CUDA programming Notes

Reference: ***Learn CUDA Programming A beginner Guide to GPU Programming and Parallel Computing***
https://www.packtpub.com/product/learn-cuda-programming/9781788996242

Compiler Setting

nvcc

-run *# This argument will run the program after compile*

-m64 *# Set the program to x64*

-I/usr/local/cuda/samples/common/inc *# include the cuda sample tools*

-gencode arch=compute_75,code=sm_75 *# 7.5 for RTX 20 series 5.6 for Jetson Nano Tegra System*

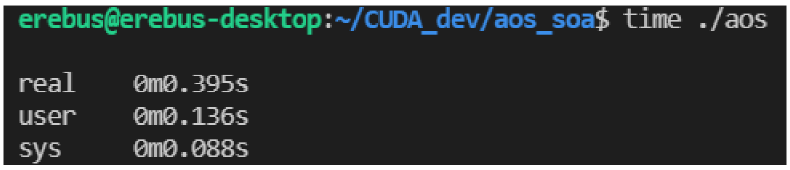
-o executive_name ./filename.cu

-Xcompiler -fopenmp # To compile with OpenMP in the host code

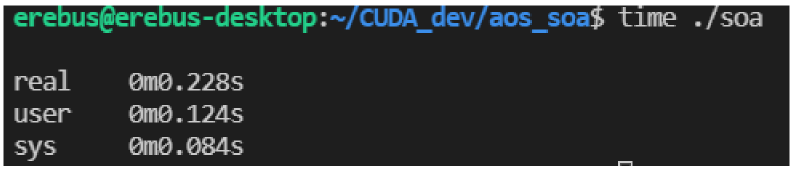
Memory Management

Coalesced Access: Sequential memory access is adjacent

Uncoalesced Access: Sequential memory access is not adjacent



```
erebus@erebus-desktop:~/CUDA_dev/aos_soa$ time ./aos

real    0m0.395s
user    0m0.136s
sys     0m0.088s
```

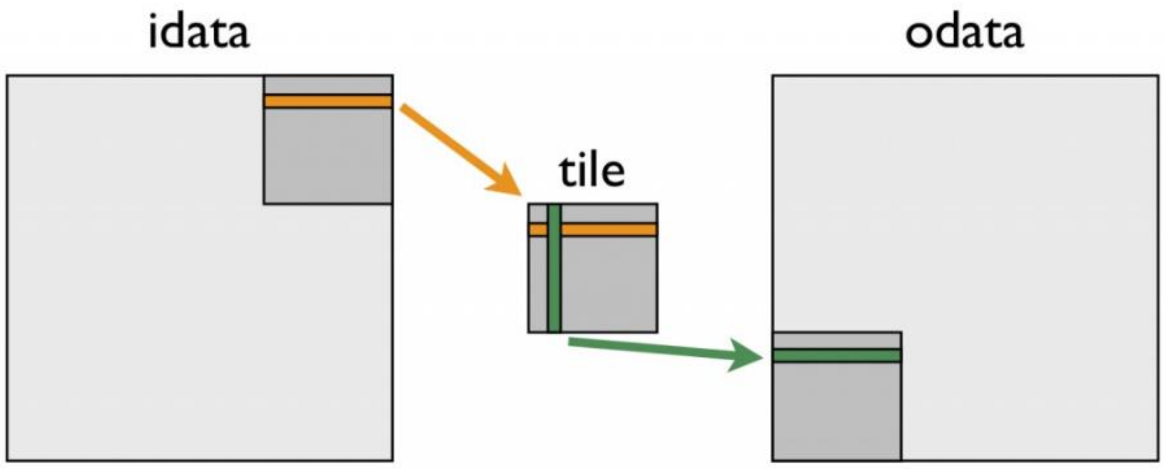


```
erebus@erebus-desktop:~/CUDA_dev/aos_soa$ time ./soa

real    0m0.228s
user    0m0.124s
sys     0m0.084s
```

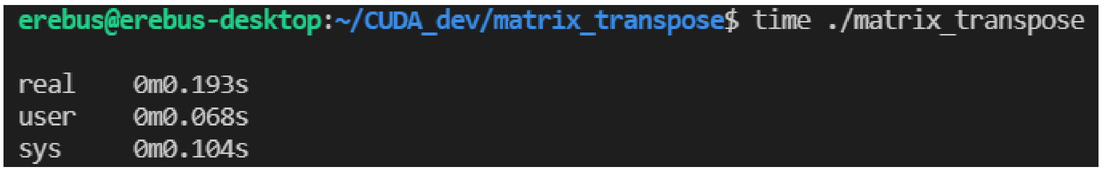
Share Memory

Need to store data into the share memory in coalesced fashion.

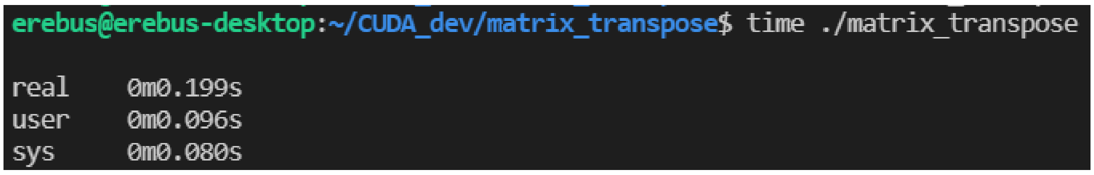
Use to hold data that will be reused many times during the execution of the kernel function.



The following kernel performs this "tiled" transpose.



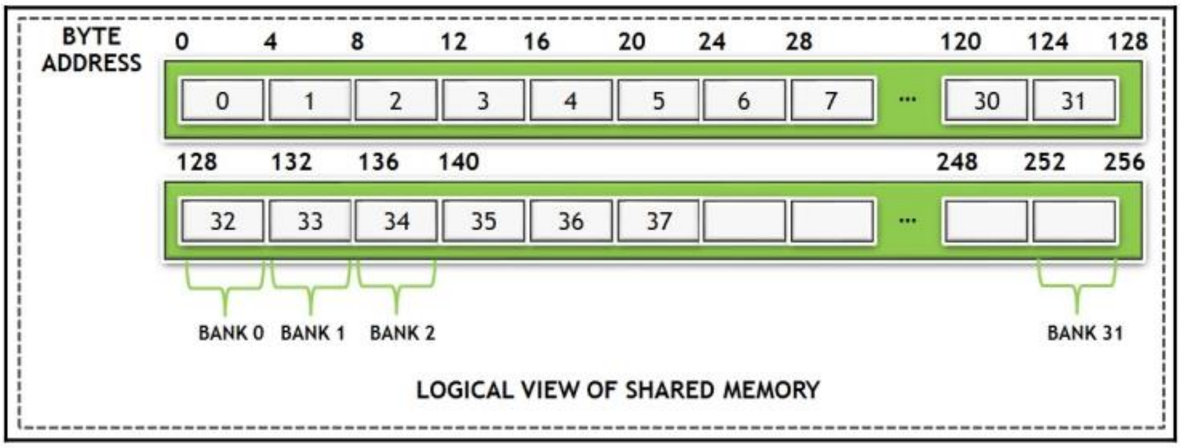
```
erebus@erebus-desktop:~/CUDA_dev/matrix_transpose$ time ./matrix_transpose

real	0m0.193s
user	0m0.068s
sys	0m0.104s
```



```
erebus@erebus-desktop:~/CUDA_dev/matrix_transpose$ time ./matrix_transpose

real	0m0.199s
user	0m0.096s
sys	0m0.080s
```

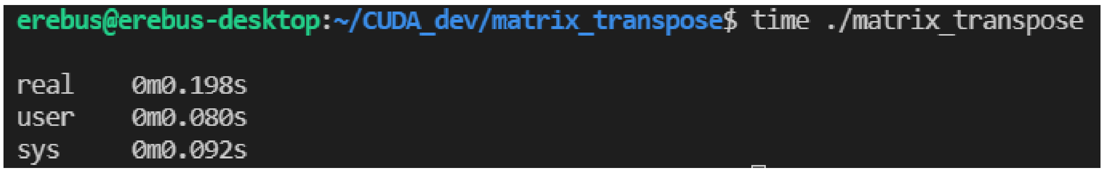
Share memory bank conflicts

Share memory is organized into banks. Accessing same bank with different thread will cause bank conflict and since one bank can only service one address per cycle, it will add a cycle of latency.



In the matrix transpose program, we have 32-way bank conflict. Pad the shared memory with a dummy can solve this problem.

```
    __shared__ int sharedMemory [BLOCK_SIZE] [BLOCK_SIZE + 1];
```



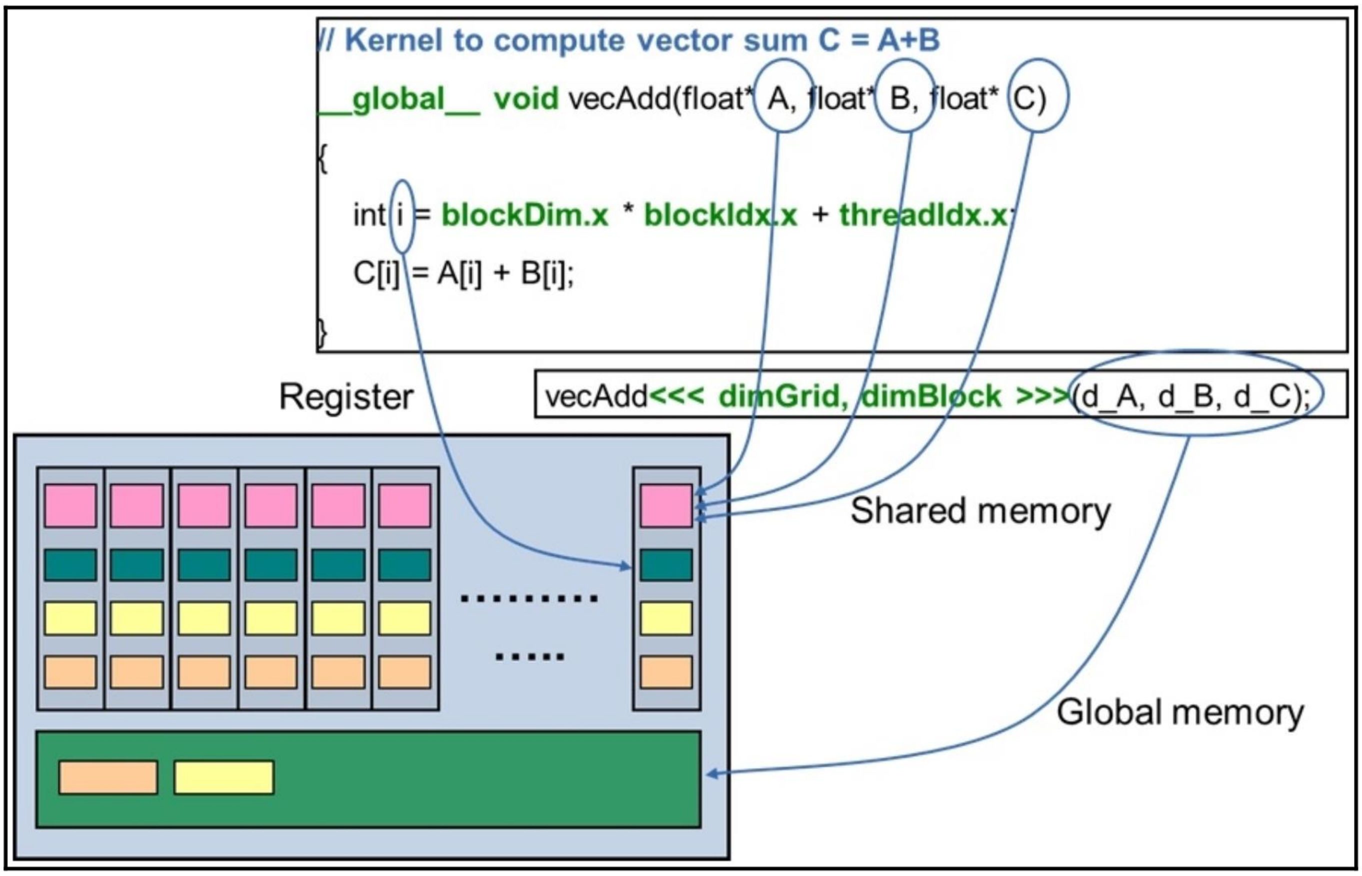
```
erebus@erebus-desktop:~/CUDA_dev/matrix_transpose$ time ./matrix_transpose

real	0m0.198s
user	0m0.080s
sys	0m0.092s
```

Read-only data/cache (texture cache)

Texture memory is a read-only memory which is used ideally when algorithm demands the entire warp to read the same address/data.

