

CUDA C++ Programming From Scratch

Kokwei Chew

Contents

1) Preparation

- Prerequisite
- Configuring CUDA Project in Visual Studio
- Compile CUDA in Project Ubuntu

2) CUDA Basics

- CUDA Example
- CUDA Terminology
- CUDA Parallelism
- Code Example

3) CUDA Optimization

- GPU Architecture
- Thread Occupancy
- Memory Access Pattern
- CUDA Stream



Preparation

Prerequisite

- 1) A laptop/PC with Nvidia CUDA supported GPU (preferably Maxwell or newer)
- 2) Install Visual studio 2015 or above.
- 3) Install Nvidia CUDA 10.x or 11.x.
- 4) OpenCV 4.5.5 (<https://opencv.org/releases/>)

Source code for examples:

<https://drive.google.com/file/d/1WRY4o1lr2loAe6L56snxT5zJDUWx9Jel/view?usp=sharing>

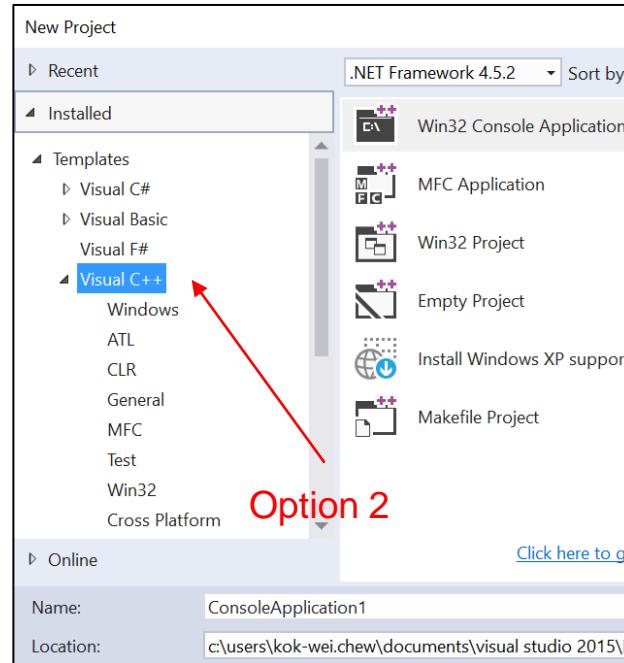
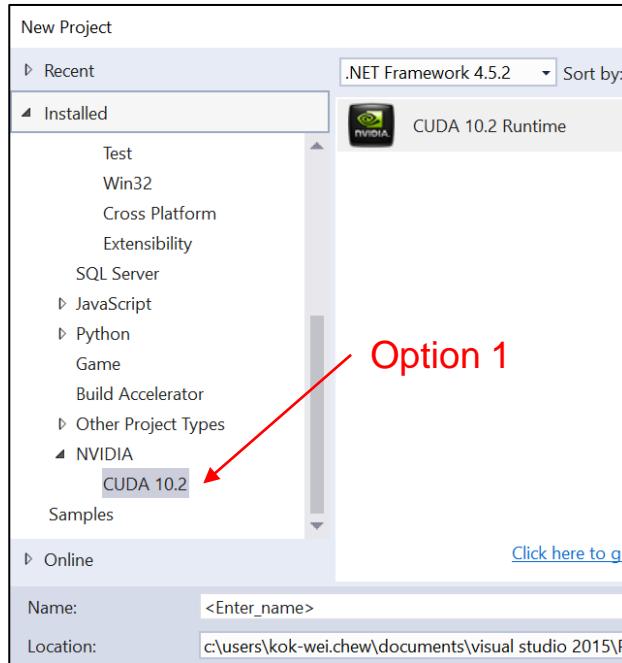
Notes:

Visual studio need to be installed before CUDA, CUDA installation will add support/extensions for visual studio automatically during installation.

Preparation

After installation, we have 2 options for creating a project with CUDA:

- 1) Directly create a CUDA project under the Nvidia CUDA tab.
- 2) Create a regular C++ project and customize later. (If CUDA is only a small part of our project)



Preparation

If option 2 is chosen, extra steps need to be done to enable compilation with CUDA.

Why: either CUDA optimization is not considered at the early stage when the project is created or CUDA not the main part of the project.

1) Add CUDA build customization to projects

At solution explorer: right click on project -> Build Dependencies -> Build Customizations -> check the CUDA 1x.x in the checkbox list

2) Change the compiler to CUDA C++ for the source files containing CUDA code. (can be skipped if the source file has been added as CUDA file)

At solution explorer: right click on source file -> Properties -> General -> Item type -> select CUDA C/C++ from the dropdown menu

3) Change the platform from x86 to x64

At solution explorer: Toolbar -> Solution Platform -> x64

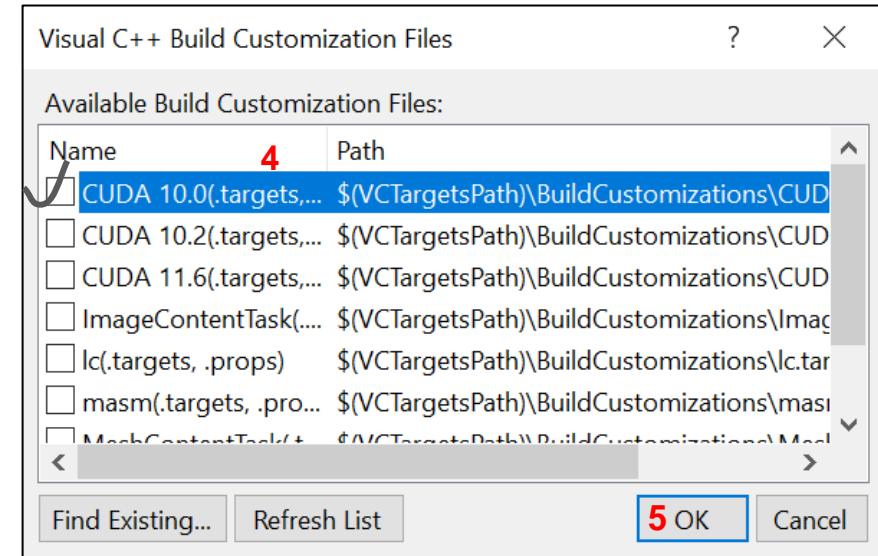
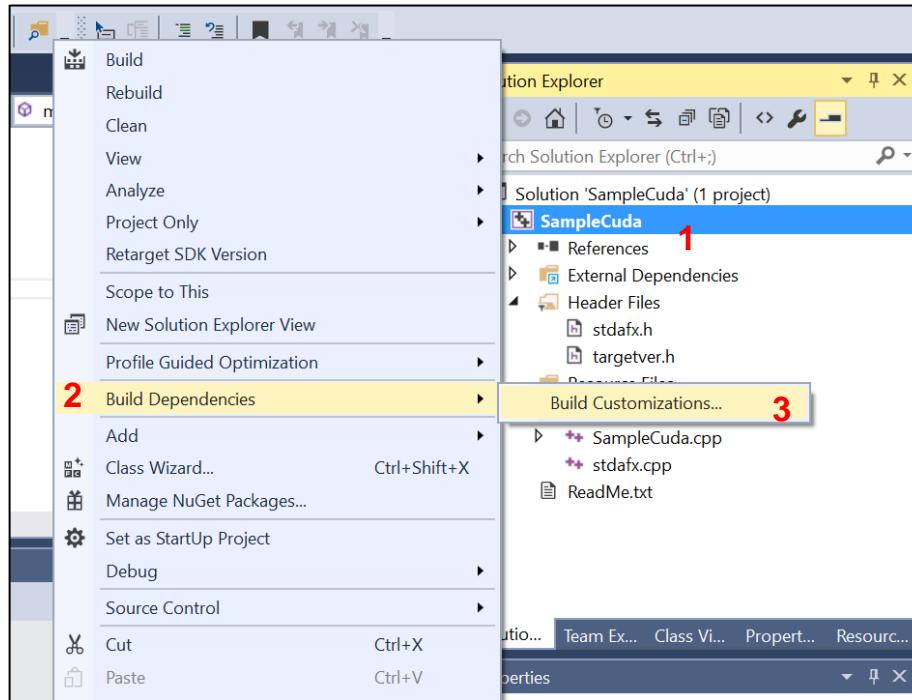
4) Ensure CUDA linker is added

At solution explorer: right click on solution -> Properties -> Linker -> Input -> Additional Dependencies

Preparation

Add CUDA build customization to projects

- At solution explorer: right click on project -> Build Dependencies -> Build Customizations -> check the CUDA 1x.x in the checkbox list

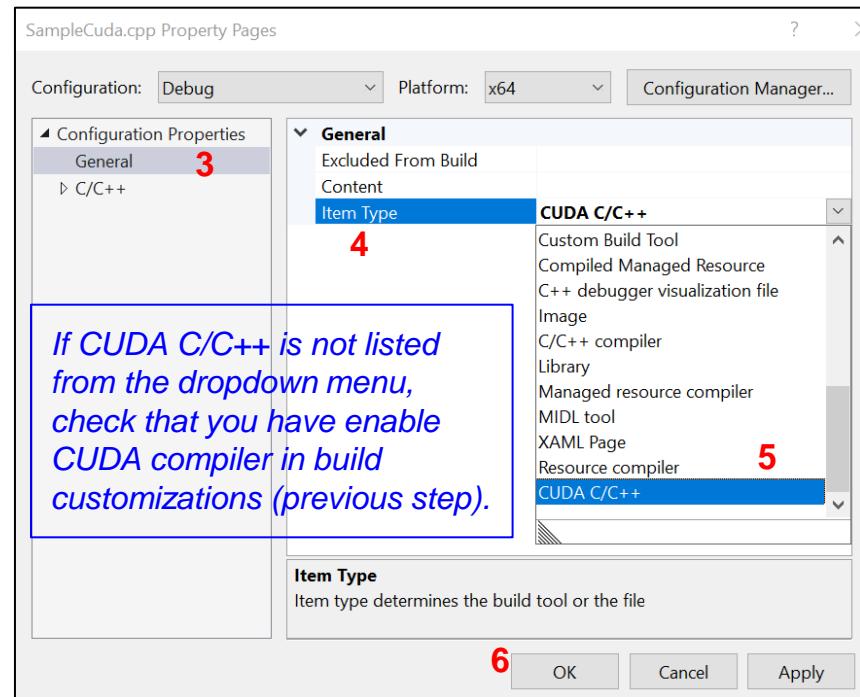
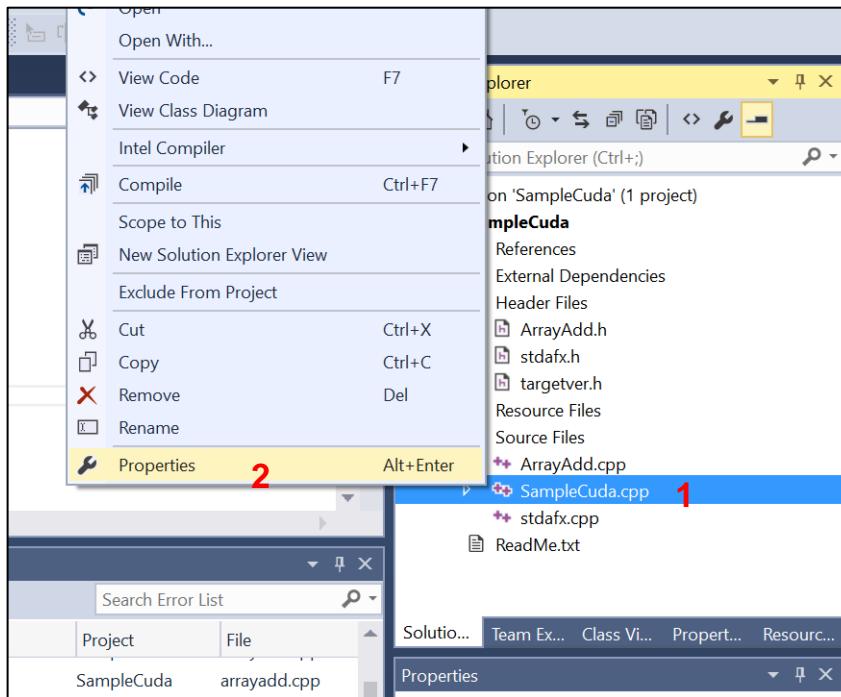


If CUDA is not present in the build
customizations, check your CUDA installation

Preparation

Change the compiler to CUDA C++ for the source files containing CUDA code.

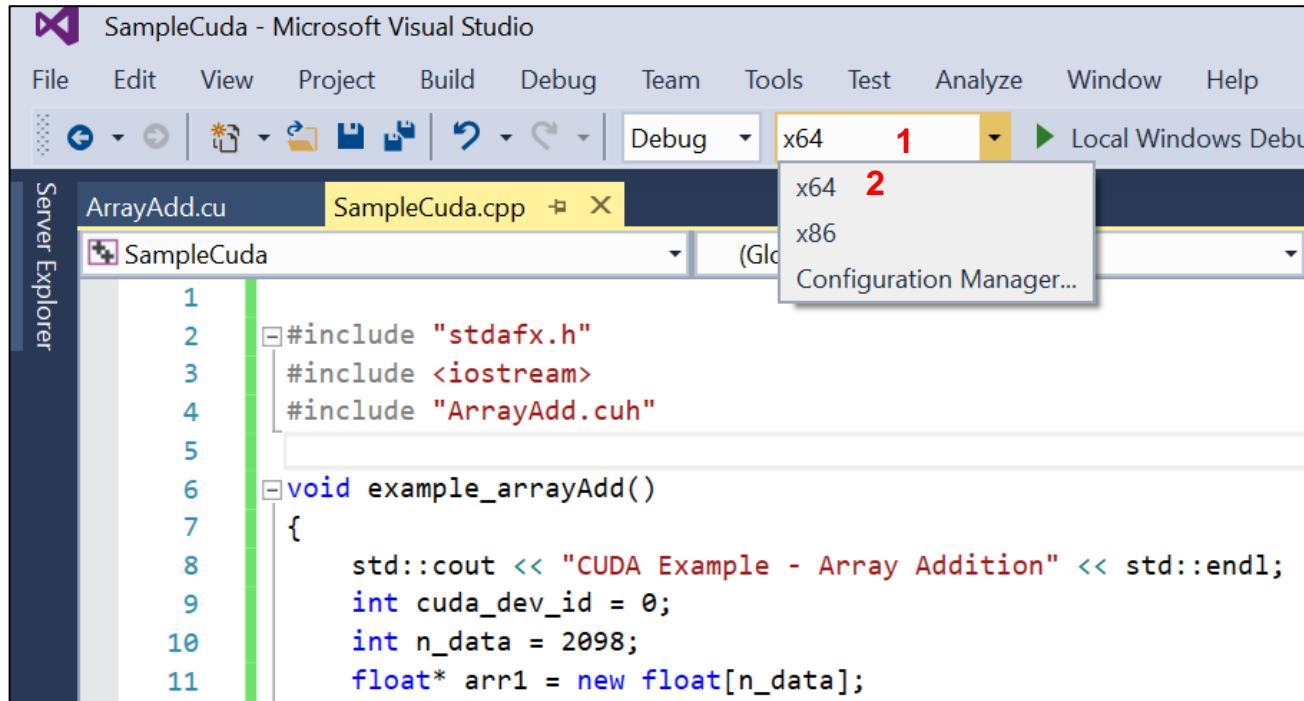
- At solution explorer: right click on source file -> Properties -> General -> Item type -> select CUDA C/C++ from the dropdown menu



Preparation

Change the platform from x86 to x64.

- At solution explorer: Toolbar -> Solution Platform -> x64



Preparation

Ensure CUDA linker is added.

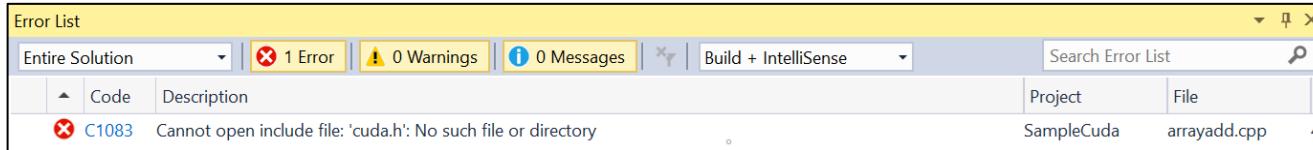
- At solution explorer: right click on solution -> Properties -> Linker -> Input -> Additional Dependencies.

