (end - start)) / CLOCKS_PER_SEC);

    // cuBLAS execution
    cublasCreate(&handle);
    start = clock();
    cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, N, N, N, &alpha, D, N, E, N, &beta, D, N);
    cudaDeviceSynchronize();
    end = clock();
    printf("cuBLAS execution time:\t %f seconds\n", ((double) (end - start)) / CLOCKS_PER_SEC);
    cublasDestroy(handle);

    // Free memory
    free(A);
    free(B);
    free(C);
    cudaFree(D);
    cudaFree(E);

    return 0;
}
```

Compile and execute it:

```
$ nvcc -o matmul matmul.cu -lcublas
$ ./matmul
```

Identify the following important parts and explain the differences in time and why cuBLAS is the fastest:

- The parameter that defines the size of the matrices
- Memory allocation for CPU
- Memory allocation for GPU
- Memory transfers to GPU
- Initialization of handle for cuBLAS
- Identify the function (name) and the level of the used cublas function

Example output for $N = 1000$:

```
$ ./matmul
CPU time:    2.768519 seconds
CUDA time:   0.019095 seconds
cuBLAS time: 0.001316 seconds
```

3.5 Activity

In physics, work done by a force is calculated as the dot product of the force vector and the displacement vector. In this activity, students are asked to use the `cublasSdot` function from cuBLAS to calculate the work done.

- Problem Statement: A force of 5 N is applied to an object, causing it to move in the same direction as the force for a distance of 10 m. Calculate the work done by the force.
- Vector Representation: Represent the force and displacement as vectors. Since the force and displacement are in the same direction, we can represent them as one-dimensional vectors:
  - Force vector, $\boldsymbol{F} = [5]$
  - Displacement vector, $\boldsymbol{d} = [10]$
- cuBLAS Implementation: Use the `cublasSdot` function to calculate the dot product of the force and displacement vectors.

Use the following template (`work_done.cu`)

Code 3.5.1: Template for calculating the dot product of the force and displacement vectors

```
#include <stdio.h>
#include <cublas_v2.h>

#define N 1

int main() {
    float h_F[N] = {5.0f};  // Host array for force
    float h_d[N] = {10.0f};  // Host array for displacement
    float *d_F, *d_d;  // Device arrays
    float result;  // Result of the dot product
    cublasHandle_t handle;  // cuBLAS handle

    // Allocate device memory with cudaMalloc for d_F and d_d


    // Copy vectors to device with cudaMemcpy


    // Create cuBLAS handle


    // Calculate dot product using cublasSdot


    // Print result
    printf("Work done: %f J\n", result);