Registers in GPU

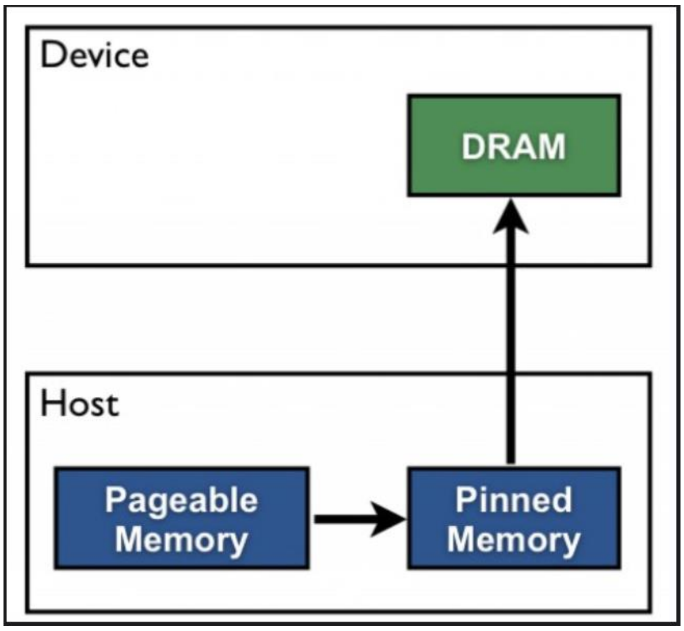
Every SM has fixed set of registers. Compiler will try to find the best number of registers per thread. When the CUDA kernel is large and has a lot of local variables and intermediate calculations (register spills), the data gets pushed to local memory, which may reside either in an L1/L2 cache or even in the global memory. The number of registers per thread plays an important role in how many blocks and threads can be active on an SM.


```
// Kernel to compute vector sum C = A+B
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```
```
vecAdd<<< dimGrid, dimBlock >>>(d_A, d_B, d_C);
```

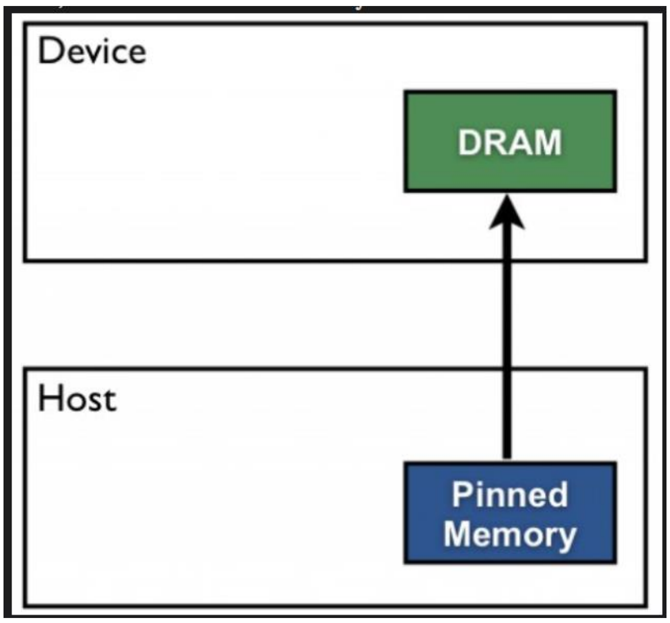
Pinned memory

By default, the GPU will not access the pageable memory. Hence, when a transfer of memory is invoked, the CUDA driver allocates the temporary pinned memory, copies the data from the default pageable memory to this temporary pinned memory, and then transfer it to the device.

Reference: https://kaibaoom.tw/2020/07/21/cuda-four-memory-access/



By using CUDA API cudaMallocHost(), the data that will be transfer to device will reside in Host Pinned Memory. However, Page-Lock memory will not be written into Swap and take a huge tow on memory consumption.



In the CUDA sample, we can find a bandwidthTest program in the Utilities section.

Unified Memory

From Pascal cars onward, cudaMallocManaged() does not allocate physical memory but allocate memory based on a first-touch basis. If the GPU first touches the variable, the page will be allocated and mapped in the GPU page table; otherwise, if the CPU first touches the variable, it will be allocated and mapped to the CPU.

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

    // initialize x and y arrays on the host
    for(int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    // Launch kernel on 1M elements on the GPU
    int blockSize = 256;
    int numBlocks = (N + blockSize - 1) / blockSize;
    add<<<numBlocks, blockSize>>>(N, x, y);
    // Synchronize with host
    cudaDeviceSynchronize();
    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for(int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i] - 3.0f));
    std::cout << "Max error " << maxError << std::endl;
    cudaFree(x);
    cudaFree(y);
```

In the code above the CPU touches the data first, hence there will be a page fault which occurs and the time of page migration gets added to the kernel time.

1) Optimize by create an initialization kernel on the GPU so that there is no page fault during the kernel run.
2) Prefetch the data.

Data prefetching are basically hints to the drive to prefetch the data that we believe will be used in the device prior to its use.

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

    // initialize x and y arrays on the host
    for(int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }
    cudaGetDevice(&device);
    // GPU (device) prefetches unified memory
    cudaMemPrefetchAsync(x, N * sizeof(float), device, NULL);
    cudaMemPrefetchAsync(y, N * sizeof(float), device, NULL);
    // Launch kernel on 1M elements on the GPU
    int blockSize = 256;
    int numBlocks = (N + blockSize - 1) / blockSize;
    add<<<numBlocks, blockSize>>>(N, x, y);
    // CPU (host prefetches memory)
    cudaMemPrefetchAsync(y, N * sizeof(float), cudaCpuDeviceId, NULL);
    // Synchronize with host
    cudaDeviceSynchronize();
    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for(int i = 0; i < N; i++)
        maxError = fmax(maxError, fabs(y[i] - 3.0f));
    std::cout << "Max error " << maxError << std::endl;
    cudaFree(x);
    cudaFree(y);
```

We can also use CUDA API cudaMemAdvise() to hint where the data will actually reside and help with multiple processor that need to simultaneously access the same data.

Reference:

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1ge37112fc1ac88d0f6bab7a945e48760a

Occupancy tuning – bounding register usage

We can increase the theoretical occupancy by limiting the register usage. Use __launch_bound__ with the kernel function __launch_bound__ (maxThreadPerBlock, minBlocksPerMultiprocessor). Additionally, we can limit the number of occupied register usage at the application level. The –maxrregcount flag to NVCC will specify the number and the compiler will reorder the register usage.

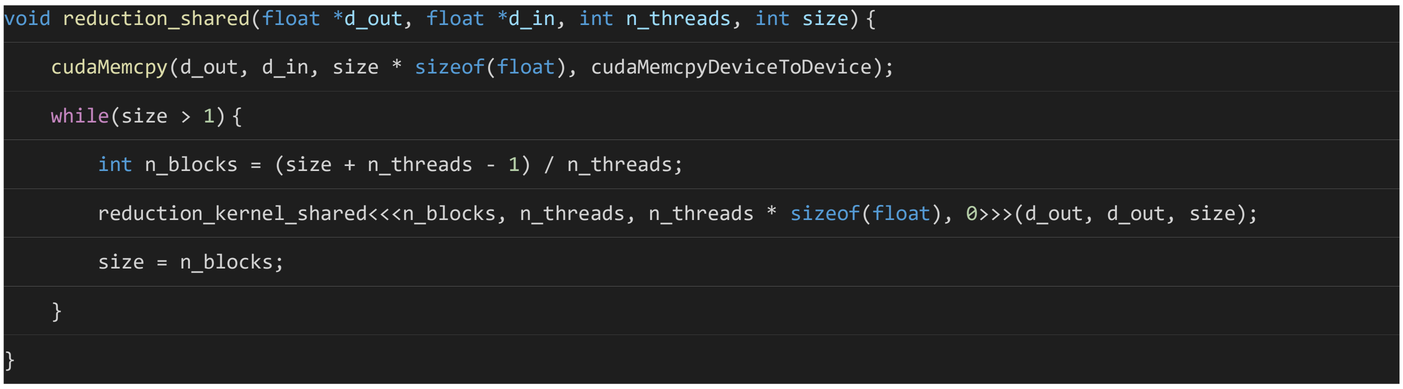
Parallel Reduction

- Using only global memory.

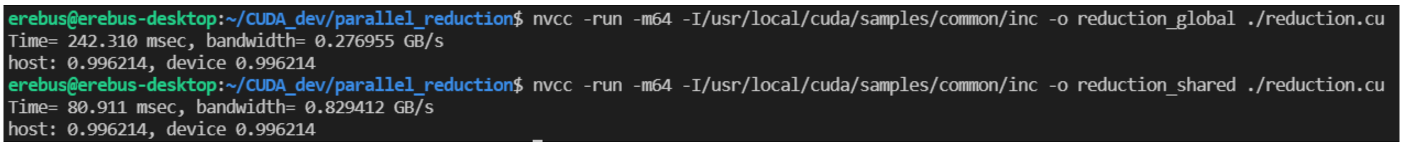
```
__global__
void global_reduction_kernel(float *data_out, float *data_in, int stride, int size){
    int idx_x = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx_x + stride < size)
        data_out[idx_x] += data_in[idx_x + stride];
}
void global_reduction(float *d_out, float *d_in, int n_threads, int size){
    int n_blocks = (size + n_threads - 1) / n_threads;
    for(int stride = 1; stride < size; stride *= 2)
        global_reduction_kernel<<<n_blocks, n_threads>>>(d_out, d_in, stride, size);
}
```

- Using shared memory.

```
__global__
void reduction_kernel_shared(float* d_out, float* d_in, unsigned int size) {
    unsigned int idx_x = blockIdx.x * blockDim.x + threadIdx.x;
    extern __shared__ float s_data[];
    s_data[threadIdx.x] = (idx_x < size) ? d_in[idx_x] : 0.f;
    __syncthreads();
    // do reduction thread synchronous reduction
    for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
        if ((idx_x & (stride * 2 - 1)) == 0)
            s_data[threadIdx.x] += s_data[threadIdx.x + stride];
        __syncthreads();
    }
    if (threadIdx.x == 0)
        d_out[blockIdx.x] = s_data[0];
}
```



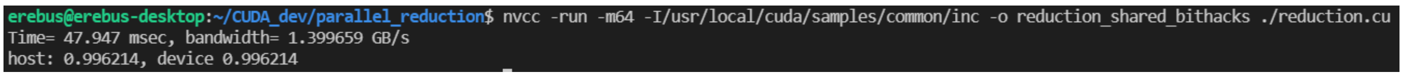
```
void reduction_shared(float *d_out, float *d_in, int n_threads, int size){
    cudaMemcpy(d_out, d_in, size * sizeof(float), cudaMemcpyDeviceToDevice);
    while(size > 1){
        int n_blocks = (size + n_threads - 1) / n_threads;
        reduction_kernel_shared<<<n_blocks, n_threads, n_threads * sizeof(float), 0>>>(d_out, d_out, size);
        size = n_blocks;
    }
}
```



```
erebus@erebus-desktop:~/CUDA_dev/parallel_reduction$ nvcc -run -m64 -I/usr/local/cuda/samples/common/inc -o reduction_global ./reduction.cu
Time= 242.310 msec, bandwidth= 0.276955 GB/s
host: 0.996214, device 0.996214
erebus@erebus-desktop:~/CUDA_dev/parallel_reduction$ nvcc -run -m64 -I/usr/local/cuda/samples/common/inc -o reduction_shared ./reduction.cu
Time= 80.911 msec, bandwidth= 0.829412 GB/s
host: 0.996214, device 0.996214
```

- Optimization – bit hacks

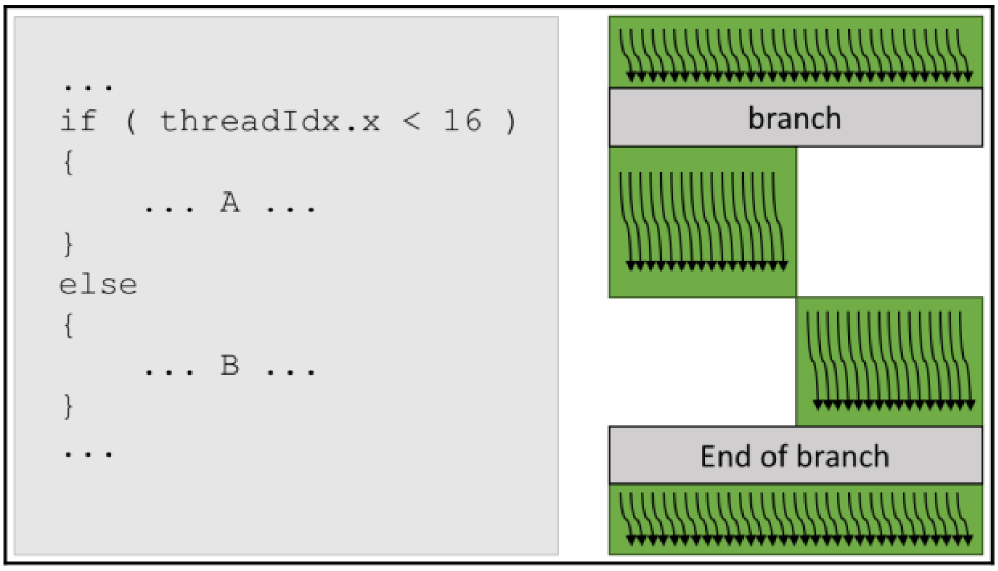
Since the stride is an exponential number of 2, we can change mod (%) operation to bitwise operation (&), which filter out all the bits except the first bit and thus left with 1 or 0.