The screenshot shows the Visual Studio interface for a CUDA project named "SampleCuda".

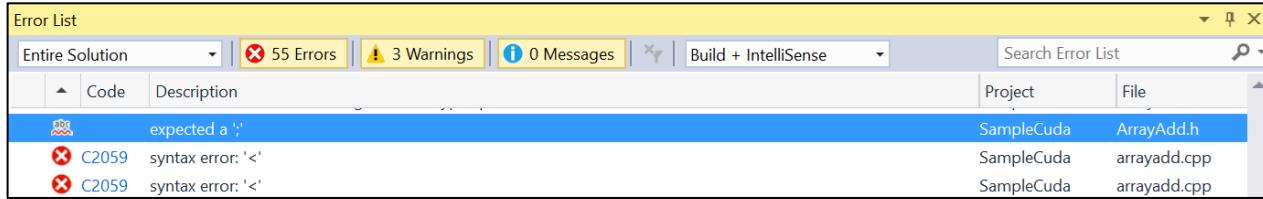
- Solution Explorer:** Shows files like "ArrayAdd.cuh", "stdafx.h", "targetver.h", "Resource Files", "Source Files" (containing "ArrayAdd.cu" and "SampleCuda.cpp"), and "ReadMe.txt". A red arrow labeled "1" points to the project node "SampleCuda".
- Properties Window:** The "Properties" tab is selected. A red arrow labeled "2" points to the "Properties" button at the bottom left.
- Property Pages:** The "SampleCuda Properties" window is open. The "Linker" category is expanded, and the "Input" section is selected (indicated by a red arrow labeled "3").
- Additional Dependencies:** The "Additional Dependencies" field contains the value "ut32.lib;uuid.lib;odbc32.lib;odbccp32.lib;". A red arrow labeled "4" points to the "Edit..." button.
- Output Window:** A callout box labeled "5" points to the entry "cudart.lib" in the "Additional Dependencies" list.
- Evaluated Value:** The "Evaluated value:" field shows "cudart.lib % (AdditionalDependencies)".
- Text Overlay:** A blue-bordered box contains the text "Add cudart.lib and other CUDA related .lib files."

Preparation

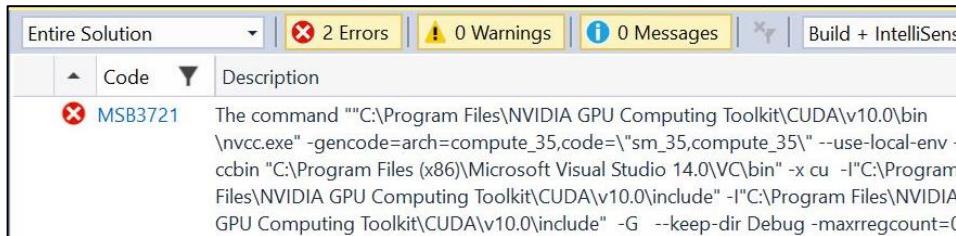
Common build errors and possible solutions



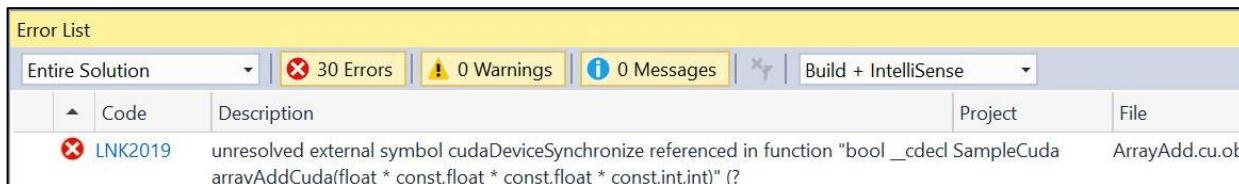
Slide 6



Slide 7



Slide 8, or need to check the build log, might due to CUDA version too new compare to VS. Try create a CUDA project directly or reinstall newer VS / older CUDA



Slide 9

Preparation

Compiling CUDA code in Ubuntu.

- Install Nvidia driver and CUDA toolkit.

```
sudo apt install nvidia-driver-515 # change 515 to specific very u need  
sudo apt install nvidia-cuda-toolkit  
sudo reboot
```

- Compile CUDA source files using nvcc.

```
nvcc main.cpp -o output_filename
```

- If CUDA code is written in other source files instead of main source file, add -x cu option.

```
nvcc main.cpp -o output_filename -x cu
```

- In case where gcc/g++ version not compatible with current nvcc, use --compiler-bindir to specify the gcc/g++ version.

```
nvcc main.cpp -o output_filename -x cu --compiler-bindir /usr/bin/g++-x
```

CUDA Basics

- 1) CUDA Example
 - Array addition
- 2) CUDA Terminology
 - Threads, blocks, grids
 - Dimensions and indices
 - CUDA syntax
- 3) CUDA Parallelism
- 4) Code Example
 - 2D convolution
 - 3D array sorting
 - Conic sections (interpolation)
 - Common errors



CUDA Example - Array Addition

```
void arrayAdd(float* dst, float* src1, float* src2, int n_data)
{
    for (int i = 0; i < n_data; i++)
    {
        dst[i] = src1[i] + src2[i];
    }
}
```

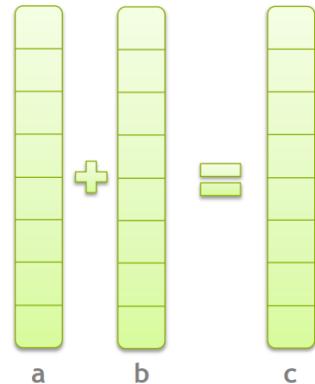
C++
Implementation

```
__global__ void addKernel1D(float *dst, float *src1, float *src2, int n_data)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    if (i < n_data) { dst[i] = src1[i] + src2[i]; }
}

bool arrayAddCuda(float* dst, float *src1, float *src2, int n_data, int cuda_dev_id)
{
    ...
    int const max_thread_size = 1024;
    int block_dim = std::min(max_thread_size, n_data);
    int grid_dim = n_data / max_thread_size + (int)((n_data%max_thread_size) > 0);
    addKernel1D << <grid_dim, block_dim >>(<_d_dst, _d_src1, _d_src2, n_data);

    ...
}
```

CUDA
Implementation



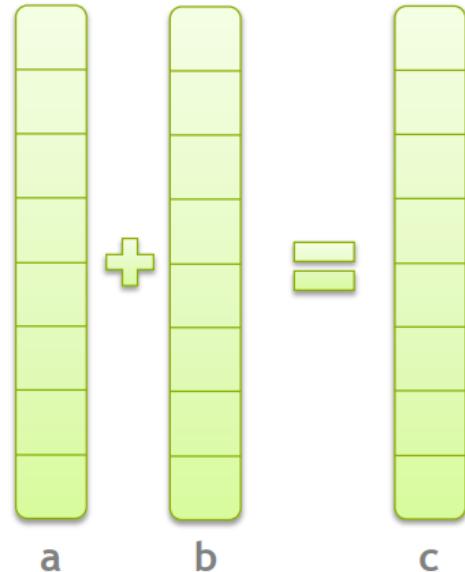
CUDA Example - Array Addition

```
global __ void addKernel1D(float *dst, float *src1, float *src2, int n_data)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    if (i < n_data) { dst[i] = src1[i] + src2[i]; }
}

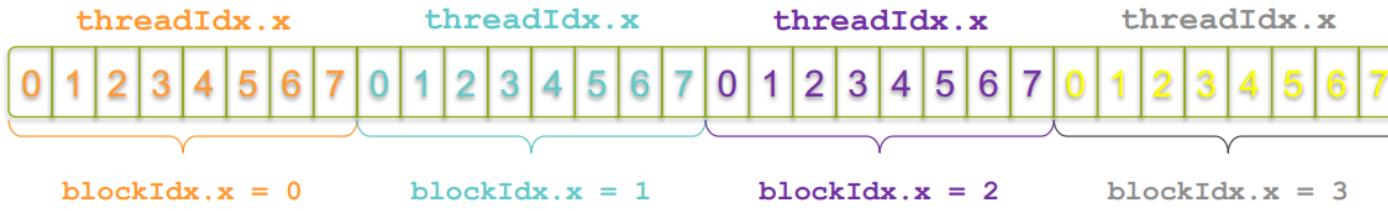
bool arrayAddCuda(float* dst, float *src1, float *src2, int n_data, int cuda_dev_id)
{
    cudaError_t cudaStatus = cudaSuccess;
    // determine thread size and block size
    int const max_thread_size = 1024;
    int block_dim = std::min(max_thread_size, n_data);
    int grid_dim = n_data / max_thread_size + (int)((n_data%max_thread_size) > 0);
    // Allocate GPU buffers for three vectors
    ...
    // Copy input vectors from host memory to GPU buffers.
    ...
    // Launch a kernel on the GPU with one thread for each element.
    addKernel1D << <grid_dim, block_dim >> >(_d_dst, _d_src1, _d_src2, n_data);
    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    ...
    // Copy output vector from GPU buffer to host memory.
    cudaStatus = cudaMemcpy(dst, _d_dst, n_data * sizeof(float), cudaMemcpyDeviceToHost);
    ...
    return true;
}
```

Device code

Host code



CUDA Example - Array Addition



Each thread is assigned to process a single element of the array determined uniquely by the its own ID (threadIdx.x, blockIdx.x)

```
__global__ void addKernel1D(float *dst, float *srcl, float *src2, int n_data)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    if (i < n_data) { dst[i] = srcl[i] + src2[i]; }
}
```

Device code: keyword “`__global__`” indicates it is called from host (CPU), execute on device (GPU).

```
bool arrayAddCuda(float* dst, float *srcl, float *src2, int n_data, int cuda_dev_id)
{
    ...
    int const max_thread_size = 1024;
    int block_dim = std::min(max_thread_size, n_data);
    int grid_dim = n_data / max_thread_size + (int)((n_data%max_thread_size) > 0);
    addKernel1D << << grid_dim, block_dim >> >(_d_dst, _d_srcl, _d_src2, n_data);
    ...
}
```

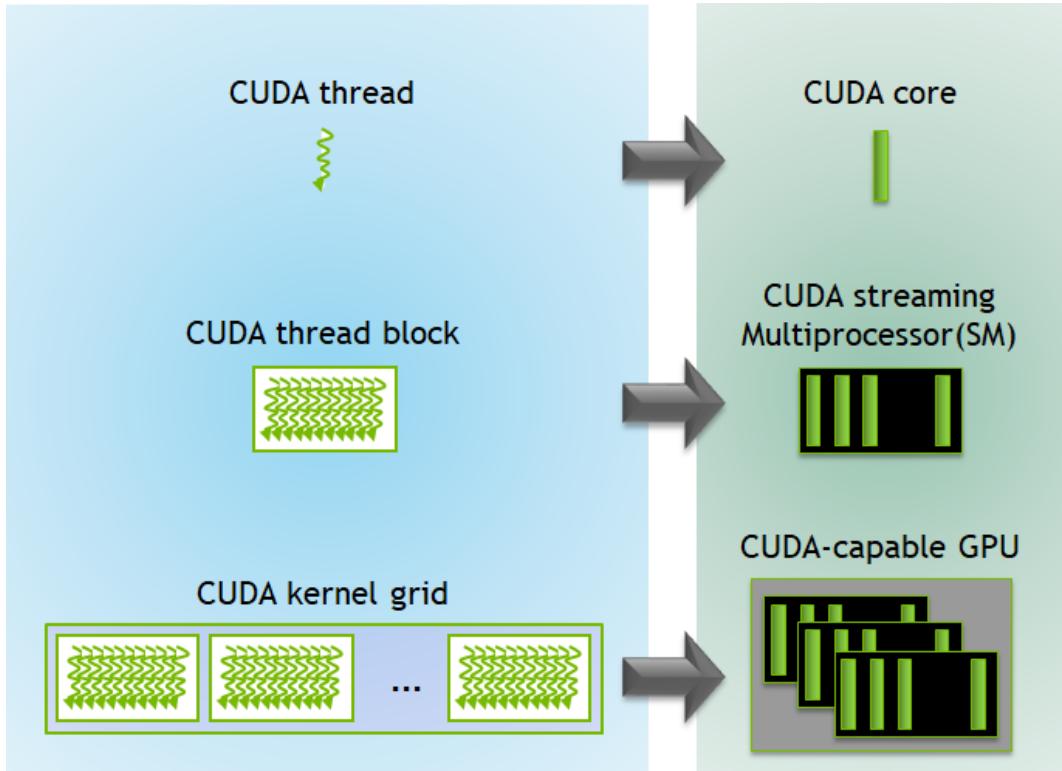
Kernel launch: triple sharp bracket “`<<< >>>`” specify the configuration (number of parallel threads and blocks) for streamed multiprocessing.

CUDA Example - Array Addition

```
cudaStatus = cudaSetDevice(cuda_dev_id);      Assign GPU for calculation (default 0 for single GPU system)
float *_d_src1 = nullptr;
cudaStatus = cudaMalloc((void**)&_d_src1, n_data * sizeof(float));  Allocate memory for variables on device
...
cudaStatus = cudaMemcpy(_d_src1, src1, n_data * sizeof(float), cudaMemcpyHostToDevice);
...                                                 Copy data from host to device
int const max_thread_size = 1024;
int block_dim = std::min(max_thread_size, n_data);
int grid_dim = n_data / max_thread_size + (int)((n_data%max_thread_size) > 0);
addKernel1D << <grid_dim, block_dim >> >(_d_dst, _d_src1, _d_src2, n_data);    Kernel launch
...
cudaStatus = cudaDeviceSynchronize();  Sync host and device before retrieving results
...
cudaStatus = cudaMemcpy(dst, _d_dst, n_data * sizeof(float), cudaMemcpyDeviceToHost);
...                                                 Copy results from device to host
cudaFree(_d_src1);  Free and release memory on device
...
cudaDeviceReset();
```

```
cudaError_t cudaStatus = cudaSuccess;
if (cudaStatus != cudaSuccess)          Cuda error status
{
    std::cout << "Failed to allocate CUDA memory, " << cudaGetStringError(cudaStatus) << "." << std::endl;
    return false;
}
```

CUDA Terminology - Threads Blocks Grids



Thread:

- ❖ Smallest execution unit (core) of a CUDA program (kernel function).

Block:

- ❖ A collection of threads formed a block.
- ❖ Threads within same block can communicate with each other.
- ❖ Maximum 1024 threads per block.

Grid:

- ❖ A grid is made up of multiple blocks.
- ❖ Maximum 65536 block per in each dimension.

Both grid or block can consist of either 1D/2D or 3D blocks/threads.