    // Cleanup hanlde nad device memory



    return 0;
}

```

cuDNN

The NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.

4.1 Convolution

Before presenting and analyzing cuDNN, we first need to describe the concept of convolution. Convolution is a mathematical operation that is pivotal in many areas of image processing. They are used as a cornerstone of various image transformation algorithms, including blurring, sharpening, edge detection, or more complex operations such as in convolutional neural networks.

A convolution for images involves applying a filter (also known as a “mask” or “kernel”) to an image. The filter is a small matrix of numbers that comprehensively scans over the image, and a computation is performed at each position to produce a new pixel value for the output image. The content of convolution include:

- Image Matrix: An image can be represented as a matrix where each cell represents a pixel. For grayscale images, the pixel value ranges from 0 to 255 (8-bit), where 0 depicts black, and 255 depicts white. For colored images, each pixel is often represented as a triad of red, green, and blue (RGB) components, where each color channel has values ranging from 0 to 255.

- nvolution Kernel: The kernel is another matrix, usually smaller than the input image, containing a pattern of numbers. The size of a kernel can vary but it is generally square (e.g., 3×3, 5×5, 7×7). The kernel values are sorted in a way to highlight certain visual features in the target image when applied.

- Convolution Operation: The convolution operation involves placing the kernel on top of the image matrix at a specific location, multiplying the kernel values by the corresponding image pixel values, then summing it all up to get a single number. This single number is the new value of the pixel in the output image for the corresponding covered area. The kernel slides across all areas of the input image to generate a complete output image.

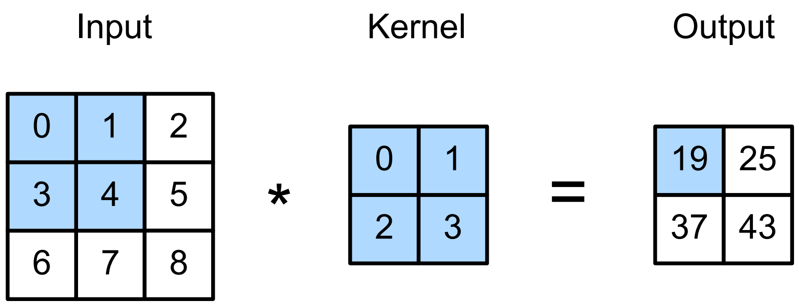


Figure 4.1: Convolution operation

4.2 Pooling

In the context of Convolutional Neural Networks (CNN), pooling is a form of non-linear down-sampling technique. Its chief objectives are to progressively reduce the spatial size of the representation to minimize the amount of parameters and computations in the network, and hence control overfitting. It also helps make the network invariant to small transformations, distortions, and translations in the input image. The pooling operations consists of:

- Pooling Layer: This is implemented as distinct phases in a CNN pipeline, usually performed after the convolutional layers. Each unit in the pooling layer summarizes the outputs of a set of neurons in the previous layer. The pooling layer operates independently on every depth slice of the input, and resizes it spatially.
- Pooling Operation: Pooling involves sliding a window (or a filter) over the input, typically with a stride larger than one to achieve down-sampling. The operation in the window could be any function that aggregates the input values such as max, average, sum etc.

One popular pooling function is Max pooling, which reports the maximum output within a rectangular neighborhood. It computes the maximum value over the input in the pooling window, or the filter, and this maximum value becomes the representative of the input values in that region. Typical filter sizes for max-pooling are 2×2 or 3×3 with a stride of 2. Max-pooling, by taking the maximum value, retains the most activated features while discarding the others, thereby providing a form of translational invariance and contributing to the performance boost of the CNN.

Pooling (and max-pooling in general) affords several notable benefits - it progressively reduces the spatial dimension, cutting down on the computation and

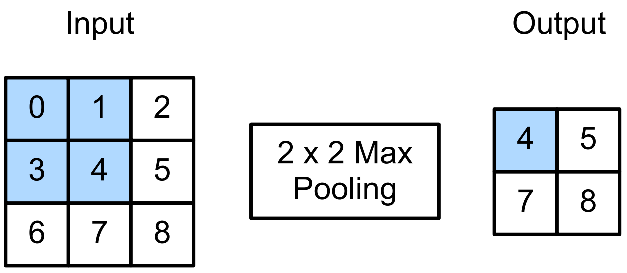


Figure 4.2: Max-pooling operation

helps combat overfitting. It also provides a form of translation invariance, which is a very desirable property for image recognition tasks where the exact location of features isn't as important as their rough relative position to other features.

4.3 cuDNN

NVIDIA cuDNN, short for CUDA Deep Neural Network, is a GPU-accelerated library created to facilitate deep learning applications. As the name suggests, it is built on top of the CUDA platform and leverages the immense parallel computing power of NVIDIA GPUs. As the demand for highly computationally intensive deep learning applications has grown, so has the importance of cuDNN in expediting the training time and giving these modern neural networks a considerable performance boost.

At its core, cuDNN is a library of optimized functions and routines which are a staple of deep learning algorithms. This includes operations such as forward and backward convolution, pooling, normalization, receptive field warping, and tensor transformation. Essentially, cuDNN provides primitive functions that act as the building blocks of deep neural networks (DNNs).

cuDNN is developed for both machine learning researchers for rapid prototyping, and production deployment with flexibility, portability, and performance in mind. It is compatible with many popular deep learning frameworks, including TensorFlow, PyTorch, and Caffe, providing developers with the flexibility to deploy this optimized library into their preferred environments.

4.3.1 Importance of cuDNN

Modern deep learning models are computationally exhaustive and memory-intensive, posing significant challenges in terms of computational resources and performance. The rise of GPUs and CUDA programming have alleviated these issues to some extent, but the arrival of cuDNN has escalated this process to a whole new level. Here's why cuDNN is vital:

1. Accelerated Performance: The high-performance cuDNN library significantly improves speed by utilizing GPU-accelerated computing. It offers highly optimized implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. The performance optimized with cuDNN can significantly reduce the training time of complex neural network models, shifting the focus from performance tuning to actual neural network architecture and design.
2. Seamless Integration: cuDNN is designed for integration into higher-level machine learning frameworks, providing developers and researchers with the flexibility to design and deploy deep learning solutions using familiar tools and environments, maximizing both productivity and performance.
3. Customizability: cuDNN allows developers to modify convolution algorithms, filter layouts, tensor formats, and other properties to suit their specific needs, adding an extra layer of customization that could optimize their applications better.
4. Portability: By abstracting the low-level CUDA details, cuDNN enables applications to run on different GPUs with varying capabilities. This means developers can write once and deploy across many NVIDIA GPU platforms, ensuring the reach and scalability of their solutions.

CNNs are the cornerstone of modern deep learning, particularly in image and video processing tasks. The convolution operation - the heart of CNNs - is computationally expensive, especially for large-scale 3D data like videos or medical images. cuDNN accelerates these operations drastically with GPU acceleration, benefiting the performance of CNNs. cuDNN provides routines for the forward and backward phases of convolutions, pooling, softmax activation, and tensor transformations. It offers multiple algorithmic choices for convolutions, including Fast Fourier Transform-based and Winograd-based algorithms.

cuDNN's optimizations are not confined to CNNs. It provides robust support for Recurrent Neural Networks (RNNs) - the de facto choice for sequence data like time-Series analysis, natural language processing, and speech recognition. cuDNN supplies a high-performance implementation for the forward and backward passes of RNNs, Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRU).

While cuDNN can bring significant performance improvements, the integration of cuDNN into applications does necessitate an understanding of its API and the underlying CUDA constructs. Developers need to ensure that their code correctly interfaces with the library, appropriately handles memory allocation, and follows the CUDA threading model.

Additional considerations around precision, quantization, and algorithmic choices like the convolution method must also be taken into account to fully leverage cuDNN's benefits. Despite these challenges, the value proposition of integrating cuDNN in terms of the performance gains is substantial enough that developers and researchers widely adopt it.

As deep learning models continue to grow in complexity and size, tools like cuDNN will become increasingly vital. By providing highly optimized routines for the fundamental operations in deep learning, NVIDIA's cuDNN library has not only accelerated the advancement of the field, but it has also given developers the ability to focus on building and improving their models rather than optimizing the low-level implementation details.

4.3.2 cuDNN program structure

A cuDNN program typically follows a structured approach to leverage the capabilities of the cuDNN library effectively. Here is a general structure for a cuDNN program:

1. Importing necessary libraries: Begin by importing the required libraries, including cuDNN, CUDA, and other standard libraries.
2. Defining the network parameters: Declare the necessary variables to store network parameters like input and output dimensions, filter sizes, strides, pad sizes, etc.
3. Allocating memory: Allocate memory on both the CPU and the GPU for input data, filter weights, and output data using cuDNN functions like `cudaMalloc` or `cudnnAlloc`.
4. Initializing cuDNN: Create and initialize a cuDNN handle using the `cudnnCreate` function. This handle will be used for most of the subsequent cuDNN function calls.
5. Setting convolution parameters: Configure the convolution parameters, such as the data type, convolution mode (e.g., convolution or cross-correlation), and padding type (e.g., zero-padding or border-replication). Set these parameters using relevant cuDNN functions like `cudnnSetConvolution2dDescriptor`.
6. Creating and configuring tensor descriptors: Define tensor descriptors for input data, filter weights, and output data using cuDNN functions like texttcudnnCreateTensorDescriptor and configure them accordingly using functions like `cudnnSetTensor4dDescriptor`.
7. Performing forward propagation: Use cuDNN functions like `cudnnConvolutionForward` to perform the forward propagation step. Set the relevant parameters, including input data, filter weights, output data, and convolution descriptor.

8. Performing backward propagation: If necessary, use cuDNN functions like `cudnnConvolutionBackwardData` and `cudnnConvolutionBackwardFilter` to perform backward propagation and update the weights of the network.
9. Synchronization and memory deallocation: Ensure all GPU computations are completed by synchronizing the GPU using `cudaDeviceSynchronize` if needed. Lastly, free the allocated memory and destroy the cuDNN handle.
10. Error handling: Handle any errors that may occur during the execution of cuDNN functions. You can use the `cudnnGetErrorString` function to get the error messages from cuDNN.

4.3.3 cuDNN descriptors

In cuDNN, descriptors are data structures used to define and describe certain properties of tensors, filters, and convolutions. These descriptors provide the necessary information to cuDNN functions for efficient processing and memory management. Here are the common types of descriptors in cuDNN and their respective contents:

1. **Tensor Descriptors:**
   - DataType: Specifies the data type of the tensor (e.g., float32, float16).
   - Dimension: Specifies the shape and size of the tensor.
2. **Filter Descriptors:**
   - DataType: Specifies the data type of the filter (e.g., float32, float16).
   - FilterType: Indicates the type of filter (e.g., spatial, temporal).
   - Format: Specifies the memory layout format of the filter (e.g., NCHW, NHWC).
   - Dimension: Specifies the shape and size of the filter.
3. **Convolution Descriptors:**
   - DataType: Specifies the data type of the input tensors and the output tensor of convolution (e.g., float32, float16).
   - Padding: Indicates the type of padding applied to the input tensor.
   - Strides: Specifies the sliding step of the filter during convolution.
   - Dilations: Specifies the dilation factor for each dimension.
   - Mode: Indicates the convolution mode (e.g., convolution, cross-correlation).
4. **Pooling Descriptors:**

- PoolingType: Specifies the type of pooling (e.g., max, average).
- WindowDimension: Specifies the size of the pooling window in each dimension.
- Padding: Indicates the type of padding applied to the input tensor.
- Strides: Specifies the sliding step of the pooling window.

These descriptors are created using specific cuDNN functions like `cudnnCreateTensorDescriptor`, `cudnnCreateFilterDescriptor`, `cudnnCreateConvolutionDescriptor`, and `cudnnCreatePoolingDescriptor`. The contents of these descriptors are set using corresponding functions like `cudnnSetTensor4dDescriptor`, `cudnnSetFilter4dDescriptor`, `cudnnSetConvolution2dDescriptor`, and `cudnnSetPooling2dDescriptor`. The values in descriptors are then used as input to various cuDNN functions for efficient operations.

4.4 Examples

4.4.1 Simple cuDNN tensor

Check the provided `cudnn_simple.cu` code. It is a simple example of a program that uses cuDNN to create a tensor descriptor and set its dimensions:

Code 4.4.1: cuDNN create a tensor descriptor and set its dimensions

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

int main() {
    cudnnHandle_t handle;
    cudnnTensorDescriptor_t tensorDesc;

    // Initialize cuDNN
    cudnnCreate(&handle);

    // Create the tensor descriptor
    cudnnCreateTensorDescriptor(&tensorDesc);

    // Set the dimensions of the tensor
    cudnnSetTensor4dDescriptor(tensorDesc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1, 1, 1, 1);

    // Cleanup
    cudnnDestroyTensorDescriptor(tensorDesc);
    cudnnDestroy(handle);

    printf("cuDNN example completed successfully.\n");

    return 0;
}
```

Compile and execute it:

```
$ nvcc -o cudnn_simple cudnn_simple.cu -lcudnn
$ ./cudnn_simple
```

This program:

- Initializes cuDNN with `cudnnCreate`
- Creates a tensor descriptor with `cudnnCreateTensorDescriptor`.
- Sets the dimensions of the tensor to $1 \times 1 \times 1 \times 1$ with `cudnnSetTensor4dDescriptor`.
  - `CUDNN_TENSOR_NCHW` parameter specifies the order (batch size, number of channels, height, width),
- Cleans up by destroying the tensor descriptor and the cuDNN handle

4.4.2 cuDNN 1D convolution

Check the provided `conv1d.cu` code. It performs 1D convolution using the CPU, CUDA, and cuDNN, and prints the execution time for each method:

Code 4.4.2: 1D convolution with CPU, CUDA, and cuDNN

```
#include <stdio.h>
#include <cudnn.h>
#include <time.h>

#define N 10000000
#define M 1000
#define BLOCK_SIZE 256

// CUDA kernel for 1D convolution
__global__ void conv1d_gpu(float *X, float *F, float *Y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        float sum = 0;
        for (int j = 0; j < M; j++) {
            if (i - j >= 0) {
                sum += X[i - j] * F[j];
            }
        }
        Y[i] = sum;
    }
}

// CPU function for 1D convolution
void conv1d_cpu(float *X, float *F, float *Y) {
    for (int i = 0; i < N; i++) {
        float sum = 0;
        for (int j = 0; j < M; j++) {
            if (i - j >= 0) {
                sum += X[i - j] * F[j];
            }
        }
        Y[i] = sum;
    }
}



int main() {
    float *h_X, *h_F, *h_Y, *d_X, *d_F, *d_Y;
    cudnnHandle_t handle;
    cudnnTensorDescriptor_t xDesc, yDesc;
    cudnnFilterDescriptor_t fDesc;
    cudnnConvolutionDescriptor_t convDesc;
    cudnnConvolutionFwdAlgo_t algo = CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM;

    size_t workspaceSize;
    void *workspace;
    clock_t start, end;
    double cpu_time_used, cuda_time_used, cudnn_time_used;

    // Allocate host and device memory
    h_X = (float*)malloc(N * sizeof(float));
    h_F = (float*)malloc(M * sizeof(float));
    h_Y = (float*)malloc(N * sizeof(float));
    cudaMalloc((void**)&d_X, N * sizeof(float));
    cudaMalloc((void**)&d_F, M * sizeof(float));
    cudaMalloc((void**)&d_Y, N * sizeof(float));

    // Initialize h_X and h_F...

    // Copy to device
    cudaMemcpy(d_X, h_X, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_F, h_F, M * sizeof(float), cudaMemcpyHostToDevice);

    // Perform 1D convolution using CUDA
    start = clock();
    conv1d_gpu<<<(N + BLOCK_SIZE - 1) / BLOCK_SIZE, BLOCK_SIZE>>>(d_X, d_F, d_Y);
    cudaDeviceSynchronize();
    end = clock();
    cuda_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("CUDA Time:\t %f\n", cuda_time_used);

    // Copy back to host
    cudaMemcpy(h_Y, d_Y, N * sizeof(float), cudaMemcpyDeviceToHost);

    // Perform 1D convolution using cuDNN
    // Create cuDNN handle and descriptors
    cudnnCreate(&handle);
    cudnnCreateTensorDescriptor(&xDesc);
    cudnnCreateTensorDescriptor(&yDesc);
    cudnnCreateFilterDescriptor(&fDesc);
    cudnnCreateConvolutionDescriptor(&convDesc);

    // Set tensor and filter descriptors
    cudnnSetTensor4dDescriptor(xDesc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1, 1, 1, N);
    cudnnSetTensor4dDescriptor(yDesc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1, 1, 1, N);
    cudnnSetFilter4dDescriptor(fDesc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, 1, 1, 1, M);
    cudnnSetConvolution2dDescriptor(convDesc, 0, 0, 1, 1, 1, 1,
                                    CUDNN_CONVOLUTION, CUDNN_DATA_FLOAT);

    // Choose convolution algorithm for version before cudnn 8
    // cudnnGetConvolutionForwardAlgorithm(handle, xDesc, fDesc, convDesc, yDesc,
                                        // CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, 0, &algo);


    // Allocate workspace
    cudnnGetConvolutionForwardWorkspaceSize(handle, xDesc, fDesc,
                                            convDesc, yDesc, algo, &workspaceSize);
    cudaMalloc(&workspace, workspaceSize);

    // Perform 1D convolution using cuDNN
    start = clock();
    float alpha = 1.0f;
    float beta = 0.0f;
    cudnnConvolutionForward(handle, &alpha, xDesc, d_X, fDesc, d_F, convDesc,
                            algo, workspace, workspaceSize, &beta, yDesc, d_Y);
    cudaDeviceSynchronize();
    end = clock();
    cudnn_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("cuDNN Time:\t %f\n", cudnn_time_used);

    // Copy back to host
    cudaMemcpy(h_Y, d_Y, N * sizeof(float), cudaMemcpyDeviceToHost);

    // Perform 1D convolution on CPU
    start = clock();
    conv1d_cpu(h_X, h_F, h_Y);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("CPU Time:\t %f\n", cpu_time_used);

    // Cleanup
    cudaFree(workspace);
    cudnnDestroyConvolutionDescriptor(convDesc);
    cudnnDestroyFilterDescriptor(fDesc);
    cudnnDestroyTensorDescriptor(yDesc);
    cudnnDestroyTensorDescriptor(xDesc);
    cudnnDestroy(handle);
    free(h_X);
    free(h_F);
    free(h_Y);
    cudaFree(d_X);
    cudaFree(d_F);
    cudaFree(d_Y);

    return 0;
}
```

Identify the following important parts and explain the differences in time and why cuDNN is the fastest:

- The parameter that defines the size of the matrices
- Memory allocation for CPU
- Memory allocation for GPU
- Memory transfers to GPU
- Initialization of descriptors for cuDNN
- Identify the functions used in cuDNN

Example output for $N = 10000000, M = 1000$:

```
$ ./conv1d
CUDA Time:   0.158909
cuDNN Time:  0.000011
CPU Time:    24.136710
```

4.5 Activity

Check the **activity** file `cublas_cudnn.cu`. This activity combines cuBLAS and cuDNN. The goal is to get two matrices: (i) multiply them using cuBLAS; (ii) apply ReLU activation function at the result using cuDNN. Use the following template and fill in the lines with **...**:

Code 4.5.1: cuDNN and cuBLAS activity

```
#include <cublas_v2.h>
#include <cudnn.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define M 3
#define K 2
#define N 4

void printMatrix(float* matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%.2f ", matrix[i * cols + j]);
        }
        printf("\n");
    }
}

// Function to initialize a matrix with random values between -1 and 1
void initializeMatrix(float* matrix, int rows, int cols) {
    srand(time(NULL));
    for (int i = 0; i < rows * cols; i++) {
        matrix[i] = (float)rand() / (float)(RAND_MAX / 2) - 1.0f;
    }
}

int main() {
    // Initialize cuBLAS and cuDNN
    cublasHandle_t cublasHandle;
    cublasCreate(&cublasHandle);

    cudnnHandle_t cudnnHandle;
    cudnnCreate(&cudnnHandle);

    // Allocate and initialize input matrices on the device
    int rowsA = M, colsA = K, rowsB = K, colsB = N;
    float* d_A;
    float* d_B;
    float* d_C;
    cudaMalloc(&d_A, sizeof(float) * rowsA * colsA);
    cudaMalloc(&d_B, sizeof(float) * rowsB * colsB);
    cudaMalloc(&d_C, sizeof(float) * rowsA * colsB);

    // Initialize d_A and d_B with random values between -1 and 1
    float* h_A = (float*)malloc(sizeof(float) * rowsA * colsA);
    float* h_B = (float*)malloc(sizeof(float) * rowsB * colsB);
    initializeMatrix(h_A, rowsA, colsA);
    initializeMatrix(h_B, rowsB, colsB);
    cudaMemcpy(d_A, h_A, sizeof(float) * rowsA * colsA, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, sizeof(float) * rowsB * colsB, cudaMemcpyHostToDevice);

    // Perform matrix multiplication using cuBLAS cublasSgemm
    const float alpha = 1.0f;
    const float beta = 0.0f;
    ...

    // Print the result of the matrix multiplication
    float* h_C = (float*)malloc(sizeof(float) * rowsA * colsB);
    cudaMemcpy(h_C, d_C, sizeof(float) * rowsA * colsB, cudaMemcpyDeviceToHost);

    printf("A:\n");
    printMatrix(h_A, M, K);
    printf("\n");

    printf("B:\n");
    printMatrix(h_B, K, N);
    printf("\n");

    printf("C before ReLU activation:\n");
    printMatrix(h_C, M, N);
    printf("\n");

    // Create a cuDNN tensor descriptor for the output of the matrix multiplication
    cudnnTensorDescriptor_t tensorDesc;
    cudnnCreateTensorDescriptor(&tensorDesc);
    cudnnSetTensor4dDescriptor(tensorDesc, CUDNN_TENSOR_NCHW,
                               CUDNN_DATA_FLOAT, 1, 1, rowsA, colsB);

    // Create a cuDNN activation descriptor for the ReLU activation function
    cudnnActivationDescriptor_t activationDesc;
    cudnnCreateActivationDescriptor(&activationDesc);
    cudnnSetActivationDescriptor(activationDesc,
                               CUDNN_ACTIVATION_RELU, CUDNN_NOT_PROPAGATE_NAN, 0.0);

    // Apply the ReLU activation function using cuDNN
    ...

    // Print the result after the ReLU activation
    cudaMemcpy(h_C, d_C, sizeof(float) * rowsA * colsB, cudaMemcpyDeviceToHost);

    printf("C after ReLU activation:\n");
    printMatrix(h_C, M, N);

    // Free the memory
    free(h_A);
    free(h_B);
    free(h_C);
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```

NCCL

NVIDIA Collective Communications Library (NCCL) is a high-performance library that provides communication routines optimized for NVIDIA GPUs. It is designed to efficiently scale multi-GPU and multi-node applications using popular interconnect technologies like InfiniBand, Ethernet, and NVLink.

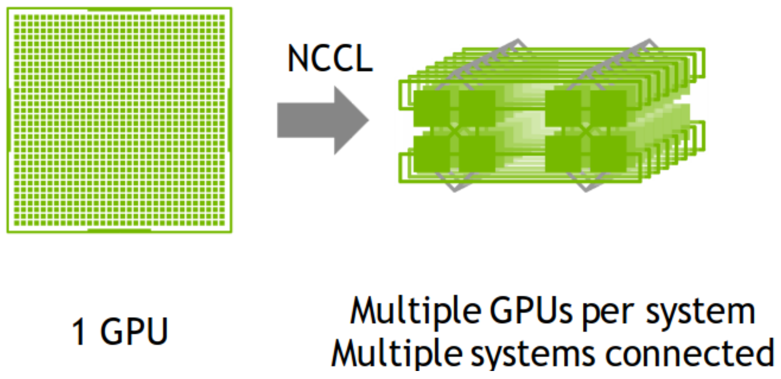


Figure 5.1: NCCL: a library developed by NVIDIA that supports high-performance multi-GPU computing.

5.1 NCCL key aspects

GPU-Aware Communication NCCL is built to leverage the unique capabilities of GPUs. It provides GPU-aware communication primitives that allow direct memory transfers between GPUs without involving the CPU. This drastically reduces communication overhead and improves performance.

Optimized for Scalability NCCL is designed to efficiently scale applications across multiple GPUs and nodes. It employs advanced algorithms and techniques, such as asynchronous and overlap communication techniques, to minimize latency and maximize throughput. This scalability is critical for large-scale parallel computing, deep learning, and scientific simulations.

Support for Various Interconnects NCCL supports a wide range of interconnect technologies, including InfiniBand, Ethernet, and NVLink. It provides optimized communication routines for each of these interconnects, allowing users to take advantage of the best available options based on their system setup.

High Performance Collectives NCCL offers optimized collective communication primitives, such as all-reduce, broadcast, reduce, scatter, and gather operations. These collective operations provide efficient ways to synchronize data across multiple GPUs and nodes, enabling effective parallelization of computations

Deep Learning Framework Integration NCCL is widely used in deep learning frameworks like TensorFlow, PyTorch, and MXNet. It provides high-performance communication backends, allowing efficient model parallelism and data parallelism across GPUs.

CUDA-Aware MPI Integration NCCL is designed to seamlessly integrate with other communication libraries, such as Message Passing Interface (MPI). It provides CUDA-aware MPI extensions to allow direct GPU-to-GPU communication in MPI applications, further enhancing performance.

Cross-Platform Support NCCL is available for various operating systems, including Linux, Windows, and macOS. It is also compatible with multiple GPU architectures, supporting NVIDIA GPUs from the Kepler, Maxwell, Pascal, and Volta generations.

In summary, NCCL plays a crucial role in multi-GPU programming by providing optimized communication primitives, scalability features, and integration with deep learning frameworks. It allows developers to leverage the computational power of multiple GPUs effectively and efficiently communicate and synchronize data. NCCL's capabilities significantly simplify multi-GPU programming and enable high-performance parallel processing on GPU-accelerated systems.

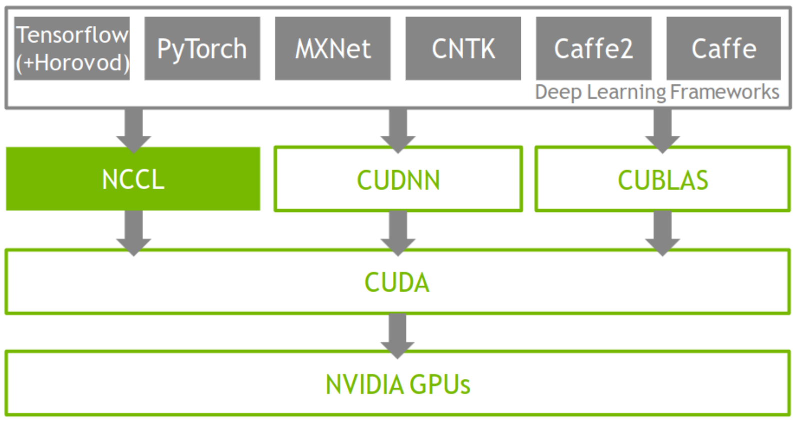


Figure 5.2: How NCCL Aids in Multi-GPU Programming

5.2 Collective communication

In NCCL (NVIDIA Collective Communications Library), collective communication refers to a set of operations that involve coordinated data exchange and synchronization between multiple GPUs. These operations are collectively performed by all participating GPUs simultaneously. NCCL provides optimized collective communication primitives that enable efficient synchronization and data exchange across multiple GPUs. Here are the key concepts related to collective communication in NCCL:

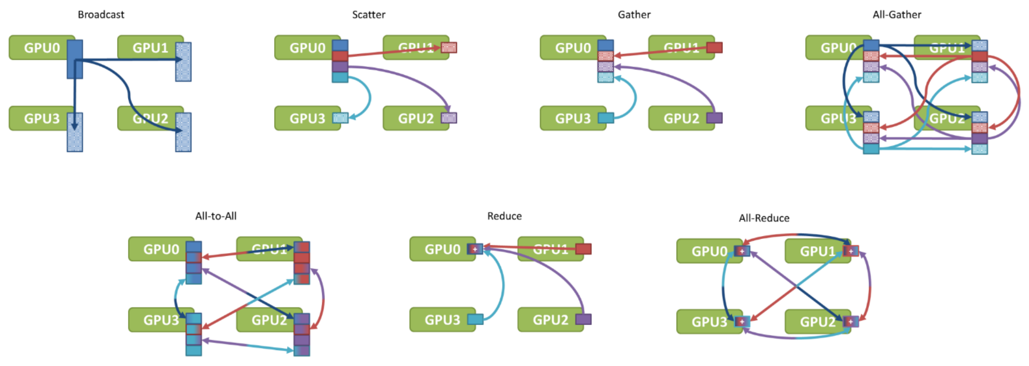


Figure 5.3: NCCL collective communication

Broadcast The broadcast operation transfers data from one GPU (the root GPU) to all other participating GPUs. The root GPU holds the data to be broadcasted. NCCL efficiently transfers the data to all the other GPUs, ensuring synchronized access to the data across all GPUs.

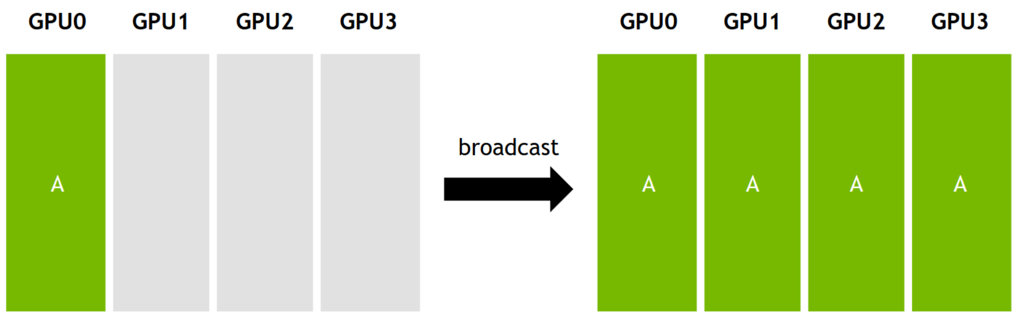


Figure 5.4: NCCL broadcast

Scatter The scatter operation distributes data from the root GPU to all other participating GPUs. The root GPU splits the data into equal-sized chunks and sends each chunk to a specific GPU. This operation is useful for parallel processing tasks that involve dividing data among multiple GPUs.

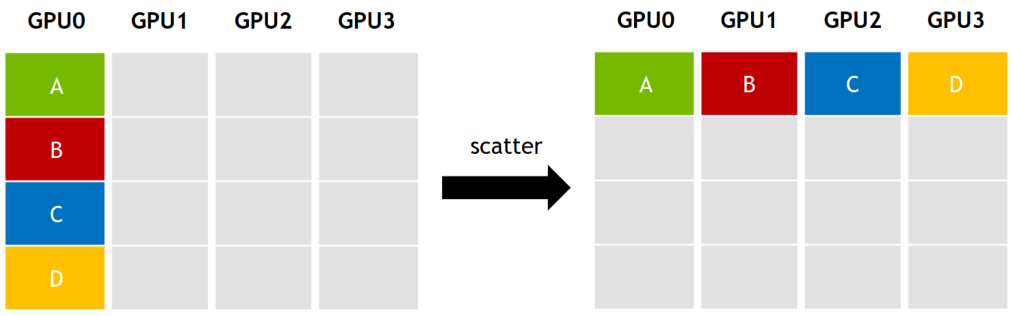


Figure 5.5: NCCL scatter

Gather The gather operation collects data from all participating GPUs and stores it on the root GPU. Each GPU provides a chunk of data, and NCCL efficiently gathers and organizes the data on the root GPU.

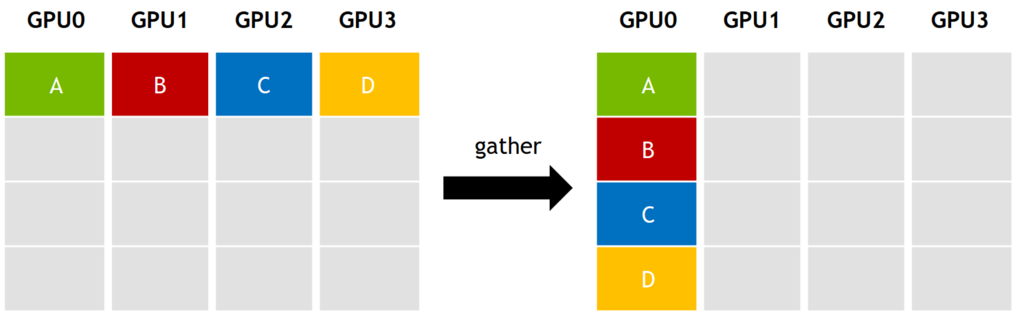


Figure 5.6: NCCL gather

All-Reduce The all-reduce operation is a collective communication primitive supported by NCCL. It takes an input data array from each GPU and performs an element-wise reduction operation across all the arrays. The result is then distributed back to all the GPUs. Common reduction operations include summation, product, minimum, and maximum.

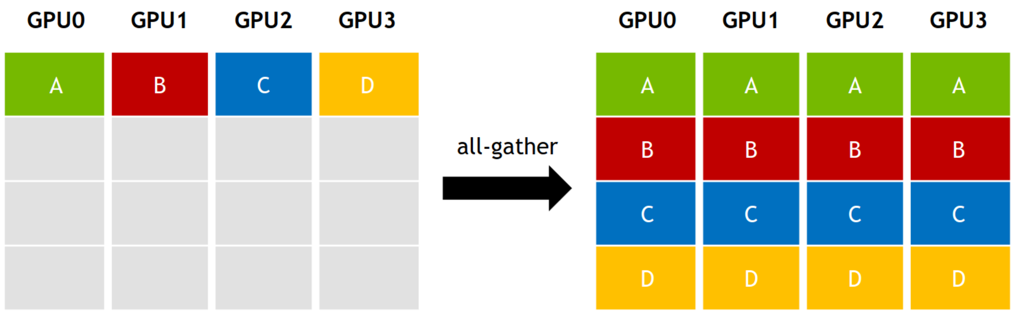


Figure 5.7: NCCL all gather

Reduce The reduce operation performs a reduction operation on data from all GPUs and stores the result on a designated target GPU. This operation is useful when a single GPU needs to collect and process data from all other GPUs collectively.

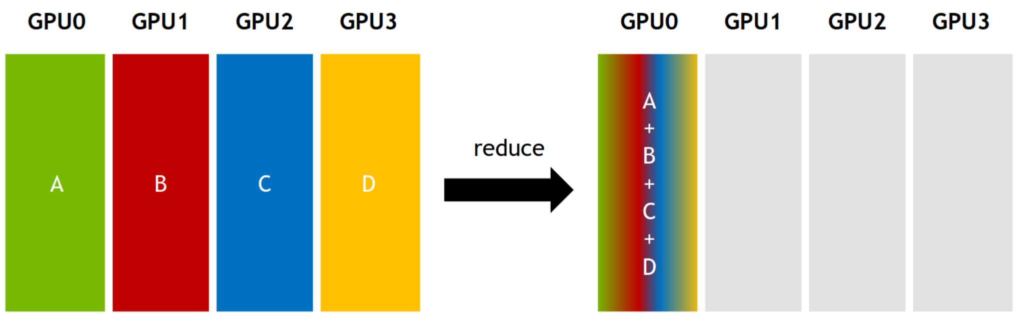


Figure 5.8: NCCL reduce

All Reduce The AllReduce operation is performing reductions on data (for example, sum, max) across devices and writing the result in the receive buffers of every rank.

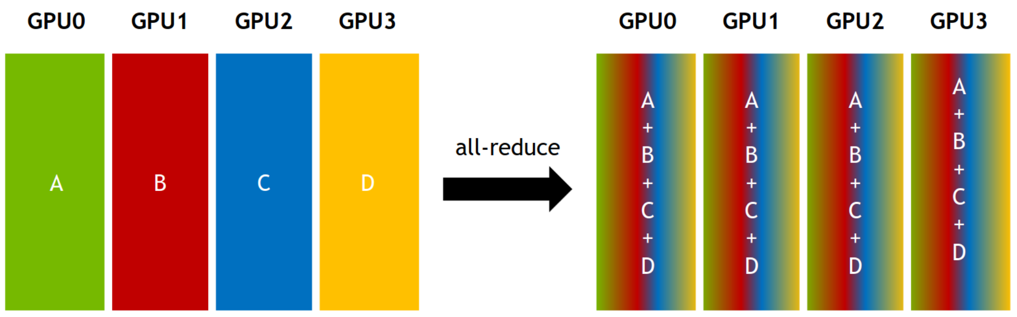


Figure 5.9: NCCL all reduce

ReduceScatter The ReduceScatter operation performs the same operation as the Reduce operation, except the result is scattered in equal blocks among ranks, each rank getting a chunk of data based on its rank index.

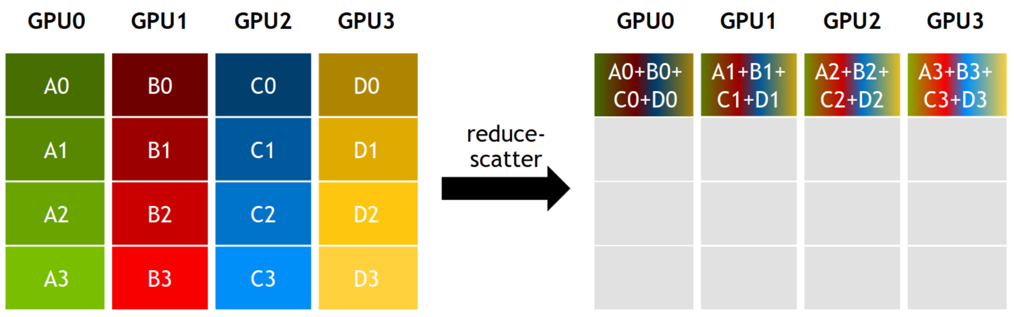


Figure 5.10: NCCL ReduceScatter

5.3 Example

Below, you can fidn the prototype of the `ncclBcast` function. It is very similar to the the `MPI_Bcast` function supported by MPI:

```
int ncclBcast (
    void*          data_p      /* in/out*/,
    size_t         count       /*   in   */,
    ncclDataType_t datatype    /*   in   */,
    int            source_proc /*   in   */,
    ncclComm_t     comm        /*   in   */,
    cudaStream_t   stream      /*   in   */
);
```

Check the provided `ncclBcast.cu` code. It implements a sample BROADCAST code using the package NCCL:

Code 5.3.1: Brodcast example based on NCCL

```
#include <nccl.h>
#include <stdio.h>
#include <stdlib.h>

__global__ void kernel(int *a)
{
  int index = threadIdx.x;

  a[index] *= 2;
  printf("%d\t", a[index]);

}/*kernel*/


void print_vector(int *in, int n){

 printf("Random values in the vector before ncclBcast\n");
 for(int i=0; i < n; i++)
  printf("%d\t", in[i]);

 printf("\n");

}/*print_vector*/


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

  int data_size = 8 ;
  int nGPUs = 0;
  cudaGetDeviceCount(&nGPUs);
  // print a message about the number of GPUs
  printf("Number of GPUs: %d\n\n", nGPUs);


  int *DeviceList = (int *) malloc (nGPUs     * sizeof(int));
  int *data       = (int*)  malloc (data_size * sizeof(int));
  int **d_data    = (int**) malloc (nGPUs     * sizeof(int*));

```

```
for(int i = 0; i < nGPUs; i++)
    DeviceList[i] = i;

/*Initializing NCCL with Multiples Devices per Thread*/
ncclComm_t* comms = (ncclComm_t*)  malloc(sizeof(ncclComm_t)  * nGPUs);
cudaStream_t* s   = (cudaStream_t*)malloc(sizeof(cudaStream_t)* nGPUs);
ncclCommInitAll(comms, nGPUs, DeviceList);

/* initialize data vector with random values from 0 to 100 */
for (int i = 0; i < data_size; i++)
    data[i] = rand() % 100;

/* print the data */
print_vector(data, data_size);

for(int g = 0; g < nGPUs; g++) {
    cudaSetDevice(DeviceList[g]);
    cudaStreamCreate(&s[g]);
    cudaMalloc(&d_data[g], data_size * sizeof(int));

    if(g == 0)  /*Copy from Host to Device*/
       cudaMemcpy(d_data[g], data, data_size * sizeof(int), cudaMemcpyHostToDevice);
}

ncclGroupStart();

                for(int g = 0; g < nGPUs; g++) {
                      cudaSetDevice(DeviceList[g]);
          /*Broadcasting it to all*/
                    ncclBcast(d_data[g], data_size, ncclInt, 0, comms[g], s[g]);
                }

ncclGroupEnd();

for (int g = 0; g < nGPUs; g++) {
    cudaSetDevice(DeviceList[g]);
    printf("\nThis is device %d\n", g);
    /*Call the CUDA Kernel: The code multiple the vector position per 2 on GPUs*/
    kernel <<< 1 , data_size >>> (d_data[g]);
    cudaDeviceSynchronize();
}

printf("\n");

for (int g = 0; g < nGPUs; g++) { /*Synchronizing CUDA Streams*/
    cudaSetDevice(DeviceList[g]);
    cudaStreamSynchronize(s[g]);
}

for(int g = 0; g < nGPUs; g++) {  /*Destroy CUDA Streams*/
    cudaSetDevice(DeviceList[g]);
    cudaStreamDestroy(s[g]);
```

