6.2.4. Shared Memory

As detailed in Variable Memory Space Specifiers shared memory is allocated using the `__shared__` memory space specifier.

Shared memory is expected to be much faster than global memory as mentioned in Thread Hierarchy and detailed in Shared Memory. It can be used as scratchpad memory (or software managed cache) to minimize global memory accesses from a CUDA block as illustrated by the following matrix multiplication example.

The following code sample is a straightforward implementation of matrix multiplication that does not take advantage of shared memory. Each thread reads one row of *A* and one column of *B* and computes the corresponding element of *C* as illustrated in Figure 8. *A* is therefore read *B.width* times from global memory and *B* is read *A.height* times.

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
```

(continues on next page)

(continued from previous page)

```
    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

The following code sample is an implementation of matrix multiplication that does take advantage of shared memory. In this implementation, each thread block is responsible for computing one square sub-matrix *Csub* of *C* and each thread within the block is responsible for computing one element of *Csub*. As illustrated in Figure 9, *Csub* is equal to the product of two rectangular matrices: the sub-matrix of *A* of dimension (*A.width, block_size*) that has the same row indices as *Csub*, and the sub-matrix of *B* of dimension (*block_size, A.width*)that has the same column indices as *Csub*. In order to fit into the device's resources, these two rectangular matrices are divided into as many square matrices of dimension *block_size* as necessary and *Csub* is computed as the sum of the products of these square matrices. Each of these products is performed by first loading the two corresponding square matrices from global memory to shared memory with one thread loading one element of each matrix, and then by having each thread compute one element of the product. Each thread accumulates the result of each of these products into a register and once done writes the result to global memory.

By blocking the computation this way, we take advantage of fast shared memory and save a lot of global memory bandwidth since *A* is only read (*B.width / block_size*) times from global memory and *B* is read (*A.height / block_size*) times.

The *Matrix* type from the previous code sample is augmented with a *stride* field, so that sub-matrices can be efficiently represented with the same type. __device__ functions are used to get and set elements and build any sub-matrix from a matrix.

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;
// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
```

(continues on next page)

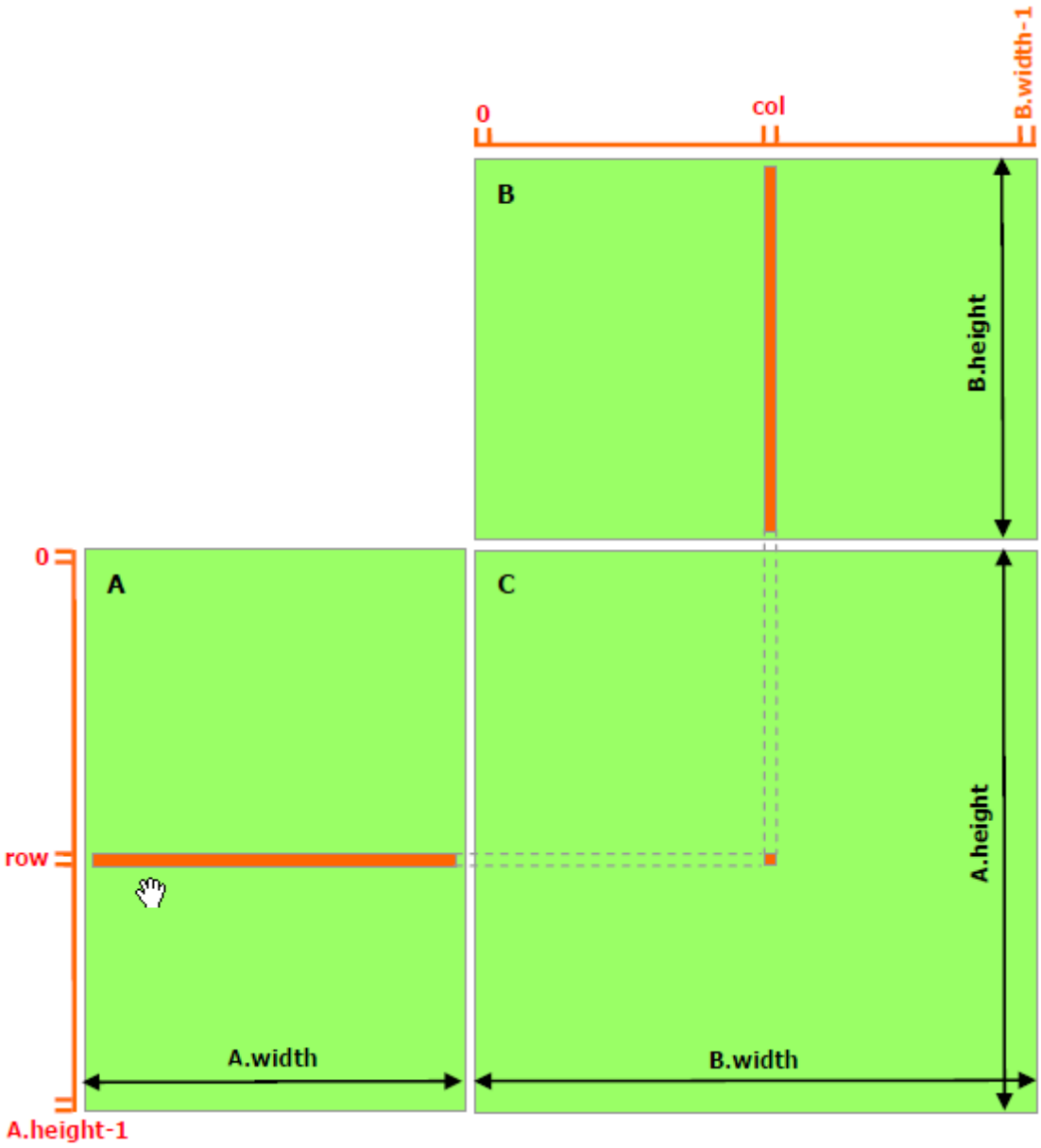

Figure 8: Matrix Multiplication without Shared Memory

(continued from previous page)