CUDA Terminology - Dimensions and Indices

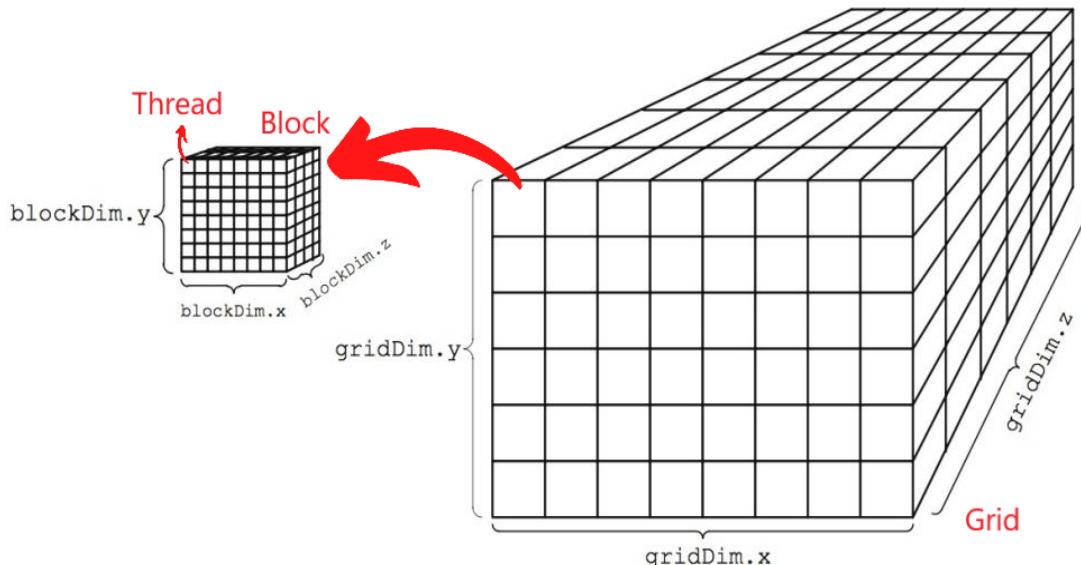


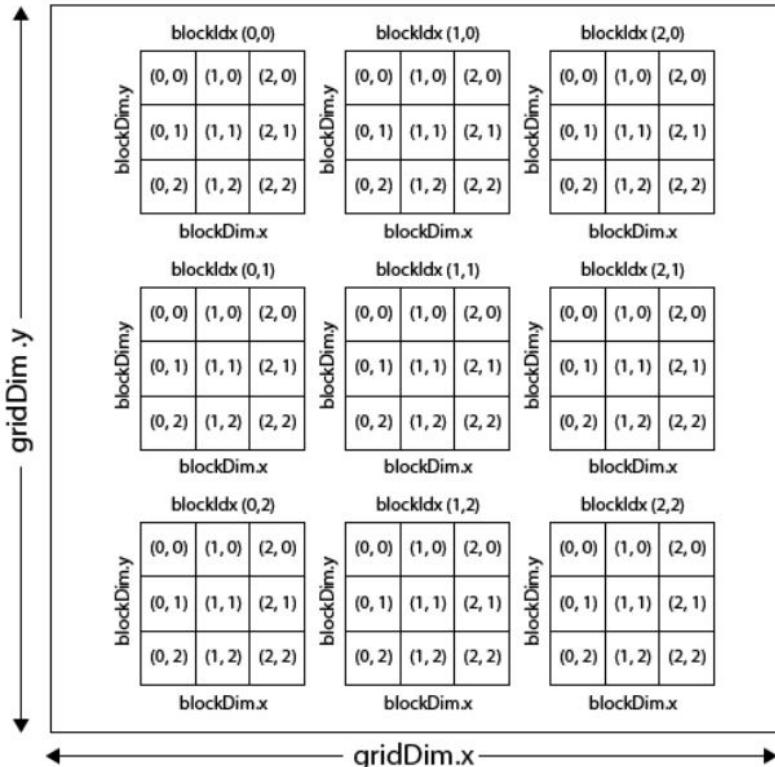
Image source: <http://harmanani.github.io/classes/csc447/Notes/Lecture15.pdf>

```
addKernel1D << <grid_dim, block_dim >> >
```

```
int block_dim = 5; // equivalent to (5,1,1)  
dim3 block_dim(5); // equivalent to (5,1,1)  
dim3 block_dim(5,4); // equivalent to (5,4,1)  
dim3 block_dim(5,4,3); // x=5, y=4, z=3
```

- ❖ A simple kernel launch will specify the block dimension (# of threads) and grid dimension (# of blocks).
- ❖ The variable for dimension can be an **integer type** (1D) or a **dim3 datastructure**.

CUDA Terminology - Dimensions and Indices



Each thread is determined uniquely by 6 indices:

- ❖ `threadIdx.x`
- ❖ `threadIdx.y`
- ❖ `threadIdx.z`
- ❖ `blockIdx.x`
- ❖ `blockIdx.y`
- ❖ `blockIdx.z`

Depends on task, if the kernel launch, some indices might not be used.

- ❖ E.g. if a 2D grid of 1D block is launched, only `threadIdx.x`, `blockIdx.x` and `blockIdx.y` is needed

CUDA Terminology - Dimensions and Indices

```
__device__  
int getGlobalIdx_1D_1D() // 1D grid 1D block  
{  
    return blockIdx.x * blockDim.x + threadIdx.x;  
}  
  
__device__  
int getGlobalIdx_1D_2D() // 1D grid 2D block  
{  
    return blockIdx.x * blockDim.x * blockDim.y  
        + threadIdx.y * blockDim.x + threadIdx.x;  
}  
  
__device__  
int getGlobalIdx_3D_3D() // 3D grid 3D block  
{  
    int blockId = blockIdx.x + blockIdx.y * gridDim.x  
        + gridDim.x * gridDim.y * blockIdx.z;  
    int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)  
        + (threadIdx.z * (blockDim.x * blockDim.y))  
        + (threadIdx.y * blockDim.x) + threadIdx.x;  
    return threadId;  
}
```

For some tasks where the computation is perform on a linear continuous data (e.g. 1D array), getting a global index (index for array) is necessary.

More detailed guide on indexing at [here](#).

CUDA Terminology - CUDA Syntax

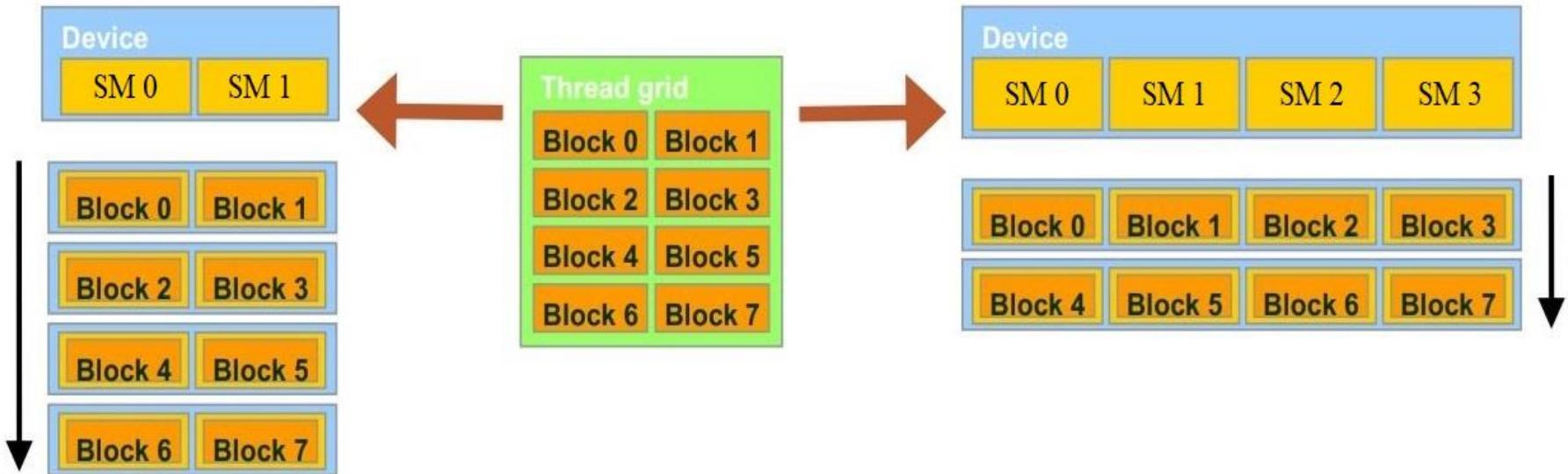
Syntax (function)	
<code>__global__</code>	declares kernel function, called on host and executed on device
<code>__device__</code>	declares device function, called and executed on device
<code>__host__</code>	declares host function, called and executed on host
<code>__device__ __host__</code>	Can called by device, executed on device or called by host, executed on host

Syntax (variable)	
<code>__device__</code>	device variable in global memory, accessible from all threads, with lifetime of application
<code>__shared__</code>	device variable in block's shared memory, accessible from all threads within a block, with lifetime of block
<code>__constant__</code>	device variable in constant memory, accessible from all threads, with lifetime of application

CUDA Parallelism

Ideally, all threads within a block is executed in parallel.

Blocks within a grid might be executed in parallel or sequentially depending on GPU architecture (# of streaming multiprocessors).



CUDA Parallelism

Only threads within same block can be synchronized and communicate with each other.

Keyword “`__syncthreads()`” in kernel function will a stop and waits for other threads.

Example, flipping an image horizontally.



CUDA Parallelism

Strategy:

- ❖ Each block handle a single row.
- ❖ Each thread of the same block handle a single pixel of the same row.

Steps:

1. Thread i read the value of the opposite pixel ($nx-1-i$).
2. Sync all threads.
3. Thread i changed the value of i-th pixel to the value of ($nx-1-i$) pixel.

```
__global__ void flipHorizontalKernel(float* src_dst, int nx, int ny)
{
    int i = blockIdx.x;
    int j = threadIdx.x;
    int loc1 = i*nx + j;
    int loc2 = i*nx + (nx - 1 - j);
    float tmp = src_dst[loc2];           // each thread read opposite pixel
    __syncthreads();                   // sync and wait all the reading done
    src_dst[loc1] = tmp;               // before changing the pixel value
}

bool flipHorizontalCuda(float* const src_dst, int const nx, int const ny, int const cuda_dev_id)
{
    ...
    int const block_dim = nx;          // each thread handle a pixel
    int const grid_dim = ny;           // along x-direction of image
    flipHorizontalKernel << <grid_dim, block_dim >> >(_d_src_dst, nx, ny);
    ...
}
```



Without
synchronization

Code Example - 2D Convolution

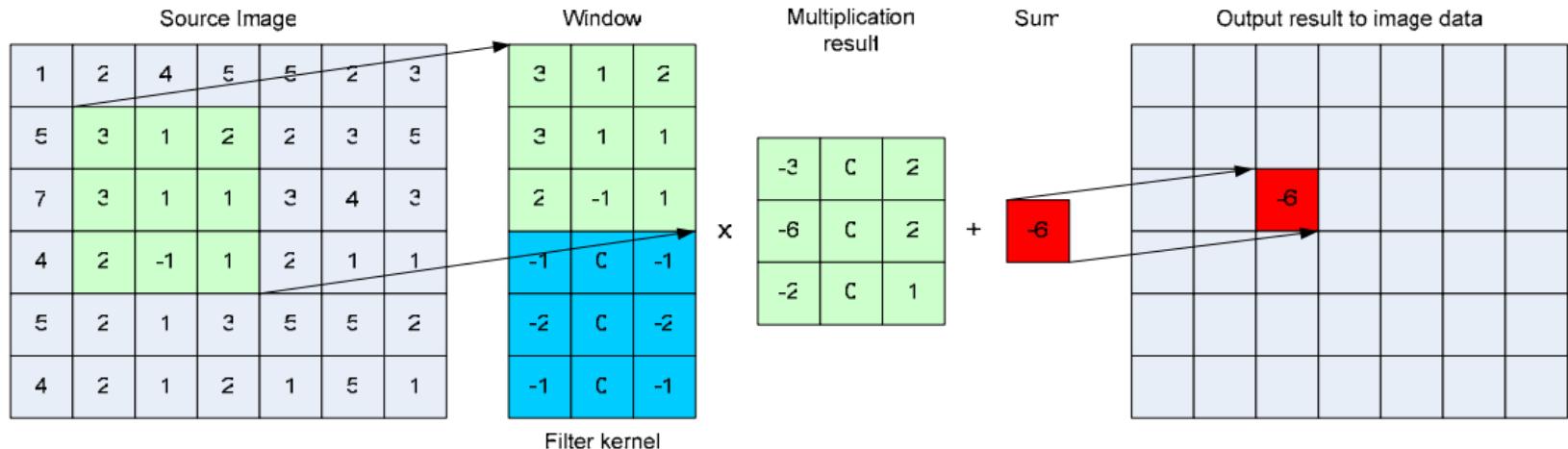


Image source: https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf

A sliding window run through the image, a simple implementation requires loops for:

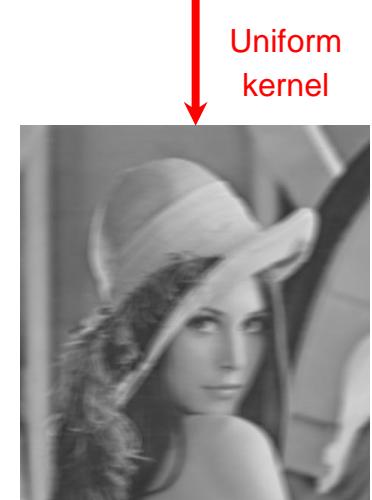
1. Pixels of output image along x-direction
2. Pixels of output image along y-direction
3. Pixels of kernel/window along x-direction
4. Pixels of kernel/window along y-direction

Code Example - 2D Convolution

```
void convolve2d(float* dst, float* src, float* kernel,
                int src_nx, int src_ny, int kern_nx, int kern_ny)
{
    int const dst_nx = src_nx - kern_nx + 1;
    int const dst_ny = src_ny - kern_ny + 1;
    for (int i = 0; i < dst_ny; i++)
    {
        for (int j = 0; j < dst_nx; j++)
        {
            int loc_dst = i*dst_nx + j;
            float sum = 0;
            for (int p = 0; p < kern_ny; p++)
            {
                for (int q = 0; q < kern_nx; q++)
                {
                    int loc_src = (i + p)*src_nx + (j+q);
                    int loc_kern = p*kern_nx + q;
                    sum += src[loc_src] * kernel[loc_kern];
                }
            }
            dst[loc_dst] = sum;
        }
    }
}
```

C++ Implementation

The boundary pixels are neglected (omitted).



Uniform
kernel

Code Example - 2D Convolution

```
__global__ void convolve2dKernel(float *dst, float *src, float *kernel,
                                 int src_nx, int src_ny, int kern_nx, int kern_ny)
{
    int dst_nx = src_nx - kern_nx + 1;
    int dst_ny = src_ny - kern_ny + 1;
    int i = blockIdx.x;                                // determine the location
    int j = threadIdx.x;                             // of the pixel by thread
    int loc_dst = i*dst_nx + j;                      // and block index
    float sum = 0;
    for (int p = 0; p < kern_ny; p++)              // calculate the sum
    {                                                 // over a sliding
        for (int q = 0; q < kern_nx; q++)           // window
        {
            int loc_src = (i + p)*src_nx + (j + q);
            int loc_kern = p*kern_nx + q;
            sum += src[loc_src] * kernel[loc_kern];
        }
    }
    dst[loc_dst] = sum;
}

bool convolve2dCuda(float* dst, float* src, float* kernel,
                    int src_nx, int src_ny, int kern_nx, int kern_ny,
                    int cuda_dev_id)
{
    ...
    int conv_nx = src_nx - kern_nx + 1;      // kernel of 1D grid of 1D block,
    int conv_ny = src_ny - kern_ny + 1;      // each threads handle a pixel
    int block_dim = conv_nx;                 // of the output image
    int grid_dim = conv_ny;
    convolve2dKernel << <grid_dim, block_dim>> > (_d_dst, _d_src, _d_kern, src_nx, src_ny, kern_nx, kern_ny);
    ...
}
```

CUDA
Implementation
(Configuration 1)

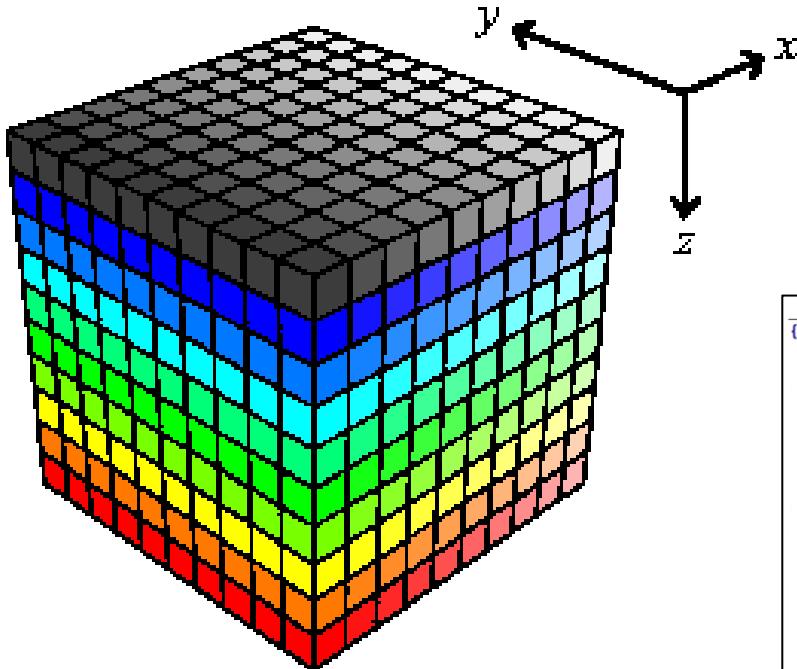
Code Example - 2D Convolution

```
__global__ void convolve2dKernel(float *dst, float *src, float *kernel,
                                 int src_nx, int src_ny, int kern_nx, int kern_ny)
{
    int dst_nx = src_nx - kern_nx + 1;
    int dst_ny = src_ny - kern_ny + 1;
    int i = blockIdx.x;
    int j = blockIdx.y;
    int p = threadIdx.x;
    int q = threadIdx.y;
    int loc_dst = i*dst_nx + j;
    __shared__ float product[1024];           // shared array to store the product
    int loc_src = (i + p)*src_nx + (j + q);
    int loc_kern = p*kern_nx + q;
    product[loc_kern] = src[loc_src] * kernel[loc_kern];
    __syncthreads();                         // sync to ensure pixel of kernel is
    float sum = 0;                          // is calculated
    if ((threadIdx.x == 0) && (threadIdx.y == 0)) // restrict 1 thread to
    {                                       // calculate the sum
        for (int m = 0; m < kern_ny*kern_nx; m++)
        { sum += product[m]; }
        dst[loc_dst] = sum;
    }
}

bool convolve2dCuda(float* dst, float* src, float* kernel,
                    int src_nx, int src_ny, int kern_nx, int kern_ny,
                    int cuda_dev_id)
{
    ...
    int conv_nx = src_nx - kern_nx + 1;      // kernel of 2D grid of 2D block,
    int conv_ny = src_ny - kern_ny + 1;      // each threads handle the product
    dim3 block_dim(kern_ny, kern_nx);        // kernel and image window
    dim3 grid_dim(conv_ny, conv_nx);
    convolve2dKernel << <grid_dim, block_dim >> > (_d_dst, _d_src, _d_kern, src_nx, src_ny, kern_nx, kern_ny);
    ...
}
```

CUDA
Implementation
(Configuration 2)

Code Example - 3D Array Sorting



A 3D array is stored as contiguous array in memory.