```
erebus@erebus-desktop:~/CUDA_dev/parallel_reduction$ nvcc -run -m64 -I/usr/local/cuda/samples/common/inc -o reduction_shared_bithacks ./reduction.cu
Time= 47.947 msec, bandwidth= 1.399659 GB/s
host: 0.996214, device 0.996214
```

- Minimizing the CUDA warp divergence effect

If a warp encounters a conditional statement or branch, its threads can diverge and serialized to execute each condition. (Branch divergence)



```
...
if ( threadIdx.x < 16 )
{
    ... A ...
}
else
{
    ... B ...
}
...
```

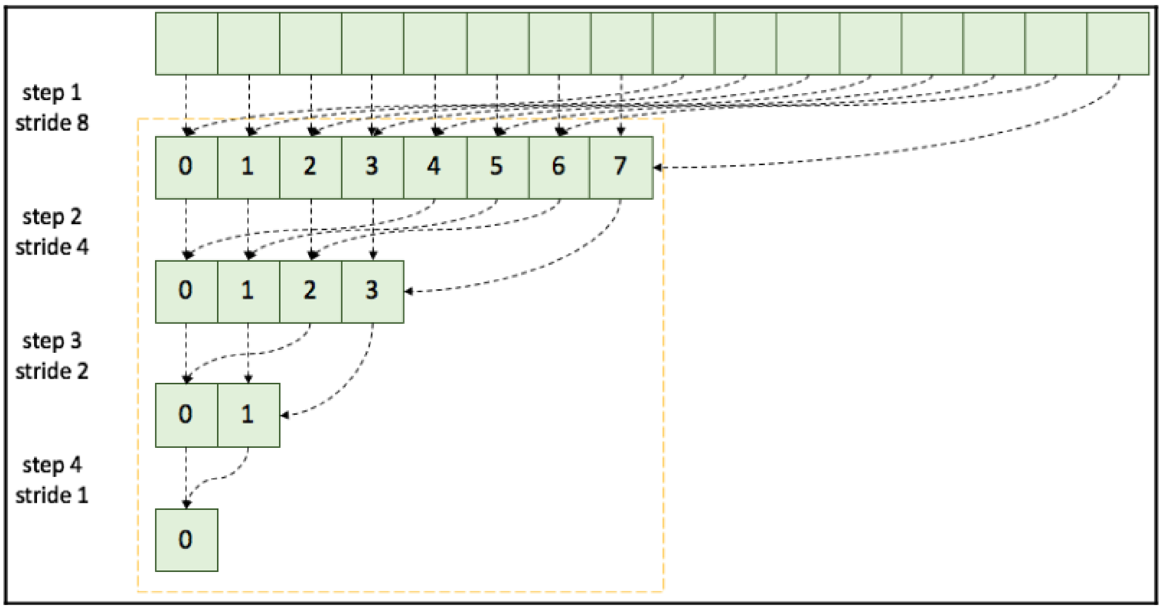
Solutions

Divergence avoidance by handling different warps to execute the branched part
Coalescing the branched part to reduce branches in a warp
Shortening the branched part; only critical parts to branched
Rearranging the data (transposing, coalescing, and so on)
Partitioning the group using tiled_partition in Cooperative Group

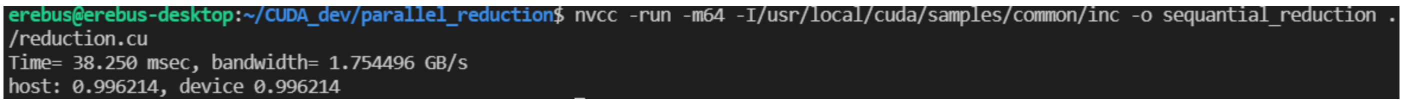
Reduction addressing, we can select one of these CUDA thread indexing strategies.

- Interleaved addressing
- Sequential addressing

In our reduction, sequential addressing can have better performance.



This design is more efficient because there is no divergence when the stride size is greater than the warp size.

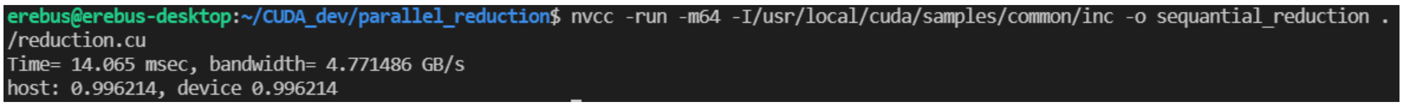


```
erebus@erebus-desktop:~/CUDA_dev/parallel_reduction$ nvcc -run -m64 -I/usr/local/cuda/samples/common/inc -o sequantial_reduction .
/reduction.cu
Time= 38.250 msec, bandwidth= 1.754496 GB/s
host: 0.996214, device 0.996214
```

- Maximizing memory bandwidth with grid-stride loops

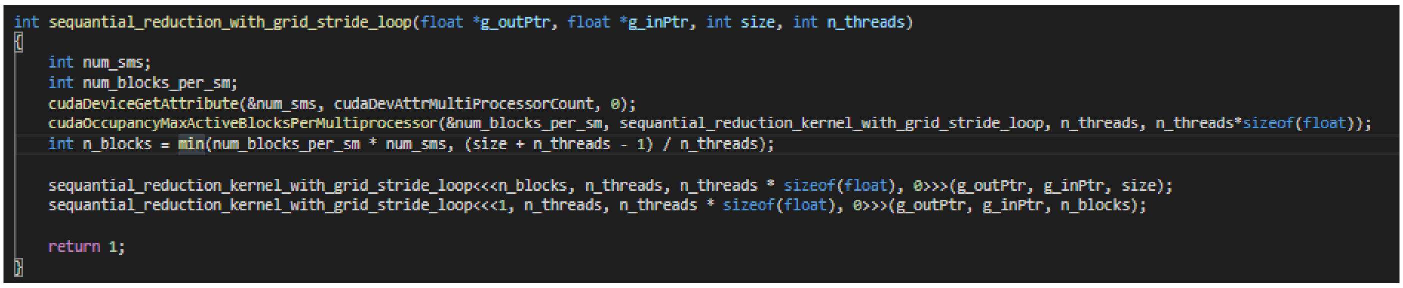
Reference: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

Rather than assume that the thread grid is large enough to cover the entire data array, this kernel loops over the data array one grid-size at a time.



```
erebus@erebus-desktop:~/CUDA_dev/parallel_reduction$ nvcc -run -m64 -I/usr/local/cuda/samples/common/inc -o sequantial_reduction .
/reduction.cu
Time= 14.065 msec, bandwidth= 4.771486 GB/s
host: 0.996214, device 0.996214
```

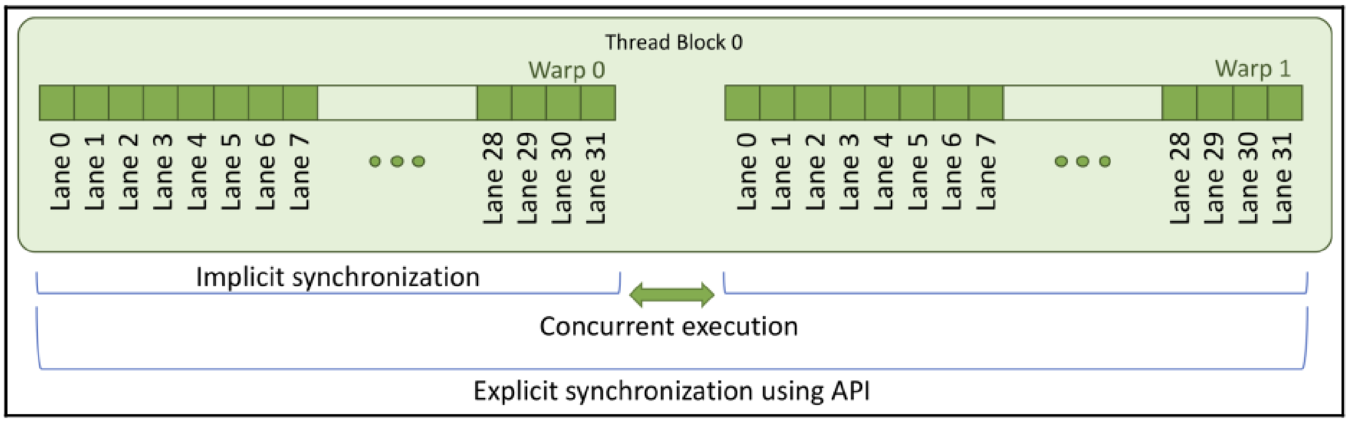
We can obtain the occupancy-aware maximum active blocks per multiprocessor using the cudaOccupancyMaxActiveBlocksPerMultiprocessor() and cudaDeviceMaxActiveBlocksPerMultiprocessor() to get the number of multiprocessors on the target GPU.



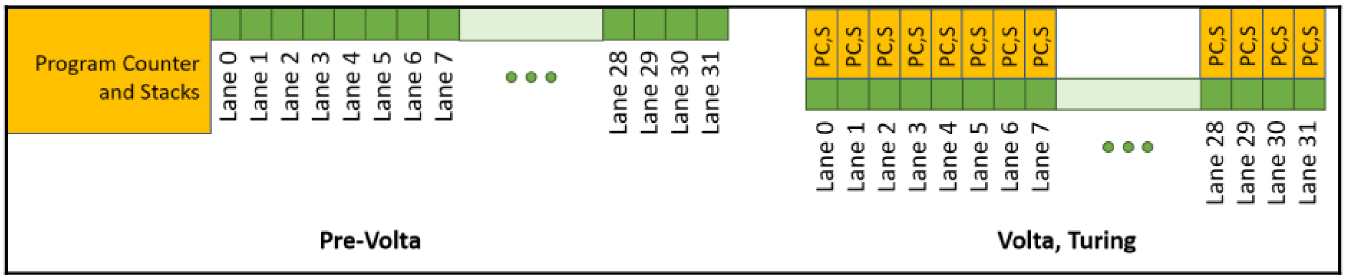
```
int sequantial_reduction_with_grid_stride_loop(float *g_outPtr, float *g_inPtr, int size, int n_threads)
{
    int num_sms;
    int num_blocks_per_sm;
    cudaDeviceGetAttribute(&num_sms, cudaDevAttrMultiProcessorCount, 0);
    cudaOccupancyMaxActiveBlocksPerMultiprocessor(&num_blocks_per_sm, sequantial_reduction_kernel_with_grid_stride_loop, n_threads, n_threads*sizeof(float));
    int n_blocks = min(num_blocks_per_sm * num_sms, (size + n_threads - 1) / n_threads);

    sequantial_reduction_kernel_with_grid_stride_loop<<<n_blocks, n_threads, n_threads * sizeof(float), 0>>>(g_outPtr, g_inPtr, size);
    sequantial_reduction_kernel_with_grid_stride_loop<<<1, n_threads, n_threads * sizeof(float), 0>>>(g_outPtr, g_inPtr, n_blocks);