```
  }

  for(int g = 0; g < nGPUs; g++)    /*Finalizing NCCL*/
     ncclCommDestroy(comms[g]);

  /*Freeing memory*/
  free(s);
  free(data);
  free(DeviceList);

  cudaFree(d_data);

  return 0;

}/*main*/
```

Compile and execute it:

```
$ nvcc ncclBcast.cu -o ncclBcast -lnccl
$ ./ncclBcast
```

Identify in the code the following components:

- Listing of GPUs (number)
- NCCL initialization
- Copy data from host to GPU
- NCCL broadcast operations
- The kernel each GPU executes
- Explain the final outcome

5.4 Activity

Check the provided `ncclReduce.cu` code. It implements dot product (scalar product) using `ncclReduce`:

Code 5.4.1: Reduce example based on NCCL

```
#include <cstdio>
#include <cstdlib>
#include <string>
#include <vector>
#include <nccl.h>

__global__ void Dev_dot(double *x, double *y, int n) {

   __shared__ double tmp[512];

   int i = threadIdx.x;
   int t = blockDim.x * blockIdx.x + threadIdx.x;

   if (t < n)
    tmp[i] = x[t];

   __syncthreads();

   for (int stride = blockDim.x / 2; stride >  0; stride /= 2) {

      if (i < stride)
         tmp[i] += tmp[i + stride];

      __syncthreads();

   }

   if (threadIdx.x == 0) {
      y[blockIdx.x] = tmp[0];
      printf("\tdot(x,y) = %1.2f\n", y[blockIdx.x]);
   }

}/*Dev_dot*/


__global__ void Dev_print(double *x) {

   int i = threadIdx.x;

   printf("%1.2f\t", x[i]);

}/*Dev_print*/


void print_vector(double *in, int n){

 for(int i=0; i < n; i++)
  printf("%1.2f\t", in[i]);

 printf("\n");

}/*print_vector*/


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

  /*Variables*/
  int nGPUs = 0;
  cudaGetDeviceCount(&nGPUs);
  // print a message about the number of GPUs
  printf("Number of GPUs: %d\n\n", nGPUs);

```

```
int *DeviceList = (int *) malloc ( nGPUs * sizeof(int));

int data_size = 8;

double *x          = (double*)    malloc(data_size * sizeof(double));
double *y          = (double*)    malloc(data_size * sizeof(double));
double **x_d_data  = (double**)   malloc(nGPUs     * sizeof(double*));
double **y_d_data  = (double**)   malloc(nGPUs     * sizeof(double*));
double **Sx_d_data = (double**)   malloc(nGPUs     * sizeof(double*));
double **Sy_d_data = (double**)   malloc(nGPUs     * sizeof(double*));

for (int i = 0; i < nGPUs; ++i)
    DeviceList[i] = i;

/*Initializing NCCL with Multiples Devices per Thread*/
ncclComm_t* comms = (ncclComm_t*)  malloc(sizeof(ncclComm_t)  * nGPUs);
cudaStream_t* s   = (cudaStream_t*)malloc(sizeof(cudaStream_t)* nGPUs);
ncclCommInitAll(comms, nGPUs, DeviceList);

/*Population vectors*/
for(int i = 0; i < data_size; i++){
    x[i] = 1;
    y[i] = 2;
}

print_vector(x, data_size);
print_vector(y, data_size);


for(int g = 0; g < nGPUs; g++) {
    cudaSetDevice(DeviceList[g]);
    cudaStreamCreate(&s[g]);

    cudaMalloc(&x_d_data[g],    data_size * sizeof(double));
    cudaMalloc(&y_d_data[g],    data_size * sizeof(double));

    cudaMalloc(&Sx_d_data[g],   data_size * sizeof(double));
    cudaMalloc(&Sy_d_data[g],   data_size * sizeof(double));

   /*Copy from Host to Devices*/
    cudaMemcpy(x_d_data[g],  x, data_size * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(y_d_data[g],  y, data_size * sizeof(double), cudaMemcpyHostToDevice);
  }

ncclGroupStart();

        for(int g = 0; g < nGPUs; g++) {
         cudaSetDevice(DeviceList[g]);
    /*Reducing x vector*/
    ncclReduce(x_d_data[g], Sx_d_data[g], data_size, ncclDouble, ncclSum, 0, comms[g], s[g]);
    /*Reducing y vector*/
    ncclReduce(y_d_data[g], Sy_d_data[g], data_size, ncclDouble, ncclSum, 0, comms[g], s[g]);
    }

  ncclGroupEnd();

  cudaSetDevice(DeviceList[0]);
  printf("\n The final result on device %d\n", 0);
  /*Call the CUDA Kernel: dot product*/
  Dev_dot <<< 1, data_size >>> (Sy_d_data[0], Sx_d_data[0], data_size);
  cudaDeviceSynchronize();


  for (int g = 0; g < nGPUs; g++) { /*Synchronizing CUDA Streams*/
      cudaSetDevice(DeviceList[g]);
      cudaStreamSynchronize(s[g]);
  }

  for(int g = 0; g < nGPUs; g++) { /*Destroy CUDA Streams*/
      cudaSetDevice(DeviceList[g]);
      cudaStreamDestroy(s[g]);
  }

  for(int g = 0; g < nGPUs; g++) /*Finalizing NCCL*/
     ncclCommDestroy(comms[g]);

  /*Freeing memory*/
  free(s);
  free(x);
  free(y);
  free(DeviceList);

  cudaFree(x_d_data);
  cudaFree(y_d_data);
  cudaFree(Sx_d_data);
  cudaFree(Sy_d_data);

  return 0;

}/*main*/
```

Perform the following:

- Identify the NCCL operation
- Modify the code so as each GPU gets the final output
- Print the output per GPU and verify its correctness
- Modify the code so as each input vector is initialized with random values from 0 to 10

NVIDIA Nsight Systems

NVIDIA Nsight Systems is a system-wide performance analysis tool designed to visualize application's algorithms, help you identify the largest opportunities to optimize, and tune to scale efficiently across any quantity of CPUs and GPUs in your computer, from laptops to DGX servers. Official link to NVIDIA Nsight Systems here.

This advanced software tool is specifically engineered to enable users to design performance-oriented applications. It offers a seamless, intuitive, and easy-to-navigate interface that provides users with in-depth insights into their system's process flow. These details pave the way for developers to analyze the data and make necessary alterations for optimal performance.

Nsight Systems is a crucial component of the NVIDIA Nsight suite, which includes a collection of debugging and profiling tools for developing on the NVIDIA platform. Known for its comprehensive user interface, it focuses on collecting higher level timing information for CUDA, OS runtime (NVTX), and OpenACC events along with system level details such as disk I/O, CPU Utilization, and OS threads and processes.

Nsight Systems offers a range of comprehensive features. It enables developers to profile their applications by running an application and recording the system's behavior over time. The recorded information includes data about CPU usage, GPU usage, threading, and much more. Developers can then drill-down into the recorded data, identify bottlenecks, and make changes to improve performance.

Before Nsight Systems, developers would have to rely on numerous different tools to get all of these details. Now, they have everything they need in a single place. This not only makes the process of profiling and tuning applications more efficient but also more accurate.

Nsight Systems uses a very intuitive and informative graphical timeline to assist developers in visualizing their applications. The graphical timeline displays the CPU and GPU usage in great detail, allowing developers to easily see where bottlenecks are occurring.

This tool is integrated with NVTX, which allows developers to annotate their code to push additional information into the timeline. This can be incredibly useful for identifying areas of the code that are executing more slowly than expected.

Furthermore, the timeline provides detailed information about GPU kernel execution, context switching among threads, GPU utilization, and various other system activities. The timeline also allows for easy navigation and zooming to present a detailed analysis of bottlenecks in the system execution sequence.

One of the main advantages of Nsight Systems is its ability to provide an understanding of an application's dependencies and their effects on performance. By examining where the application is CPU-bound, you can determine the areas to focus on for optimization. By reviewing the GPU usage, you can see whether the GPU is fully utilized or not. If not, this may indicate bottlenecks in the CPU or problems with the interaction between the CPU and GPU.

In addition, Nsight Systems can identify and display potential oversubscription or inefficient usage of CPUs. To gain a better understanding of the application's execution, this tool allows filtering various event groups, which helps developers focus their investigation onto specific problem areas and avoid unnecessary information.

Moreover, Nsight Systems can assist in identifying issues related to API misuse. By comparing results across multiple devices, you can detect patterns or anomalies that may indicate a problem in the way the API is being utilized.

Nsight Systems supports analysis of various types of applications including CUDA, Direct3D, OpenCL, Vulkan, and more. It works on multiple platforms including Windows, Linux, and QNX. It supports remote collection of data on Linux and QNX systems, which can prove to be quite beneficial in many scenarios.

Nsight Systems also integrates well with other tools in the NVIDIA Nsight suite. For instance, if you want to dive into the details of GPU kernel execution following a system-wide performance analysis, that can be done seamlessly by opening the collected report in Nsight Compute, which provides kernel-level profiling.

Using Nsight Systems, developers can trace workloads across CPU cores and CUDA streams, and visualize the overlap of CUDA, NVTX, and OS runtime events. This gives a clear overview of how software components are interacting, enabling developers to better locate and fix areas of inefficiency.

In conclusion, NVIDIA Nsight Systems is an advanced system-wide performance analysis tool that allows developers to visualize their application's behavior, identify large optimization opportunities, and tune their application to perform efficiently across any quantity of CPUs and GPUs; ranging from laptops to DGX servers. By offering comprehensive features in a single tool, Nsight Systems proves to be an invaluable asset to developers aiming to execute performance-oriented applications. The complete list of features can be found here

6.1 Nsight Systems CLI

NVIDIA Nsight Systems command line tool is called `nsys`. Launching the tool:

```
$ nsys --version
NVIDIA Nsight Systems version 2022.1.3.3-1c7b5f7
$ nsys --help
The most commonly used nsys commands are:
    profile       Run an application and capture its profile into a QDSTRM file.
    launch        Launch an application ready to be profiled.
    start         Start a profiling session.
    stop          Stop a profiling session and capture its profile into a QDSTRM file.
    cancel        Cancel a profiling session and discard any collected data.
    stats         Generate statistics from an existing nsys-rep or SQLite file.
    status        Provide current status of CLI or the collection environment.
    shutdown      Disconnect launched processes from the profiler and shutdown the profiler.
    sessions list List active sessions.
    export        Export nsys-rep file into another format.
    analyze       Run rules on an existing nsys-rep or SQLITE file.
    nvprof        Translate nvprof switches to nsys switches and execute collection.
```

The Nsight Systems CLI provides a simple interface to collect on a target without using the GUI. The collected data can then be copied to any system and analyzed later. The general command in `nsys` has the following format:

```
$ nsys [command_switch][optional command_switch_options][application] [optional application_options]
```

6.2 Example

Check the provided `addarrays.cu` code. This is a simple program that adds the elements of two arrays on the GPU.

Code 6.2.1: Adding two arrays

```
#include <iostream>
#include <math.h>

// CUDA Kernel function to add the elements of two arrays on the GPU
__global__ void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;

  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x, *y;

  // Allocate Unified Memory – accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 256>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

Compile it:

```
$ nvcc -o addarrays addarrays.cu
```

Now, it is time to profile it using `nsys`:

```
$ nsys profile --stats=true --force-overwrite true -o report ./addarrays
```

In this command, the arguments have the following meaning:

- `nsys profile`: Starts a profiling session with Nsight Systems.
- `--stats=true`: Prints a summary of the profiling results to the console.
- `--force-overwrite true`: Overwrites any existing profiling data with the same name.
- `-o report`: Saves the profiling data to a file named `report.qdrep`.

An example of the profiling output is shown below:

```
Operating System Runtime API Statistics:

Time (%)  Total Time (ns)  Num Calls    Avg (ns)       Med (ns)      Min (ns)    Max (ns)     StdDev (ns)      Name
--------  ---------------  ---------  -------------  -------------  ---------  -----------  ------------- ----------
    73.9    1,113,804,107          3  371,268,035.7  393,747,147.0  1,061,513  718,995,447  359,494,459.3 sem_wait
    18.0      270,611,330         19  14,242,701.6  10,069,455.0      2,558  100,129,004   24,284,703.5  poll
     4.7       70,166,271        504     139,218.8      11,618.5      1,065   25,241,326    1,357,498.6  ioctl
     2.8       42,662,842         15   2,844,189.5      82,218.0     26,742   20,534,983    5,898,506.4  sem_timedwait
     0.3        5,215,754         42     124,184.6      20,918.5      9,801    2,409,020      439,191.3  mmap64
     0.1        1,340,510          8     167,563.8       7,612.0      2,082      701,533      300,156.7  fread
     0.1          906,253         22      41,193.3      14,498.0      2,473      371,659       83,668.8  mmap
     0.0          621,904         59      10,540.7       9,399.0      3,922       20,438        3,690.5  open64
     0.0          406,006          5      81,201.2      76,247.0     66,261      107,800       16,075.5  pthread_create
     0.0          342,166         54       6,336.4       5,529.0      2,249       21,263        3,412.2  fopen
     0.0          135,187          7      19,312.4      10,124.0      7,211       67,683       21,717.4  fgets
     0.0          103,903         44       2,361.4       2,235.0      1,271        4,655          667.5  fclose
     0.0           65,263          6      10,877.2       8,978.0      6,505       20,603        5,074.5  munmap
     0.0           35,225         23       1,531.5       1,248.0      1,010        4,013          805.5  fcntl
     0.0           34,385          5       6,877.0       6,067.0      5,440       10,062        1,928.2  open
     0.0           22,066         11       2,006.0       1,814.0      1,181        3,415          798.2  read
...

CUDA API Statistics:

Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)    Min (ns)    Max (ns)    StdDev (ns)       Name
--------  ---------------  ---------  ------------  ------------  ---------  -----------  -------------  ----------------
    97.5      147,204,541          2  73,602,270.5  73,602,270.5     59,116  147,145,425  104,005,726.5  cudaMallocManaged
     2.0        3,023,309          1   3,023,309.0   3,023,309.0  3,023,309    3,023,309            0.0  cudaDeviceSynchro
     0.5          708,519          2     354,259.5     354,259.5    260,222      448,297      132,989.1  cudaFree
     0.0           41,804          1      41,804.0      41,804.0     41,804       41,804            0.0  cudaLaunchKernel
     0.0            1,732          1       1,732.0       1,732.0      1,732        1,732            0.0  cuModuleGetLoadi
...


CUDA Kernel Statistics:

Time (%)  Total Time (ns)  Instances   Avg (ns)     Med (ns)    Min (ns)   Max (ns)   StdDev (ns)             Name
--------  ---------------  ---------  -----------  -----------  ---------  ---------  -----------  ---------------------
   100.0        2,979,577          1  2,979,577.0  2,979,577.0  2,979,577  2,979,577          0.0  add(int, float *, fl

...


CUDA Memory Operation Statistics (by time):

Time (%)  Total Time (ns)  Count  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)              Operation
--------  ---------------  -----  --------  --------  --------  --------  -----------  ---------------------------------
    64.6          755,192     48  15,733.2   3,967.5     2,558    85,630     24,110.7  [CUDA Unified Memory memcpy HtoD]
    35.4          413,676     24  17,236.5   3,855.5     1,183    98,590     27,704.2  [CUDA Unified Memory memcpy DtoH]
...
```

6.2.1 Usage of NVTX

If you check careful the output of the profiler, you will see at the top the following output:

```
[3/8] Executing 'nvtxsum' stats report
SKIPPED: home/.../report.sqlite does not contain NV Tools Extension (NVTX) data.
```

NVTX, or the NVIDIA Tools Extensions SDK, is a powerful mechanism for annotating events, code ranges, and resources in applications. It is designed to improve the profiling experience by providing detailed and specific information about your program's behavior, which can be extremely helpful in performance optimization

While performance analysis tools provide developers with a high-level overview of where bottlenecks may be occurring, it can be challenging to link these general observations back to specific parts of the codebase. NVTX helps bridge this gap by allowing developers to mark certain sections of the code and adding user-defined events in the application's timeline that can be tracked by profiling tools.

The NVTX tool features a C-based API used for annotating events, code ranges, and resources. These annotated points can be viewed in correlation with other metrics in a GPU profiling tool's timeline, such as Nsight Systems. They can be useful in marking when specific events occur in the application, denoting the start and end of a region of code for timing purposes, and improving tool's ability to correlate work across threads or devices.

For instance, developers can insert an NVTX marker into the application code to represent the beginning and end of a task. When the application is run by a profiler like Nsight Systems, the timeline generated will show these tasks as individual ranges. This allows developers to easily see how long each task takes and how they interact and overlap with each other. This granularity of information can assist in pinpointing where code parallelism is less than optimal or where bottlenecks are occurring.

Moreover, NVTX offers color-coding for ranges and events, making it easier to spot certain tasks in the timeline. Also, developers can customize the labels assigned to these events or ranges, which can be useful in finding specific parts of the code in the profiling tool.

Another notable function of NVTX is the capacity to interact with the CUDA Memory Allocator. It can track memory allocations, deallocations, and migrations when integrated with the CUDA Memory Allocator, providing more context for memory-related activities in the application's timeline.

Speaking of integration, NVTX is designed for seamless compatibility with NVIDIA's graphical profiling tools like Nsight Systems and others. This direct integration enables smoother handoff between tools and ultimately contributes to a productive proficiency analysis workflow.

In essence, the NVIDIA Tools Extension SDK (NVTX) is an unmissable tool in any GPU developer's performance analysis toolbox. It offers a mechanism to mark certain areas of code and events, helping connect the observation from a profiling tool back to the specific parts of the application code. By doing so, it lets developers focus their optimization efforts in the part of the code that will result in the most significant performance improvements.

Check the provided `addarraysNVTX.cu` code. In this version, we have added a call to `nvtxRangePushA` before the CUDA kernel is launched and a call to `nvtxRangePop` after the kernel has finished executing:

Code 6.2.2: Adding two arrays using NVTX

```
#include <iostream>
#include <math.h>
```

```
#include <nvToolsExt.h>

// CUDA Kernel function to add the elements of two arrays on the GPU
__global__ void add(int n, float *x, float *y)
{
  int index = threadIdx.x;
  int stride = blockDim.x;

  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x, *y;

  // Allocate Unified Memory - accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Start NVTX range for kernel execution
  nvtxRangePushA("CUDA Kernel Execution");

  // Run kernel on 1M elements on the GPU
  add<<<1, 256>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // End NVTX range for kernel execution
  nvtxRangePop();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

Compile and profile it:

```
$ nvcc -o addarraysNVTX addarraysNVTX.cu -lnvToolsExt
$ nsys profile --stats=true --force-overwrite true -o report ./addarraysNVTX
```

Check the new addition to the output:

```
[3/8] Executing 'nvtxsum' stats report
NVTX Range Statistics:
Time (%)  Total Time (ns)  Instances   Avg (ns)     Med (ns)     Min (ns)   Max (ns)   StdDev (ns)   Style           Range
--------  ---------------  ---------  -----------  -----------  ---------  ---------  -----------  -------  ---------------------
100.0       3,978,097          1  3,978,097.0  3,978,097.0  3,978,097  3,978,097         0.0  PushPop  CUDA Kernel Execution
```

6.3 Activity

Previously, we analysed the code for performing 1D convolution (`conv1d.cu` 4.4.2). On that code perform the following:

- Use the NVTX API to monitor the activity of the convolution using cuDNN
- Use the NVTX API to monitor the activity of the convolution using CUDA
- Profile the code using NVIDIA Nsight Systems and report the findings

NVIDIA Nsight Compute

NVIDIA Nsight Compute is an interactive kernel profiler for GPU-accelerated applications. It provides detailed performance metrics and API debugging via a user interface (UI) and command line tool (CLI). The tool provides testing that offers actionable feedback utilized to maximize the performance of CUDA kernels. It is part of NVIDIA's suite of profiling and debugging tools giving developers a detailed and fine-grained view of kernel function's internal GPU computations' statistical behavior to assist in optimization. Official link to NVIDIA Nsight Compute here.

When running GPU-accelerated applications, it's crucial to understand where the performance bottlenecks are, as they can tremendously affect the overall system speed. Nsight Compute comes in to play by helping you identify these bottlenecks. Loaded with features ranging from high-level performance metrics to in-depth hardware and function statistics, this debugging tool provides you the capacity to tune CUDA-based applications to make them run significantly faster.

The NVIDIA Nsight Compute tool performs a whitespace profiling model. It focuses on either a single CUDA kernel at a time to allow for very detailed profiling statistics or screens of "whitespace" through the whole CUDA context in masses. The tool identifies issues with kernel execution or memory accesses that go beyond high-level metrics.

It is worth noting that Nsight Compute runs on several platforms that include Windows, Linux, and MacOS, and it is designed to work with Turing, Volta, Pascal, and Maxwell GPUs. The tool has an edge over NVIDIA Visual Profiler and other past profiling tools due to its advanced capabilities and features.

The user interface is designed to provide an in-depth overview of kernel performance. It neatly organizes the performance metrics in various sections, including an overview, kernel properties, memory workload analysis, compute workload analysis, and more. Each of these sections shows a wealth of data that developers can use to understand the efficiency of their GPU kernels at a granular level.

This tool collects a substantial amount of low-level performance information, but one key feature is its focus on actionable data. Each metric can give developers a clear idea about what the problem is and how to fix it. In the user interface, columns are populated with rule-based guidance that offers tips to improve performance based on the observed metrics. This level of detail can provide developers with a roadmap to modify their code and realize performance improvements.

Another interesting feature of Nsight Compute is its customization flexibility. Using the Nsight Compute SDK, users can create custom profiling metrics or ”sections” to be displayed in the profiling report. This means you can extend the tool's capabilities and modify it to fit your specific use case or application.

Moreover, with Nsight Compute, you can profile your application in a single process or in a multi-process, multi-context mode. Single process profiling is useful for understanding the behavior of a particular kernel in your application. In contrast, the multi-process, multi-context mode is useful when kernels from different CUDA contexts or processes interact on the GPU.

Comparison of profiling results is another beneficial functionality. With Nsight Compute, you can compare the results of multiple profiling reports, which is often useful when iterating on an application's performance. By comparing the profiling results before and after a certain code modification, developers can accurately measure the impact of their changes.

In addition, the tool provides timeline correlation with Nsight Systems, which is another part of NVIDIA's Nsight tool suite. This feature can be useful to get an overall system-wide performance view and analyze a particular kernel's behavior.

The command-line interface (CLI) version of the tool is handy for those who prefer scripting or running automated profiling tasks. The CLI version also allows for remote profiling, which means profiling data can be collected on a remote machine and then transferred and analyzed on a local machine. This can be great for profiling on systems with limited resources or in scenarios where multiple simultaneous profiling tasks are required.

Moreover, Nsight Compute supports interoperability with CUDA-MEMCHECK that provides memory error detection capability. Thus, developers can spot and fix memory-related issues in CUDA applications alongside performance tuning, providing a holistic development and debugging environment.

In summary, NVIDIA Nsight Compute is a powerful interactive kernel profiler aimed at profiling and optimizing CUDA applications. It offers intricate performance metrics, actionable insights, a highly user-friendly interface, and the flexibility of customization which are all instrumental in refining the efficiency of GPU kernels. Regardless of where the application stands in the development process - be it early prototype or near completion - this versatile profiling tool offers value like no other and is a must-have in the arsenal of every CUDA developer. The complete list of features can be found here.

7.1 Nsight Compute CLI

NVIDIA Nsight Compute command line tool is called `ncu`. Launching the tool:

```
$ ncu --version
NVIDIA (R) Compute Command Line Profiler
Copyright (c) 2012-2019 NVIDIA Corporation
Version 2019.1 (Build 25595643)
$ ncu --help
General Options:
    -h [ --help ]                        Print this help message.
    -v [ --version ]                     Print the version number.
    --mode arg (=launch-and-attach)      Select the mode of interaction
                                       with the target application
    ...
```

The Nsight Compute CLI provides a simple interface to collect on a target without using the GUI. The collected data can then be copied to any system and analyzed later. The general command in `ncu` has the following format:

```
$ ncu [options] [program] [program-arguments]
```

7.2 Example

Check the provided `addarrays.cu` 6.2.1 code that we analyzed in the previous chapter. As we aforementioned, this is a simple program that adds the elements of two arrays on the GPU.

Compile and profile it:

```
$ nvcc -o addarrays addarrays.cu
$ ncu ./addarrays
```

After running this command, Nsight Compute will profile your application and print a summary of the profiling results to the console. You can also use the `-o` option to save the profiling data to a file:

```
$ ncu -o report ./addarrays
```

7.2.1 Other useful commands

You can find below a list of other useful commands for Nsight Compute:

- Tells Nsight Compute to print a summary of the profiling results for each kernel:

  ```
  $ ncu --summary per-kernel ./addarrays
  ```

- If you're interested in the number of instructions executed and the memory bandwidth used:

  ```
  $ ncu --metrics inst_executed,mem_bandwidth ./addarrays
  ```

- Collect and display a specific set of metrics:

  ```
  $ ncu --metrics smsp__inst_executed.avg.pct_of_peak_sustained_active,\
  smsp__cycles_elapsed.avg.per_second ./addarrays
  ```

  - average percentage of peak sustained active instructions executed
  - average number of cycles elapsed per second

- Collect and display all metrics:

  ```
  $ ncu --metrics all ./addarrays
  ```

- Collect and display a specific section of the report:

  ```
  $ ncu --sections SpeedOfLight  ./addarrays
  ```

  - "SpeedOfLight" section includes metrics related to the theoretical and achieved performance of the GPU.

- Profile a specific kernel:

  ```
  $ ncu --kernel-id ::add: ./addarrays
  ```

  - average percentage of peak sustained active instructions executed
  - average number of cycles elapsed per second

7.3 Activity

Previously, we analysed the code for performing matrix multiplication (`matmul.cu` 3.4.1). On that code perform the following:

- Use the Nsights Compute to monitor the activity of the program .
- Use the Nsights Compute to monitor the activity of the CUDA kernel.