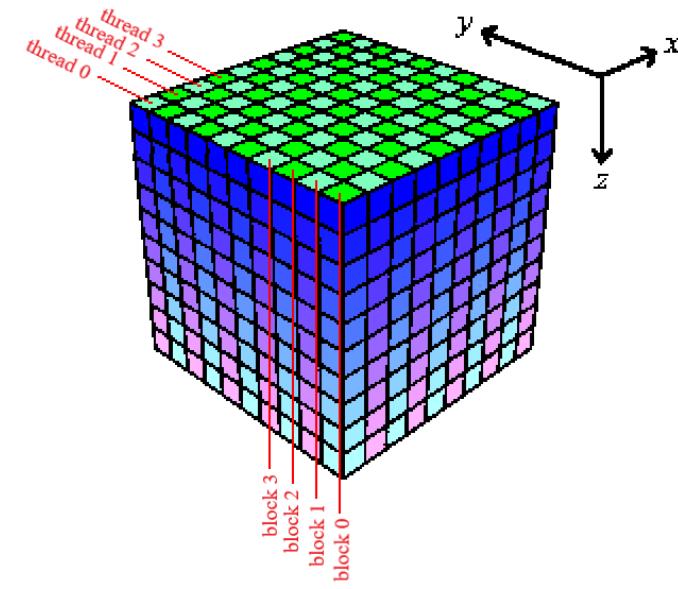
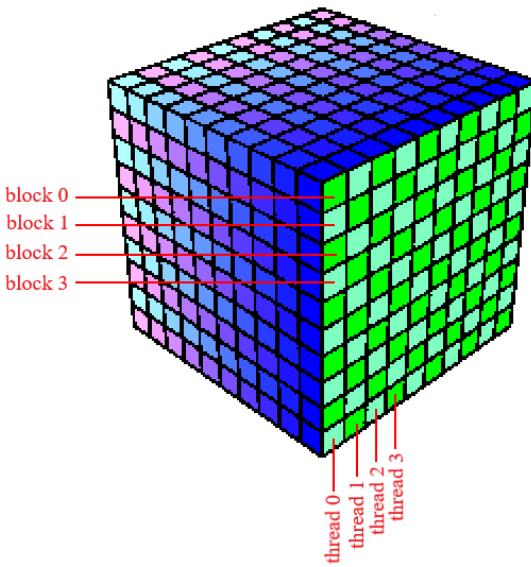
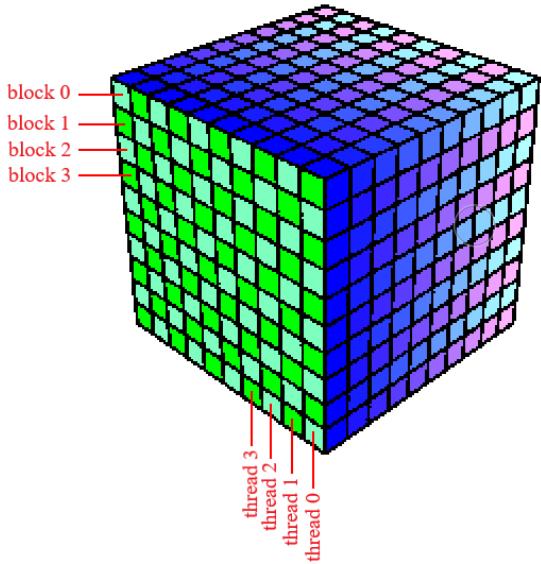
Depending on sorting direction, the flattened distance (stride) between adjacent elements differs.

An example of selection sort algorithm modified for sorting with arbitrary starting index and stride.

```
_device_ _host_ void selectionSort(float* arr, int start_idx, int stride, int n_data)
{ // sorting an array start from index 'start_idx' with 'stride' between sorting elements
    int idx_i, idx_j, idx_min;
    float tmp_min;
    for (int i = 0; i < n_data -1; i++)
    {
        idx_i = start_idx + (i*stride);
        idx_min = idx_i;
        tmp_min = arr[idx_i];
        for (int j = i+1; j < n_data; j++)
        {
            idx_j = start_idx + (j*stride);
            if (arr[idx_j] < tmp_min) { idx_min = idx_j; tmp_min = arr[idx_j]; }
        }
        arr[idx_min] = arr[idx_i];
        arr[idx_i] = tmp_min;
    }
}
```



Code Example - 3D Array Sorting



Sorting along X-axis

- ❖ Each block handle a X-Y plane
- ❖ Block size = Ny
- ❖ Grid size = Nz
- ❖ Start index $Nx * Ny * \text{block_idx} + Nx * \text{thread_idx}$
- ❖ Distance between neighbor (stride) = 1

Sorting along Y-axis

- ❖ Each block handle a X-Y plane
- ❖ Block size = Nx
- ❖ Grid size = Nz
- ❖ Start index $Nx * ny * \text{block_idx} + \text{thread_idx}$
- ❖ Distance between neighbor (stride) = nx

Each thread responsible for sorting a row

Sorting along Z-axis

- ❖ Each block handle a X-Z plane
- ❖ Block size = Ny
- ❖ Grid size = nx
- ❖ Start index $Nx * \text{block_idx} + \text{thread_idx}$
- ❖ Distance between neighbor (stride) = Ny * Nx

Code Example - 3D Array Sorting

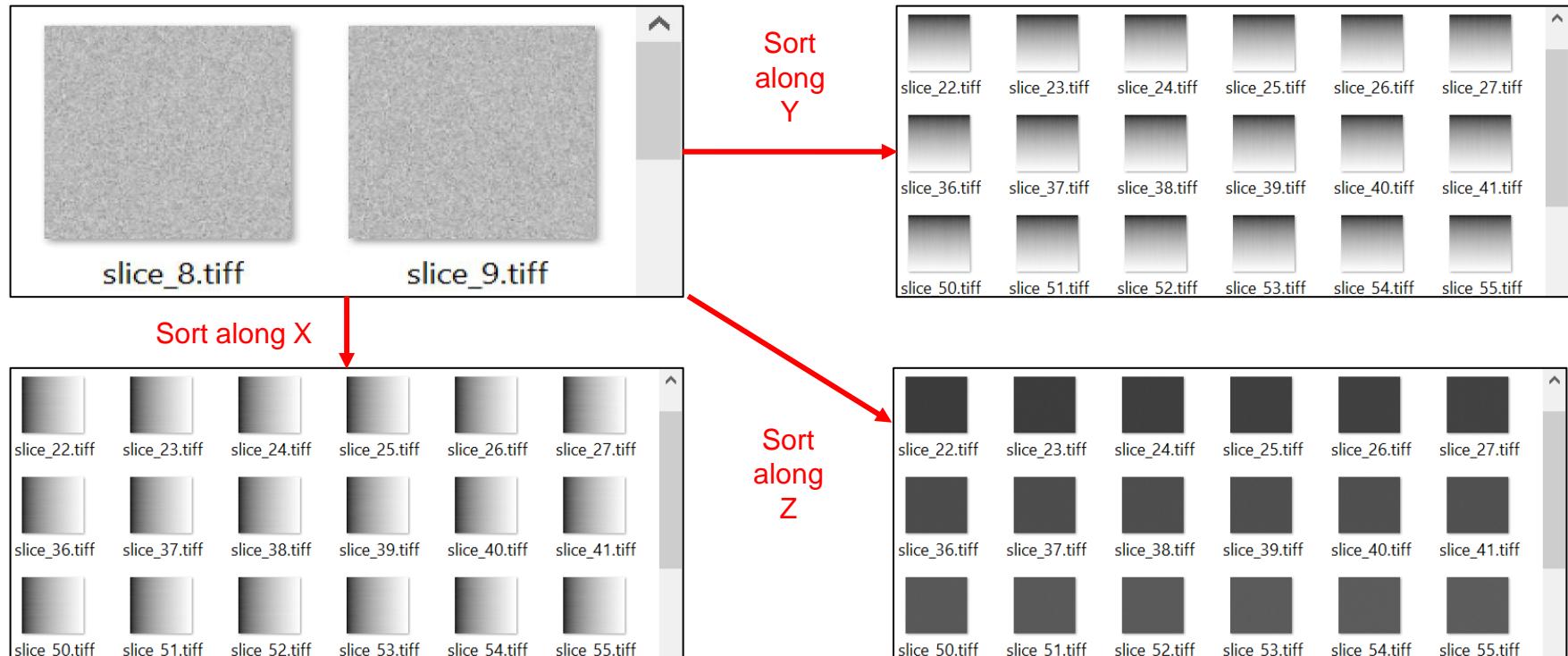
```
__global__ void sortKernel(float *src_dst, int n_data_sort,
                           int start_idx_stridel, int start_idx_stride2,
                           int sort_stride)
{
    int i = blockIdx.x;
    int j = threadIdx.x;
    int start_idx = j*start_idx_stridel + i*start_idx_stride2;
    selectionSort(src_dst, start_idx, sort_stride, n_data_sort);
    //insertionSort(src_dst, start_idx, sort_stride, n_data_sort);
    //quickSort(src_dst, start_idx, sort_stride, 0, n_data_sort-1);
    //heapSort(src_dst, start_idx, sort_stride, n_data_sort);
}
```

The start_idx for each thread is determined by the blockIdx, threadIdx and the number of elements between the 1st element of the axis called start_stride.

Code Example - 3D Array Sorting

```
if ((axis == 'x') || (axis == 'X'))
{ // sort along x
    int block_dim = ny; int grid_dim = nz;
    int n_data_sort = nx;
    int start_idx_stridel = nx;
    int start_idx_stride2 = nx*ny;
    int sort_stride = 1;
    sortKernel << <grid_dim, block_dim>>>(_d_src_dst, n_data_total, start_idx_stridel, start_idx_stride2, sort_stride);
}
else if ((axis == 'y') || (axis == 'Y'))
{ // sort along y
    int block_dim = nx; int grid_dim = nz;
    int n_data_sort = ny;
    int start_idx_stridel = 1;
    int start_idx_stride2 = nx*ny;
    int sort_stride = nx;
    sortKernel << <grid_dim, block_dim>>>(_d_src_dst, n_data_total, start_idx_stridel, start_idx_stride2, sort_stride);
}
else if ((axis == 'z') || (axis == 'Z'))
{ // sort along z
    int block_dim = nx; int grid_dim = ny;
    int n_data_sort = nz;
    int start_idx_stridel = 1;
    int start_idx_stride2 = nx;
    int sort_stride = nx*ny;
    sortKernel << <grid_dim, block_dim>>>(_d_src_dst, n_data_total, start_idx_stridel, start_idx_stride2, sort_stride);
}
```

Code Example - 3D Array Sorting



Using stack images of random numbers to verify results

Code Example - Conic Sections

A conic is the curve obtained as the intersection of a plane, with the surface of a cone.

- ❖ Circle:
 - Plane intersect cone at angle=0.
- ❖ Ellipse:
 - Plane intersect cone at angle between 0 and slanted side of the cone.
- ❖ Parabola:
 - Plane intersect cone at an angle parallel to the slanted side of the cone.
- ❖ Hyperbola:
 - Plane intersect cone at an angle greater than the slanted side of the cone.

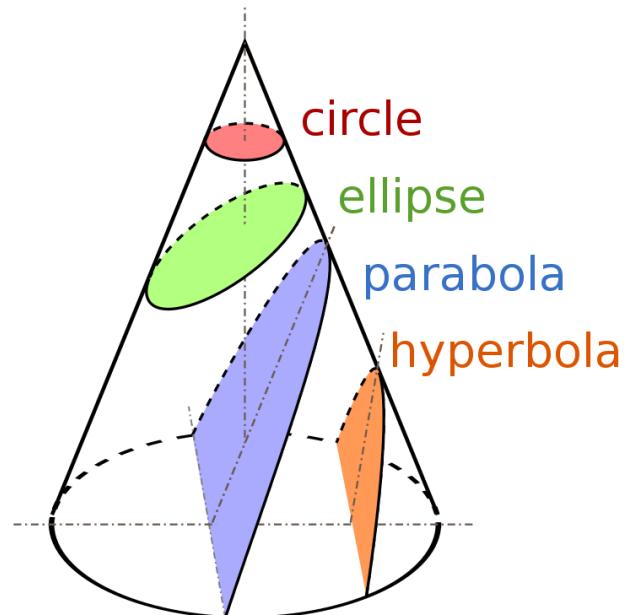


Image source:
https://en.wikipedia.org/wiki/Conic_section

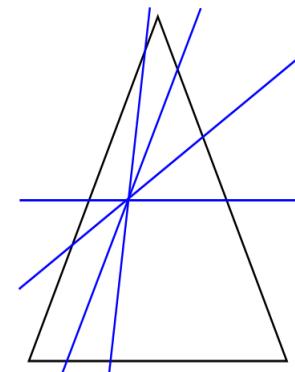
Code Example - Conic Sections

The coordinates of 2D plane slicing the cone is identified and used for interpolation to get the respective conic curves.

```
int block_dim = dst_nx; // width of the slice
int grid_dim = dst_ny; // height of the slice

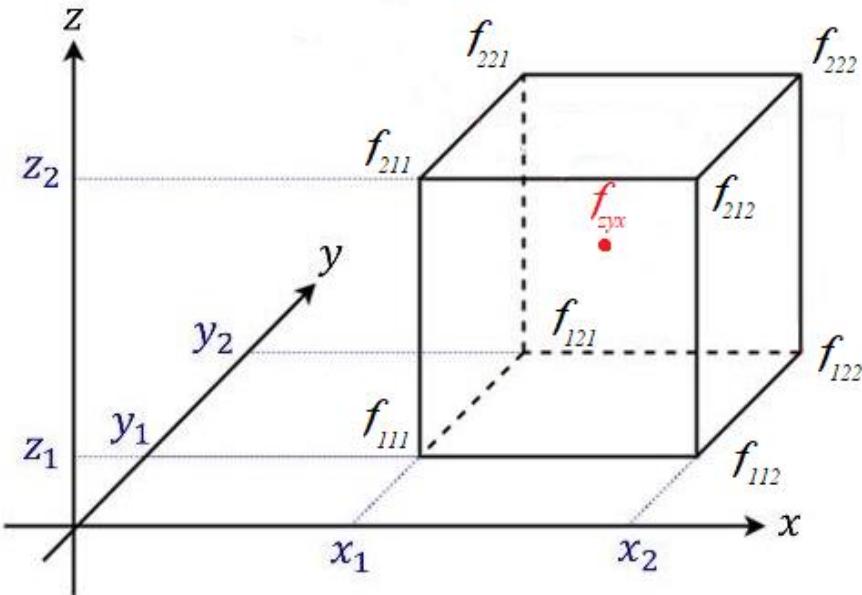
slicingKernel << <grid_dim, block_dim >> >(&_d_dst[i*dst_nx*dst_ny], dst_nx, dst_ny,
    _d_src, src_nx, src_ny, src_nz, x0, y0, z0, angle);
```

```
__global__ void slicingKernel(float *dst, int dst_nx, int dst_ny,
    float *src, int src_nx, int src_ny, int src_nz,
    float x0, float y0, float z0, float angle)
{
    int i = threadIdx.x;
    int j = blockIdx.x;
    int loc_dst = i + j*dst_nx;
    float src_y = y0 + j / factor - (float)(dst_ny) / 2.0;
    float src_x0 = i - (float)(dst_nx) / 2.0;
    float src_z0 = 0;
    float src_x = (src_x0)*std::cos(angle) - (src_z0)*std::sin(angle) + x0;
    float src_z = (src_x0)*std::sin(angle) + (src_z0)*std::cos(angle) + z0;
    dst[loc_dst] = trilinearInterpolate(src, src_nx, src_ny, src_nz, src_x, src_y, src_z);
}
```