    return 1;
}
```

Warp-level primitive programming

Historically, CUDA provide only one explicit synchronization API __syncthreads().

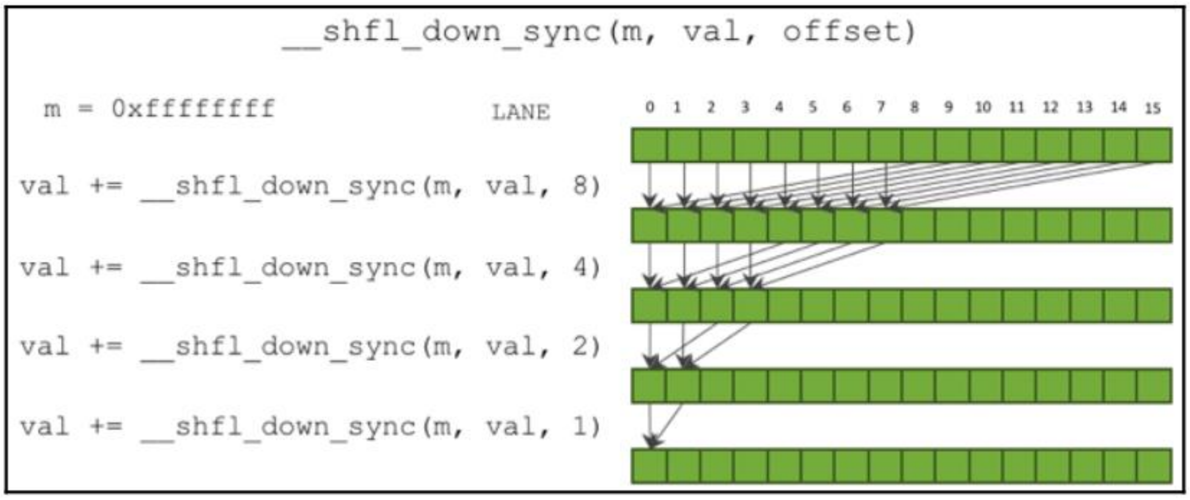


However, Volta and Turing have an enhanced thread control model, where each thread can execute a different instruction, while they keep its SIMT programming model. The independent thread scheduling enables each CUDA thread to have its program counter and allows sets of participating threads in a warp.

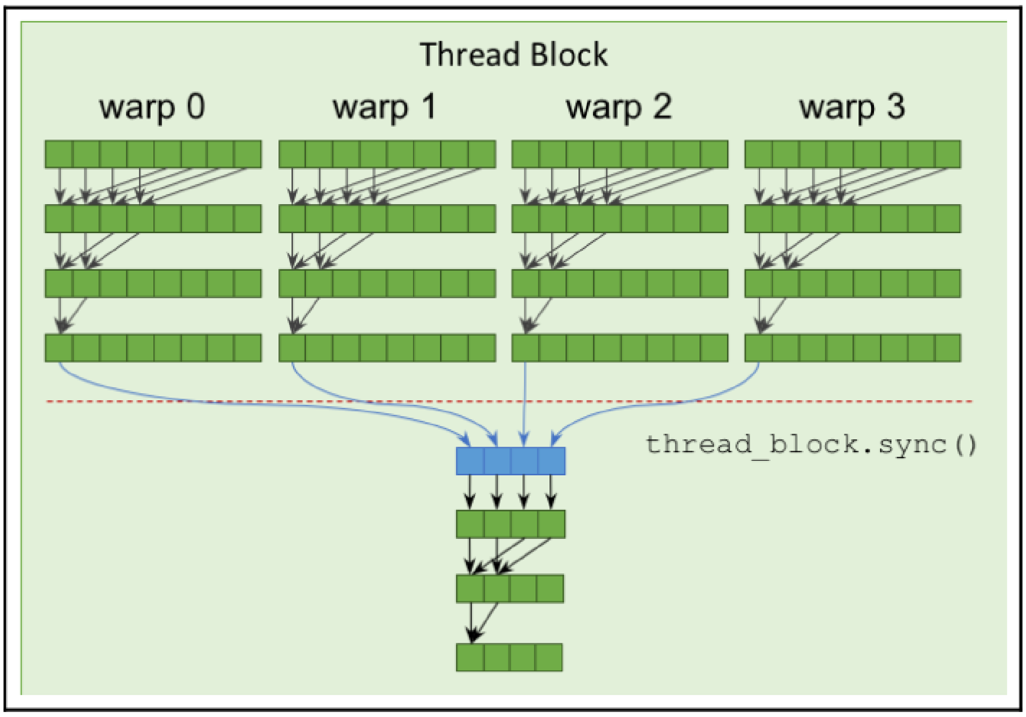


	Warp-level primitive functions
Identifying active threads	`__activemask()`
Masking active threads	`__all_sync()`, `__any_sync()`, `__uni_sync()`, `__ballot_sync()` `__match_any_sync()`, `__match_all_sync()`
Synchronized data exchange	`__shfl_sync()`, `__shfl_up_sync()`, `__shfl_down_sync()`, `__shfl_xor_sync()`
Threads synchronization	`__syncwarp()`

Parallel reduction with warp primitives

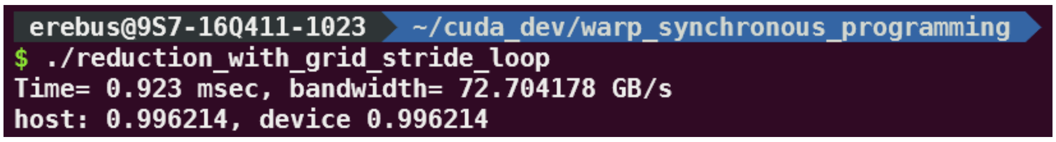


```
__shfl_down_sync(m, val, offset)

m = 0xffffffff
val += __shfl_down_sync(m, val, 8)
val += __shfl_down_sync(m, val, 4)
val += __shfl_down_sync(m, val, 2)
val += __shfl_down_sync(m, val, 1)
```

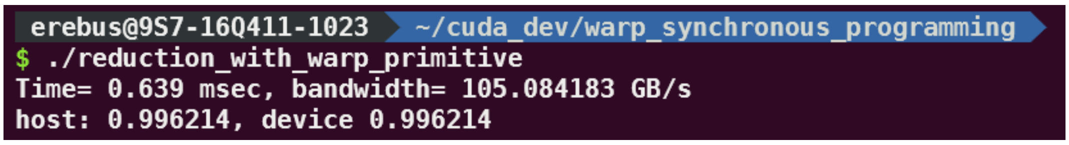


Each warp's reduction result is stored to shared memory to share with other warp. Then the final block-wise reduction can be obtained by doing warp-wise collection again.

Reference: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/



```
erebus@9S7-16Q411-1023  ~/cuda_dev/warp_synchronous_programming
$ ./reduction_with_grid_stride_loop
Time= 0.923 msec, bandwidth= 72.704178 GB/s
host: 0.996214, device 0.996214
```



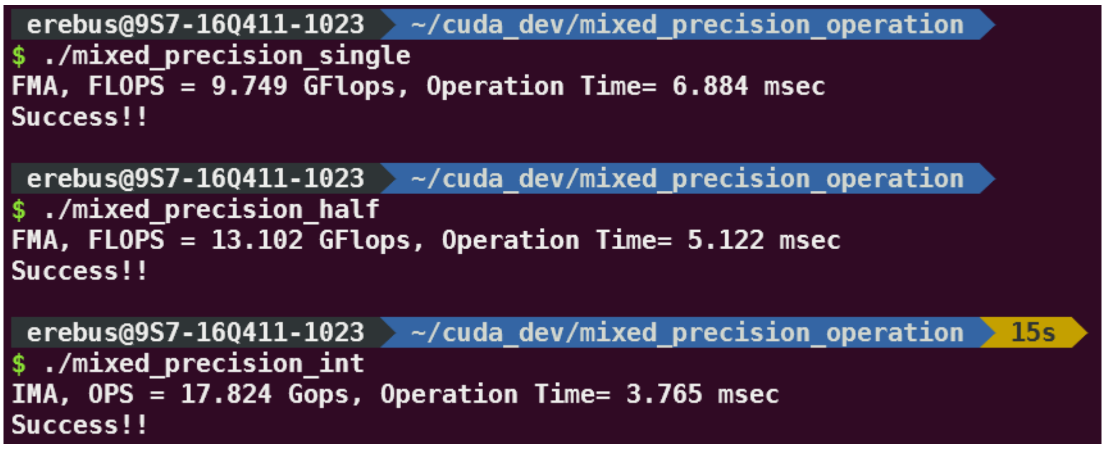
```
erebus@9S7-16Q411-1023  ~/cuda_dev/warp_synchronous_programming
$ ./reduction_with_warp_primitive
Time= 0.639 msec, bandwidth= 105.084183 GB/s
host: 0.996214, device 0.996214
```

Low/Mixed Precision Operations

Reference: https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/

Low precision operation computation has the benefit of reduced memory bandwidth and higher computing throughput compared with high-precision computing. GPU will use SIMD operation with some specific APIs.

These SIMD operations is similar to the C intrinsic.



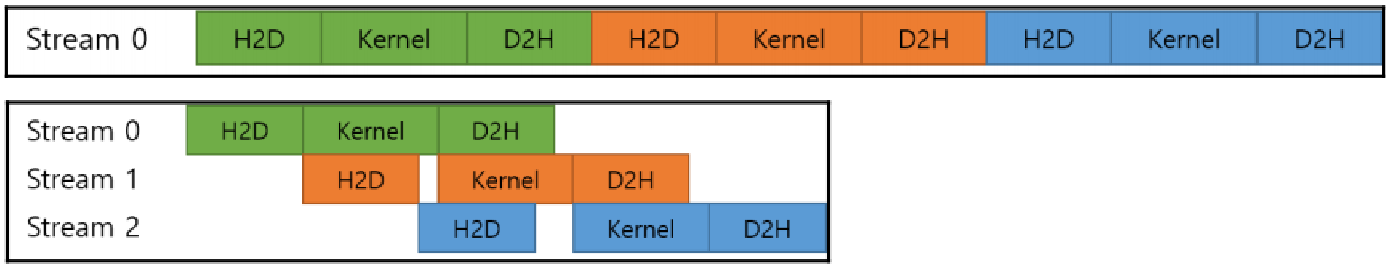
```
erebus@9S7-16Q411-1023  ~/cuda_dev/mixed_precision_operation
$ ./mixed_precision_single
FMA, FLOPS = 9.749 GFlops, Operation Time= 6.884 msec
Success!!

erebus@9S7-16Q411-1023  ~/cuda_dev/mixed_precision_operation
$ ./mixed_precision_half
FMA, FLOPS = 13.102 GFlops, Operation Time= 5.122 msec
Success!!