```
{
    return A.elements[row * A.stride + col];
}
// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value)
{
    A.elements[row * A.stride + col] = value;
}
// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
 __device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                         + BLOCK_SIZE * col];
    return Asub;
}
// Thread block size
#define BLOCK_SIZE 16
// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
    cudaMemcpyHostToDevice);
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);
    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,
               cudaMemcpyDeviceToHost);
    // Free device memory
```

(continues on next page)

(continued from previous page)

```
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}
// Matrix multiplication kernel called by MatMul()
 __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;
    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;
    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;
    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);
        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();
        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];
        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }
    // Write Csub to device memory
    // Each thread writes one element
    SetElement(Csub, row, col, Cvalue);
}
```

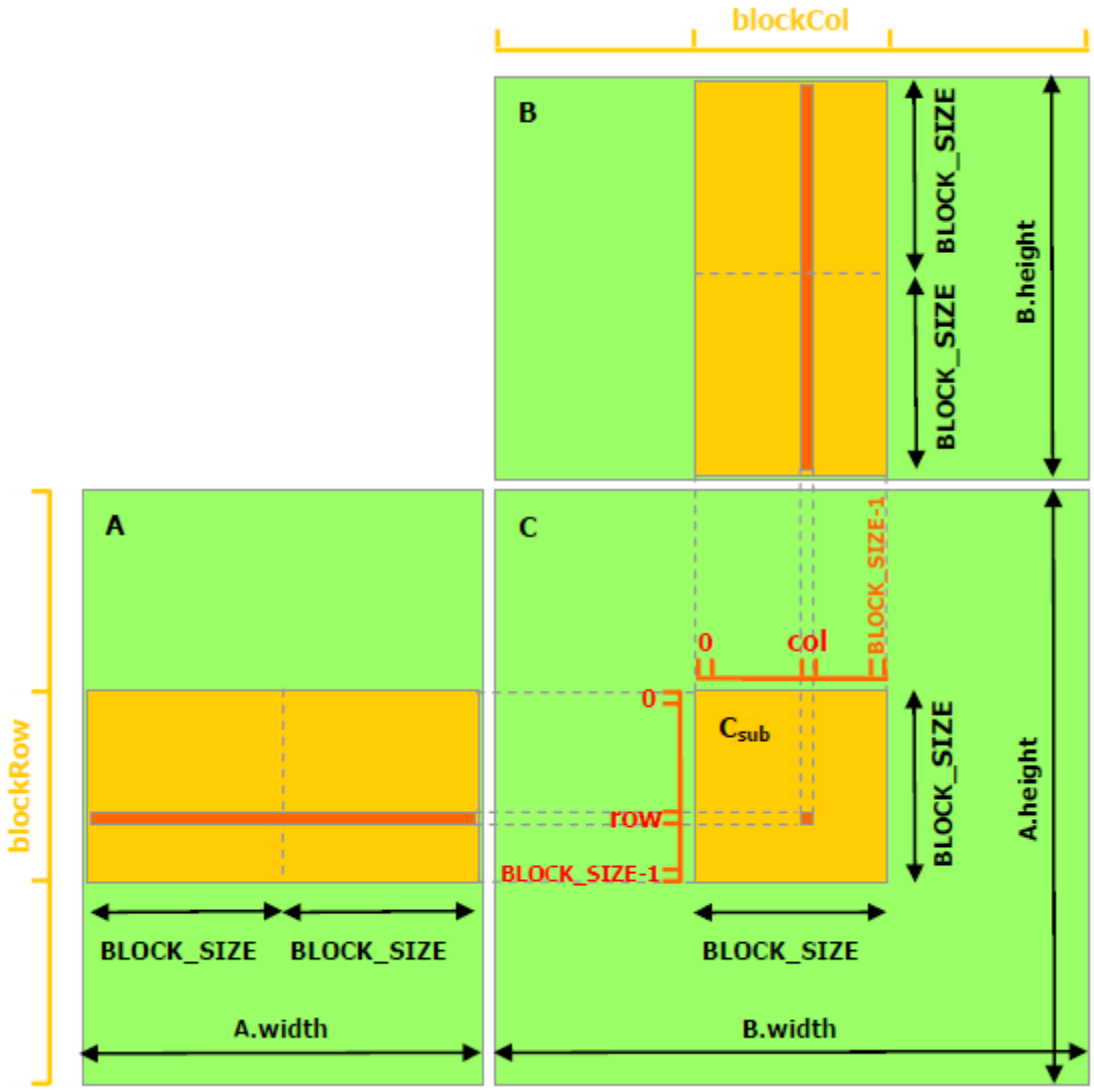

Figure 9: Matrix Multiplication with Shared Memory

6.2.5. Distributed Shared Memory

Thread block clusters introduced in compute capability 9.0 provide the ability for threads in a thread block cluster to access shared memory of all the participating thread blocks in a cluster. This partitioned shared memory is called *Distributed Shared Memory*, and the corresponding address space is called Distributed shared memory address space. Threads that belong to a thread block cluster, can