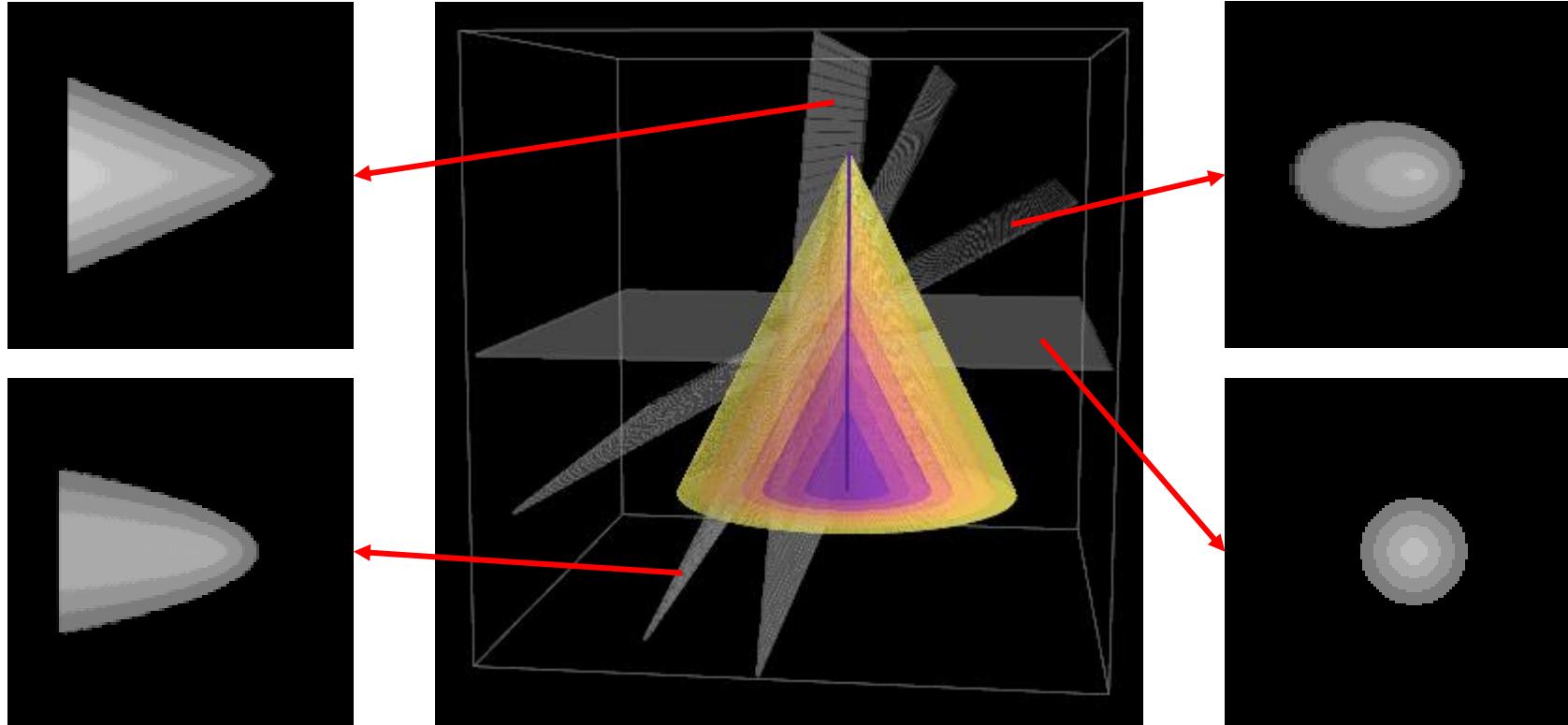
Code Example - Conic Sections (Interpolation)

Trilinear interpolation



```
device float trilinearInterpolate(float *src,
int src_nx, int src_ny, int src_nz, float x, float y, float z)
{
    float delta = 1.0e-3;
    float x1 = floorf(x);
    float x2 = ceilf(x);
    float y1 = floorf(y);
    float y2 = ceilf(y);
    float z1 = floorf(z);
    float z2 = ceilf(z);
    int x1_idx = (int)x1;
    int x2_idx = (int)x2;
    int y1_idx = (int)y1;
    int y2_idx = (int)y2;
    int z1_idx = (int)z1;
    int z2_idx = (int)z2;
    float f111 = src[x1_idx + y1_idx*src_nx + z1_idx*src_nx*src_ny];
    float f112 = src[x2_idx + y1_idx*src_nx + z1_idx*src_nx*src_ny];
    float f121 = src[x1_idx + y2_idx*src_nx + z1_idx*src_nx*src_ny];
    float f122 = src[x2_idx + y2_idx*src_nx + z1_idx*src_nx*src_ny];
    float f211 = src[x1_idx + y1_idx*src_nx + z2_idx*src_nx*src_ny];
    float f212 = src[x2_idx + y1_idx*src_nx + z2_idx*src_nx*src_ny];
    float f221 = src[x1_idx + y2_idx*src_nx + z2_idx*src_nx*src_ny];
    float f222 = src[x2_idx + y2_idx*src_nx + z2_idx*src_nx*src_ny];
    float f11x = f111 + (f112 - f111)*(x - x1) / max(x2 - x1, delta);
    float f12x = f121 + (f122 - f121)*(x - x1) / max(x2 - x1, delta);
    float f21x = f211 + (f212 - f211)*(x - x1) / max(x2 - x1, delta);
    float f22x = f221 + (f222 - f221)*(x - x1) / max(x2 - x1, delta);
    float f1yx = f11x + (f12x - f11x)*(y - y1) / max(y2 - y1, delta);
    float f2yx = f21x + (f22x - f21x)*(y - y1) / max(y2 - y1, delta);
    float fzyx = f1yx + (f2yx - f1yx)*(z - z1) / max(z2 - z1, delta);
    return fzyx;
}
```

Code Example - Conic Sections



Code Example - Common Errors

Illegal memory access:

- ❖ Accessing array with index out its bound.
 - Error when indexing the array with threadIdx and blockIdx.
 - Memory allocated for array is less than intended (Ctrl-C + Ctrl-V during malloc).
- ❖ Using dynamically allocated array in kernel without specifying during kernel launch.

Invalid argument:

- ❖ Probably happened when copy data from host to device, the size of data to be copied is larger than the size allocated (Ctrl-C + Ctrl-V fault again)

Invalid configuration:

- ❖ Launching a kernel with block size (# of threads) larger than designed (1024).

Too many resources requested for launch:

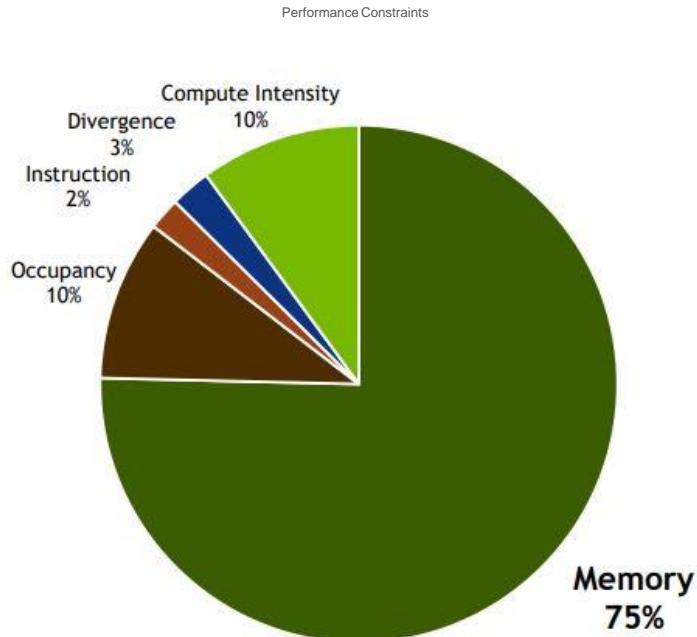
- ❖ Too many local variable used/declared within a thread, which exceed the register size of the SM. Reducing # of threads per block might resolve this issue.

CUDA Optimization

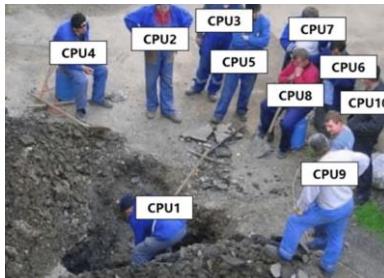
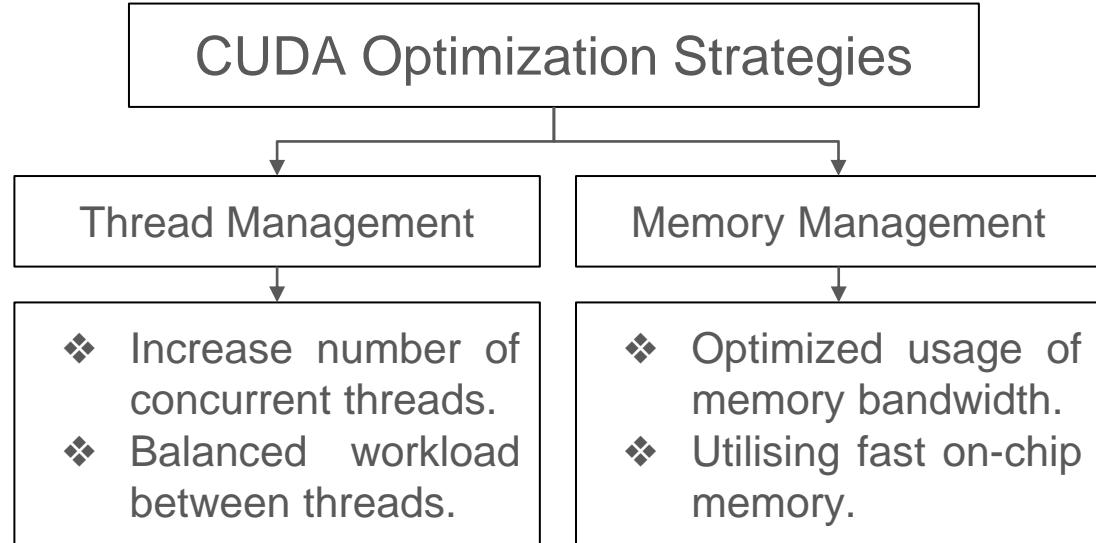
- 1) GPU Architecture
 - Threads execution
 - Memory hierarchy
- 2) Thread Occupancy
 - Kernel launch configuration
 - Thread Divergence
- 3) Memory Access Pattern
 - Memory coalescing
 - Shared memory & bank conflicts
 - Texture
- 4) CUDA Stream



CUDA Optimization



Source: <https://ondemand.gputechconf.com/gtc/2017/presentation/s7122-stephen-jones-cuda-optimization-tips-tricks-and-techniques.pdf>



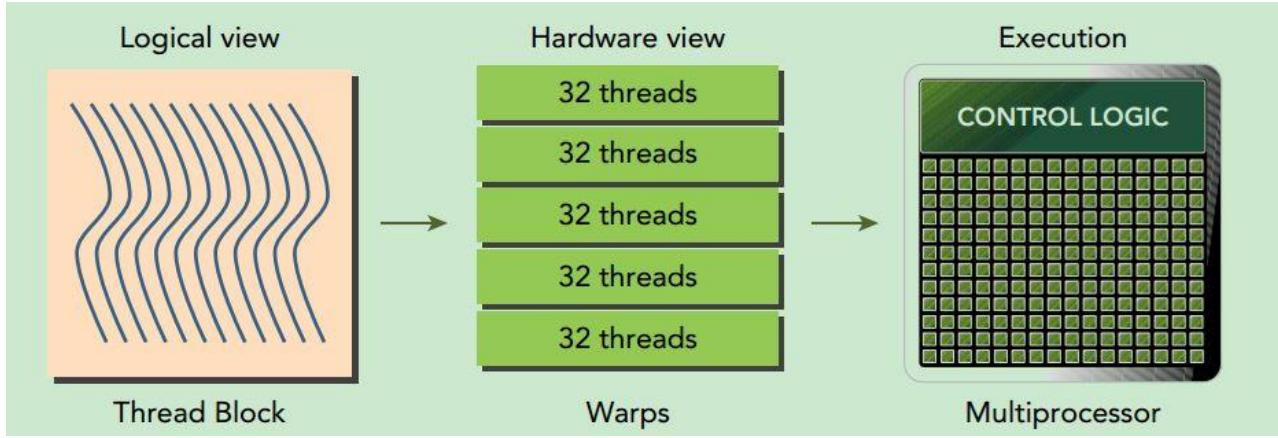
GPU Architecture

RTX 3090 Specs		Remarks
Base Clock	1395 MHz	
CUDA Cores	10496	Equivalent to ALU in CPU, or ‘warp’ in CUDA programming.
SM Count	82	Equivalent to a physical core in CPU
Memory Size	24 GB	Stores global memory of a CUDA program.
Bandwidth	936.2 GB/s	
L1 Cache	128 KB per SM	Stores shared memory of a CUDA program.
L2 Cache	6 MB	Stores global or local memory of a CUDA program.
FP32 (float) performance	35.58 TFLOPS	

Source: <https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622>



GPU Architecture - Threads Execution



Logical perspective

- ❖ A thread block is a collection of threads organized in a 1D, 2D, or 3D layout.

Hardware perspective

- ❖ A thread block is a 1D collection of warps. Threads in a thread block are organized in a 1D layout, and each set of 32 consecutive threads forms a warp.

A warp consists of 32 consecutive threads and all threads in a warp are executed in Single Instruction Multiple Thread (SIMT) manner.

