Unit-V

Kernel Execution Model and Optimization Strategies: Kernel execution with CUDA streams, Pipelining the GPU execution, The CUDA callback function, CUDA streams with priority, Kernel execution time estimation using CUDA events, CUDA dynamic parallelism, Grid level cooperative groups, Multi Process Service.

Kernel execution with CUDA streams

CUDA streams are a fundamental concept in NVIDIA's CUDA programming model. They provide a way to overlap the execution of multiple operations on a GPU, enabling concurrent processing and improved performance. CUDA streams allow developers to organize and manage asynchronous execution of tasks, such as kernel launches and memory transfers, on the GPU.

The usage of CUDA streams

- Streams are created using the **cudaStreamCreate** function

  cudaStream_t   stream;

  cudaStreamCreate(&stream);

- Kernel Launch with Streams: When launching a kernel, you can specify the stream in which the kernel should execute.

  kernelName<<<gridDim, blockDim, sharedMem, stream>>>(arg1, arg2, ...);

  kernelName: The name of the kernel function you want to launch.
  <<<gridDim, blockDim, sharedMem, stream>>>:
  gridDim: The dimensions of the grid. It can be a 1D, 2D, or 3D structure.
  blockDim: The dimensions of each block within the grid. Similar to gridDim, it can be 1D, 2D, or 3D.
  sharedMem: An integer specifying the amount of dynamic shared memory to be allocated per block in bytes. This is optional and can be omitted if your kernel does not use shared memory.
  stream: A CUDA stream in which the kernel should be executed. This is an optional parameter and can be omitted if you are not working with streams.
  arg1, arg2, ...: The arguments to the kernel function.

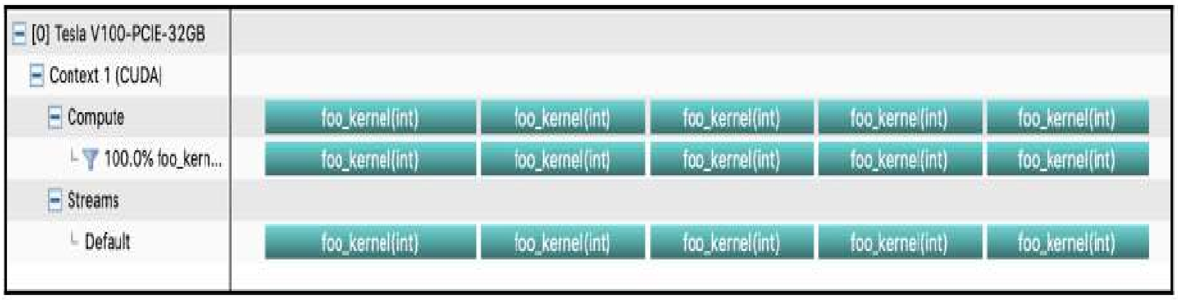
- **Destroying Streams:** Streams should be destroyed after use.

```
cudaStreamDestroy(stream);
```

First, let's write an application that uses the default CUDA stream, as follows:

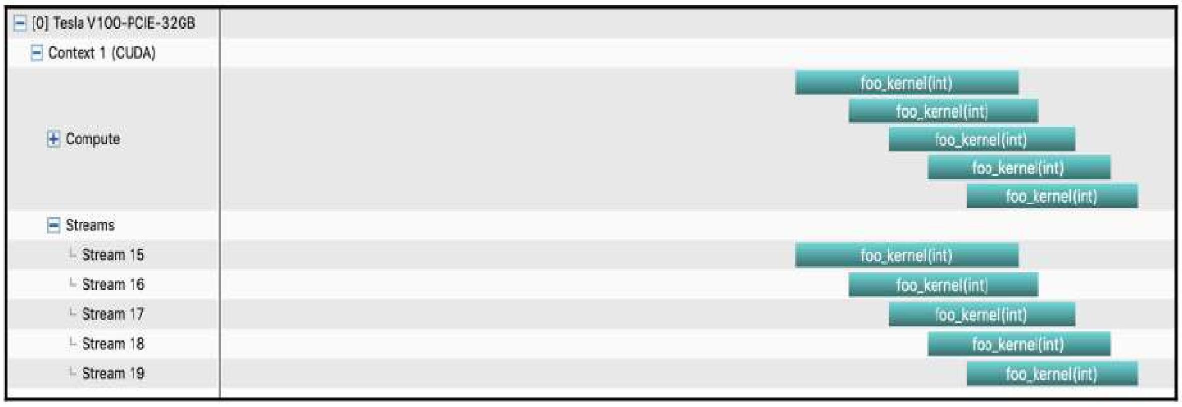
```
 __global__ void foo_kernel(int step)
 {
 printf("loop: %d\n", step);
 }
int main()
{
for (int i = 0; i < 5; i++) // CUDA kernel call with the default stream
foo_kernel<<< 1, 1, 0, 0 >>>(i);
 cudaDeviceSynchronize();
return 0;
}
```

As you can see in the code, we call the kernel function with the stream ID as 0, because the identification value of the default stream is 0



Multiple CUDA Streams

```
__global__ void foo_kernel(int step)
{
printf("loop: %d\n", step);
}
 int main()
{
int n_stream = 5;
cudaStream_t *ls_stream;
ls_stream = (cudaStream_t*) new cudaStream_t[n_stream];
// create multiple streams
 for (int i = 0; i < n_stream; i++)
cudaStreamCreate(&ls_stream[i]);
// execute kernels with the CUDA stream each
 for (int i = 0; i < n_stream; i++)
foo_kernel<<< 1, 1, 0, ls_stream[i] >>>(i);
// synchronize the host and GPU
cudaDeviceSynchronize();
 // terminates all the created CUDA streams
for (int i = 0; i < n_stream; i++)
cudaStreamDestroy(ls_stream[i]);
delete [] ls_stream;
 return 0;
 }
```



As you can see at the bottom of the screenshot, five individual streams execute the same kernel function concurrently and their operations are overlapped with each other. From this, we can discern two features of the streams, as follows:

1. Kernel executions are asynchronous with the host.

2. CUDA operations in different streams are independent of each other

Stream-level synchronization:

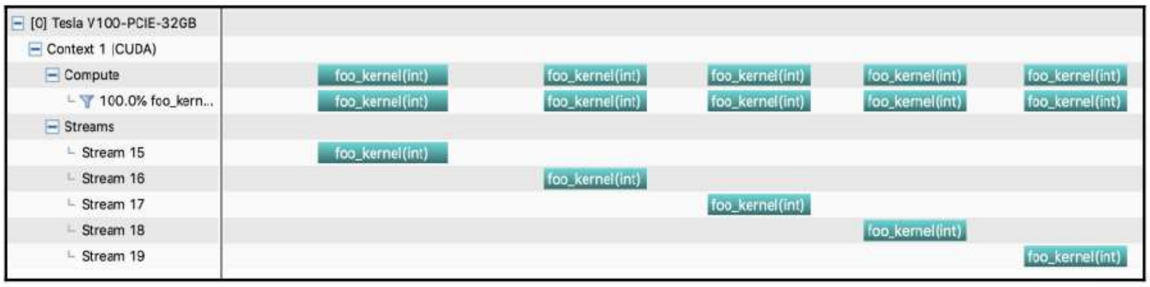
In the context of CUDA (Compute Unified Device Architecture), stream-level synchronization refers to coordinating the execution of multiple CUDA streams. CUDA streams are sequences of operations that execute on a GPU. Each stream can contain a series of kernel launches, memory transfers, and other GPU operations.

The previous example shows concurrent operations without synchronization within the loop. However, we can halt the host to execute the next kernel execution by using the cudaStreamSynchronize() function. The following code shows an example of using stream synchronization at the end of the kernel execution:

// execute kernels with the CUDA stream each

```
for (int i = 0; i < n_stream; i++)
{
foo_kernel<<< 1, 1, 0, ls_stream[i] >>>(i);
cudaStreamSynchronize(ls_stream[i]);
}
```

The following screenshot shows the result:



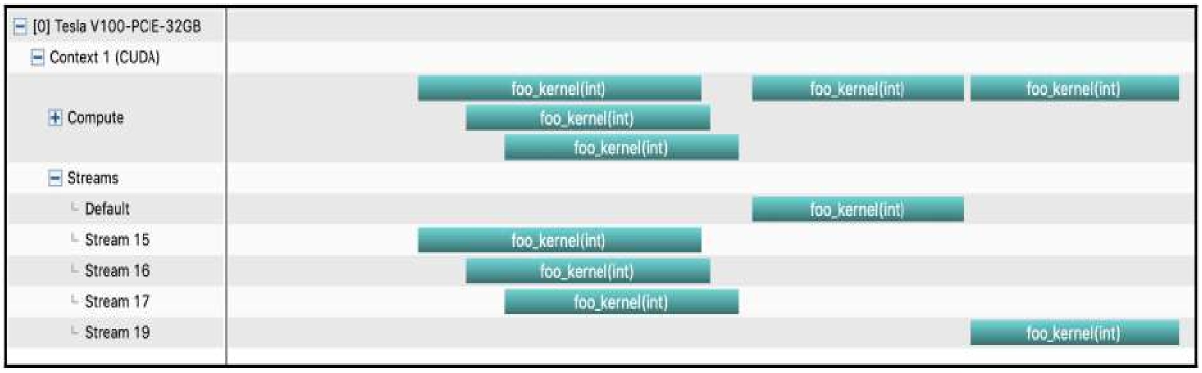
As you can see, all the kernel executions have no overlapping points, although they are executed with the different streams. Using this feature, we can let the host wait for the specific stream operation to start with the result.

Working with the default stream

In CUDA, you can use multiple streams to achieve concurrent execution of GPU operations. Each stream operates independently, allowing you to overlap the execution of kernels and memory transfers. However, the default stream (stream 0) operates synchronously, meaning that operations within the default stream are executed sequentially.

To use multiple streams in CUDA, you typically create and manage them explicitly in your code. Here's a simple example that demonstrates the use of multiple streams:

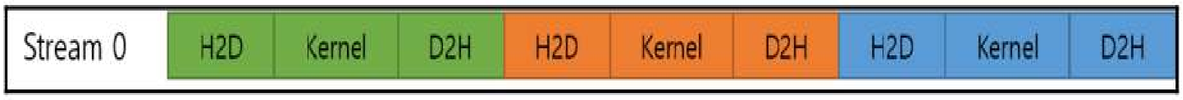
```
for (int i = 0; i < n_stream; i++)
   if (i == 3)
        foo_kernel<<< 1, 1, 0, 0 >>>(i);
 else
      foo_kernel<<< 1, 1, 0, ls_stream[i] >>>(i)
```



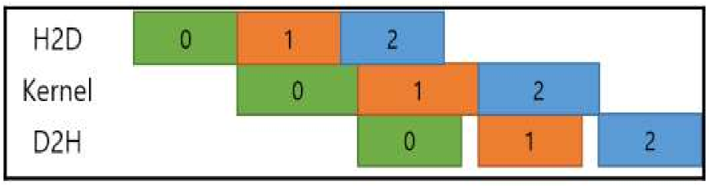
Pipelining the GPU execution

In computing, pipelining is a technique where the processing of multiple instructions or tasks is overlapped to improve overall throughput. When we execute the kernel function, we need to transfer data from the host to the GPU. Then, we transfer the result back from the GPU to the host. However, the kernel execution is basically asynchronous in that the host and GPU can operate concurrently with each other. One of the major benefits of multiple streams is overlapping the data transfer with the kernel execution. By overlapping the kernel operation and data transfer, we can conceal the data transfer overhead and increase overall performance.

The following diagram shows an example of iterative operations that transfer data between the host and kernel executions:

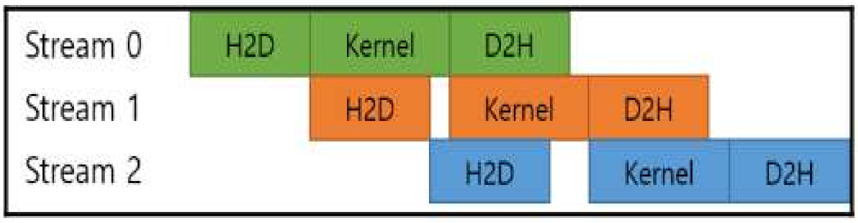


The following diagram shows the operation when the data transfer can be executed like a normal kernel operation, and handled along with the stream:



Regarding CUDA streams, all CUDA operations—data transfers and kernel executions—are sequential in the same stream. However, those can operate simultaneously along with the different streams.

The following diagram shows overlapped data transfer with the kernel operation for multiple streams:



To enable such a pipelining operation, CUDA has three prerequisites:

1. The host memory should be allocated as pinned memory—CUDA provides the cudaMallocHost() and cudaFreeHost() functions for this purpose. Or static memory(Normal Array declaration).

2. Transfer data between the host and GPUs without blocking the host—CUDA provides the cudaMemcpyAsync() function for this purpose.