erebus@9S7-16Q411-1023  ~/cuda_dev/mixed_precision_operation  15s
$ ./mixed_precision_int
IMA, OPS = 17.824 Gops, Operation Time= 3.765 msec
Success!!
```

All the kernel calls and data transfers are handled by the CUDA stream. By default, it will be default stream. CUDA can execute multiple operations concurrently by using the multiple streams.

1. Kernel executions are asynchronous with the host.
2. CUDA operations in different streams are independent of each other.

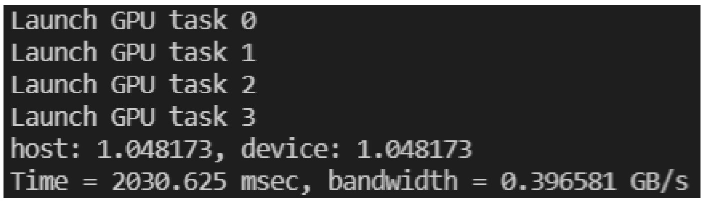
Concept of GPU pipelining

We can overlap the data transfer and kernel operation with multiple CUDA streams. This help CUDA to hide the data transfer delay.



To enable pipelining operation

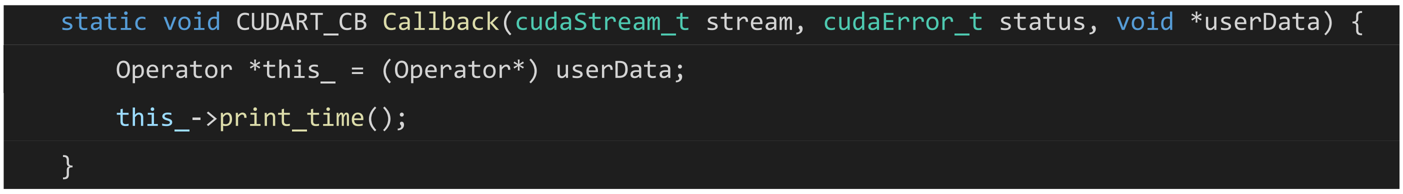
1. The host memory should be allocated as pinned memory (cudaMallocHost(), cudaFreeHost())
2. Transfer data between the host and GPUs without blocking the host (cudaMemcpyAsync())
3. Manage each operation along with the different CUDA streams to have concurrent operations



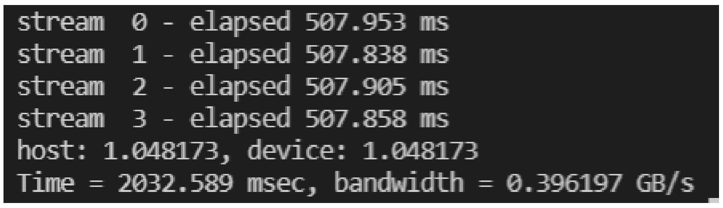
```
Launch GPU task 0
Launch GPU task 1
Launch GPU task 2
Launch GPU task 3
host: 1.048173, device: 1.048173
Time = 2030.625 msec, bandwidth = 0.396581 GB/s
```

The CUDA callback function

The CUDA callback function is callable host function to be executed by the GPU execution context.



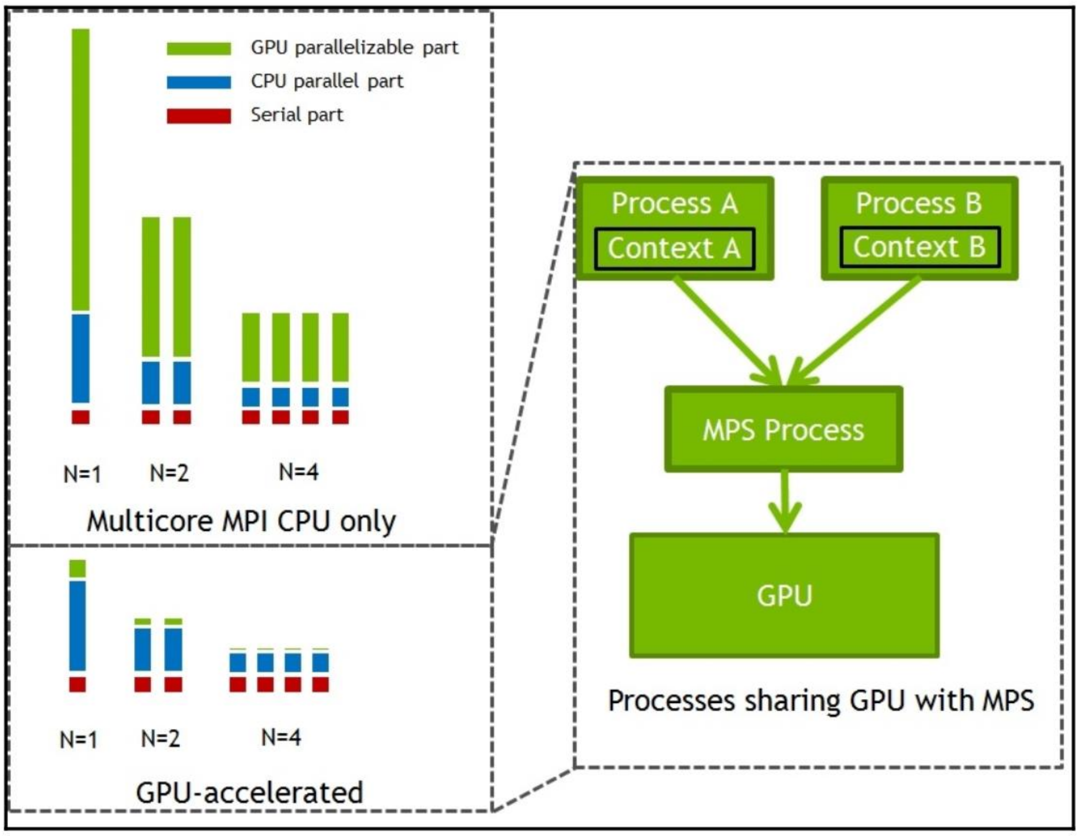
```
static void CUDART_CB Callback(cudaStream_t stream, cudaError_t status, void *userData) {
    Operator *this_ = (Operator*) userData;
    this_->print_time();
}
```



```
stream  0 - elapsed 507.953 ms
stream  1 - elapsed 507.838 ms
stream  2 - elapsed 507.905 ms
stream  3 - elapsed 507.858 ms
host: 1.048173, device: 1.048173
Time = 2032.589 msec, bandwidth = 0.396197 GB/s
```

Multi-Process Service

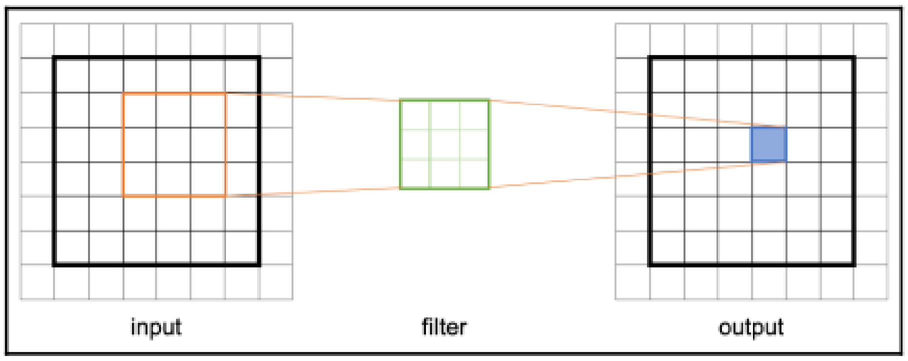
MPS enables different processes to execute their kernels simultaneously on a GPU to fully utilize GPU resources.



When one MPI process is unable to saturate the entire GPU and a significant part of the code is also running on the CPU.

Parallel Programming Patterns

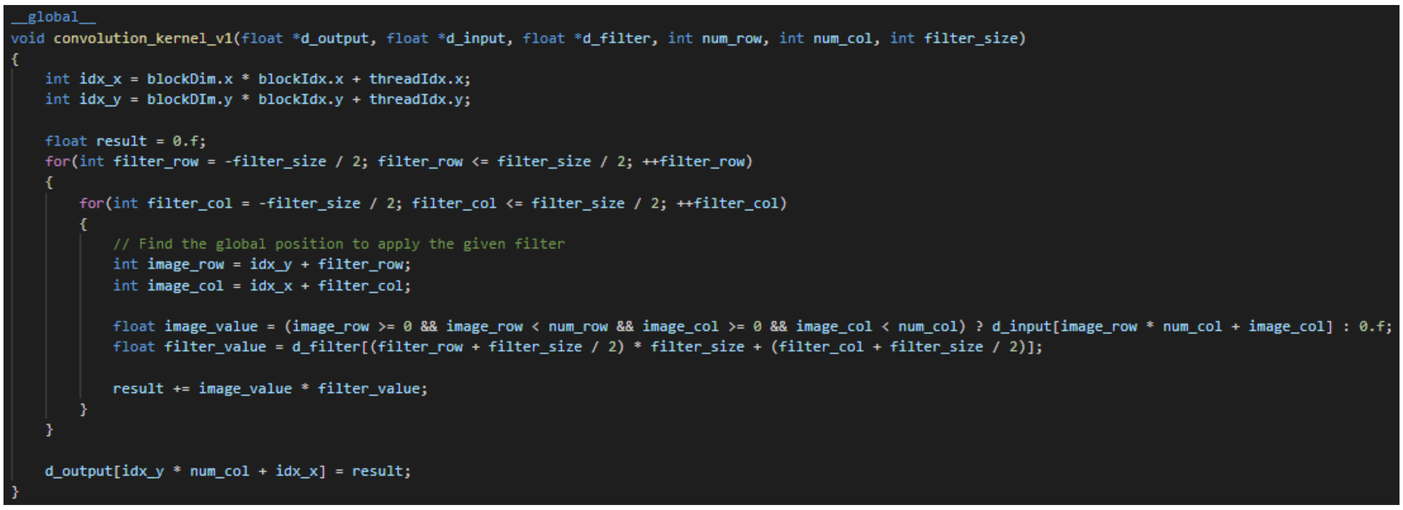
Convolution



The design of the kernel function

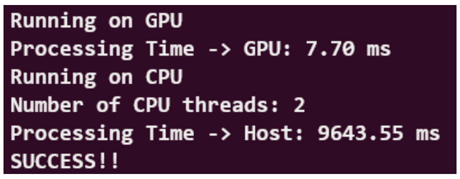
- Each CUDA thread generates one filtered output
- Each CUDA thread applies the filter's coefficients to the data
- The filter shape is a box filter

Naïve Kernel



```
__global__
void convolution_kernel_v1(float *d_output, float *d_input, float *d_filter, int num_row, int num_col, int filter_size)
{
    int idx_x = blockDim.x * blockIdx.x + threadIdx.x;
    int idx_y = blockDIm.y * blockIdx.y + threadIdx.y;

    float result = 0.f;
    for(int filter_row = -filter_size / 2; filter_row <= filter_size / 2; ++filter_row)
    {
        for(int filter_col = -filter_size / 2; filter_col <= filter_size / 2; ++filter_col)
        {
            // Find the global position to apply the given filter
            int image_row = idx_y + filter_row;
            int image_col = idx_x + filter_col;

            float image_value = (image_row >= 0 && image_row < num_row && image_col >= 0 && image_col < num_col) ? d_input[image_row * num_col + image_col] : 0.f;
            float filter_value = d_filter[(filter_row + filter_size / 2) * filter_size + (filter_col + filter_size / 2)];

            result += image_value * filter_value;
        }
    }

    d_output[idx_y * num_col + idx_x] = result;
}
```

The kernel function fetches input data and the filter for every operation and does not reuse all the data.

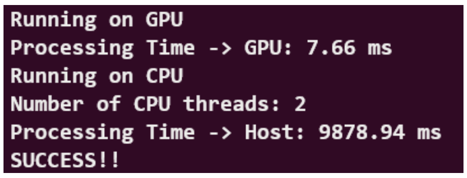


```
Running on GPU
Processing Time -> GPU: 7.70 ms
Running on CPU
Number of CPU threads: 2
Processing Time -> Host: 9643.55 ms
SUCCESS!!
```

Optimization Strategy

Using Constant Memory

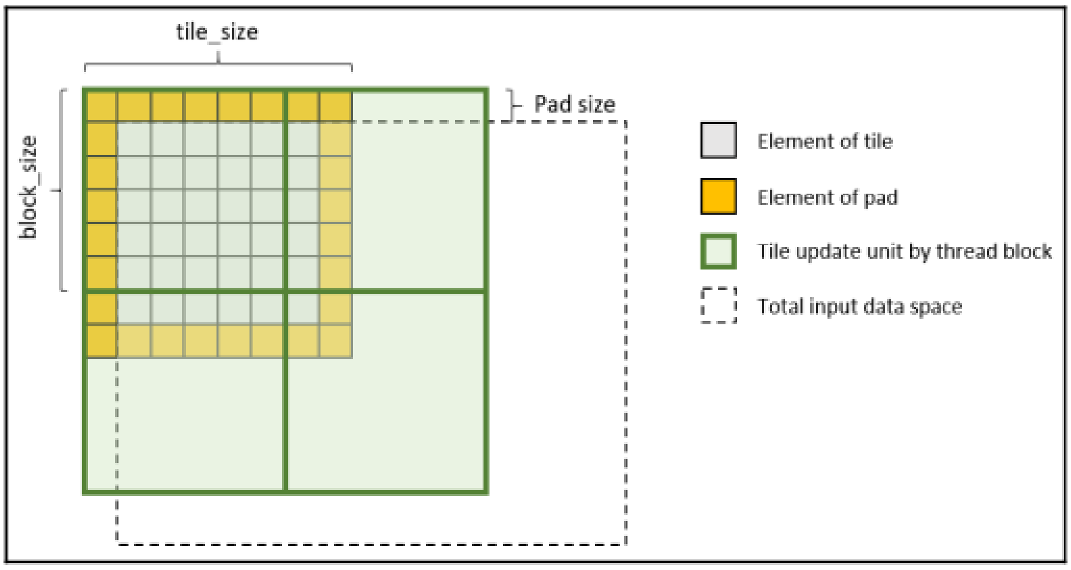
*__constant__ float c_filter[MAX_FILTER_LENGTH * MAX_FILTER_LENGTH];*

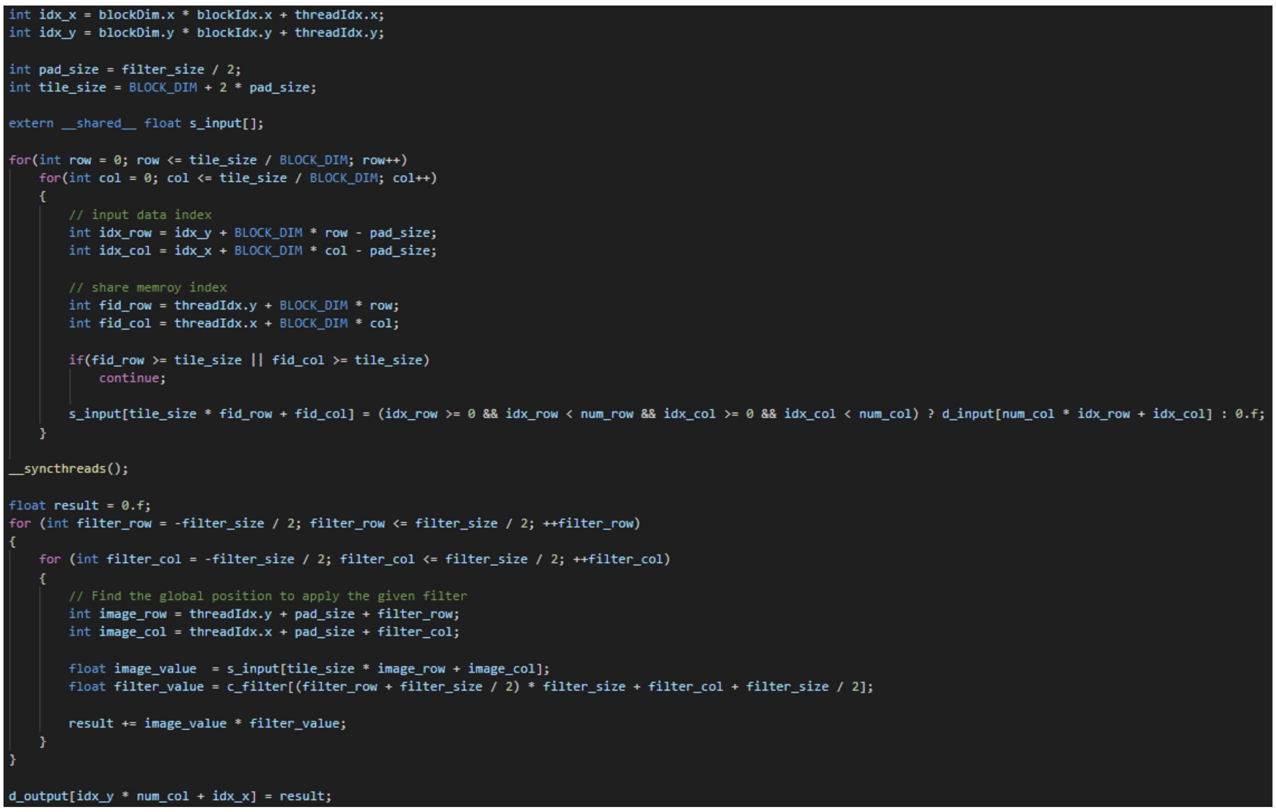