GPU Architecture - Memory Hierarchy

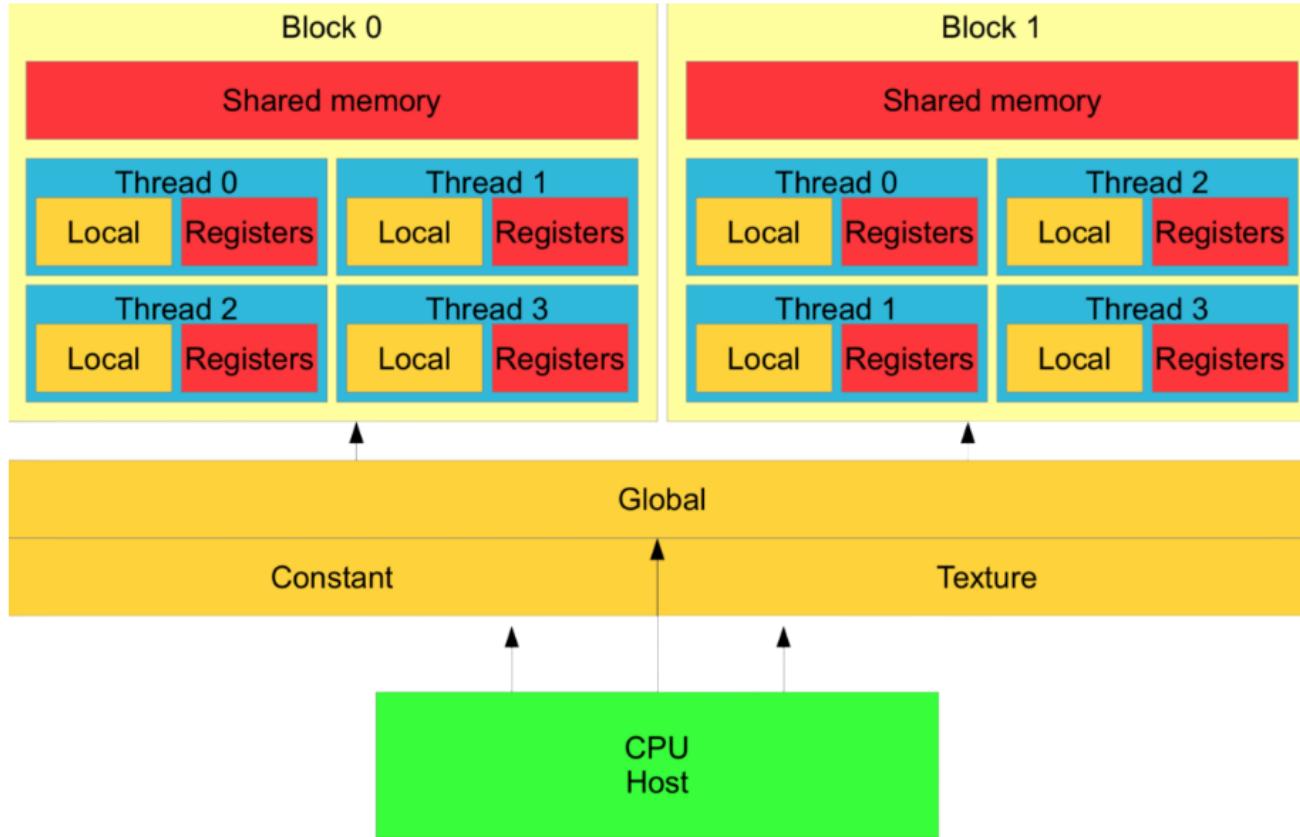


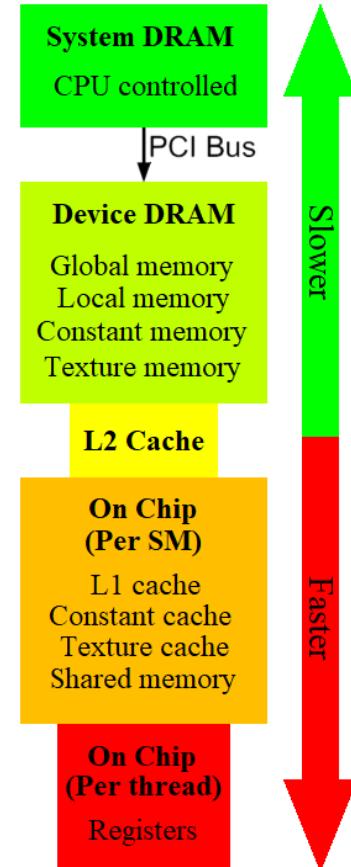
Image source: <http://thebeardsage.com/cuda-memory-hierarchy/>

GPU Architecture - Memory Hierarchy

Memory	Scope	Speed	Relative cycle time (number of instructions)
Global	All	Slow, cached	200+
Constant	All	Slow, cached	200+
Texture	All	Slow, cached	2~200+
Local*	Thread	Slow, cached	200+
Shared	Block	Fast	2~3
Register	Thread	Fast	1

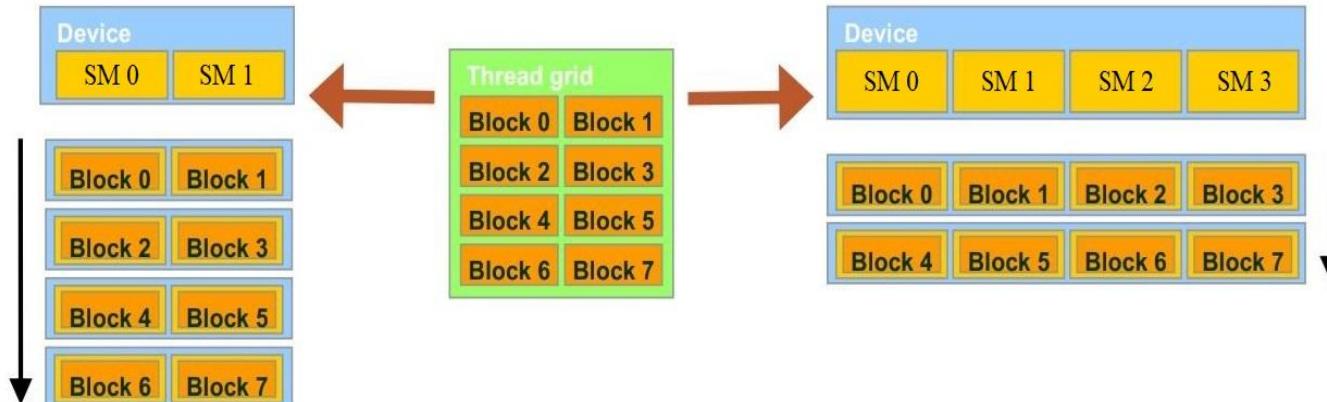
Source: http://shodor.org/media/content/petascale/materials/GPGPU/presentations/Optimizing_CUDA_pdf

* the term 'local' refers to local variable in programming context, the actual location for the variable might be stored in register, cache or global memory.



Thread Occupancy - Kernel Launch Configuration

- ❖ Number of threads per block is limited to 1024.
- ❖ Max number of blocks per grid is $65535 \times 65535 \times 65535$.
- ❖ A CUDA block is a programming abstraction. From hardware perspective, blocks might be executed in parallel or serial.
- ❖ For serial execution of blocks, a new block could not be started until the all the threads in previous block completed the tasks.



Thread Occupancy - Kernel Launch Configuration

- ❖ Balanced workload
 - The block size can be taken as large as possible, reducing the overhead cost for warp scheduler in a block.
- ❖ Imbalance workload
 - Block size should be smaller but depends on the work nature. Using large block size will cause the all threads to wait for the slowest thread and reduce the thread occupancy.
- ❖ Avoid irregular block size (not a multiple of 32),
 - e.g. a block of 33 or 515 threads, instead of 32 or 512.
- ❖ Optimal block size requires empirical tuning.

Thread Occupancy - Thread Divergence

If code branches for threads within a warp. The warp will execute the threads in *then* statement followed by the *else* statement serially.

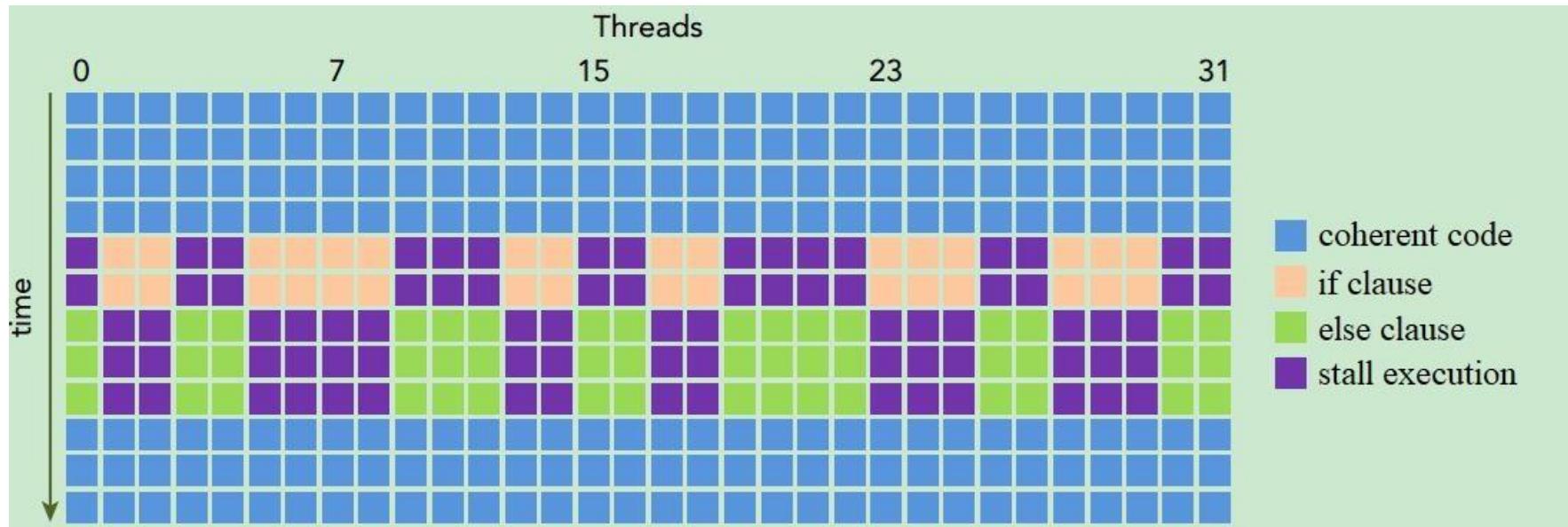


Image source: <https://hanxiao.me/cuda-programming-note-11-warp/>

Thread Occupancy - Thread Divergence

Example: sort algo comparison

- ❖ A simpler algorithm like selection sort with less (almost zero) branching perform the best despite its $O(n^2)$ time complexity.
- ❖ GPU can't make use of 'smarter' algorithm like quick sort or heap sort which requires sophisticated logic.

Sort algorithm	Durations (seconds) for sorting 262144 sets of array with length 512	
	CPU (single thread)	GPU (Global memory)
Selection sort	26.9	4.2
Insertion sort	40.9	7.2
Heap sort	11.2	5.4
Quick sort	6.3	10.8

$n \log(n)$ alike algorithm with simple logic >>> [bitonic sort](#)

Thread Occupancy - Thread Divergence

Example: sort algo comparison

- ❖ A simpler algorithm like selection sort with less (almost zero) branching perform the best despite its $O(n^2)$ time complexity.
- ❖ GPU can't make use of 'smarter' algorithm like quick sort or heap sort which requires sophisticated logic.

排序算法	运行时长 (秒)	
	CPU (单线程)	GPU (Global memory)
选择排序	26.9	4.2
插入排序	40.9	7.2
堆排序	11.2	5.4
快速排序	6.3	10.8

$n \log(n)$ alike algorithm with simple logic >>> [bitonic sort](#)

Thread Occupancy - Thread Divergence

Bitonic sort

- ❖ Implements based on bitonic sequence.
- ❖ Recursively constructing a bitonic sequence from 2 subsequences, 1 in ascending order while the other in descending order.
- ❖ Time complexity: $O(n(\log n)^2)$
- ❖ Advantage: easy to implement as sorting network for massive parallelization.
- ❖ Drawback: can only perform on array with size 2^N (virtual padding to 2^N required for array with arbitrary size).

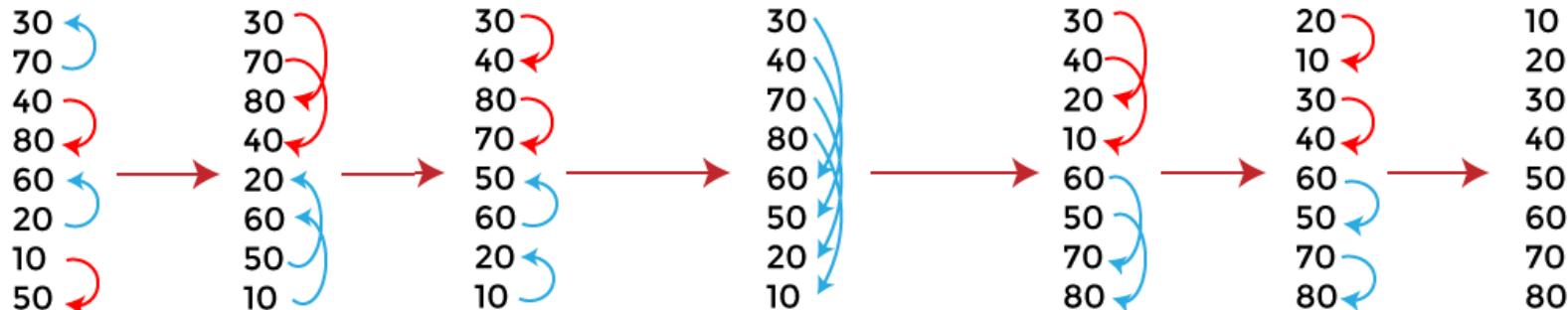


Image source: <https://www.javatpoint.com/bitonic-sort>

Thread Occupancy - Thread Divergence

Bitonic sort

- ❖ Blue (arrow down) represent sort in ascending order
- ❖ Green (arrow up) represent sort in descending order
- ❖ The algorithm has **consistent logic (instructions)** regardless of the input data.

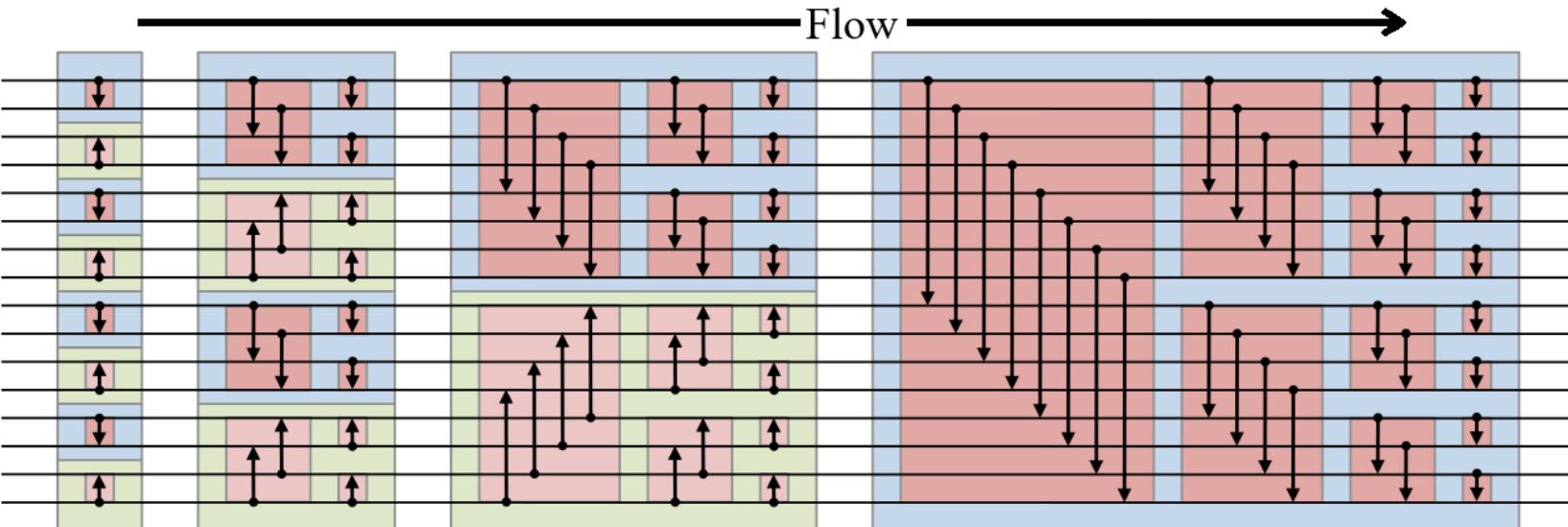


Image source: https://en.wikipedia.org/wiki/Bitonic_sorter

Thread Occupancy - Thread Divergence

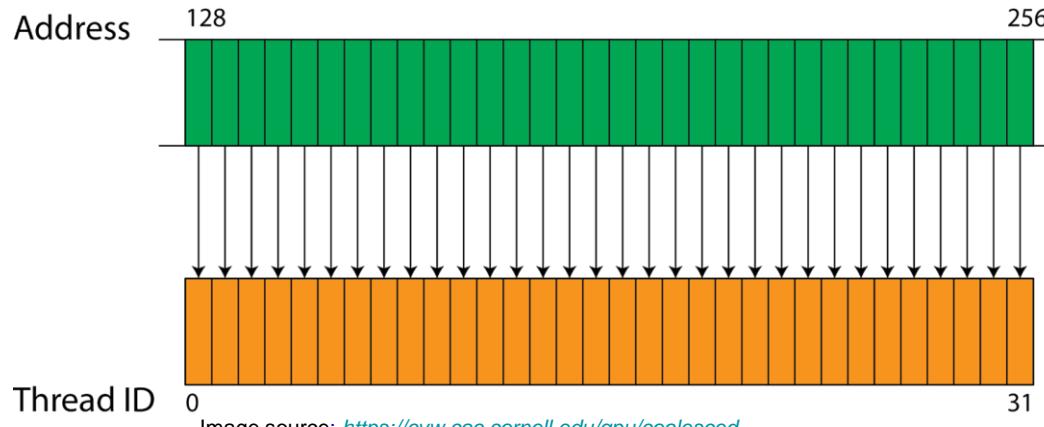
Bitonic sort comparison with other sorting algorithms:

- ❖ With **consistent instructions issued to all the threads**, bitonic sort outperformed quick sort and heap sort.