read, write or perform atomics in the distributed address space, regardless whether the address belongs to the local thread block or a remote thread block. Whether a kernel uses distributed shared memory or not, the shared memory size specifications, static or dynamic is still per thread block. The size of distributed shared memory is just the number of thread blocks per cluster multiplied by the size of shared memory per thread block.

Accessing data in distributed shared memory requires all the thread blocks to exist. A user can guarantee that all thread blocks have started executing using `cluster.sync()` from Cluster Group API. The user also needs to ensure that all distributed shared memory operations happen before the exit of a thread block, e.g., if a remote thread block is trying to read a given thread block's shared memory, user needs to ensure that the shared memory read by remote thread block is completed before it can exit.

CUDA provides a mechanism to access to distributed shared memory, and applications can benefit from leveraging its capabilities. Lets look at a simple histogram computation and how to optimize it on the GPU using thread block cluster. A standard way of computing histograms is do the computation in the shared memory of each thread block and then perform global memory atomics. A limitation of this approach is the shared memory capacity. Once the histogram bins no longer fit in the shared memory, a user needs to directly compute histograms and hence the atomics in the global memory. With distributed shared memory, CUDA provides an intermediate step, where a depending on the histogram bins size, histogram can be computed in shared memory, distributed shared memory or global memory directly.

The CUDA kernel example below shows how to compute histograms in shared memory or distributed shared memory, depending on the number of histogram bins.

```
#include <cooperative_groups.h>

// Distributed Shared memory histogram kernel
__global__ void clusterHist_kernel(int *bins, const int nbins, const int bins_per_
↪block, const int *__restrict__ input,
                                   size_t array_size)
{
  extern __shared__ int smem[];
  namespace cg = cooperative_groups;
  int tid = cg::this_grid().thread_rank();

  // Cluster initialization, size and calculating local bin offsets.
  cg::cluster_group cluster = cg::this_cluster();
  unsigned int clusterBlockRank = cluster.block_rank();
  int cluster_size = cluster.dim_blocks().x;

  for (int i = threadIdx.x; i < bins_per_block; i += blockDim.x)
  {
    smem[i] = 0; //Initialize shared memory histogram to zeros
  }

  // cluster synchronization ensures that shared memory is initialized to zero in
  // all thread blocks in the cluster. It also ensures that all thread blocks
  // have started executing and they exist concurrently.
```

(continues on next page)

(continued from previous page)

```
  cluster.sync();

  for (int i = tid; i < array_size; i += blockDim.x * gridDim.x)
  {
    int ldata = input[i];

    //Find the right histogram bin.
    int binid = ldata;
    if (ldata < 0)
      binid = 0;
    else if (ldata >= nbins)
      binid = nbins - 1;

    //Find destination block rank and offset for computing
    //distributed shared memory histogram
    int dst_block_rank = (int)(binid / bins_per_block);
    int dst_offset = binid % bins_per_block;

    //Pointer to target block shared memory
    int *dst_smem = cluster.map_shared_rank(smem, dst_block_rank);

    //Perform atomic update of the histogram bin
    atomicAdd(dst_smem + dst_offset, 1);
  }

  // cluster synchronization is required to ensure all distributed shared
  // memory operations are completed and no thread block exits while
  // other thread blocks are still accessing distributed shared memory
  cluster.sync();

  // Perform global memory histogram, using the local distributed memory histogram
  int *lbins = bins + cluster.block_rank() * bins_per_block;
  for (int i = threadIdx.x; i < bins_per_block; i += blockDim.x)
  {
    atomicAdd(&lbins[i], smem[i]);
  }
}
```

The above kernel can be launched at runtime with a cluster size depending on the amount of distributed shared memory required. If histogram is small enough to fit in shared memory of just one block, user can launch kernel with cluster size 1. The code snippet below shows how to launch a cluster kernel dynamically based depending on shared memory requirements.

```
// Launch via extensible launch
{
  cudaLaunchConfig_t config = {0};
  config.gridDim = array_size / threads_per_block;
  config.blockDim = threads_per_block;

  // cluster_size depends on the histogram size.
  // ( cluster_size == 1 ) implies no distributed shared memory, just thread block
↪local shared memory
  int cluster_size = 2; // size 2 is an example here
  int nbins_per_block = nbins / cluster_size;

  //dynamic shared memory size is per block.
```

(continues on next page)

(continued from previous page)

```
  //Distributed shared memory size =  cluster_size * nbins_per_block * sizeof(int)
  config.dynamicSmemBytes = nbins_per_block * sizeof(int);

  CUDA_CHECK(::cudaFuncSetAttribute((void *)clusterHist_kernel,
↪cudaFuncAttributeMaxDynamicSharedMemorySize, config.dynamicSmemBytes));

  cudaLaunchAttribute attribute[1];
  attribute[0].id = cudaLaunchAttributeClusterDimension;
  attribute[0].val.clusterDim.x = cluster_size;
  attribute[0].val.clusterDim.y = 1;
  attribute[0].val.clusterDim.z = 1;

  config.numAttrs = 1;
  config.attrs = attribute;

  cudaLaunchKernelEx(&config, clusterHist_kernel, bins, nbins, nbins_per_block, input,
↪ array_size);
}
```

6.2.6. Page-Locked Host Memory

The runtime provides functions to allow the use of *page-locked* (also known as *pinned*) host memory (as opposed to regular pageable host memory allocated by `malloc()`):

- `cudaHostAlloc()` and `cudaFreeHost()` allocate and free page-locked host memory;
- `cudaHostRegister()` page-locks a range of memory allocated by `malloc()` (see reference manual for limitations).

Using page-locked host memory has several benefits:

- Copies between page-locked host memory and device memory can be performed concurrently with kernel execution for some devices as mentioned in Asynchronous Concurrent Execution.
- On some devices, page-locked host memory can be mapped into the address space of the device, eliminating the need to copy it to or from device memory as detailed in Mapped Memory.
- On systems with a front-side bus, bandwidth between host memory and device memory is higher if host memory is allocated as page-locked and even higher if in addition it is allocated as write-combining as described in Write-Combining Memory.

Note: Page-locked host memory is not cached on non I/O coherent Tegra devices. Also, `cudaHostRegister()` is not supported on non I/O coherent Tegra devices.

The simple zero-copy CUDA sample comes with a detailed document on the page-locked memory APIs.