```
Running on GPU
Processing Time -> GPU: 7.66 ms
Running on CPU
Number of CPU threads: 2
Processing Time -> Host: 9878.94 ms
SUCCESS!!
```

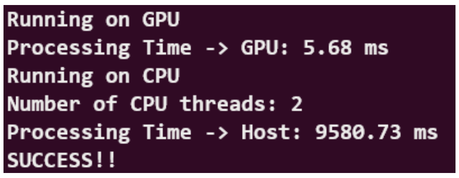
The performance is a little bit better.

Using shared memory and tiling input data



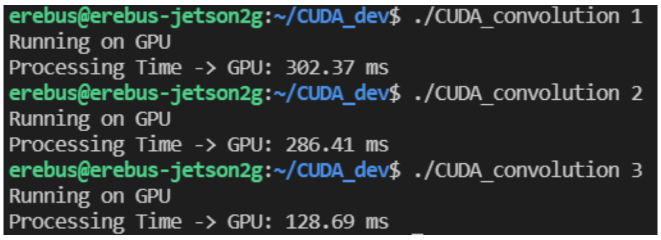


```
int idx_x = blockDim.x * blockIdx.x + threadIdx.x;
int idx_y = blockDim.y * blockIdx.y + threadIdx.y;

int pad_size = filter_size / 2;
int tile_size = BLOCK_DIM + 2 * pad_size;

extern __shared__ float s_input[];

for(int row = 0; row <= tile_size / BLOCK_DIM; row++)
    for(int col = 0; col <= tile_size / BLOCK_DIM; col++)
    {
        // input data index
        int idx_row = idx_y + BLOCK_DIM * row - pad_size;
        int idx_col = idx_x + BLOCK_DIM * col - pad_size;

        // share memroy index
        int fid_row = threadIdx.y + BLOCK_DIM * row;
        int fid_col = threadIdx.x + BLOCK_DIM * col;

        if(fid_row >= tile_size || fid_col >= tile_size)
            continue;

        s_input[tile_size * fid_row + fid_col] = (idx_row >= 0 && idx_row < num_row && idx_col >= 0 && idx_col < num_col) ? d_input[num_col * idx_row + idx_col] : 0.f;
    }

__syncthreads();

float result = 0.f;
for (int filter_row = -filter_size / 2; filter_row <= filter_size / 2; ++filter_row)
{
    for (int filter_col = -filter_size / 2; filter_col <= filter_size / 2; ++filter_col)
    {
        // Find the global position to apply the given filter
        int image_row = threadIdx.y + pad_size + filter_row;
        int image_col = threadIdx.x + pad_size + filter_col;

        float image_value  = s_input[tile_size * image_row + image_col];
        float filter_value = c_filter[(filter_row + filter_size / 2) * filter_size + filter_col + filter_size / 2];

        result += image_value * filter_value;
    }
}

d_output[idx_y * num_col + idx_x] = result;
```



```
Running on GPU
Processing Time -> GPU: 5.68 ms
Running on CPU
Number of CPU threads: 2
Processing Time -> Host: 9580.73 ms
SUCCESS!!
```

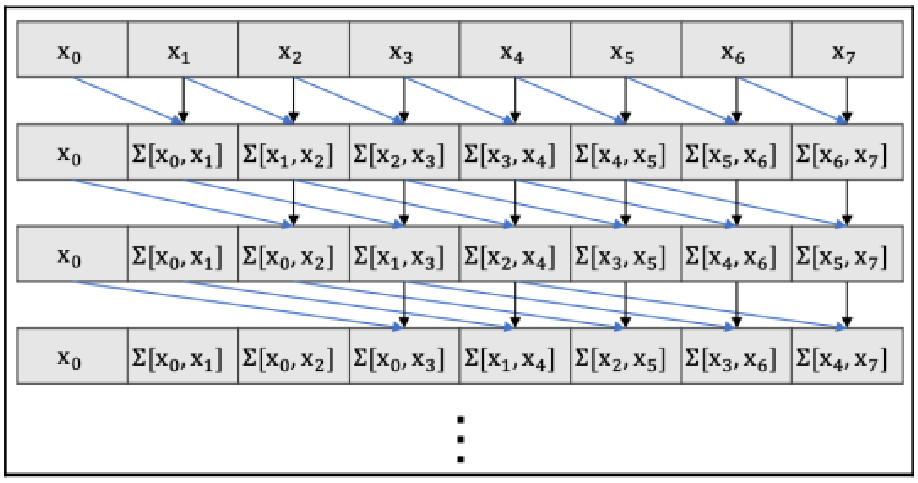
Result Ran on Jetson Nano (2G model)



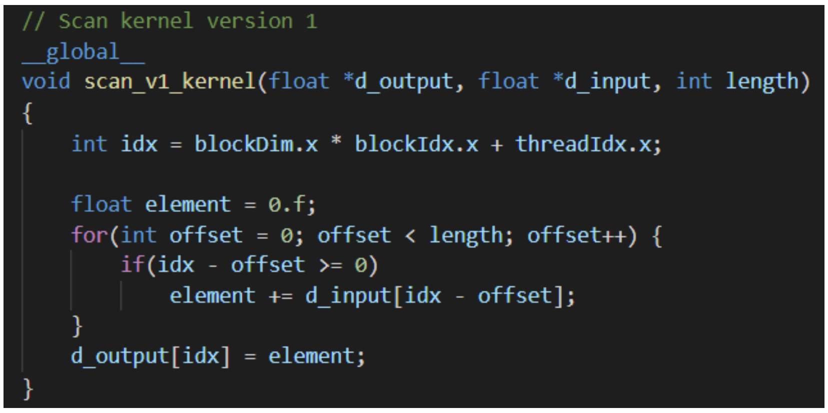
```
erebus@erebus-jetson2g:~/CUDA_dev$ ./CUDA_convolution 1
Running on GPU
Processing Time -> GPU: 302.37 ms
erebus@erebus-jetson2g:~/CUDA_dev$ ./CUDA_convolution 2
Running on GPU
Processing Time -> GPU: 286.41 ms
erebus@erebus-jetson2g:~/CUDA_dev$ ./CUDA_convolution 3
Running on GPU
Processing Time -> GPU: 128.69 ms _
```

Prefix Sum (Prefix Scan)

Parallel prefix-scan

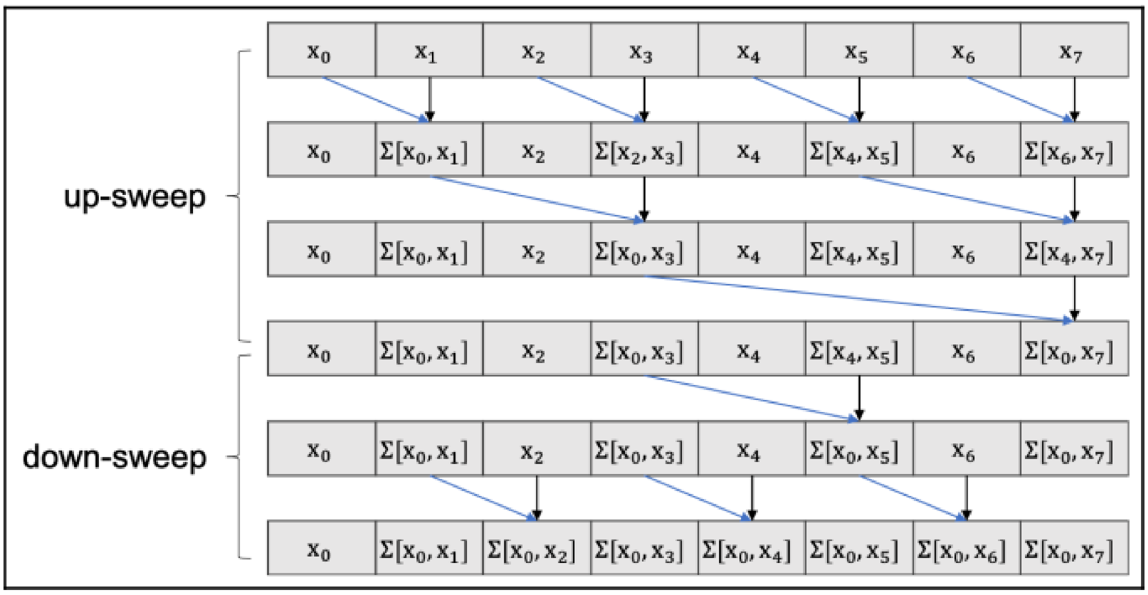


x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
x_0 $\Sigma[x_0,x_1]$ $\Sigma[x_1,x_2]$ $\Sigma[x_2,x_3]$ $\Sigma[x_3,x_4]$ $\Sigma[x_4,x_5]$ $\Sigma[x_5,x_6]$ $\Sigma[x_6,x_7]$
x_0 $\Sigma[x_0,x_1]$ $\Sigma[x_0,x_2]$ $\Sigma[x_1,x_3]$ $\Sigma[x_2,x_4]$ $\Sigma[x_3,x_5]$ $\Sigma[x_4,x_6]$ $\Sigma[x_5,x_7]$
x_0 $\Sigma[x_0,x_1]$ $\Sigma[x_0,x_2]$ $\Sigma[x_0,x_3]$ $\Sigma[x_1,x_4]$ $\Sigma[x_2,x_5]$ $\Sigma[x_3,x_6]$ $\Sigma[x_4,x_7]$
⋮



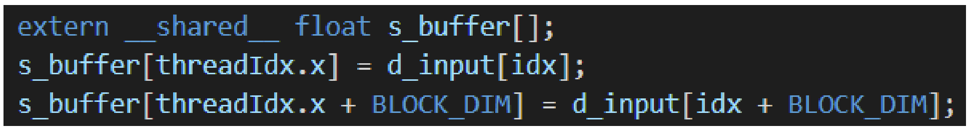
```
// Scan kernel version 1
__global__
void scan_v1_kernel(float *d_output, float *d_input, int length)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;

    float element = 0.f;
    for(int offset = 0; offset < length; offset++) {
        if(idx - offset >= 0)
            element += d_input[idx - offset];
    }
    d_output[idx] = element;
}
```

Blelloch Scan



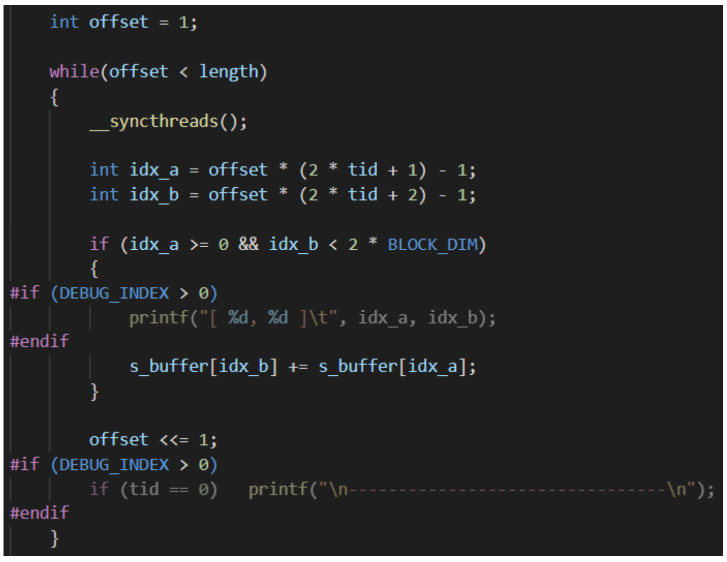
up-sweep
down-sweep
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
x_0 $\Sigma[x_0, x_1]$ x_2 $\Sigma[x_2, x_3]$ x_4 $\Sigma[x_4, x_5]$ x_6 $\Sigma[x_6, x_7]$
x_0 $\Sigma[x_0, x_1]$ x_2 $\Sigma[x_0, x_3]$ x_4 $\Sigma[x_4, x_5]$ x_6 $\Sigma[x_4, x_7]$
x_0 $\Sigma[x_0, x_1]$ x_2 $\Sigma[x_0, x_3]$ x_4 $\Sigma[x_4, x_5]$ x_6 $\Sigma[x_0, x_7]$
x_0 $\Sigma[x_0, x_1]$ x_2 $\Sigma[x_0, x_3]$ x_4 $\Sigma[x_0, x_5]$ x_6 $\Sigma[x_0, x_7]$
x_0 $\Sigma[x_0, x_1]$ $\Sigma[x_0, x_2]$ $\Sigma[x_0, x_3]$ $\Sigma[x_0, x_4]$ $\Sigma[x_0, x_5]$ $\Sigma[x_0, x_6]$ $\Sigma[x_0, x_7]$

There are two steps based on the stride controls. The strides are increased and decreased exponentially. This algorithm can generate outputs that are double the size of CUDA threads.



```
extern __shared__ float s_buffer[];
s_buffer[threadIdx.x] = d_input[idx];
s_buffer[threadIdx.x + BLOCK_DIM] = d_input[idx + BLOCK_DIM];
```

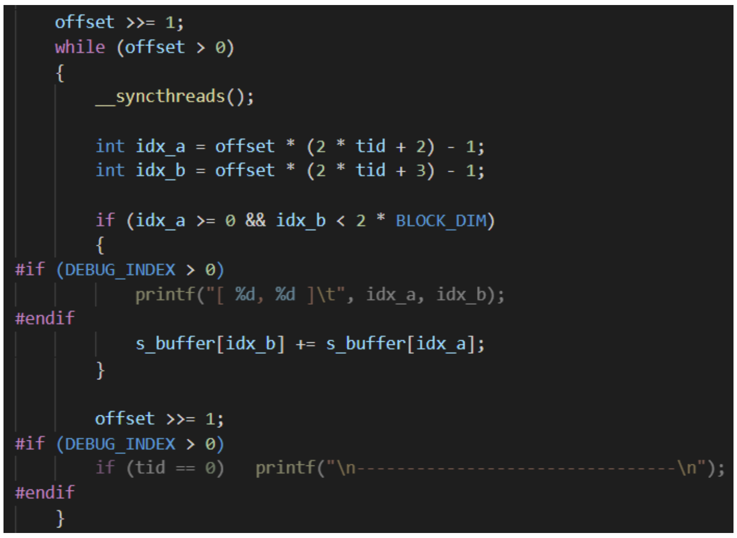
Up-sweeping



```
    int offset = 1;

    while(offset < length)
    {
        __syncthreads();

        int idx_a = offset * (2 * tid + 1) - 1;
        int idx_b = offset * (2 * tid + 2) - 1;

        if (idx_a >= 0 && idx_b < 2 * BLOCK_DIM)
        {
#if (DEBUG_INDEX > 0)
            printf("[ %d, %d ]\t", idx_a, idx_b);
#endif
            s_buffer[idx_b] += s_buffer[idx_a];
        }

        offset <<= 1;
#if (DEBUG_INDEX > 0)
        if (tid == 0)   printf("\n--------------------------------\n");
#endif
    }
```

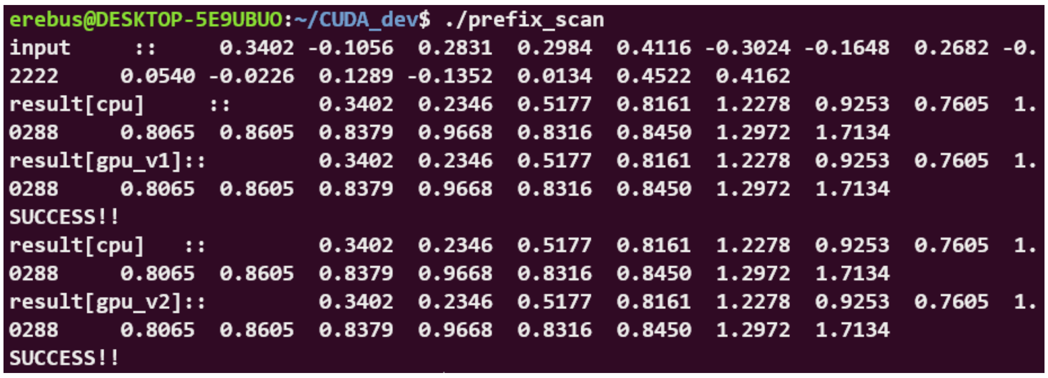
We use the if statement in the compiling state instead of using it in the runtime state because having branch statement in the CUDA kernel will slow down the performance.

Down-sweeping



```
    offset >>= 1;
    while (offset > 0)
    {
        __syncthreads();

        int idx_a = offset * (2 * tid + 2) - 1;
        int idx_b = offset * (2 * tid + 3) - 1;

        if (idx_a >= 0 && idx_b < 2 * BLOCK_DIM)
        {
#if (DEBUG_INDEX > 0)
            printf("[ %d, %d ]\t", idx_a, idx_b);
#endif
            s_buffer[idx_b] += s_buffer[idx_a];
        }

        offset >>= 1;
#if (DEBUG_INDEX > 0)
        if (tid == 0)   printf("\n-------------------------------\n");
#endif
    }
```

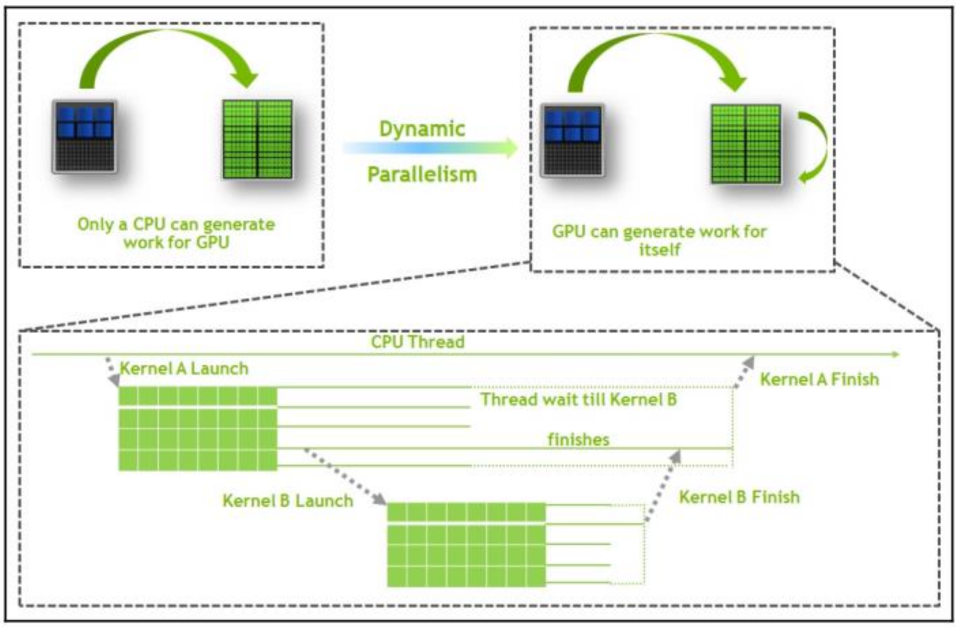
Result

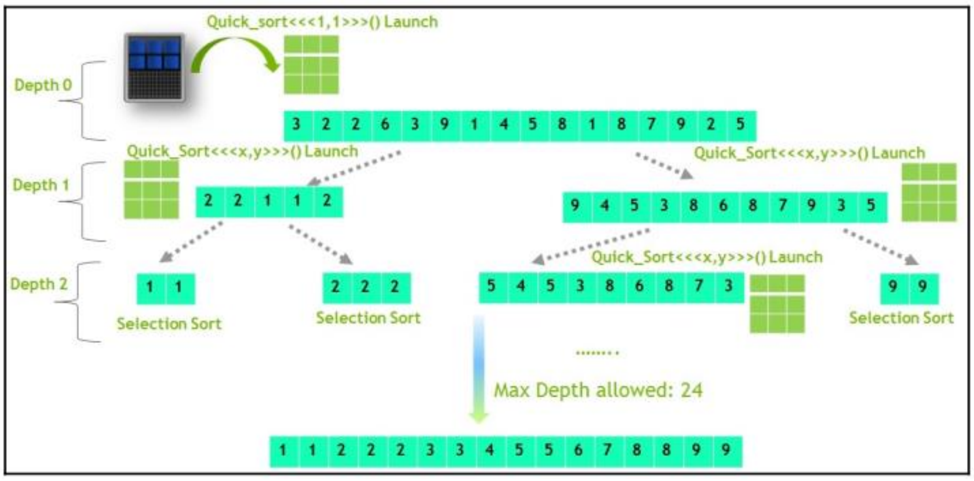


```
erebus@DESKTOP-5E9UBUO:~/CUDA_dev$ ./prefix_scan
input     ::     0.3402 -0.1056  0.2831  0.2984  0.4116 -0.3024 -0.1648  0.2682 -0.
2222     0.0540 -0.0226  0.1289 -0.1352  0.0134  0.4522  0.4162
result[cpu]     ::       0.3402  0.2346  0.5177  0.8161  1.2278  0.9253  0.7605  1.
0288     0.8065  0.8605  0.8379  0.9668  0.8316  0.8450  1.2972  1.7134
result[gpu_v1]::         0.3402  0.2346  0.5177  0.8161  1.2278  0.9253  0.7605  1.
0288     0.8065  0.8605  0.8379  0.9668  0.8316  0.8450  1.2972  1.7134
SUCCESS!!
result[cpu]   ::         0.3402  0.2346  0.5177  0.8161  1.2278  0.9253  0.7605  1.
0288     0.8065  0.8605  0.8379  0.9668  0.8316  0.8450  1.2972  1.7134
result[gpu_v2]::         0.3402  0.2346  0.5177  0.8161  1.2278  0.9253  0.7605  1.
0288     0.8065  0.8605  0.8379  0.9668  0.8316  0.8450  1.2972  1.7134
SUCCESS!!
```

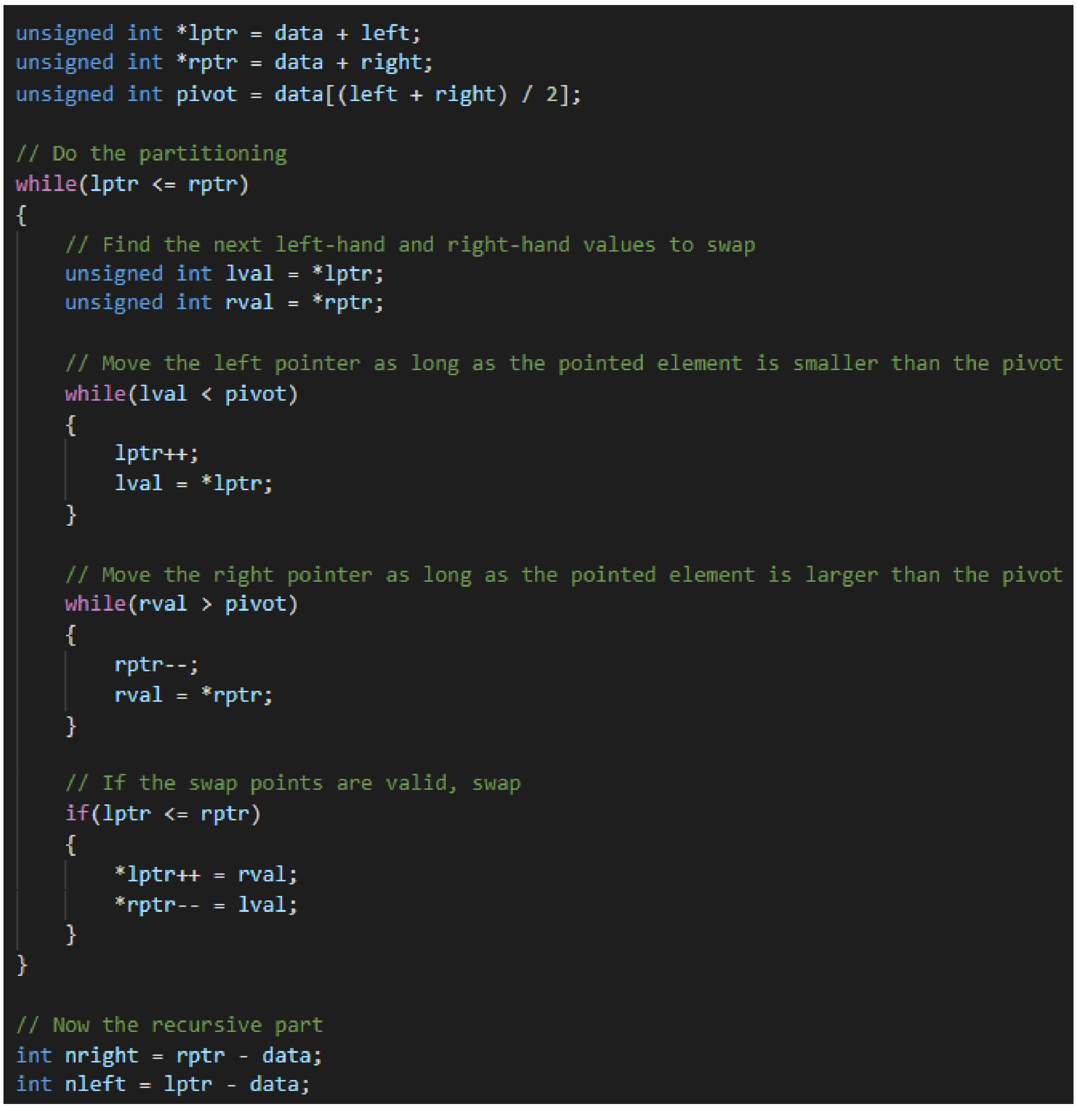
Quicksort

To implement Quicksort, the kernels should be launched recursively. Therefore, CUDA dynamic parallelism is needed. Dynamic parallelism allows the thread within a kernel to launch new kernels form the GPU without pass control back to the CPU host.

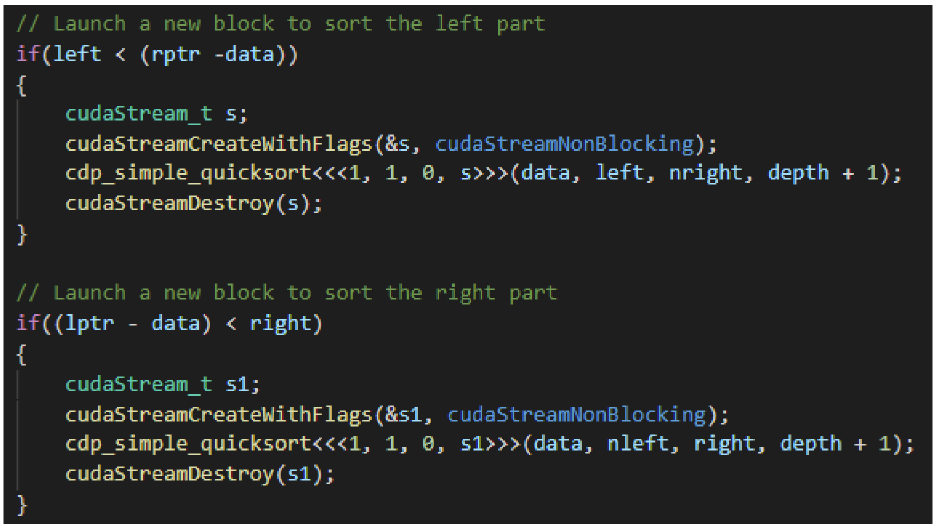

Only a CPU can generate work for GPU
Dynamic Parallelism
GPU can generate work for itself
CPU Thread
Kernel A Launch
Kernel A Finish
Thread wait till Kernel B finishes
Kernel B Launch
Kernel B Finish



For each subarray, the kernel will launch extra two kernels: one for the left array and one for the right array. Recursion stops after the max depth of the kernel has been reached or the number of the elements is less than 32 (warp size). In order to launch kernels asynchronous and independently, a stream should be created before every kernel launch.

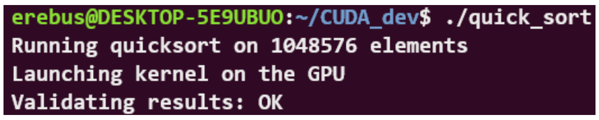

```
unsigned int *lptr = data + left;
unsigned int *rptr = data + right;
unsigned int pivot = data[(left + right) / 2];

// Do the partitioning
while(lptr <= rptr)
{
    // Find the next left-hand and right-hand values to swap
    unsigned int lval = *lptr;
    unsigned int rval = *rptr;

    // Move the left pointer as long as the pointed element is smaller than the pivot
    while(lval < pivot)
    {
        lptr++;
        lval = *lptr;
    }

    // Move the right pointer as long as the pointed element is larger than the pivot
    while(rval > pivot)
    {
        rptr--;
        rval = *rptr;
    }

    // If the swap points are valid, swap
    if(lptr <= rptr)
    {
        *lptr++ = rval;
        *rptr-- = lval;
    }
}

// Now the recursive part
int nright = rptr - data;
int nleft = lptr - data;
```

Create Stream for the dynamic launched CUDA kernels.



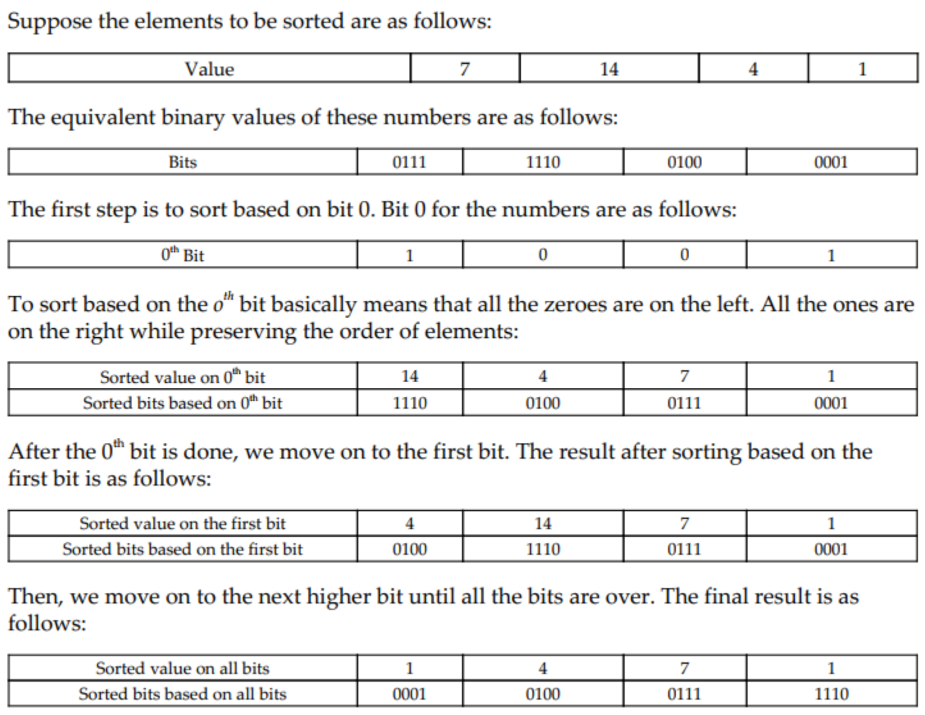
```
// Launch a new block to sort the left part
if(left < (rptr -data))
{
    cudaStream_t s;
    cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
    cdp_simple_quicksort<<<1, 1, 0, s>>>(data, left, nright, depth + 1);
    cudaStreamDestroy(s);
}

// Launch a new block to sort the right part
if((lptr - data) < right)
{
    cudaStream_t s1;
    cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
    cdp_simple_quicksort<<<1, 1, 0, s1>>>(data, nleft, right, depth + 1);
    cudaStreamDestroy(s1);
}
```

In order to enable dynamic parallelism, *-rdc=true* compiler flag need to be added to the nvcc compiler. The following is the result of the program run with 1024 * 1024 elements.



```
erebus@DESKTOP-5E9UBUO:~/CUDA_dev$ ./quick_sort
Running quicksort on 1048576 elements
Launching kernel on the GPU
Validating results: OK
```

Radix Sort



Suppose the elements to be sorted are as follows:

Value	7	14	4	1

The equivalent binary values of these numbers are as follows:

Bits	0111	1110	0100	0001

The first step is to sort based on bit 0. Bit 0 for the numbers are as follows:

0^{th} Bit	1	0	0	1

To sort based on the *o^{th}* bit basically means that all the zeroes are on the left. All the ones are on the right while preserving the order of elements:

Sorted value on 0^{th} bit	14	4	7	1
Sorted bits based on 0^{th} bit	1110	0100	0111	0001

After the 0^{th} bit is done, we move on to the first bit. The result after sorting based on the first bit is as follows:

Sorted value on the first bit	4	14	7	1
Sorted bits based on the first bit	0100	1110	0111	0001

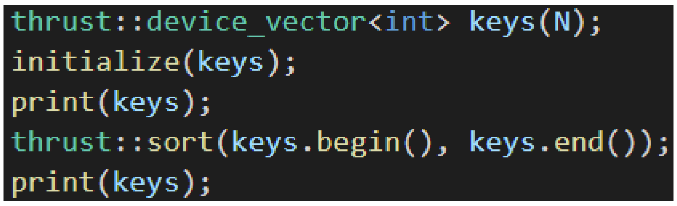
Then, we move on to the next higher bit until all the bits are over. The final result is as follows:

Sorted value on all bits	1	4	7	1
Sorted bits based on all bits	0001	0100	0111	1110

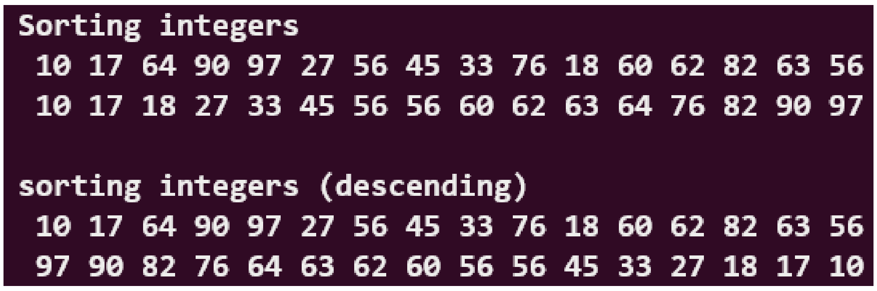
Implementation

Use Thrust library, which implements a generic radix sort.

Using the library provides different type of sorting methods, including radix sort for integers and floats. Similar to STL vectors, a custom comparator can be passed to the function.



```
thrust::device_vector<int> keys(N);
initialize(keys);
print(keys);
thrust::sort(keys.begin(), keys.end());
print(keys);
```



```
Sorting integers
 10 17 64 90 97 27 56 45 33 76 18 60 62 82 63 56
 10 17 18 27 33 45 56 56 60 62 63 64 76 82 90 97

sorting integers (descending)
 10 17 64 90 97 27 56 45 33 76 18 60 62 82 63 56
 97 90 82 76 64 63 62 60 56 56 45 33 27 18 17 10
```