Sort algorithm	Durations (seconds) for sorting 262144 sets of array with length 512	
	CPU (single thread)	GPU (Global memory)
Selection sort	26.9	4.2
Insertion sort	40.9	7.2
Heap sort	11.2	5.4
Quick sort	6.3	10.8
Bitonic sort (sequential)	19.9	1.8

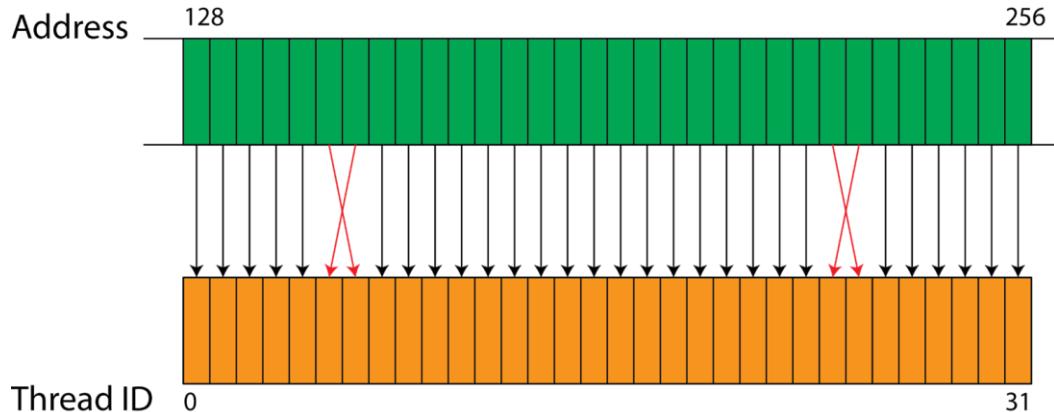
Memory Access Pattern - Memory Coalescing

- ❖ When access to global memory is unavoidable, the access can be coalesced (grouped) into fewer transactions.
- ❖ For coalesced memory access, threads within a single warp need to access consecutive address of data in the DRAM.

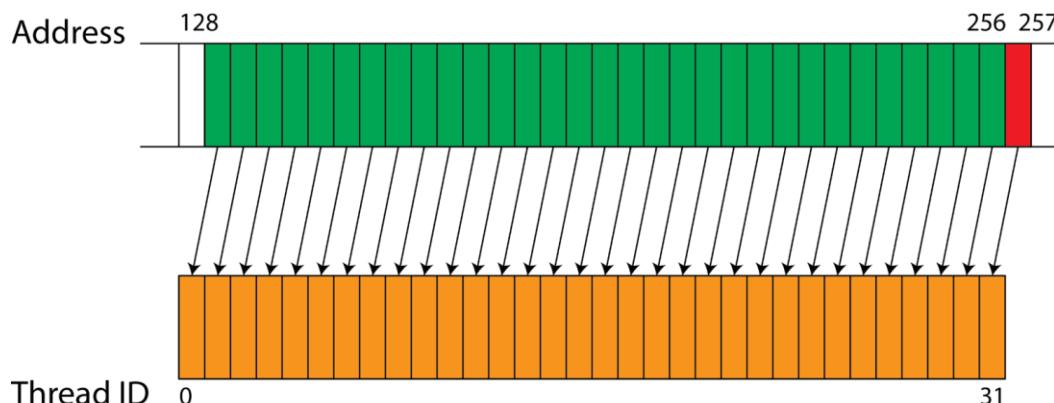


Aligned and sequential,
1 transaction needed.

Memory Access Pattern - Memory Coalescing



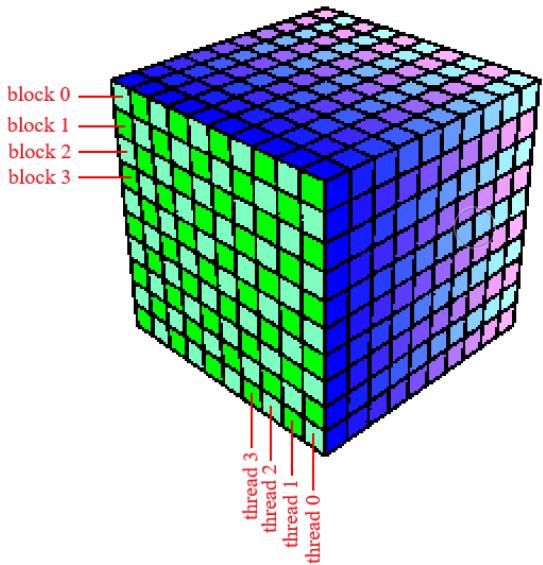
Aligned but not sequential,
1 transaction needed. Not
supported for older hardware.



Mis-aligned,
2 transaction needed.

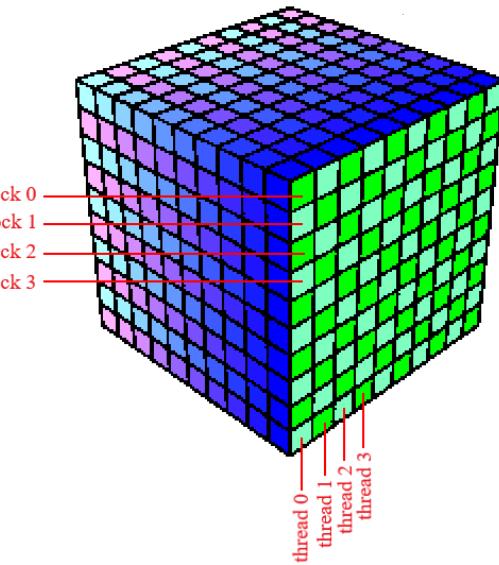
Memory Access Pattern - Memory Coalescing

Example: sorting along different direction



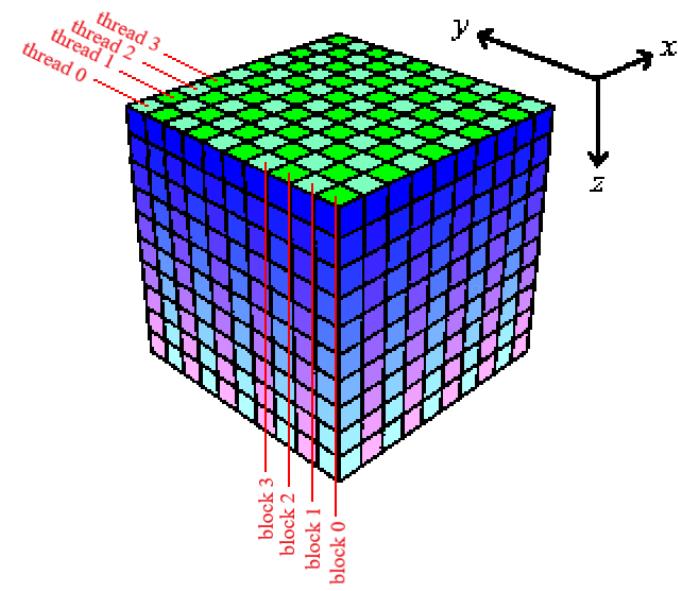
Sorting along x-axis

- ❖ Each thread handle a row in X-Y plane
- ❖ Duration (seconds)
 - > 10.2 (selection)
 - > 10.1 (bitonic)



Sorting along y-axis

- ❖ Each thread handle a row in X-Y plane
- ❖ Duration: (seconds)
 - > 4.2 (selection)
 - > 1.8 (bitonic)



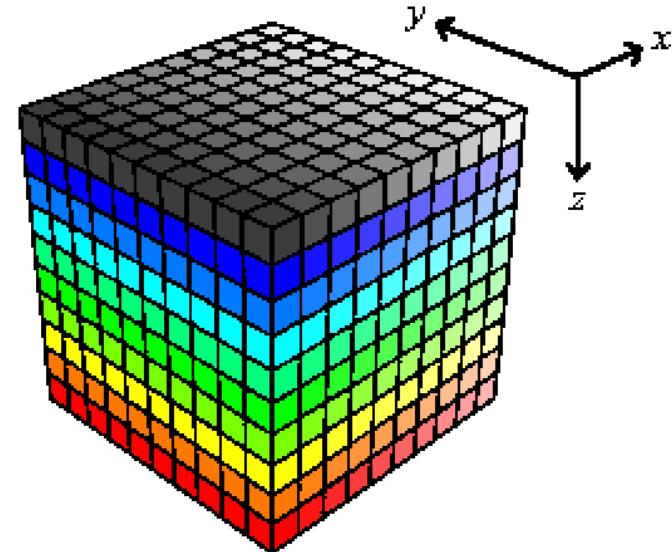
Sorting along z-axis

- ❖ Each thread handle a row in X-Z plane
- ❖ Duration: (seconds)
 - > 4.3 (selection)
 - > 1.9 (bitonic)

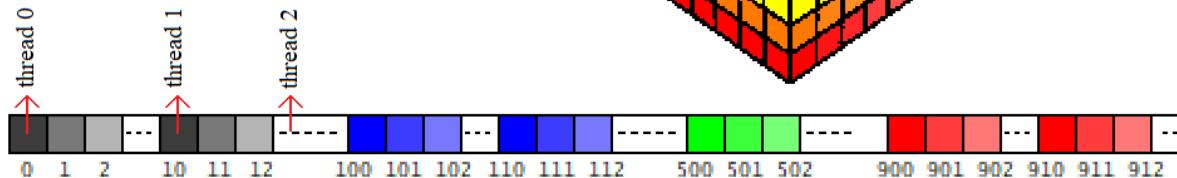
Memory Access Pattern - Memory Coalescing

Example: sorting along different direction

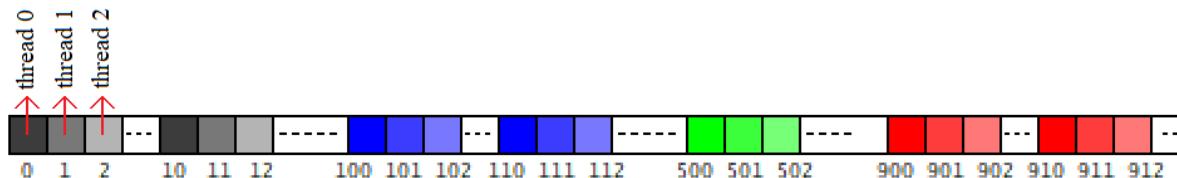
- ❖ Sorting along X
 - Starting index for each array = 0, Nx, 2*Nx, 3*Nx ...
- ❖ Sorting along Y/Z
 - Starting index for each array = 0, 1, 2, 3 ...
- ❖ For sorting along Y or Z, starting index for array is consecutive, higher chances for coalesced transaction.



Sort along X



Sort along Y/Z



Memory Access Pattern - Shared Memory

- ❖ When data is small and requires often reuse, the data can be loaded into shared memory and process.
- ❖ Shared memory speed is close to register speed (150x of global).
- ❖ Example of task can utilise shared memory:
 - Sorting
 - Convolution
 - Matrix multiplication

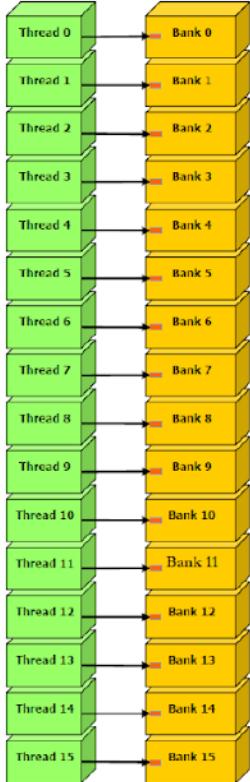
Memory Access Pattern - Shared Memory

Bank Conflict

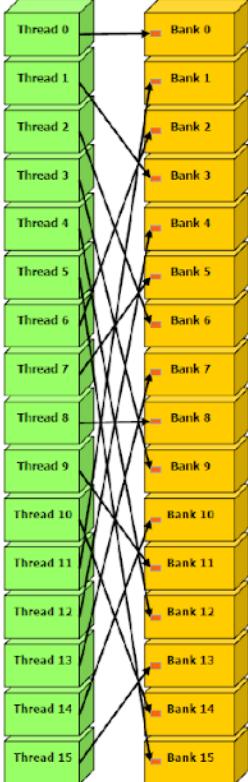
- ❖ More than 1 threads access memory within a bank (32-bit) in shared memory.
- ❖ When bank conflicts occurs, warp execution become serialized (reduce parallelism).
- ❖ For single precision float, can neglect this issue unless 2 or more threads accessing same location in an array.

Memory Access Pattern - Shared Memory

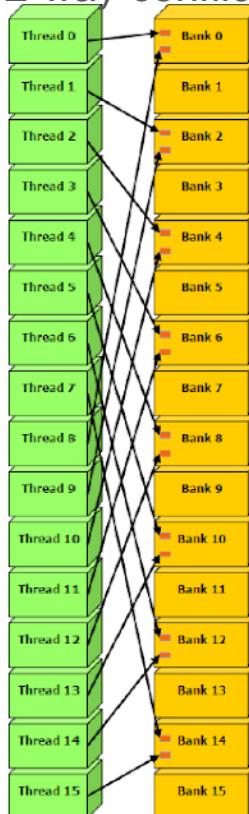
No bank conflict



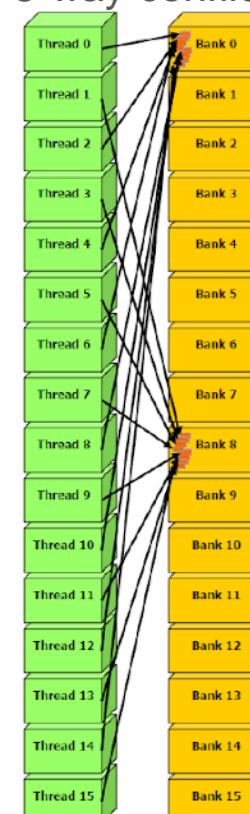
No bank conflict



2-way conflict



8-way conflict



Memory Access Pattern - Shared Memory

Example: bitonic sorting network (parallel)

- ❖ Each array is sort by threads within a block (not to confuse with previous method)
- ❖ Each thread responsible for comparing and swapping 2 elements connected by vertical arrow.
- ❖ No duplicated access for same element at any instance (no bank conflict).