3. Manage each operation along with the different CUDA streams to have concurrent operations

Building a pipelining execution

```
const int N = 5; // size of the vectors

// CUDA kernel for vector addition

__global__ void vectorAdd(float *a, float *b, float *c, int size)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
      c[tid] = a[tid] + b[tid];
}

int main()
{
  const int blockSize = 256;

  const int numBlocks = (N + blockSize - 1) / blockSize;

  // Allocate and initialize host vectors
 int h_a[5] = {1, 1, 1, 1, 1}, h_b[5] = {2, 2, 2, 2, 2}, h_c[5];

  // Allocate device vectors
  int *d_a, *d_b, *d_c;
  cudaMalloc((void**)&d_a, N * sizeof(int));
  cudaMalloc((void**)&d_b, N * sizeof(int));
  cudaMalloc((void**)&d_c, N * sizeof(int));

  // Create CUDA streams
  cudaStream_t stream1, stream2;
  cudaStreamCreate(&stream1);
  cudaStreamCreate(&stream2);
  // Copy data to device asynchronously using streams
  cudaMemcpyAsync(d_a, h_a, N * sizeof(int), cudaMemcpyHostToDevice, stream1);
  cudaMemcpyAsync(d_b, h_b, N * sizeof(int), cudaMemcpyHostToDevice, stream2);

  // Launch the kernel asynchronously using stream1
  vectorAdd<<<numBlocks, blockSize, 0, stream1>>>(d_a, d_b, d_c, N);

  // Copy data back to host asynchronously using stream2
  cudaMemcpyAsync(h_c, d_c, N * sizeof(int), cudaMemcpyDeviceToHost, stream2);

  // Synchronize streams to ensure all operations are completed
  cudaStreamSynchronize(stream1);
  cudaStreamSynchronize(stream2);

  // Verify the result
  for (int i = 0; i < N; ++i) {
     printf("%f ", h_c[i]);
  }

  // Free resources
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  // Destroy streams
  cudaStreamDestroy(stream1);
  cudaStreamDestroy(stream2);

  return 0;

}
```

The CUDA callback function

CUDA callback functions are functions provided by the CUDA runtime that allow you to register your own functions to be called at specific points during the execution of a CUDA application. Callbacks can used in two ways: the GPU requests some data/computation from the CPU, or the GPU feeds debugging/checkpointing information to the CPU.

The main CUDA callback functions include:

- **Error Callbacks (cudaSetErrorCallback):** You can register a callback function to be called whenever a CUDA runtime API call returns an error. This can be useful for custom error handling or logging.

  cudaSetErrorCallback(errorCallback, nullptr);

  errorCallback: This is a pointer to the user-defined error callback function.The function takes three parameters:

  error: The CUDA error code associated with the error.

  description: A string describing the error.

userData: A user-specified pointer that you can use to pass additional data to the callback function.**nullptr**: is the **userData** parameter passed to the callback function. It can be used to pass additional data to the callback if needed.

- **Event Callbacks (cudaEventCreateWithCallback):** You can register a callback function to be called when a CUDA event is recorded. This is useful for synchronizing CPU and GPU tasks.

cudaEventCreateWithCallback(&event, cudaEventDefault, eventCallback, nullptr);

&event: This is the address of the **cudaEvent_t** variable where the newly created event will be stored.

cudaEventDefault: This parameter specifies the flags associated with the event. **cudaEventDefault** is a default flag value, and you can customize it based on your requirements.

eventCallback: This is the callback function that will be called when the event reaches the specified state. It's a user-defined function that you need to implement with the following parameters.

stream: The stream with which the event is associated.

status: The status of the event.

userData: A user-specified pointer that you can use to pass additional data to the callback function. **Nullptr** is the **userData** parameter passed to the callback function. It can be used to pass additional data to the callback if needed. In this case, it's set to **nullptr**, indicating that no additional data is being passed.

// Error callback function

```
void errorCallback(cudaError_t error, const char *description, void *userData)

{

   printf( "CUDA error: %s ", description);

}
// Event callback function

void eventCallback(cudaStream_t stream, cudaError_t status, void *userData)
 {

Printf("CUDA Event Callback: Event completed with status %s", cudaGetErrorString(status) );

}
__global__ void myKernel() {

   // Some kernel code

}

int main()
{
   // Set error callback
   cudaSetErrorCallback(errorCallback, nullptr);

   // Create event with callback
   cudaEvent_t event;
   cudaEventCreateWithCallback(&event, cudaEventDefault, eventCallback, nullptr);

   // Allocate device data
   int *d_data;
   cudaMalloc((void**)&d_data, sizeof(int));

   // Launch kernel asynchronously
   myKernel<<<1, 1, 0, nullptr>>>(d_data);

   // Record event in the default stream
   cudaEventRecord(event, nullptr);

   // Synchronize with the event
   cudaEventSynchronize(event);

   // Cleanup
   cudaFree(d_data);
   cudaEventDestroy(event);

   return 0;
}
```

CUDA streams with priority

By default, all CUDA streams have equal priority so they can execute their operations in the right order. However, starting with CUDA 11.0, NVIDIA introduced the concept of stream priorities. Stream priorities allow you to assign different priorities to CUDA streams, influencing the order in which they are scheduled for execution. Higher priority streams are scheduled to run before lower priority streams.

Priorities in CUDA

➢ To use streams with priorities, we need to obtain the available priorities from the GPU first. We can obtain these using the cudaDeviceGetStreamPriorityRange() function. Its output is two numeric values, which are the lowest and highest priority values.

cudaError_t cudaDeviceGetStreamPriorityRange(int* leastPriority, int* greatestPriority);

- **leastPriority**: Pointer to an integer in which the function stores the minimum supported stream priority.
- **greatestPriority**: Pointer to an integer in which the function stores the maximum supported stream priority.

The **cudaDeviceGetStreamPriorityRange** function returns **cudaSuccess** on success or an error code if there's an issue.

➢ Then, we can create a priority stream using the cudaStreamCreateWithPriority() function .This function is part of the CUDA runtime API, and it allows you to create a CUDA stream with a specified priority. This function was introduced in CUDA 10.0 and is used to create streams with different priorities, influencing the order in which they are scheduled for execution on the GPU.

cudaError_t cudaStreamCreateWithPriority(cudaStream_t* pStream, unsigned int flags, int priority);

- **pStream**: Pointer to a variable that will receive the newly created stream.
- **flags**: Flags that can be used to specify additional stream creation options. Typically, this is set to **cudaStreamDefault**.
- **priority**: The priority of the stream. A higher priority value indicates a higher-priority stream.

Stream execution with priorities

```
void checkCudaError(cudaError_t cudaStatus, const char* errorMessage)
{
  if (cudaStatus != cudaSuccess) {
```

```
printf("%s", cudaGetErrorString(cudaStatus) );
  }
}

int main() {
  // Get the priority range supported by the device
  int leastPriority, greatestPriority;
  cudaError_t cudaStatus = cudaDeviceGetStreamPriorityRange(&leastPriority, &greatestPriority);
  checkCudaError(cudaStatus, "cudaDeviceGetStreamPriorityRange failed");
  // Create streams with different priorities
  int highPriority = greatestPriority;
  int lowPriority = leastPriority;
  cudaStream_t  streamHigh, streamLow;
  cudaStatus = cudaStreamCreateWithPriority(&streamHigh, cudaStreamDefault, highPriority);
  checkCudaError(cudaStatus, "cudaStreamCreateWithPriority (High Priority) failed");
  cudaStatus = cudaStreamCreateWithPriority(&streamLow, cudaStreamDefault, lowPriority);
  checkCudaError(cudaStatus, "cudaStreamCreateWithPriority (Low Priority) failed");
  // Use the created streams for kernel launches or memory transfers
  // Destroy the streams when done
  cudaStatus = cudaStreamDestroy(streamHigh);
  checkCudaError(cudaStatus, "cudaStreamDestroy (High Priority) failed");
  cudaStatus = cudaStreamDestroy(streamLow);
  checkCudaError(cudaStatus, "cudaStreamDestroy (Low Priority) failed");
  return 0;
}
```

Kernel execution time estimation using CUDA events

CUDA events are a mechanism provided by the CUDA API to allow for precise timing and synchronization in GPU applications. They can be used to measure the execution time of CUDA kernels and to synchronize the CPU with the GPU.

Here's an overview of how CUDA events work:

- **Creating Events:** To use CUDA events, you need to create them. This is done using the **cudaEventCreate** function:

cudaEvent_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

- **Recording Events:** You can record events at specific points in your code, typically before and after the region of code you want to measure. This is done using the **cudaEventRecord** function:

  cudaEventRecord(start);

  // Code to be timed

  cudaEventRecord(stop);

- **Synchronizing Events:** To ensure that the recorded events have completed, you can use the **cudaEventSynchronize** function:

  cudaEventSynchronize(stop);

  This is especially important when measuring the execution time of a kernel to make sure that the GPU has finished executing the kernel.

- **Calculating Elapsed Time:** The elapsed time between two events can be calculated using the **cudaEventElapsedTime** function:

  float elapsedTime;

  cudaEventElapsedTime(&elapsedTime, start, stop);

  The **elapsedTime** variable will contain the time between the **start** and **stop** events in milliseconds.

- **Destroying Events:** Once you're done using the events, it's important to destroy them to free up resources:

  cudaEventDestroy(start);

  cudaEventDestroy(stop);

Sample Program

```
__global__ void myKernel()
 {
   // Some computation
}

int main()
{
   cudaEvent_t start, stop;
   cudaEventCreate(&start);
   cudaEventCreate(&stop);
```

```
    // Record the start event
    cudaEventRecord(start);

    // Launch the kernel
    myKernel<<<1, 1>>>();

    // Record the stop event
    cudaEventRecord(stop);

    // Synchronize to ensure accurate timing
    cudaEventSynchronize(stop);

    // Calculate and print the elapsed time
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, start, stop);

    // Destroy the events
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}
```

Multiple stream estimation

When dealing with multiple CUDA streams, you might want to estimate the overall execution time, taking into account the asynchronous nature of stream execution. The basic idea is to use events to measure the time taken by each stream and synchronize with the CPU to ensure accurate measurements

```
// Example kernel
__global__ void myKernel(int* data, int size)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

      data[tid] = data[tid] * 2;

}

int main()
{
  const int dataSize = 5;
  const int blockSize= 256;
  const int gridSize = (dataSize + blockSize - 1) / blockSize;

  int h_data[5] = {1, 2, 3, 4, 5};
```

```
int* d_data[2];  // Two streams

for (int i = 0; i < 2; ++i) {
   // Allocate device memory for each stream
   cudaMalloc((void**)&d_data[i], dataSize * sizeof(int));
   // Copy data from host to device
   cudaMemcpy(d_data[i], h_data, dataSize * sizeof(int), cudaMemcpyHostToDevice);
}

cudaStream_t stream[2];

for (int i = 0; i < 2; ++i) {
   // Create streams
   cudaStreamCreate(&stream[i]);
}

cudaEvent_t start[2], stop[2];

for (int i = 0; i < 2; ++i) {
   // Create events
   cudaEventCreate(&start[i]);
   cudaEventCreate(&stop[i]);
}

// Record the start events
for (int i = 0; i < 2; ++i) {
   cudaEventRecord(start[i], stream[i]);
}

// Launch the kernels in parallel streams
for (int i = 0; i < 2; ++i) {
   myKernel<<<gridSize, blockSize, 0, stream[i]>>>(d_data[i], dataSize);
}

// Record the stop events
for (int i = 0; i < 2; ++i) {
   cudaEventRecord(stop[i], stream[i]);
}

// Synchronize to ensure accurate timing
for (int i = 0; i < 2; ++i) {
   cudaStreamSynchronize(stream[i]);
}

// Calculate and print the elapsed time for each stream
for (int i = 0; i < 2; ++i) {
   float elapsedTime;
```

```
        cudaEventElapsedTime(&elapsedTime, start[i], stop[i]);
        printf("Stream %d execution time is %f ms\n", i, elapsedTime);
    }

    // Clean up
    for (int i = 0; i < 2; ++i) {
        cudaFree(d_data[i]);
        cudaStreamDestroy(stream[i]);
        cudaEventDestroy(start[i]);
        cudaEventDestroy(stop[i]);
    }

    return 0;
}
```

CUDA dynamic parallelism

Dynamic parallelism in CUDA refers to the ability of a CUDA kernel to launch other kernels. In traditional CUDA programming, kernels are launched from the host, and they run on the GPU. With dynamic parallelism, a kernel running on the GPU can itself launch additional kernels. CUDA dynamic parallelism (CDP) is a device runtime feature that enables nested calls from device functions

```
__global__ void childKernel(int* data, int index)
{
    data[index] = data[index] + 1;
}

__global__ void parentKernel(int* data, int size)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
        childKernel<<<1, 1>>>(data, tid);
        cudaDeviceSynchronize();  // Ensure child kernel completes
}

int main() {
    const int dataSize = 5;
    const int blockSize = 256;
    const int gridSize = (dataSize + blockSize - 1) / blockSize;

    int h_data[dataSize] = {1, 2, 3, 4, 5};
    int* d_data;

    cudaMalloc((void**)&d_data, dataSize * sizeof(int));
    cudaMemcpy(d_data, h_data, dataSize * sizeof(int), cudaMemcpyHostToDevice);

    // Launch parent kernel
    parentKernel<<<gridSize, blockSize>>>(d_data, dataSize);
    cudaDeviceSynchronize();

    // Copy data back to host
    cudaMemcpy(h_data, d_data, dataSize * sizeof(int), cudaMemcpyDeviceToHost);

    // Print modified values
    for (int i = 0; i < dataSize; ++i)
{
        printf("Modified value at index %d: %d\n", i, h_data[i]);
    }

    cudaFree(d_data);

    return 0;
}
```

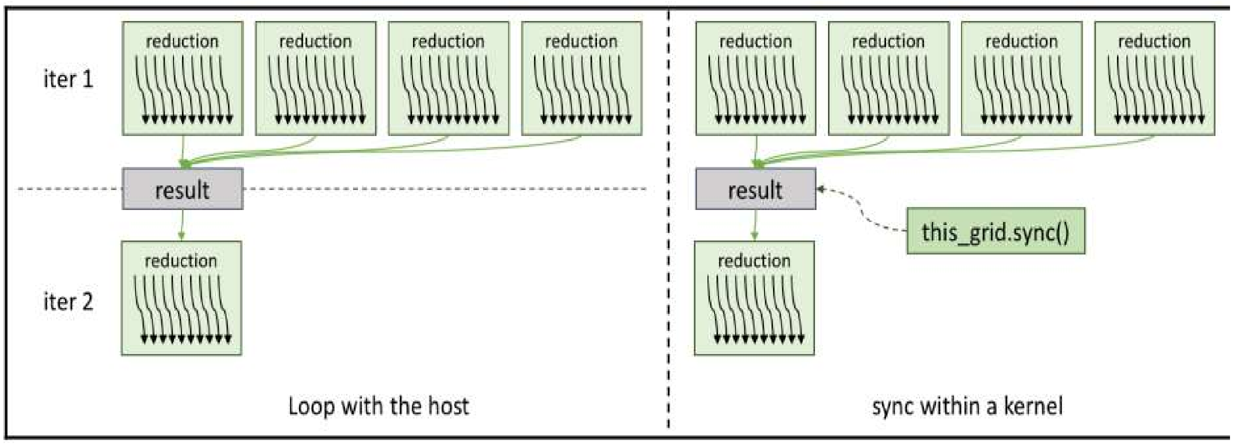
Recursion using Dynamic Parallelism

```
const int blockSize = 5;
const int dataSize = 5;
// Declaration of the recursive_kernel function
__global__ void recursive_kernel(int* data, int size, int depth);
int main() {
    int h_data[dataSize] = {1, 2, 3, 4, 5};
    int* d_data;
    cudaMalloc((void**)&d_data, dataSize * sizeof(int));
    cudaMemcpy(d_data, h_data, dataSize * sizeof(int), cudaMemcpyHostToDevice);
    // Launch the recursive kernel from the host
    int gridDim = (dataSize + blockSize - 1) / blockSize;
    recursive_kernel<<<gridDim, blockSize>>>(d_data, dataSize, 3);
    cudaDeviceSynchronize();
    // Copy the result back to the host
    cudaMemcpy(h_data, d_data, dataSize * sizeof(int), cudaMemcpyDeviceToHost);
    // Print the modified data
    printf("Modified data: ");
    for (int i = 0; i < dataSize; ++i) {
        printf("%d ", h_data[i]);
    }
    printf("\n");
    // Cleanup
    cudaFree(d_data);

    return 0;
}

// Implementation of the recursive_kernel function
__global__ void recursive_kernel(int* data, int size, int depth)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (depth > 0 && tid < size)
{
        data[tid] *= 2;
        recursive_kernel<<<1, size>>>(data, size, depth - 1);
    }
}
```

Grid level cooperative groups

Cooperative groups in CUDA allow you to work with groups of threads in a more flexible and efficient manner. Cooperative groups can be categorized by their grouping targets: warp-level, block-level, and grid-level groups Grid level cooperative groups, in particular, provide a way to synchronize threads across multiple blocks within a grid.

Reduction example with grid.sync()



For implementing cooperative groups we require two functions: grid_group & grid.sync().

➢ The **grid_group** is part of the **cooperative_groups** library in CUDA, introduced to provide a more flexible way for threads in different thread blocks to synchronize and collaborate within a grid. This is particularly useful when you need cooperation or communication between different thread blocks.

To make all the thread blocks in grid_group synchronize, the total number of active thread blocks in the grid should not exceed the number of maximum active blocks for the kernel function and the device. The maximum active block size on a GPU is a multiplication of the maximum amount of active blocks per SM and the number of streaming multiprocessors. The violation of this rule can result in deadlock or undefined behavior. We can obtain the maximum amount of active thread blocks of a kernel function per SM using the cudaOccupancyMaxActiveBlocksPerMultiprocessor() function, by passing the kernel function and block size information.

➢ **grid.sync()** is used to synchronize all threads within the grid. All threads within the grid must reach the **grid.sync()** statement. No thread in any block is allowed to proceed beyond the **grid.sync()** until all threads in all blocks have reached it.Once all threads in the grid have reached this point, they will be synchronized.This synchronization allows you to perform collective operations or share data between blocks.

Example

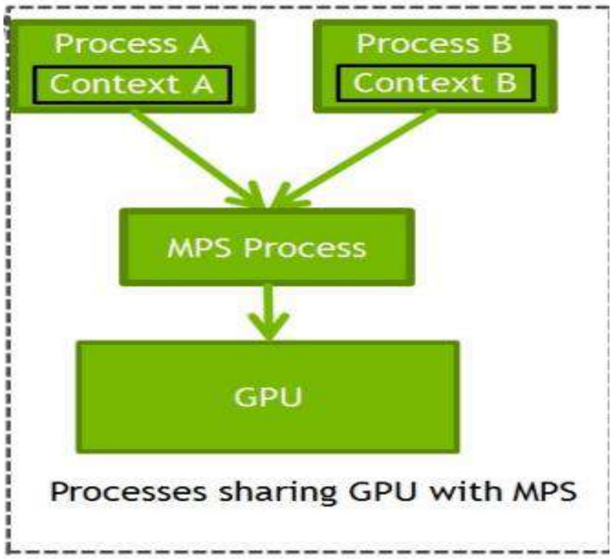
```
namespace cg = cooperative_groups;
__global__ void gridLevelKernel(int* data, int size)
 {
   cg::grid_group grid = cg::this_grid();
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   // Work on data assigned to the block
   // Synchronize all threads within the grid
   grid.sync();
   // Perform some computation collectively across all blocks
   // Synchronize again
```

```
    grid.sync();
}
int main() {
    const int blockSize = 256;
    const int gridSize = 4;
    const int dataSize = blockSize * gridSize;
    int* d_data;
    cudaMalloc((void**)&d_data, dataSize * sizeof(int));
    // Launch the kernel with multiple blocks
    gridLevelKernel<<<gridSize, blockSize>>>(d_data, dataSize);
    // Wait for the kernel to finish
    cudaDeviceSynchronize();
    // Cleanup
    cudaFree(d_data);
    return 0;
}
```

Multi-Process Service

- Multi-Process Service is a feature provided by NVIDIA GPUs that allows concurrent execution of kernels from different CPU processes on the same GPU. This is particularly beneficial when multiple applications or processes are trying to utilize the GPU simultaneously.
- By default, when multiple CPU processes try to use the GPU concurrently, the GPU time-slices between them. This means that the GPU switches between executing kernels from different processes over time.The default time-slicing approach can lead to underutilization of GPU resources because even if a kernel doesn't fully use the GPU's compute resources, it has to wait for its turn in the time slice.
- To address this issue and fully utilize GPU resources, MPS mode is introduced. In MPS mode, different CPU processes can execute their kernels simultaneously on the GPU. This allows for better parallelism and resource utilization.It is particularly useful in scenarios where you have multiple applications running concurrently, each using the GPU.

- While MPS mode can be beneficial, it's essential to consider factors such as memory usage, context switching, and potential contention for resources. Proper management of MPS mode is required to avoid resource conflicts between concurrent processes.



To have the multi-processes application scenario to a single GPU we will use MPI.MPI is commonly used for parallel computing, where multiple processes work together to solve a larger computational problem. These processes can run on different CPU cores, GPUs, or even on different nodes within a cluster.

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

int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   printf("Hello from process %d of %d\n", rank, size);

   MPI_Finalize();
   return 0;
}
```

Advanced Concepts in CUDA

Scalable Multi-GPU Programming: Solving a linear equation using Gaussian elimination, GPU Direct peer to peer, GPU Direct RDMA, CUDA streams . Parallel Programming Patterns in CUDA: Matrix multiplication optimization, Convolution, Prefix sum (scan), Compact and split, Quicksort in CUDA using dynamic parallelism, Radix sort. Overview of Libraries : cuBLAS, cuRAND , cuFFT, NPP, cuDNN

Solving a linear equation using Gaussian elimination

Gaussian elimination is a method used to solve a system of linear equations. The idea is to transform the system into an equivalent, but simpler, system in order to find the solution. Here's a general outline of the Gaussian elimination process:

$$x + y + z = 3$$

$$x + 2y + 3z = 0$$

$$x + 3y + 2z = 3$$

Equation 1: System of linear equations to solve

- **Create an augmented matrix**:

  Combine the coefficient matrix and the constant terms into one matrix.

$$\begin{aligned} x + y + z &= 3 \\ x + 2y + 3z &= 0 \\ x + 3y + 2z &= 3 \end{aligned} \quad \rightarrow \quad \left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 1 & 2 & 3 & 0 \\ 1 & 3 & 2 & 3 \end{array}\right]$$

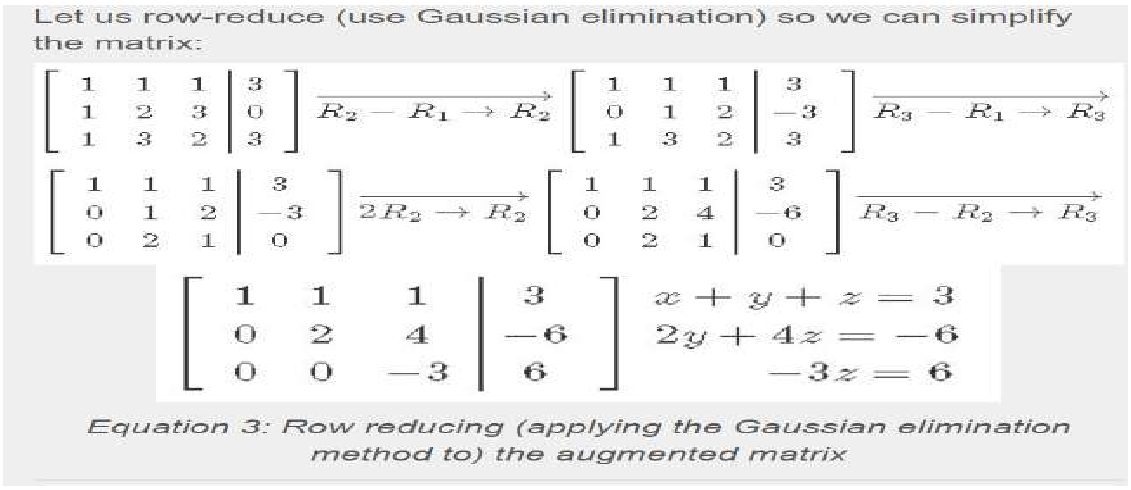
Equation 2: Transcribing the linear system into an augmented matrix

- **Apply row operations to transform the matrix into upper triangular matrix:**

  Swap rows if necessary to move the row with the leftmost non-zero element to the top.

  Multiply a row by a non-zero scalar.

  Add or subtract multiples of one row from another to eliminate entries below the pivot.



Let us row-reduce (use Gaussian elimination) so we can simplify the matrix:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 1 & 2 & 3 & 0 \\ 1 & 3 & 2 & 3 \end{array}\right] \xrightarrow{R_2 - R_1 \to R_2} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & 1 & 2 & -3 \\ 1 & 3 & 2 & 3 \end{array}\right] \xrightarrow{R_3 - R_1 \to R_3}$$

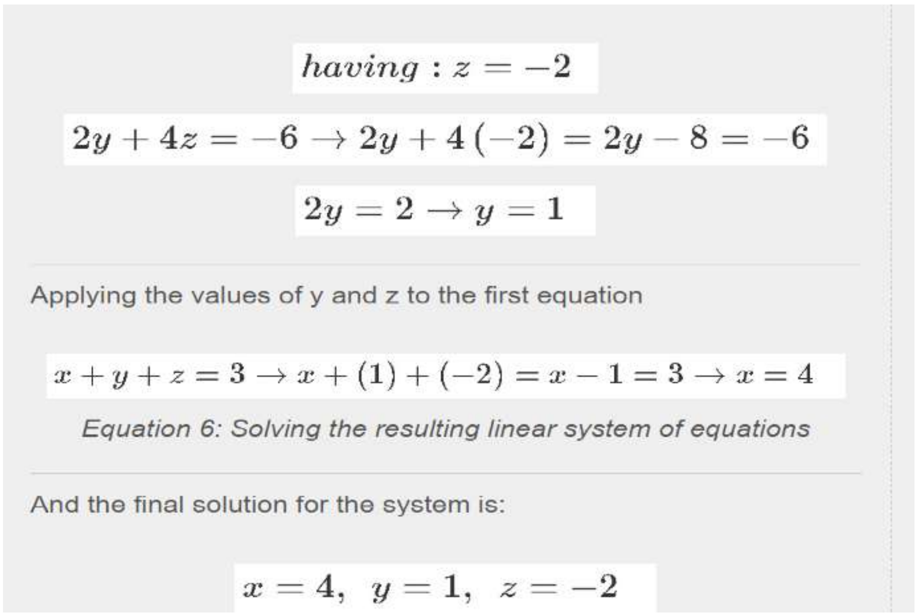
$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & 1 & 2 & -3 \\ 0 & 2 & 1 & 0 \end{array}\right] \xrightarrow{2R_2 \to R_2} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & 2 & 4 & -6 \\ 0 & 2 & 1 & 0 \end{array}\right] \xrightarrow{R_3 - R_2 \to R_3}$$

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & 2 & 4 & -6 \\ 0 & 0 & -3 & 6 \end{array}\right] \quad \begin{aligned} x + y + z &= 3 \\ 2y + 4z &= -6 \\ -3z &= 6 \end{aligned}$$

Equation 3: Row reducing (applying the Gaussian elimination method to) the augmented matrix

➢ **Back substitution**:

Starting from the last equation, solve for the variables one by one.



$$having : z = -2$$

$$2y + 4z = -6 \to 2y + 4(-2) = 2y - 8 = -6$$

$$2y = 2 \to y = 1$$

Applying the values of y and z to the first equation

$$x + y + z = 3 \to x + (1) + (-2) = x - 1 = 3 \to x = 4$$

Equation 6: Solving the resulting linear system of equations

And the final solution for the system is:

$$x = 4, \quad y = 1, \quad z = -2$$

Sequential CPU code

```
for (int n = 0; n < N; n++) {
  // M: number of equations, N: number of unknowns
  for (int pr = 0; pr < M; pr++) {
    // finding the pivot row
    // if pr satisfies condition for pivot i.e. is non zero
    if (AB[pr * N + n] != 0) {
      // Pivot found in column n, at row pr
      break;
```

```
        }
    }

    for (int r = 0; r < M; r++) {
        // reduce all other eligible rows using the pivot row
        double ratio = AB[r * N + n] / AB[pr * N + n];
        for (int nn = n; nn < N + 1; nn++) {
            AB[r * N + nn] -= (ratio * AB[pr * N + nn]);
        }
    }
}
```

Multi-GPU Code:

```
#include <stdio.h>
#include <cuda_runtime.h>
#define N 3 // Number of equations
#define M 3 // Number of unknowns
__global__ void forwardElimination(float *mat, int pivot_column) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < M && row > pivot_column) {
        double ratio = mat[row * N + pivot_column] / mat[pivot_column * N + pivot_column];
        for (int col = pivot_column; col <= N; col++) {
            mat[row * N + col] -= ratio * mat[pivot_column * N + col];
        }
    }
}

int main() {
    float *h_mat, *d_mat;
    // Allocate memory on the host
    h_mat = (float*)malloc(M * N * sizeof(float));

    // Initialize host matrix (replace with your input matrix)
    // Allocate memory on each GPU
    cudaMalloc(&d_mat, M * N * sizeof(float));
    // Transfer data from host to device
    cudaMemcpy(d_mat, h_mat, M * N * sizeof(float), cudaMemcpyHostToDevice);
    // Gaussian elimination on multiple GPUs
    for (int pivot_column = 0; pivot_column < N - 1; pivot_column++) {
      // Launch a kernel on each GPU to perform forward elimination
      int num_blocks = (M + 255) / 256;
      forwardElimination<<<num_blocks, 256>>>(d_mat, pivot_column);
          // Synchronize GPUs before moving to the next pivot column
      cudaDeviceSynchronize();
    }
    // Transfer result from device to host
    cudaMemcpy(h_mat, d_mat, M * N * sizeof(float), cudaMemcpyDeviceToHost);
    // Print the result
    // Free memory on the host and device
    free(h_mat);
    cudaFree(d_mat);
    return 0;
}
```

GPU Direct peer to peer

In a typical GPU setup, when data needs to be transferred between two GPUs, it often involves copying the data from GPU 1 to the system memory (host memory), and then copying it from the system memory to GPU 2. This process is known as a "staging" transfer and can introduce latency and consume additional system resources. So, GPU Direct Peer-to-Peer (P2P) is a technology that allows direct communication between GPUs (Graphics Processing Units) without involving the CPU (Central Processing Unit) or main system memory. This can significantly improve data transfer speeds between GPUs and reduce latency in certain parallel computing tasks.

GPU Direct can be classified into the following major categories:

- **Peer to peer (P2P) transfer between GPU:** Allows CUDA programs to use high speed Direct Memory Transfer (DMA) to copy data between two GPUs in the same system. It also allows optimized access to the memory of other GPUs within the same system.

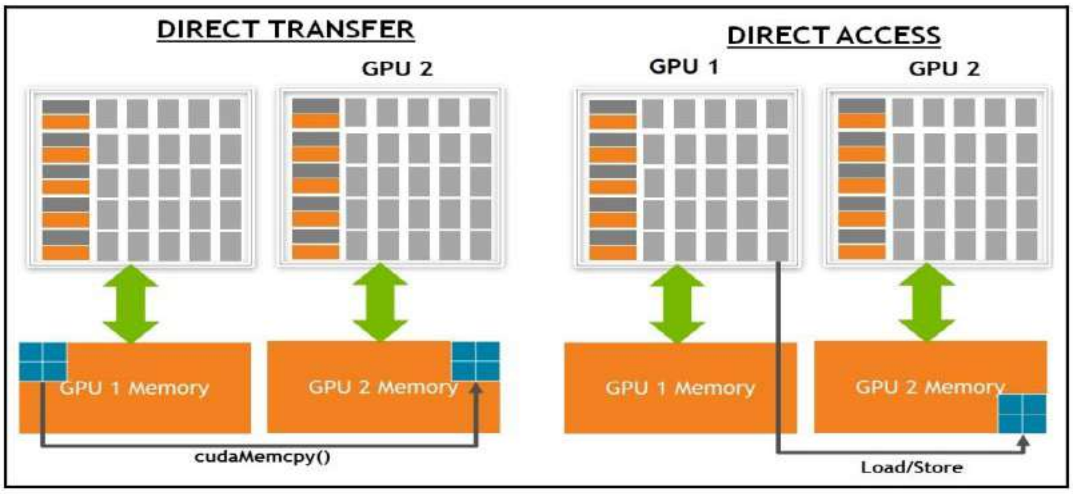
- **Accelerated communication between network and storage:** GPU Direct facilitates direct access to CUDA (Compute Unified Device Architecture) memory from external devices, such as network adapters or storage devices. This direct access helps eliminate the need for data to be copied multiple times between system memory and GPU memory, streamlining the communication process. This feature is supported from CUDA 3.1 onward.
- **GPUDirect for video:** GPUDirect for Video addresses the specific needs of video processing applications, providing a direct and efficient pathway for video data between devices and GPU memory. I This technology optimizes pipelines for frame-based video devices. It allows low-latency communication with OpenGL, DirectX, or CUDA and is supported from CUDA 4.2 onward
- **GPUDirect RDMA:** It enables direct memory access between GPUs in different nodes within a cluster, facilitating efficient data transfers for parallel computing workloads. This feature is supported from CUDA 5.0 and later.

The GPUDirect P2P feature allows the following:

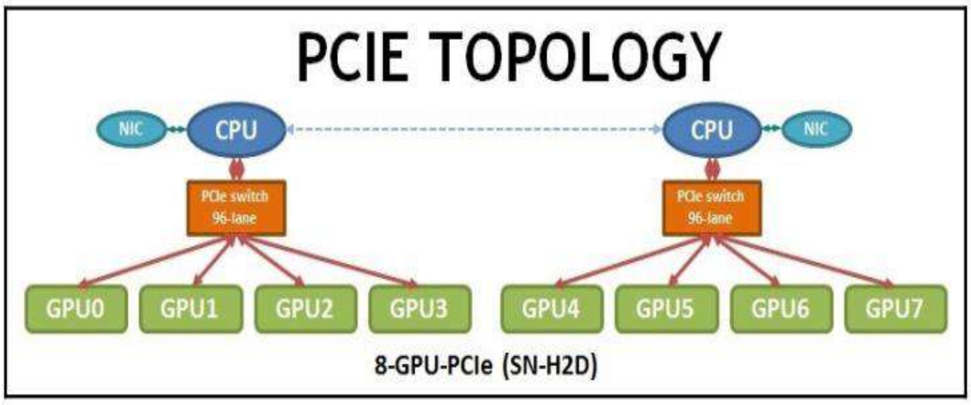
GPUDirect transfers: cudaMemcpy() initiates a DMA copy from GPU 1's memory to GPU 2's memory.

Direct access: GPU 1 can read or write GPU 2's memory (load/store)

The following diagram demonstrates these features

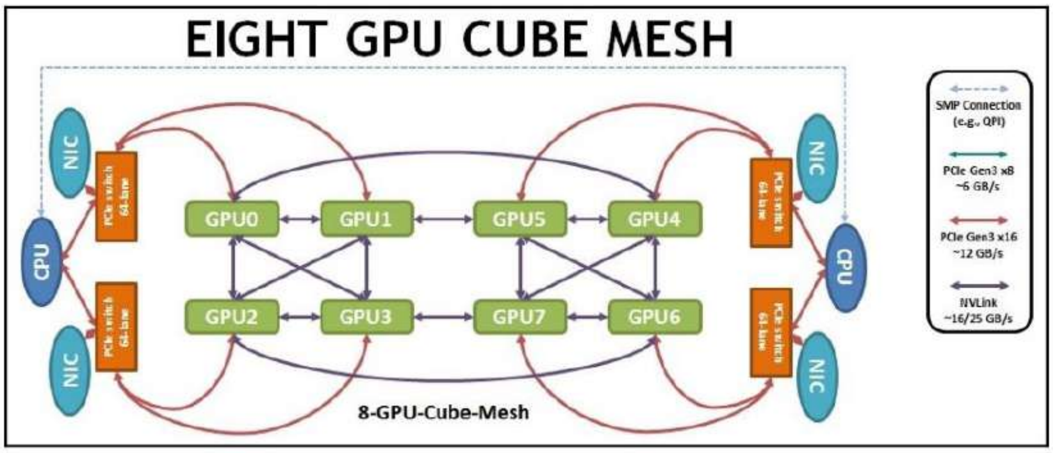


To understand the advantage of P2P, it is necessary to understand the PCIe bus specification. This was created with the intention of optimally communicating through interconnects such as InfiniBand to other nodes. The following is a sample PCIe topology where eight GPUs are being connected to various CPUs and NIC/InfiniBand cards.



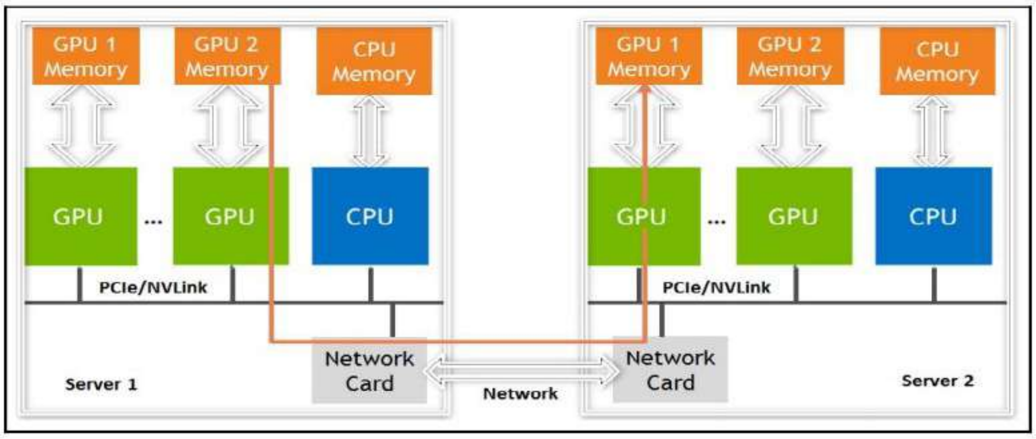
In the preceding diagram, P2P transfer is allowed between GPU0 and GPU1 as they are both situated in the same PCIe switch. However, GPU0 and GPU4 cannot perform a P2P transfer as PCIe P2P communication is not supported between two I/O Hubs (IOHs). QPI (QuickPath Interconnect) is a high-speed, point-to-point interconnect technology developed by Intel o connect processors and other components within a computer system. The nature of the QPI link connecting the two CPUs ensures that a direct P2P copy between GPU memory is not possible if the GPUs reside on different PCIe domains. Thus, a copy from the memory of GPU0 to the memory of GPU4 requires copying over the PCIe link to the memory attached to CPU0, then transferring it over the QPI link to CPU1 and over the PCIe again to GPU4. As you can imagine, this process adds a significant amount of overhead in terms of both latency and bandwidth.

The following diagram shows another system where GPUs are connected to each other via an NVLink interconnect that supports P2P transfers:(NVLink stands for "NVIDIA Link." It is a high-speed, high-bandwidth interconnect technology developed by NVIDIA to facilitate efficient communication between various computing components, including GPUs (Graphics Processing Units) and other high-performance computing elements) .



GPU Direct RDMA

In a cluster environment, we would like to make use of GPUs across multiple nodes. GPUDirect RDMA allows direct communication between GPUs across a cluster. It was first supported by CUDA 5.0 with the Kepler GPU card. In the following diagram, we can see the GPUDirect RDMA, that is, GPU 2 in Server 1 communicating directly with GPU 1 in Server 2.



CUDA-aware MPI

A CUDA-aware MPI is used to leverage GPUDirect RDMA for optimized inter-node communication. (CUDA-aware MPI allows MPI-enabled applications to communicate directly with GPU memory. This means that MPI processes can exchange data without the need for intermediate copies between GPU and CPU memory).

All the latest versions of the MPI libraries support the GPUDirect feature. MPI libraries that support for NVIDIA GPUDirect and Unified Virtual Addressing (UVA) enable the following:

- CUDA-aware MPI allows MPI-enabled applications to directly transfer data to and from GPU memory using an API (Application Programming Interface).
- CUDA-aware MPI libraries can automatically distinguish between device (GPU) memory and host (CPU) memory without requiring explicit hints from the user.
- The programmer's productivity increases as less application code needs to be changed for data transfers across multiple MPI ranks.

The following code is an example of using non-CUDA-aware MPI calls:

```
//MPI rank 0: Passing s_buf residing in GPU memory is to be transferred to CPU memory
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);
```

//MPI rank 1: r_buf received buffer needs to be transferred to GPU memory before being used in GPU

```
MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD,&status);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

With a CUDA-aware MPI library, this is not necessary; the GPU buffers can be directly passed to MPI, as shown in the following code:

//MPI rank 0

```
MPI_Send(s_buf_d,size,MPI_CHAR,1,100,MPI_COMM_WORLD);
```

//MPI rank n-1

```
MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
```

CUDA streams

Using multiple streams to overlap data transfers with kernel execution

The steps that we need to follow to overlap data transfers with kernel execution or to launch multiple kernels concurrently are as follows

1. **Declare the host memory to be pinned, as shown in the following code snippet:**

   ```
   cudaMallocHost(&hostInput1, inputLength*sizeof(float));
   cudaMallocHost(&hostInput2, inputLength*sizeof(float));
    cudaMallocHost(&hostOutput, inputLength*sizeof(float));
   ```

2. **Create a Stream object, as shown in the following code snippet:**

   ```
   for (i = 0; i < 4; i++)
   cudaStreamCreateWithFlags(&stream[i],cudaStreamNonBlocking)
   ```

   - **cudaStreamCreateWithFlags** is a CUDA function used to create a CUDA stream.
   - **&stream[i]** is the pointer to the newly created stream, and **i** is likely an index indicating which stream you are creating.
   - **cudaStreamNonBlocking** is a flag that indicates that operations in the stream can be non-blocking. It means that kernel launches and other operations in this stream may overlap with other CUDA streams, potentially improving overall throughput.

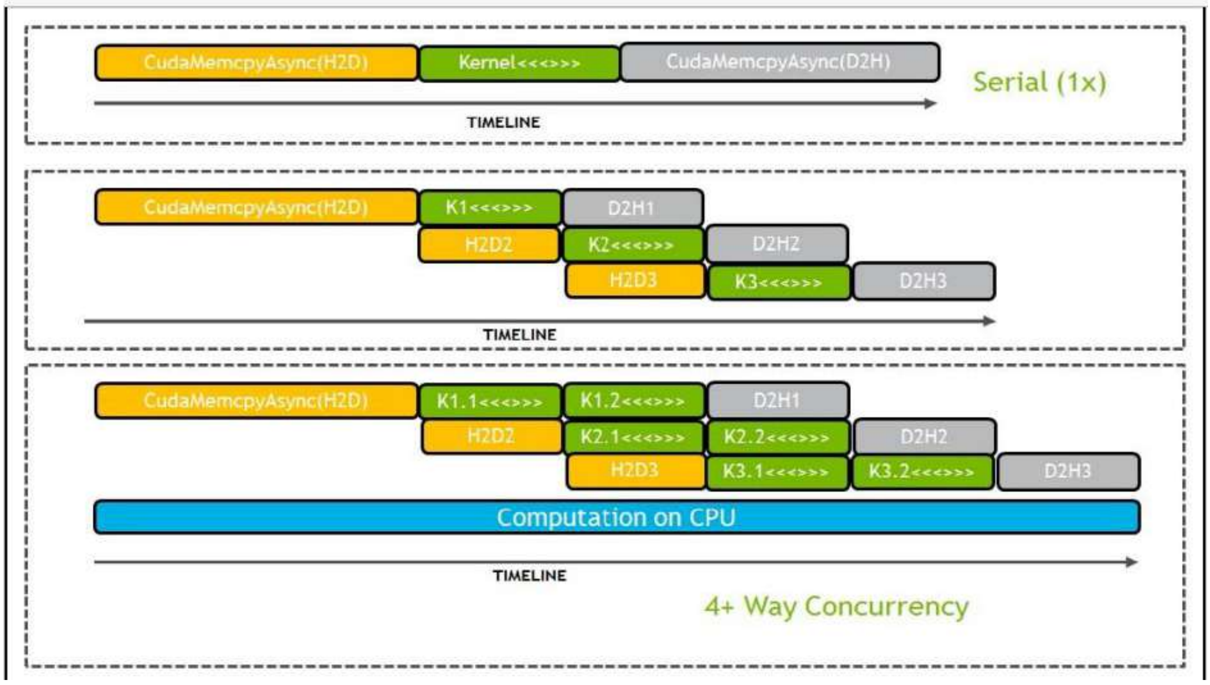
3. **Call the CUDA kernel and memory copies with the stream flag, as shown in the following code snippet:**

   ```
    for (i = 0; i < inputLength; i += Seglen * 4) {
   ```

```
   for (k = 0; k < 4; k++) {
   cudaMemcpyAsync(... , cudaMemcpyHostToDevice, stream[k]);
   cudaMemcpyAsync(... , cudaMemcpyHostToDevice, stream[k]);
    vecAdd<<<Gridlen,256,0,stream[k]>>>(...);
    } }
```

As we can see, instead of performing the vector addition in one shot by copying the whole array once, instead we chunk the array into segments and copy the segments asynchronously. Kernel execution is also done asynchronously in the respective streams.

Every GPU has two memory copy engines. One is responsible for the host to device transfer while the other is responsible for the device to host transfer. Hence, the two memory copies, which happen in opposite directions, can be overlapped. Also, the memory copies can be overlapped with the compute kernels. This can result in n-way concurrency, as shown in the following diagram:



Using multiple streams to run kernels on multiple devices

```
// Kernel to perform vector addition
__global__ void vectorAddition(int* a, int* b, int* c, int size) {
   int idx = threadIdx.x;
   c[idx] = a[idx] + b[idx];
}
int main()
```

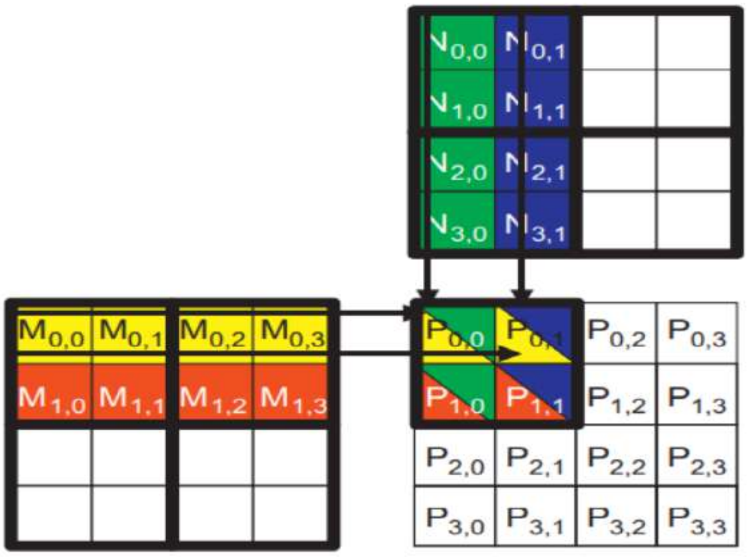
```
// Number of GPUs
int numGPUs;
cudaGetDeviceCount(&numGPUs);
// Allocate host memory
int h_a[5] = {1, 1, 1, 1, 1}, h_b[5] = {2, 2, 2, 2, 2}, h_c[5];
// Loop over each GPU
for (int gpu = 0; gpu < numGPUs; ++gpu) {
   // Set the current GPU
   cudaSetDevice(gpu);
   // Allocate device memory
   int *d_a, *d_b, *d_c;
   cudaMalloc((void**)&d_a, 5 * sizeof(int));
   cudaMalloc((void**)&d_b, 5 * sizeof(int));
   cudaMalloc((void**)&d_c, 5 * sizeof(int));
   // Copy input vectors from host to device
   cudaMemcpyAsync(d_a, h_a, 5 * sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpyAsync(d_b, h_b, 5 * sizeof(int), cudaMemcpyHostToDevice);
   // Launch vector addition kernel on a stream
   cudaStream_t stream1,stream2;
   cudaStreamCreate(&stream1);
   cudaStreamCreate(&stream1);
   vectorAddition<<<1, 5, 0, stream1>>>(d_a, d_b, d_c, 5);
   // Copy result vector from device to host
   cudaMemcpyAsync(h_c, d_c, 5 * sizeof(int), cudaMemcpyDeviceToHost,stream2);
   // Synchronize the stream
   cudaStreamSynchronize(stream1);     cudaStreamSynchronize(stream2);
   // Clean up for the current GPU
   cudaFree(d_a);     cudaFree(d_b);     cudaFree(d_c);
   cudaStreamDestroy(stream1);     cudaStreamDestroy(stream2);
}
// Print the result vector
   return 0;}
```

Parallel Programming Patterns in CUDA

Matrix multiplication optimization

Matrix multiplication optimization is achieved by using a tiling approach in CUDA.It involves breaking the matrices into smaller tiles or blocks and processing these blocks in parallel. This can improve data locality and reduce the number of global memory accesses.

In the figure below M,N are input matrices and P is a output matrix each of size 4*4.Using tiling approach M is divided into tiles of each 2*2(4 Tiles).



Without tiling: In order to calculate one output element, a thread will need to access one entire row of input A and one entire column of input B, for calculating the dot product. In our example, that is **8 accesses per thread**.

With tiling: Each thread ends up loading two elements from input A and two elements from input B, which totals up to **4 accesses per thread.**

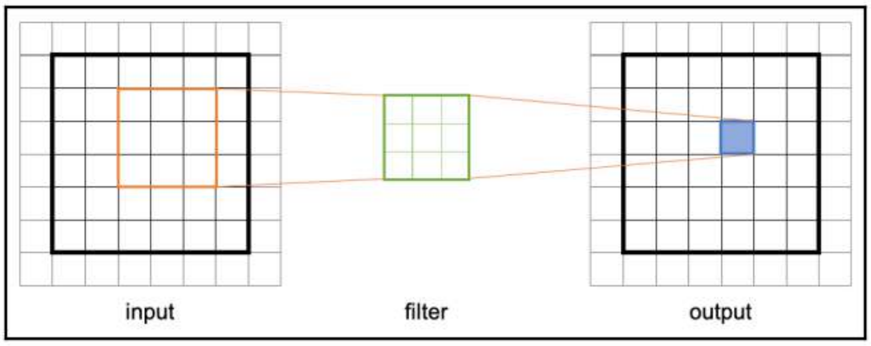
```
#include <stdio.h>

#define TILE_SIZE 16

__global__void matrixMul(int *a, int *b, int *c, int matrixSize) {

    __shared__ int tileA[TILE_SIZE][TILE_SIZE];

    __shared__ int tileB[TILE_SIZE][TILE_SIZE];

    int bx = blockIdx.x, by = blockIdx.y;

    int tx = threadIdx.x, ty = threadIdx.y;
    // Identify the row and column of the d_c element to compute
    int row = by * TILE_SIZE + ty;
    int col = bx * TILE_SIZE + tx;
    int result = 0;
    // Loop over the tiles of the input matrices
    for (int t = 0; t < matrixSize / TILE_SIZE; ++t) {
        // Collaboratively load tiles into shared memory
        tileA[ty][tx] = a[row * matrixSize + t * TILE_SIZE + tx];
        tileB[ty][tx] = b[(t * TILE_SIZE + ty) * matrixSize + col];
        // Synchronize to make sure the tiles are loaded
        __syncthreads();
        // Compute the product for the tile
        for (int k = 0; k < TILE_SIZE; ++k) {
            result += tileA[ty][k] * tileB[k][tx];
        }
        // Synchronize before loading the next tile
        __syncthreads();
    }
    // Write the result to the output matrix
    c[row * matrixSize + col] = result;
}
int main() {
    const int matrixSize = 1024;
    const int matrixBytes = matrixSize * matrixSize * sizeof(int);
```

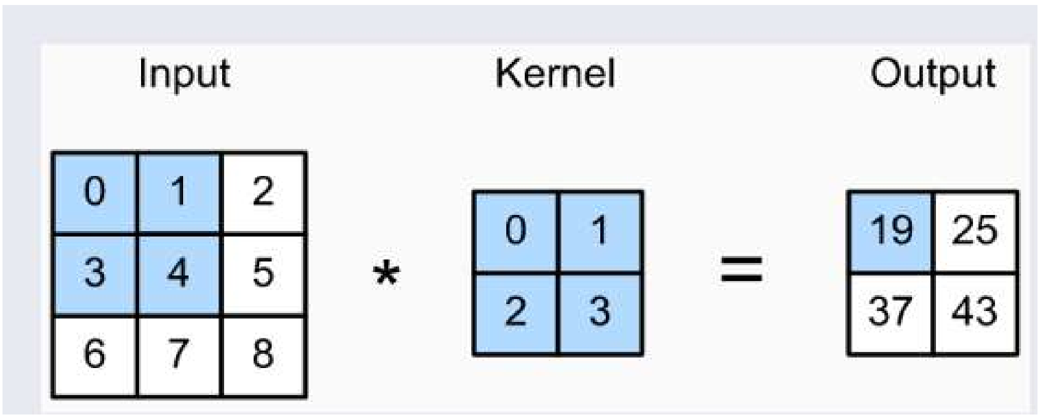
```
// Host matrices

int *h_a, *h_b, *h_c;

h_a = (int*)malloc(matrixBytes);

h_b = (int*)malloc(matrixBytes);

h_c = (int*)malloc(matrixBytes);

// Initialize matrices

for (int i = 0; i < matrixSize * matrixSize; ++i) {

   h_a[i] = i;       h_b[i] = i;

}

// Device matrices

int *d_a, *d_b, *d_c;

cudaMalloc((void**)&d_a, matrixBytes);

cudaMalloc((void**)&d_b, matrixBytes);

cudaMalloc((void**)&d_c, matrixBytes);

// Copy host matrices to device

cudaMemcpy(d_a, h_a, matrixBytes, cudaMemcpyHostToDevice);

cudaMemcpy(d_b, h_b, matrixBytes, cudaMemcpyHostToDevice);

// Define grid and block dimensions

dim3 dimGrid(matrixSize / TILE_SIZE, matrixSize / TILE_SIZE);

dim3 dimBlock(TILE_SIZE, TILE_SIZE);

// Launch the kernel

matrixMul<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, matrixSize);

// Copy the result back to host

cudaMemcpy(h_c, d_c, matrixBytes, cudaMemcpyDeviceToHost);
```

```
    // Print some elements of the result matrix for verification
    for (int i = 0; i < 5; ++i) {
        for (int j = 0; j < 5; ++j) {
            printf("%d ", h_c[i * matrixSize + j]);
        }
        printf("\n");
    }
    // Free memory
    free(h_a);    free(h_b);    free(h_c);
    cudaFree(d_a);    cudaFree(d_b);    cudaFree(d_c);
    return 0;
}
```

Convolution

Convolution is extensively used in image processing for operations like blurring, sharpening, edge detection, and noise reduction. Filters or kernels are applied to images through convolution to extract features or enhance specific characteristics. The convolutional operation consists of source data and a filter. The filter is also known as a kernel. By applying the filter against the input data, we can obtain the modified result. A two-dimensional convolution is shown in the following diagram:





We need to consider a couple of concepts when we implement convolution operation, that is, kernel and padding.

- **Kernel (Filter):**

The kernel is a small matrix of learnable parameters. It represents the set of coefficients or weights that are applied to local regions of the input data.During the convolution operation, the kernel slides or convolves across the input data, and at each position, the element-wise product of the kernel and the input data is computed, producing a single value in the output feature map.The weights of the kernel are learned through the training process, allowing the network to automatically adapt and extract relevant features from the input.

- **Padding:**

Padding involves adding extra virtual space (usually filled with zeros) around the input data. This is done to ensure that the convolutional operation can be applied to the edges of the input without losing information.The amount of padding is controlled by the "padding size." Zero-padding is common, where zeros are added around the input data.Padding is essential to prevent the reduction of spatial dimensions after convolution, especially in deep networks. It helps maintain the spatial resolution of the input and output feature maps.

- There are different types of padding:

  Valid (No Padding): No padding is applied..

  Same (Zero Padding): Enough padding is added to keep the spatial dimensions of the output the same as the input.

  simple convolutional operation

```
__global__ void convolution2D(int *input, int *output, int width, int height) {
   int tx = threadIdx.x + blockIdx.x * blockDim.x;
   int ty = threadIdx.y + blockIdx.y * blockDim.y;
```

```
    int result = 0;
    for (int i = 0; i < MASK_WIDTH; ++i) {
      for (int j = 0; j < MASK_WIDTH; ++j) {
        int row = ty - (MASK_WIDTH - 1) / 2 + i;
        int col = tx - (MASK_WIDTH - 1) / 2 + j;
        if (row >= 0 && row < height && col >= 0 && col < width) {
          result += input[row * width + col] * 1;  // Simple filter with all 1s
        }       }
    }
    output[ty * width + tx] = result;
}
int main() {
  // ... (variable declarations and input initialization)
  // Allocate device memory
  cudaMalloc((void**)&d_input, N * N * sizeof(int));
  cudaMalloc((void**)&d_output, N * N * sizeof(int));
  // Copy input data to device
  cudaMemcpy(d_input, input, N * N * sizeof(int), cudaMemcpyHostToDevice);
  // Define block and grid dimensions
  dim3 dimBlock(16, 16);  // 16x16 threads per block
  dim3 dimGrid((N - 1) / dimBlock.x + 1, (N - 1) / dimBlock.y + 1);
  // Launch the kernel
  convolution2D<<<dimGrid, dimBlock>>>(d_input, d_output, N, N);
  // Copy the result back to the host
  cudaMemcpy(output, d_output, N * N * sizeof(int), cudaMemcpyDeviceToHost);
  // ... (print some elements of the result matrix and free memory)
  return 0;
}
```

This kernel function fetches input data and a filter for the very operation and does not reuse all the data. Considering the performance impact from memory inefficiency, we need to design our kernel code so that we can reuse the loaded data.

```
#define MASK_WIDTH 3
```

```
#define TILE_SIZE 16
#define N 1024
__global__ void convolution2D(int *input, int *output, int width, int height)
{
   __shared__ int N_ds[TILE_SIZE + MASK_WIDTH - 1][TILE_SIZE + MASK_WIDTH - 1];
   int tx = threadIdx.x;
   int ty = threadIdx.y;
   int row_o = blockIdx.y * TILE_SIZE + ty;
   int col_o = blockIdx.x * TILE_SIZE + tx;
   int row_i = row_o - (MASK_WIDTH - 1) / 2;
   int col_i = col_o - (MASK_WIDTH - 1) / 2;
   // Load data into shared memory
   if (row_i >= 0 && row_i < height && col_i >= 0 && col_i < width)
 {
      N_ds[ty][tx] = input[row_i * width + col_i];
   }
else
{
      N_ds[ty][tx] = 0;
   }
   __syncthreads();

   // Apply convolution with a 3x3 box filter
   int result = 0;
   if (ty < TILE_SIZE && tx < TILE_SIZE) {
      for (int i = 0; i < MASK_WIDTH; ++i) {
         for (int j = 0; j < MASK_WIDTH; ++j) {
            result += N_ds[ty + i][tx + j];
         }
      }
   }

   // Store result
   if (row_o < height && col_o < width) {
      output[row_o * width + col_o] = result / (MASK_WIDTH * MASK_WIDTH);
   }
}

int main()
 {
   int input[N][N];
   int output[N][N];
   int *d_input, *d_output;

   // Initialize input data (for simplicity, using sequential values)
   for (int i = 0; i < N * N; ++i) {
      input[0][i] = i;
```

```
    }

    // Allocate device memory
    cudaMalloc((void**)&d_input, N * N * sizeof(int));
    cudaMalloc((void**)&d_output, N * N * sizeof(int));

    // Copy input data to device
    cudaMemcpy(d_input, input, N * N * sizeof(int), cudaMemcpyHostToDevice);

    // Define grid and block dimensions
    dim3 dimGrid((N - 1) / TILE_SIZE + 1, (N - 1) / TILE_SIZE + 1);
    dim3 dimBlock(TILE_SIZE, TILE_SIZE);

    // Launch the kernel
    convolution2D<<<dimGrid, dimBlock>>>(d_input, d_output, N, N);

    // Copy the result back to the host
    cudaMemcpy(output, d_output, N * N * sizeof(int), cudaMemcpyDeviceToHost);
    // Print some elements of the result matrix for verification
    // Free device memory
    cudaFree(d_input);    cudaFree(d_output);

    return 0;
}
```

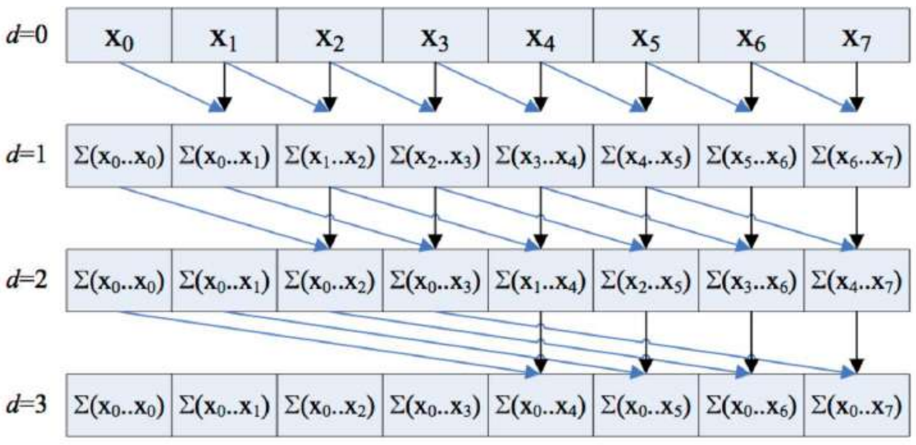
In this example, the convolution is performed with a 3x3 box filter, and the result is averaged by dividing by the total number of elements in the filter. The use of shared memory (**N_ds**) helps improve memory access patterns and can lead to better performance, especially for larger input matrices and more complex filters.

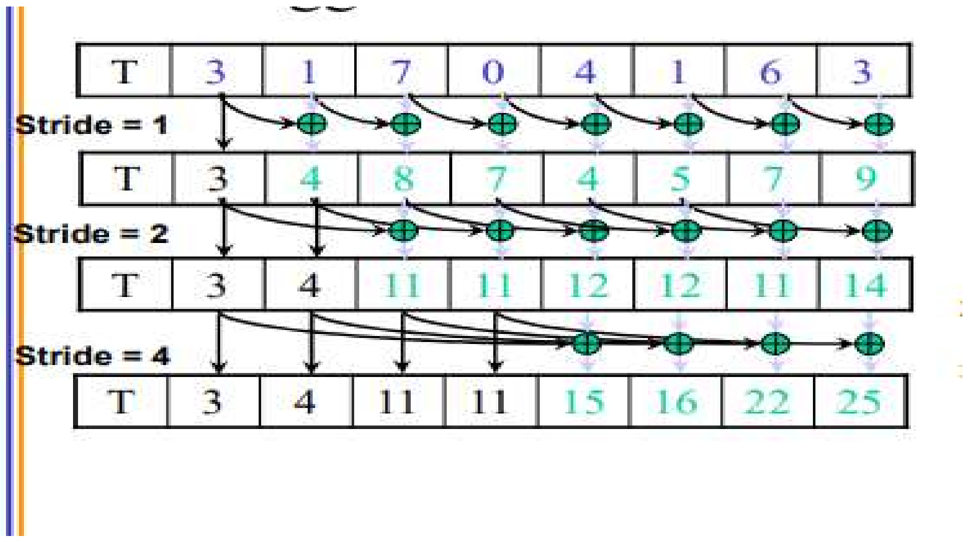
Prefix sum

Prefix sum (scan) is used to obtain a cumulative number array from the given input numbers array. For example, we can make a prefix-sum sequence as follows:

Input numbers	1	2	3	4	5	6
Prefix sums	1	3	6	10	15	21

It differs from parallel reduction since reduction just generates the total operation output from the given input data. On the other hand, scan generates outputs from each operation. The easiest way to solve this problem is to iterate all the inputs to generate the output. However, it would take a long time and would be inefficient in GPUs. Hence, the mild approach can parallelize the prefix-sum operation, as follows:

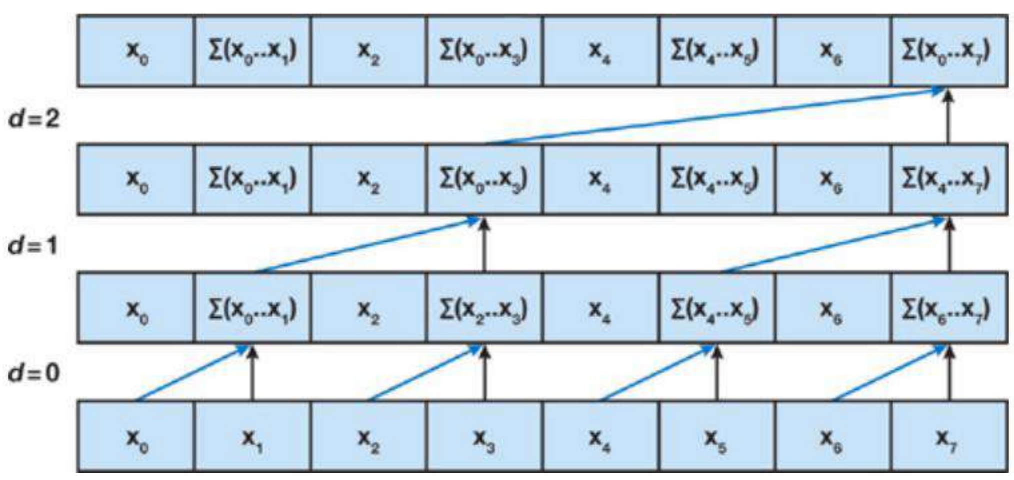

d=0
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
d=1
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_1..x_2)$ $\Sigma(x_2..x_3)$ $\Sigma(x_3..x_4)$ $\Sigma(x_4..x_5)$ $\Sigma(x_5..x_6)$ $\Sigma(x_6..x_7)$
d=2
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_1..x_4)$ $\Sigma(x_2..x_5)$ $\Sigma(x_3..x_6)$ $\Sigma(x_4..x_7)$
d=3
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_0..x_4)$ $\Sigma(x_0..x_5)$ $\Sigma(x_0..x_6)$ $\Sigma(x_0..x_7)$



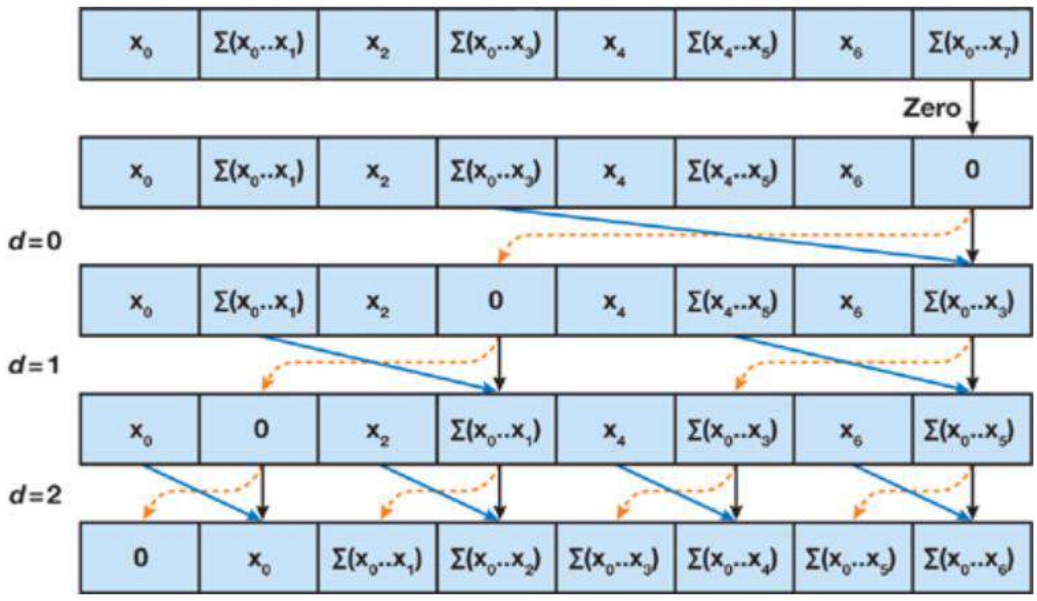
```
#define N 1024

__global__ void prefixSum(int *input, int *output, int n)
 {
   int thid = threadIdx.x;

   // Perform parallel reduction to calculate partial sums
   for (int offset = 1; offset < n; offset *= 2)
{
      if (thid + offset < n) {
         output[thid + offset] += output[thid];
      }
      __syncthreads();
   }
}

int main()
 {
   int *h_input = new int[N];
   int *h_output = new int[N];
   int *d_input, *d_output;
```

```
    // Initialize input data (for simplicity, using sequential values)
    for (int i = 0; i < N; ++i)
  {
        h_input[i] = i;
    }

    // Allocate device memory
    cudaMalloc((void**)&d_input, N * sizeof(int));
    cudaMalloc((void**)&d_output, N * sizeof(int));

    // Copy input data to device
    cudaMemcpy(d_input, h_input, N * sizeof(int), cudaMemcpyHostToDevice);

    // Define block and grid dimensions
    dim3 dimBlock(N, 1);
    dim3 dimGrid(1, 1);

    // Launch the prefix sum kernel
    prefixSum<<<dimGrid, dimBlock>>>(d_input, d_output, N);

    // Copy the result back to the host
    cudaMemcpy(h_output, d_output, N * sizeof(int), cudaMemcpyDeviceToHost);

    // Print some elements of the result array for verification
    for (int i = 0; i < 10; ++i) {
        printf("%d\t",h_output[i]);
    }

    // Free device memory
    cudaFree(d_input);    cudaFree(d_output);
    delete[] h_input;    delete[] h_output;
    return 0;
  }
```

There is another optimized approach named. This method generates prefixsum outputs by increasing and decreasing the strides exponentially. This method's procedure is shown in the following diagram:



Upsweep(Reduce Phase)

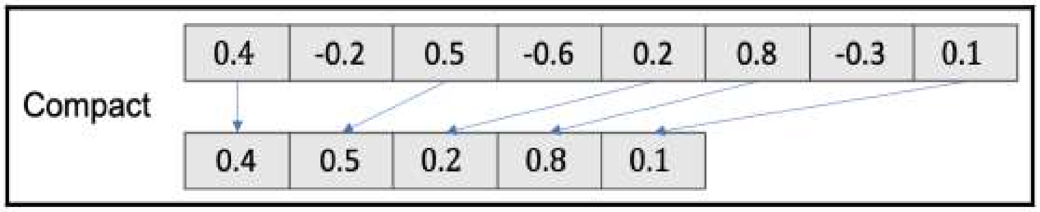


Downsweep

```
#define N 1024
__global__ void upsweep(int *data, int stride) {
   int thid = threadIdx.x;
   int offset = 2 * stride * thid;
   data[offset + 2 * stride - 1] += data[offset + stride - 1];
}

__global__ void downsweep(int *data, int stride) {
   int thid = threadIdx.x;
   int offset = 2 * stride * thid;

   int temp = data[offset + stride - 1];
   data[offset + stride - 1] = data[offset + 2 * stride - 1];
   data[offset + 2 * stride - 1] += temp;
}

__global__ void initializeLastElement(int *data, int n) {
   // Set the last element to 0 for inclusive scan
   data[n - 1] = 0;
}
int main() {
   int *h_data = new int[N];
   int *d_data;
   // Initialize input data (for simplicity, using sequential values)
   for (int i = 0; i < N; ++i) {
       h_data[i] = i;
   }
   // Allocate device memory
   cudaMalloc((void**)&d_data, N * sizeof(int));

   // Copy input data to device
   cudaMemcpy(d_data, h_data, N * sizeof(int), cudaMemcpyHostToDevice);

   // Define block dimensions
   dim3 dimBlock(N / 2, 1);

   // Upsweep (reduce) phase
   for (int stride = 1; stride < N; stride *= 2) {
       dim3 dimGrid(1, 1);
       upsweep<<<dimGrid, dimBlock>>>(d_data, stride);
   }
   // Initialize last element to 0
   initializeLastElement<<<1, 1>>>(d_data, N);

   // Downsweep (inclusive scan) phase
   for (int stride = N / 2; stride > 0; stride /= 2) {
       dim3 dimGrid(1, 1);
       downsweep<<<dimGrid, dimBlock>>>(d_data, stride);
   }
   // Copy the result back to the host
   cudaMemcpy(h_data, d_data, N * sizeof(int), cudaMemcpyDeviceToHost);

   // Print some elements of the result array for verification
   // Free device memory
   cudaFree(d_data);    delete[] h_data;
   return 0;
}
```

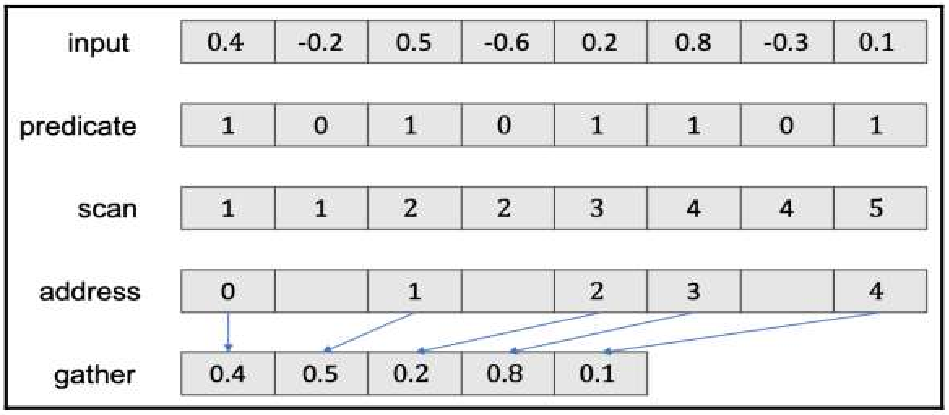
Compact and split

A compact operation typically refers to the process of removing unwanted elements from an array, leaving only the desired elements. In CUDA, this often involves creating a new array that contains only the elements that satisfy a certain condition. The operation can be performed efficiently in parallel by different threads. For example, if we want to use the compact operation for the positive elements in an array, then the operation is as follows:



In parallel programming, we have a different approach for compact operation that can utilize multiple cores using the parallel prefix-sum operation. First, we mark the data to check whether it meets the condition or not (that is, predicate), and then we do the prefix-sum operation. The output of prefix-sum will be the index of the marked values, so we can obtain the gathered array by copying them. Since all of these tasks can be done in parallel, we can obtain the gathered array in four steps.

The following diagram shows an example of a compact operation:



To implement a compact operation, we will write several kernel functions that can do the required operation for each step and call those last:

1.Let's write a kernel function that can make a predicate array by checking whether each element's value is greater than zero or not:

```
__global__ void predicate_kernel(float *d_predicates, float *d_input, int length)
{
int idx = blockDim.x * blockIdx.x + threadIdx.x;
if (idx >= length) return;
d_predicates[idx] = d_input[idx] > FLT_ZERO;
}
```

2. Then, we have to perform a prefix-sum operation for that predicate array. We will reuse the previous implementation here. After that, we can write a kernel function that can detect the address

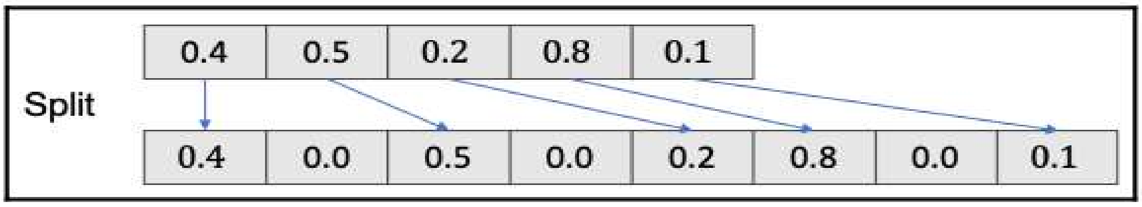
of the scanned array and gather the target elements as output:

```
 __global__ void pack_kernel(float *d_output, float *d_input, float *d_predicates, float *d_scanned,
int length)
{
 int idx = blockDim.x * blockIdx.x + threadIdx.x;
 if (idx >= length) return;
if (d_predicates[idx] != 0.f)
{ // addressing
 int address = d_scanned[idx] - 1;
// gather
d_output[address] = d_input[idx];
} }
```

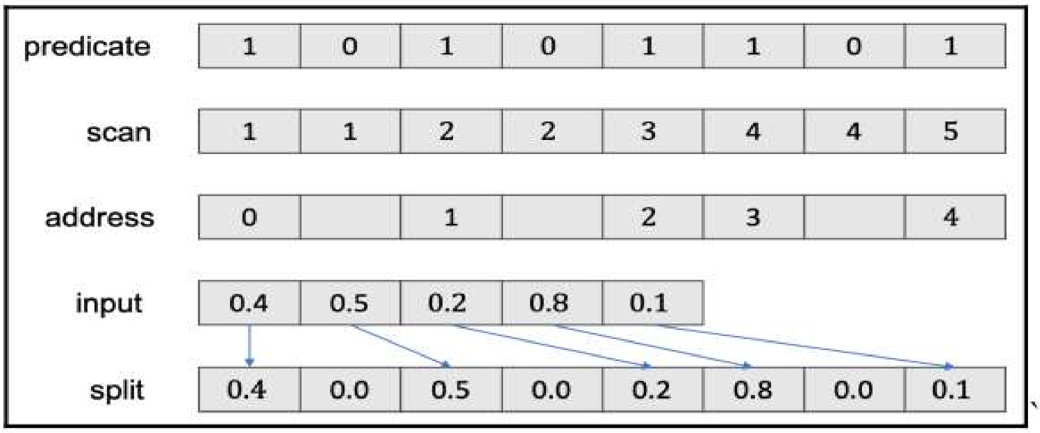
3. Now, let's call them all together to make a compact operation:

```
// predicates
predicate_kernel<<< GRID_DIM, BLOCK_DIM >>>(d_predicates, d_input, length);
// scan
 scan_v2(d_scanned, d_predicates, length);
// addressing & gather (pack)
pack_kernel<<< GRID_DIM, BLOCK_DIM >>>(d_output, d_input, d_predicates, d_scanned, length);
```

A split operation typically refers to dividing data into multiple parts, often based on a certain criterion. This can be useful when you want to perform parallel processing on different subsets of the data.



We can also do this in parallel using prefix-sum. Firstly, we refer to the predicate array and do the prefix-sum. Since the outputs are each element's address, we can distribute them easily. The following diagram shows how this operation can be done.



Let's write the split kernel function, as follows: _

```
_global__ void split_kernel(float *d_output, float *d_input, float *d_predicates, float *d_scanned, int length)
{
int idx = blockDim.x * blockIdx.x + threadIdx.x;
if (idx >= length) return;
if (d_predicates[idx] != 0.f)
{ // address
 int address = d_scanned[idx] - 1;
// split
d_output[idx] = d_input[address];
} }
```

2. Now, we can call the kernel function, as follows:

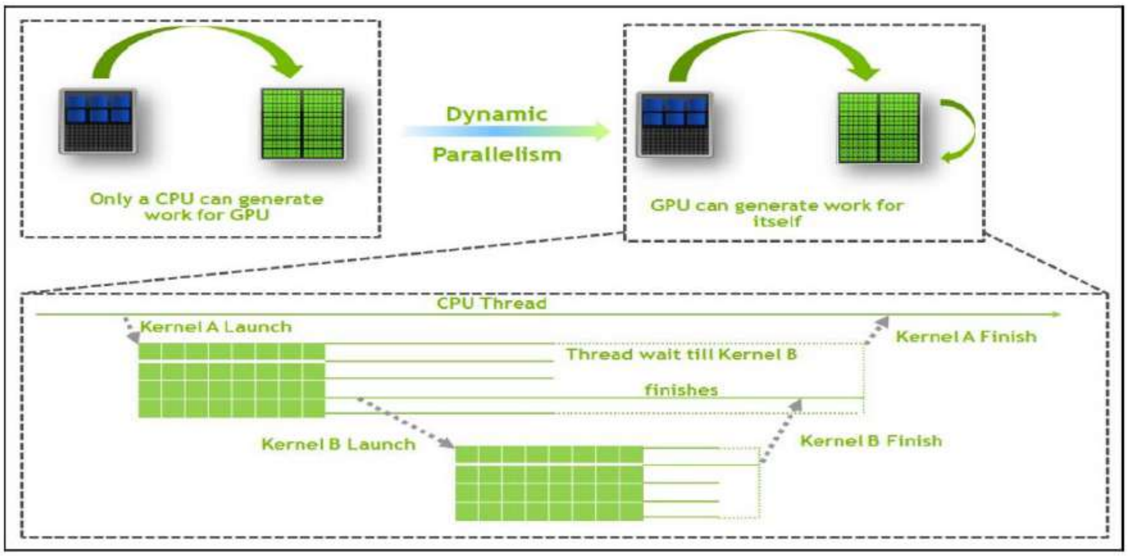
```
cudaMemcpy(d_input, d_output, sizeof(float) * length, cudaMemcpyDeviceToDevice);
cudaMemset(d_output, 0, sizeof(float) * length);
split_kernel<<< GRID_DIM, BLOCK_DIM >>>(d_output, d_input, d_predicates, d_scanned, length);
```

Quicksort in CUDA using dynamic parallelism

QuickSort is a divide-and-conquer sorting algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

The Quicksort algorithm demands launching kernels recursively. So far, the algorithms we have seen call the kernel once via the CPU. After the kernel has finished executing, we return to the CPU thread and then relaunch it. Doing this results in giving back control to the CPU, and may also result in data transfer between CPU and GPU, which is a costly operation. It used to be very difficult to efficiently implement algorithms such as Quicksort on GPUs that demand features such as recursion. With the GPU architecture 3.5 and CUDA 5.0 onwards, a new feature was introduced called dynamic parallelism.

Dynamic parallelism allows the threads within a kernel to launch new kernels from the GPU without returning control back to the CPU. The word dynamic comes from the fact that it is dynamically based on the runtime data. Multiple kernels can be launched by threads at once. The following diagram simplifies this explanation.


Only a CPU can generate work for GPU
Dynamic Parallelism
GPU can generate work for itself
CPU Thread
Kernel A Launch
Kernel A Finish
Thread wait till Kernel B finishes
Kernel B Launch
Kernel B Finish

```
const int MAX_DEPTH = 16; // Maximum recursion depth
__global__ void quicksort(int* array, int left, int right, int depth);
__device__ int partition(int* array, int left, int right) {
   int pivot = array[right];
   int i = left - 1;
   for (int j = left; j < right; j++) {
      if (array[j] <= pivot) {
         i++;
         int temp = array[i];
         array[i] = array[j];
         array[j] = temp;
      }
   }
   int temp = array[i + 1];
   array[i + 1] = array[right];
   array[right] = temp;

   return i + 1;
}
```

```
__global__ void quicksort(int* array, int left, int right, int depth) {
   if (depth <= 0 || left >= right) {
      // Use a simple sorting algorithm for small chunks or when recursion is deep enough
      for (int i = left + 1; i <= right; i++) {
         int j = i;
         while (j > left && array[j - 1] > array[j]) {
            int temp = array[j];
            array[j] = array[j - 1];
            array[j - 1] = temp;
            j--;
         }
      }
      return;
   }
   int pivotIndex = partition(array, left, right);
   // Launch two child kernels with new partitions
   if (left < pivotIndex) {
      quicksort<<<1, 1>>>(array, left, pivotIndex - 1, depth - 1);
   }
   if (pivotIndex < right) {
      quicksort<<<1, 1>>>(array, pivotIndex + 1, right, depth - 1);
   }
}
int main() {
   const int arraySize = 1000; // Adjust the size as needed
   const int arrayBytes = arraySize * sizeof(int);
   int* hostArray = new int[arraySize];
   int* deviceArray;
   // Initialize array with random values
   for (int i = 0; i < arraySize; i++) {
      hostArray[i] = rand() % 1000;
   }
   // Allocate and copy array to the device
   cudaMalloc((void**)&deviceArray, arrayBytes);
   cudaMemcpy(deviceArray, hostArray, arrayBytes, cudaMemcpyHostToDevice);
   // Launch quicksort kernel
   quicksort<<<1, 1>>>(deviceArray, 0, arraySize - 1, MAX_DEPTH);
   cudaDeviceSynchronize();
   // Copy the sorted array back to host
   cudaMemcpy(hostArray, deviceArray, arrayBytes, cudaMemcpyDeviceToHost);
   // Cleanup
   delete[] hostArray;
   cudaFree(deviceArray);
   return 0;
}
```

Radix Sort

Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys. Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit's value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order.

The detailed steps are as follows −

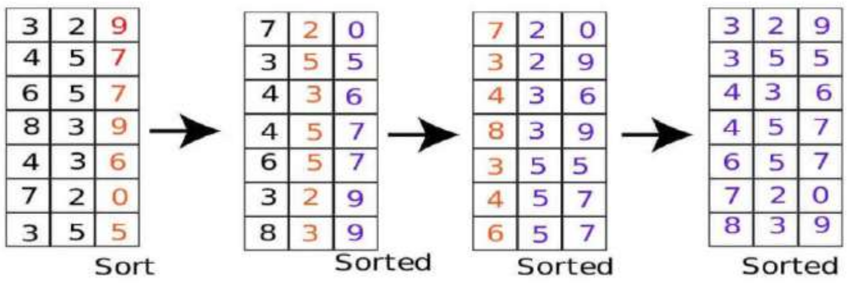
Step 1 − Check whether all the input elements have same number of digits. If not, check for numbers that have maximum number of digits in the list and add leading zeroes to the ones that do not.

Step 2 − Take the least significant digit of each element.

Step 3 − Sort these digits using counting sort logic and change the order of elements based on the output achieved. For example, if the input elements are decimal numbers, the possible values each digit can take would be 0-9, so index the digits based on these values.

Step 4 − Repeat the Step 2 for the next least significant digits until all the digits in the elements are sorted.

Step 5 − The final list of elements achieved after kth loop is the sorted output.



Implementation using warp level primitives

```
#define NUM_BITS 8
#define NUM_BUCKETS 1 << NUM_BITS
__device__ int getDigit(int num, int digit, int numBits)
 {
   return (num >> digit) & ((1 << numBits) - 1);
}

__global__ void radixSort(int* data, int numElements)
 {
   extern __shared__ int sharedData[];
```

```
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  int laneId = threadIdx.x % warpSize;
  for (int bit = 0; bit < sizeof(int) * 8; bit += NUM_BITS)
{
    int digitMask = (1 << NUM_BITS) - 1;
    int shift = bit;

    // Warp-level radix sort
    for (int i = 0; i < NUM_BITS; ++i)
 {
      int digit = getDigit(data[tid], shift, NUM_BITS);

      // Create a histogram using warp-level primitives
      unsigned int mask = (digitMask << laneId);
      int warpHistogram = __popc(__ballot_sync(mask, (digitMask & digit) == laneId));

      // Compute the prefix sum
      int warpPrefixSum = __popc(__ballot_sync(mask, laneId < digit));

      // Update the position of the current element
      int position = warpPrefixSum + warpHistogram;
      sharedData[position] = data[tid];

      __syncthreads();

      // Copy data back to global memory
      data[tid] = sharedData[threadIdx.x];

      shift += NUM_BITS;
    }
  }
}

int main() {
  // Initialize data on the host
  const int numElements = 1024;
  int data[numElements];
  for (int i = 0; i < numElements; ++i) {
    data[i] = rand() % 1000;
  }

  // Allocate and copy data to the device
  int* d_data;
  cudaMalloc((void**)&d_data, numElements * sizeof(int));
  cudaMemcpy(d_data, data, numElements * sizeof(int), cudaMemcpyHostToDevice);

  // Set grid and block dimensions
```

```
    dim3 gridSize(1, 1, 1);
    dim3 blockSize(1024, 1, 1);

    // Run the radix sort kernel
    radixSort<<<gridSize, blockSize, blockSize.x * sizeof(int)>>>(d_data, numElements);

    // Copy data back to the host
    cudaMemcpy(data, d_data, numElements * sizeof(int), cudaMemcpyDeviceToHost);

    // Free allocated memory on the device
    cudaFree(d_data);

    // Print sorted data
    for (int i = 0; i < numElements; ++i) {
        printf("%d ", data[i]);
    }

    return 0;
}
```

Thrust-based radix sort

Thrust is a parallel template library for CUDA, and it provides a high-level interface for GPU programming. It includes various parallel algorithms, and it has a radix sort implementation that you can use directly without having to write low-level CUDA code.

The steps involved in making use of Thrust for radix sort are as follows:

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
int main() {
    // Generate random data on the host
    thrust::host_vector<int> hostData(1024);
    for (int i = 0; i < 1024; ++i) {
        hostData[i] = rand() % 1000;
    }

    // Transfer data to the device
    thrust::device_vector<int> deviceData = hostData;

    // Use Thrust to perform radix sort on the device
    thrust::sort(deviceData.begin(), deviceData.end());

    thrust::copy(deviceData.begin(), deviceData.end(), hostData.begin());
    // Transfer data back to the host

    // Print sorted data
    return 0;
}
```

cuBLAS

cuBLAS, or CUDA Basic Linear Algebra Subroutines, is a GPU-accelerated library for performing basic linear algebra operations on NVIDIA GPUs. It is part of the CUDA Toolkit, which is a set of tools and libraries for parallel computing using NVIDIA GPUs.

cuBLAS provides a collection of optimized routines that cover a wide range of linear algebra operations, such as matrix multiplication, matrix-vector operations, and various matrix factorizations. These operations are crucial in many scientific and engineering applications, including machine learning, computer graphics, and scientific simulations.cuBLAS provides a variety of functions covering basic linear algebra operations. Here are some common functions and operations provided by cuBLAS:

1. **Matrix-Matrix Operations:**
   - **cublasSgemm**, **cublasDgemm**: Single and double precision general matrix-matrix multiplication.
   - **cublasSgemmBatched**, **cublasDgemmBatched**: Batched version of matrix-matrix multiplication.
2. **Matrix-Vector Operations:**
   - **cublasSgemv**, **cublasDgemv**: Single and double precision general matrix-vector multiplication.
   - **cublasSgemvBatched**, **cublasDgemvBatched**: Batched version of matrix-vector multiplication.

cuRAND

The **curand** library in CUDA (Compute Unified Device Architecture) provides functions for generating random numbers on NVIDIA GPUs. The **curand** library is specifically designed to offer high-performance random number generation for CUDA applications. Here are some key points about using **curand** in CUDA:

➢ **Initialization:**

curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);

This line creates a new random number generator (**gen**) and sets it to use the default pseudo-random number generator algorithm. The **CURAND_RNG_PSEUDO_DEFAULT** parameter indicates that you want to use the default PRNG provided by the library, which is typically a fast and high-quality generator.

curandSetPseudoRandomGeneratorSeed(gen, seed);

This line sets the seed for the pseudo-random number generator. The **seed** variable is an input value that you can choose to initialize the generator. The seed determines the starting point of the random sequence. If you use the same seed, you will get the same sequence of random numbers. If you change the seed, you will get a different sequence.

- **Random Number Generation:**

Once the generator is initialized, you can use functions like **curandGenerate** or **curandGenerateUniform** to generate random numbers.

curandGenerateUniform(gen, devData, numElements);

gen: This is the random number generator object that you previously created and initialized using **curandCreateGenerator** and **curandSetPseudoRandomGeneratorSeed**. It specifies the generator to be used for generating random numbers.

devData: This is a pointer to the device memory where the generated random numbers will be stored. The data type of **devData** should match the desired output data type (e.g., **float** or **double**).

numElements: This is the number of random numbers to generate and store in the **devData** array.

- **Cleanup:** After using the generator, it's important to free the allocated resources:

curandDestroyGenerator(gen);

cuFFT

CuFFT (CUDA Fast Fourier Transform) is a library provided by NVIDIA for performing Fast Fourier Transforms (FFT) on NVIDIA GPUs using CUDA. FFT is an efficient algorithm for computing the discrete

Fourier transform and is widely used in various signal processing and scientific computing applications. Here are some of the key functions in CuFFT:

- **cufftResult cufftPlan(cufftHandle *plan, int rank, const int* n, cufftType type, int batch);**

plan: A pointer to a **cufftHandle** variable that will be populated with the FFT plan.
rank: An integer representing the number of dimensions (1, 2, or 3) for the FFT.
n: An array of integers specifying the size of each dimension of the FFT.
type: The data type of the input and output arrays. It can be **CUFFT_R2C** for real-to-complex, **CUFFT_C2R** for complex-to-real, or **CUFFT_C2C** for complex-to-complex transforms.
batch: The number of FFTs to be executed in parallel. If **batch** is set to 1, it means a single FFT is performed. If **batch** is greater than 1, it indicates that multiple independent FFTs will be performed simultaneously on different sets of input data.

- **cufftResult cufftExecC2C(cufftHandle plan, cufftComplex *idata, cufftComplex *odata, int direction);**

plan: The FFT plan handle, which was previously created using functions like **cufftPlan1d**, **cufftPlan2d**, or **cufftPlan3d**. This handle contains information about the FFT configuration (size, type, etc.).
idata: A pointer to the input complex data array. This array contains the input sequence in the time (or spatial) domain.
odata: A pointer to the output complex data array. This array will contain the transformed sequence in the frequency domain after the FFT operation.
direction: An integer indicating the direction of the FFT. It can take one of the following values:
CUFFT_FORWARD (1): Perform a forward FFT (from time to frequency domain).
CUFFT_INVERSE (-1): Perform an inverse FFT (from frequency to time domain).
The **cufftExecC2C** function computes the FFT or its inverse according to the specified direction and updates the **odata** array with the result.

- **Destroy Plan Function: cufftDestroy**- Destroy an FFT plan when it is no longer needed.

NPP

NVIDIA Performance Primitives (NPP) is a library of functions that provides a collection of functions organized into different domains, including image processing, signal processing, statistics, and more. It is intended to be used alongside CUDA for developing high-performance GPU-accelerated applications.

Here are some key points about NPP:

Usage: To use NPP, you typically include the appropriate header files and link against the NPP library. For example, to use image processing functions, you might include the **npp.h** header and link against **libnppc**, **libnppig**, etc.

Image Processing: NPP offers a wide range of image processing functions such as filtering, color conversion, morphology operations, and more. For example, you can use functions like **nppiFilter_32f_C1R** for 2D filtering on 32-bit floating-point images.

Signal Processing: NPP provides functions for signal processing operations, including convolution, correlation, FFT, and other operations. For instance, you can use functions like **nppsFFTInitAlloc** and **nppsFFTForward_32f_C1R** for fast Fourier transforms.

Statistics and Math: NPP includes functions for statistical operations, basic math operations, and other numerical routines. Functions like **nppiMinMax_32f_C1R** can be used for finding the minimum and maximum values in an image

cuDNN

cuDNN (CUDA Deep Neural Network library)provides optimized implementations for a variety of deep learning operations, including convolution, pooling, normalization, activation functions, recurrent neural networks (RNNs), and more.

- **Convolution:**

Function: cudnnConvolutionForward

Description: Optimized implementation of the forward pass of convolutional layers. It is used for computing the output of a convolutional layer given input data and convolutional filters.

- **Pooling:**

Function: cudnnPoolingForward

Description: Optimized implementation of pooling operations, including max pooling and average pooling. It is used for downsampling feature maps, reducing spatial dimensions.

- **Normalization:**

  Functions: cudnnLRNCrossChannelForward, **cudnnBatchNormalizationForwardTraining**

  Description:

  cudnnLRNCrossChannelForward: Implements Local Response Normalization (LRN) across channels.

  cudnnBatchNormalizationForwardTraining: Implements batch normalization, which normalizes activations across mini-batches.

- **Activation Functions:**

  Function: cudnnActivationForward

  Description: Optimized implementation of activation functions, including sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU). It is used for introducing non-linearity into neural networks.

- **Recurrent Neural Networks (RNNs):**

  Function: cudnnRNNForwardTraining, **cudnnRNNForwardInference**

  Description:

cudnnRNNForwardTraining: Optimized implementation of the forward pass in training mode for RNNs.

cudnnRNNForwardInference: Optimized implementation of the forward pass in inference mode for RNNs.

- **Tensor Operations:**

  Functions: cudnnAddTensor, **cudnnScaleTensor**

  Description:

  cudnnAddTensor: Element-wise addition of two tensors.

  cudnnScaleTensor: Element-wise scaling of a tensor.

- **Softmax:**

  Function: cudnnSoftmaxForward

  Description: Optimized implementation of the softmax activation function. It is commonly used in the output layer of a neural network for multi-class classification problems.

- **Dropout:**

  Function: cudnnDropoutForward

  Description: Implementation of the dropout operation, which randomly sets a fraction of input units to zero during training to prevent overfitting.