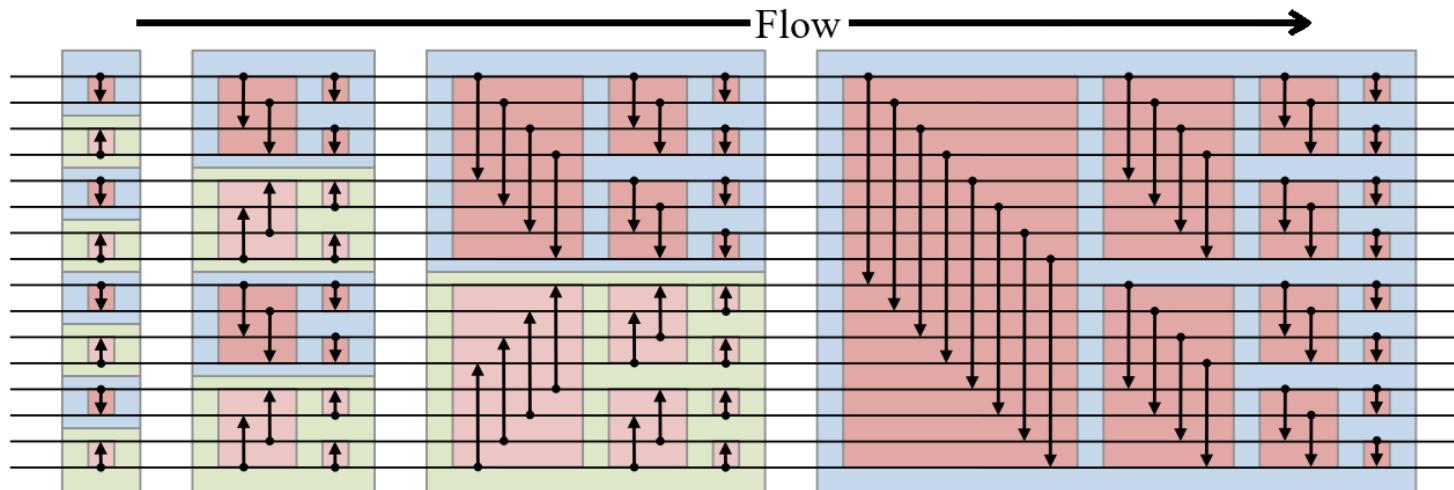


Image source: https://en.wikipedia.org/wiki/Bitonic_sorter

Memory Access Pattern - Shared Memory

```
__global__ void sortingNetworKernel(float *src_dst, int n_data_sort, int n_data_total, int start_idx, int sort_stride)
{// assume array size is less than 1024
const int n_round = ceil(log2f(n_data_sort));
const int t_idx1 = threadIdx.x;
const int t_idx2 = threadIdx.x + blockDim.x;
__shared__ float buf[4096];           // allocate share memory for array
buf[t_idx1] = src_dst[start_idx + t_idx1*sort_stride];
buf[t_idx2] = src_dst[start_idx + t_idx2*sort_stride];
__syncthreads();
int nbswap0 = 1;
for (int i = 0; i < n_round; i++)
{
    int nbswap = nbswap0;
    for (int j = i; j >= 0; j--)
    {
        int n_set = n_virtual / nbswap / 2;
        int k = threadIdx.x / nbswap;
        int m = threadIdx.x % nbswap;
        int loc1 = k*nbswap * 2 + m;
        int loc2 = loc1 + nbswap;
        int reverse = (loc1 / nbswap0 / 2) % 2;
        bitonicSwap(buf[loc1], buf[loc2], reverse);
        nbswap /= 2;
        __syncthreads();           // sync thread during each flow
    }
    nbswap0 *= 2;
}                                     // copy data back to global memory
src_dst[start_idx + t_idx1*sort_stride] = buf[t_idx1];
src_dst[start_idx + t_idx2*sort_stride] = buf[t_idx2];
return;
}
```

Tasks	Durations (seconds)	
	Global memory	Shared memory
Sort along X	1.26	1.12
Sort along Y	4.09	1.00
Sort along Z	7.09	0.99

Memory Access Pattern - Texture

- ❖ When dealing with multi-dimensional data (2D, 3D) and memory coalescing can't be achieved, **texture can be used for faster caching.**
- ❖ Textures are the mechanism by which data in global memory is retrieved through those computational units inside kernels running on the shader hardware.
- ❖ Advantage of texture:
 - Fast caching
 - Hardware level interpolation/filtering
- ❖ Disadvantages:
 - Read only

Memory Access Pattern - Texture

- ❖ Usage:
 - When data need to be accessed very often.
- ❖ Steps for using textures:
 1. Create a CUDA array
 2. Copy data to CUDA array
 3. Bind CUDA array to texture
 4. Access array data using texture API

Memory Access Pattern - Texture

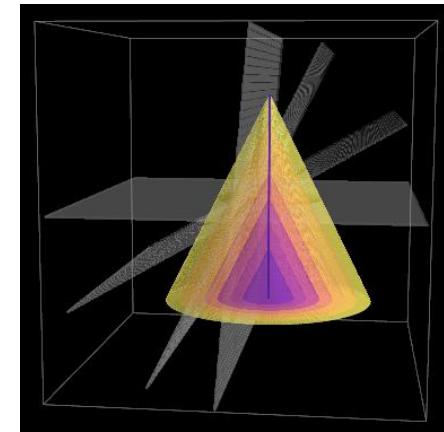
Example: slicing a 3D cone along various directions (conic section).

```
// cuda array creation
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
cudaArray *_d_cu_arr;
cudaStatus = cudaMalloc3DArray(&_d_cu_arr, &channelDesc, make_cudaExtent(src_nx * sizeof(float), src_ny, src_nz), 0);
// copy array
cudaMemcpy3DParms copy_params = { 0 };
copy_params.srcPtr = make_cudaPitchedPtr(_d_src, src_nx * sizeof(float), src_ny, src_nz);
copy_params.dstArray = _d_cu_arr;
copy_params.extent = make_cudaExtent(src_nx * 1, src_ny, src_nz);
copy_params.kind = cudaMemcpyDeviceToDevice;
cudaStatus = cudaMemcpy3D(&copy_params);
```

Create CUDA array and copy data

```
// bind to texture
cudaTextureObject_t _tex_src;
cudaResourceDesc tex_res;
memset(&tex_res, 0, sizeof(cudaResourceDesc));
tex_res.resType = cudaResourceTypeArray;
tex_res.res.array.array = _d_cu_arr;
cudaTextureDesc tex_desc;
memset(&tex_desc, 0, sizeof(cudaTextureDesc));
tex_desc.normalizedCoords = false;
tex_desc.filterMode = cudaFilterModeLinear;
tex_desc.addressMode[0] = cudaAddressModeClamp; // clamp
tex_desc.addressMode[1] = cudaAddressModeClamp;
tex_desc.addressMode[2] = cudaAddressModeClamp;
tex_desc.readMode = cudaReadModeElementType;
cudaStatus = cudaCreateTextureObject(&_tex_src, &tex_res, &tex_desc, NULL);
```

Create texture object and bind to CUDA array



Memory Access Pattern - Texture

Example: slicing a 3D cone along various directions (conic section).

```
int block_dim = dst_nx;
dim3 grid_dim(dst_ny, n_angle);

slicingWithTextureKernel << <grid_dim, block_dim >> >(_d_dst, dst_nx, dst_ny,
    _tex_src, src_nx, src_ny, src_nz, x0, y0, z0, _d_angle);
```

Pass texture object as parameter to kernel function

```
__global__ void slicingWithTextureKernel(float *dst, int dst_nx, int dst_ny,
    cudaTextureObject_t tex_src, int src_nx, int src_ny, int src_nz,
    float x0, float y0, float z0, float *angle_s)
{
    float factor = 1.0;
    int i = threadIdx.x;
    int j = blockIdx.x;
    int k = blockIdx.y;
    int loc_dst = i + j*dst_nx + k*dst_nx*dst_ny;
    float interp_y = y0 + j / factor - (float)(dst_ny) / 2.0 / factor;
    float src_x0 = i - (float)(dst_nx) / 2.0;
    float src_z0 = 0;
    float interp_x = (src_x0)*std::cos(angle_s[k]) / factor - (src_z0)*std::sin(angle_s[k]) / factor + x0;
    float interp_z = (src_x0)*std::sin(angle_s[k]) / factor + (src_z0)*std::cos(angle_s[k]) / factor + z0;
    dst[loc_dst] = tex3D<float>(tex_src, interp_x, interp_y, interp_z);
}
```

Accessing texture data using texture API

Duration reduced from 1.8s to 1.3 using texture.

Memory Access Pattern - Texture

Texture description

Normalize coordinate
to 0.0 and 1.0

```
tex3D<float>(tex_src, interp_x, interp_y, interp_z);
```

```
tex_desc.normalizedCoords = false;  
tex_desc.filterMode = cudaFilterModeLinear;  
tex_desc.addressMode[0] = cudaAddressModeClamp;  
tex_desc.addressMode[1] = cudaAddressModeClamp;  
tex_desc.addressMode[2] = cudaAddressModeClamp;
```

Interpolation mode
outside of region

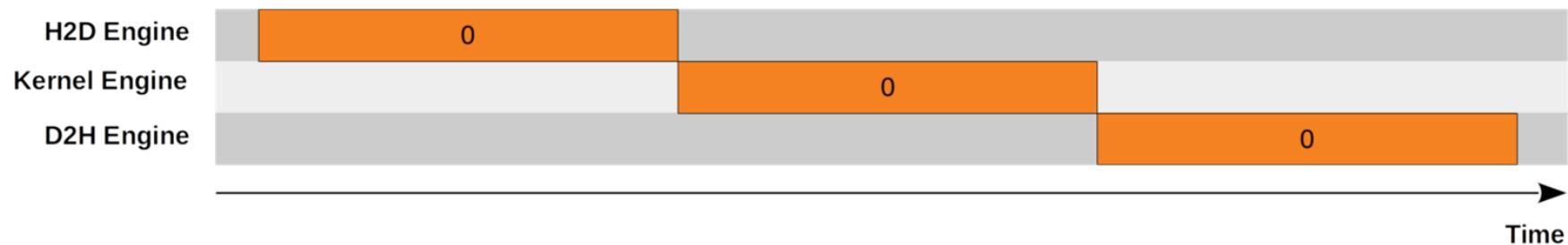
cudaAddressModeClamp	cudaAddressModeBorder	cudaAddressModeWrap	cudaAddressModeMirror
Set to nearest value	Set to 0	Periodic	Mirror
AAA ABCD DDD	000 ABCD 000	BCD ABCD ABC	DCB ABCD CBA

CUDA Stream

- ❖ Simplest CUDA program contains:
 1. Copy data from host to device (H2D)
 2. Kernel launch
 3. Copy data from device to host (D2H)
- ❖ CUDA program can be executed in serial mode or concurrent mode
- ❖ Serial modal:
 - copy data and kernel launch run in serial
 - Each operation starts only when the previous is completed.
- ❖ Concurrent mode:
 - A single task is divided into N smaller portion (aka stream).
 - Each portion perform copy data and kernel launch concurrently.

CUDA Stream

Serial Model



Concurrent Model

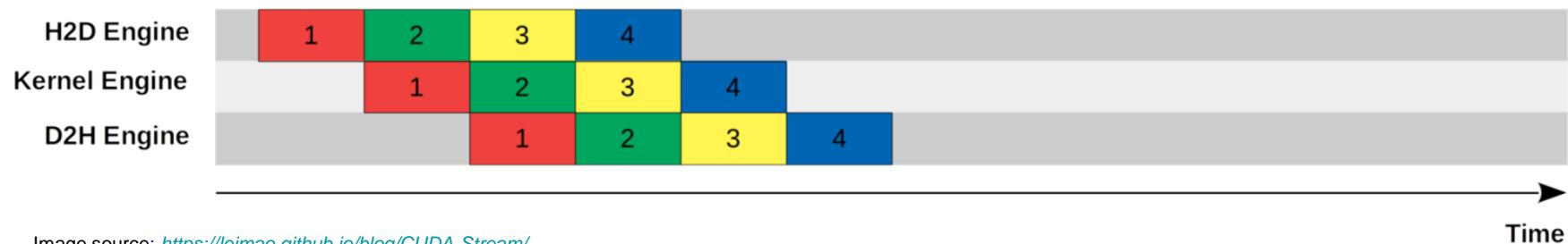


Image source: <https://leimao.github.io/blog/CUDA-Stream/>

CUDA Stream

```
// create stream0
cudaStream_t stream0;           Create stream
cudaStreamCreate(&stream0);

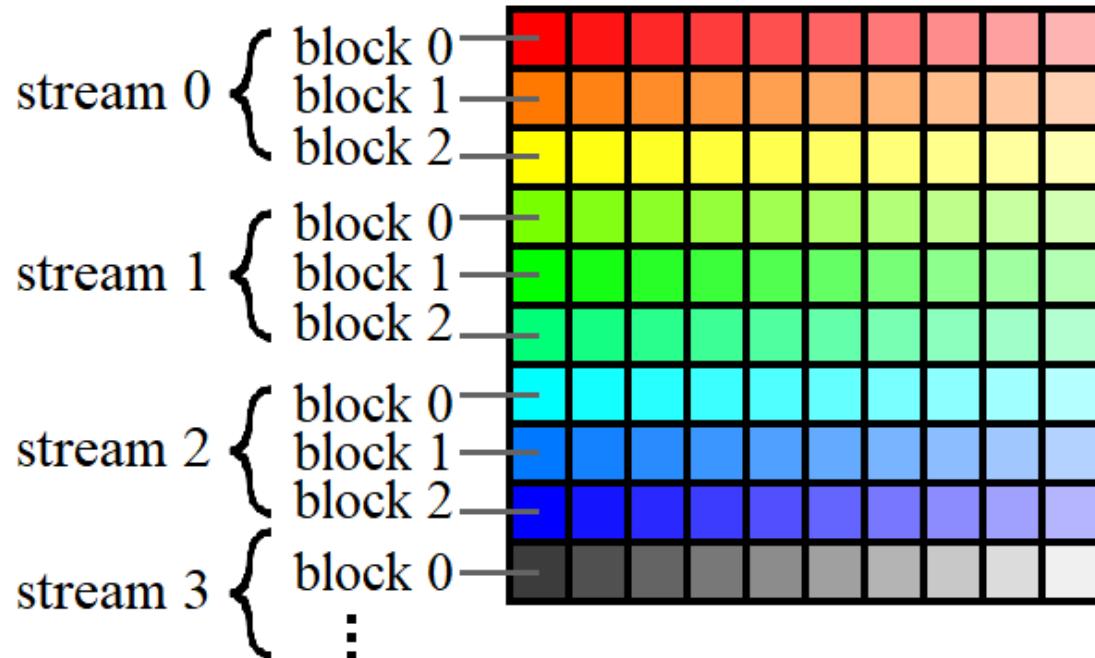
// copy data async
cudaMemcpyAsync(dst, src, size, cudaMemcpyHostToDevice, stream0)
    Use cudaMemcpyAsyn and specify the stream it belongs

// perform calculation
funcKernel << <grid_dim, block_dim, 0, stream0 >> >(param0, param1);
    3rd parameter in the arrow bracket specify
    maximum allowed dynamic memory of a kernel
    Kernel launch and specify the
    stream it belongs at 4th parameter

// copy results async
cudaMemcpyAsync(dst, src, size, cudaMemcpyDeviceToHost, stream0);
```

CUDA Stream

Example: sorting 2D array along X-direction



- ❖ Without stream
 - Each block handle a row
 - A total of N_y blocks will launch
 - Duration 0.21s
- ❖ With 4 streams
 - Each stream handle $N_y/4$ blocks
 - Each block handle a row
 - Duration 0.13s

CUDA Stream

Example: sorting 2D array along X-direction

```
// declare stream
const int n_stream = 4;
cudaStream_t stream_s[n_stream];

for (int i = 0; i < n_stream; i++)
{
    // create stream
    cudaStreamCreate(&stream_s[i]);
    // copy data async
    cudaMemcpyAsync(&_d_src_dst[n_data_stream*i], &src_dst[n_data_stream*i],
                   n_data_stream * sizeof(float), cudaMemcpyHostToDevice, stream_s[i]);
    // perform calculation
    sortKernel << <grid_dim, block_dim, 0, stream_s[i] >>
    (&_d_src_dst[n_data_per_stream*i], n_data_sort, n_data_stream,
     start_idx_stride1, start_idx_stride2, sort_stride);
    // copy results async
    cudaMemcpyAsync(&src_dst[n_data_stream*i], &_d_src_dst[n_data_stream*i],
                   n_data_stream * sizeof(float), cudaMemcpyDeviceToHost, stream_s[i]);
}
```

References & Useful Links

- ❖ <https://www.cs.utexas.edu/~rossbach/cs380p/papers/cuda-programming.pdf>
- ❖ <http://harmanani.github.io/classes/csc447/Notes/Lecture15.pdf>
- ❖ <https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf>
- ❖ <https://zhuanlan.zhihu.com/p/360727546>
- ❖ <http://thebeardsage.com/cuda-memory-hierarchy/>
- ❖ http://shodor.org/media/content/petascale/materials/GPGPU/presentations/Optimizing_CUDA_pdf
- ❖ <http://cuda-programming.blogspot.com/2013/02/bank-conflicts-in-shared-memory-in-cuda.html>
- ❖ <https://leimao.github.io/blog/CUDA-Stream/>
- ❖ <https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf>