



CUDA C++ Programming Guide

Design Guide

Table of Contents

Chapter 1. Introduction.....	1
1.1. The Benefits of Using GPUs.....	1
1.2. CUDA®: A General-Purpose Parallel Computing Platform and Programming Model.....	2
1.3. A Scalable Programming Model.....	3
1.4. Document Structure.....	5
Chapter 2. Programming Model.....	7
2.1. Kernels.....	7
2.2. Thread Hierarchy.....	8
2.3. Memory Hierarchy.....	10
2.4. Heterogeneous Programming.....	11
2.5. Compute Capability.....	14
Chapter 3. Programming Interface.....	15
3.1. Compilation with NVCC.....	15
3.1.1. Compilation Workflow.....	16
3.1.1.1. Offline Compilation.....	16
3.1.1.2. Just-in-Time Compilation.....	16
3.1.2. Binary Compatibility.....	17
3.1.3. PTX Compatibility.....	17
3.1.4. Application Compatibility.....	17
3.1.5. C++ Compatibility.....	18
3.1.6. 64-Bit Compatibility.....	18
3.2. CUDA Runtime.....	19
3.2.1. Initialization.....	19
3.2.2. Device Memory.....	20
3.2.3. Device Memory L2 Access Management.....	23
3.2.3.1. L2 cache Set-Aside for Persisting Accesses.....	23
3.2.3.2. L2 Policy for Persisting Accesses.....	23
3.2.3.3. L2 Access Properties.....	25
3.2.3.4. L2 Persistence Example.....	25
3.2.3.5. Reset L2 Access to Normal.....	26
3.2.3.6. Manage Utilization of L2 set-aside cache.....	27
3.2.3.7. Query L2 cache Properties.....	27
3.2.3.8. Control L2 Cache Set-Aside Size for Persisting Memory Access.....	27
3.2.4. Shared Memory.....	27
3.2.5. Page-Locked Host Memory.....	33

3.2.5.1. Portable Memory.....	33
3.2.5.2. Write-Combining Memory.....	33
3.2.5.3. Mapped Memory.....	34
3.2.6. Asynchronous Concurrent Execution.....	35
3.2.6.1. Concurrent Execution between Host and Device.....	35
3.2.6.2. Concurrent Kernel Execution.....	35
3.2.6.3. Overlap of Data Transfer and Kernel Execution.....	36
3.2.6.4. Concurrent Data Transfers.....	36
3.2.6.5. Streams.....	36
3.2.6.6. CUDA Graphs.....	40
3.2.6.7. Events.....	49
3.2.6.8. Synchronous Calls.....	49
3.2.7. Multi-Device System.....	49
3.2.7.1. Device Enumeration.....	49
3.2.7.2. Device Selection.....	50
3.2.7.3. Stream and Event Behavior.....	50
3.2.7.4. Peer-to-Peer Memory Access.....	51
3.2.7.5. Peer-to-Peer Memory Copy.....	51
3.2.8. Unified Virtual Address Space.....	52
3.2.9. Interprocess Communication.....	52
3.2.10. Error Checking.....	53
3.2.11. Call Stack.....	54
3.2.12. Texture and Surface Memory.....	54
3.2.12.1. Texture Memory.....	54
3.2.12.2. Surface Memory.....	63
3.2.12.3. CUDA Arrays.....	67
3.2.12.4. Read/Write Coherency.....	67
3.2.13. Graphics Interoperability.....	67
3.2.13.1. OpenGL Interoperability.....	68
3.2.13.2. Direct3D Interoperability.....	70
3.2.13.3. SLI Interoperability.....	75
3.2.14. External Resource Interoperability.....	75
3.2.14.1. Vulkan Interoperability.....	76
3.2.14.2. OpenGL Interoperability.....	83
3.2.14.3. Direct3D 12 Interoperability.....	84
3.2.14.4. Direct3D 11 Interoperability.....	89
3.2.14.5. NVIDIA Software Communication Interface Interoperability (NVSCI).....	97
3.2.15. CUDA User Objects.....	101

3.3. Versioning and Compatibility.....	103
3.4. Compute Modes.....	104
3.5. Mode Switches.....	105
3.6. Tesla Compute Cluster Mode for Windows.....	105
Chapter 4. Hardware Implementation.....	107
4.1. SIMT Architecture.....	107
4.2. Hardware Multithreading.....	109
Chapter 5. Performance Guidelines.....	110
5.1. Overall Performance Optimization Strategies.....	110
5.2. Maximize Utilization.....	110
5.2.1. Application Level.....	110
5.2.2. Device Level.....	111
5.2.3. Multiprocessor Level.....	111
5.2.3.1. Occupancy Calculator.....	113
5.3. Maximize Memory Throughput.....	115
5.3.1. Data Transfer between Host and Device.....	115
5.3.2. Device Memory Accesses.....	116
5.4. Maximize Instruction Throughput.....	120
5.4.1. Arithmetic Instructions.....	120
5.4.2. Control Flow Instructions.....	126
5.4.3. Synchronization Instruction.....	126
5.5. Minimize Memory Thrashing.....	126
Appendix A. CUDA-Enabled GPUs.....	128
Appendix B. C++ Language Extensions.....	129
B.1. Function Execution Space Specifiers.....	129
B.1.1. __global__.....	129
B.1.2. __device__.....	129
B.1.3. __host__.....	129
B.1.4. Undefined behavior.....	130
B.1.5. __noinline__ and __forceinline__.....	130
B.2. Variable Memory Space Specifiers.....	131
B.2.1. __device__.....	131
B.2.2. __constant__.....	131
B.2.3. __shared__.....	131
B.2.4. __managed__.....	132
B.2.5. __restrict__.....	132
B.3. Built-in Vector Types.....	134

3.2.4. 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 7](#). *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);

    // 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);
}

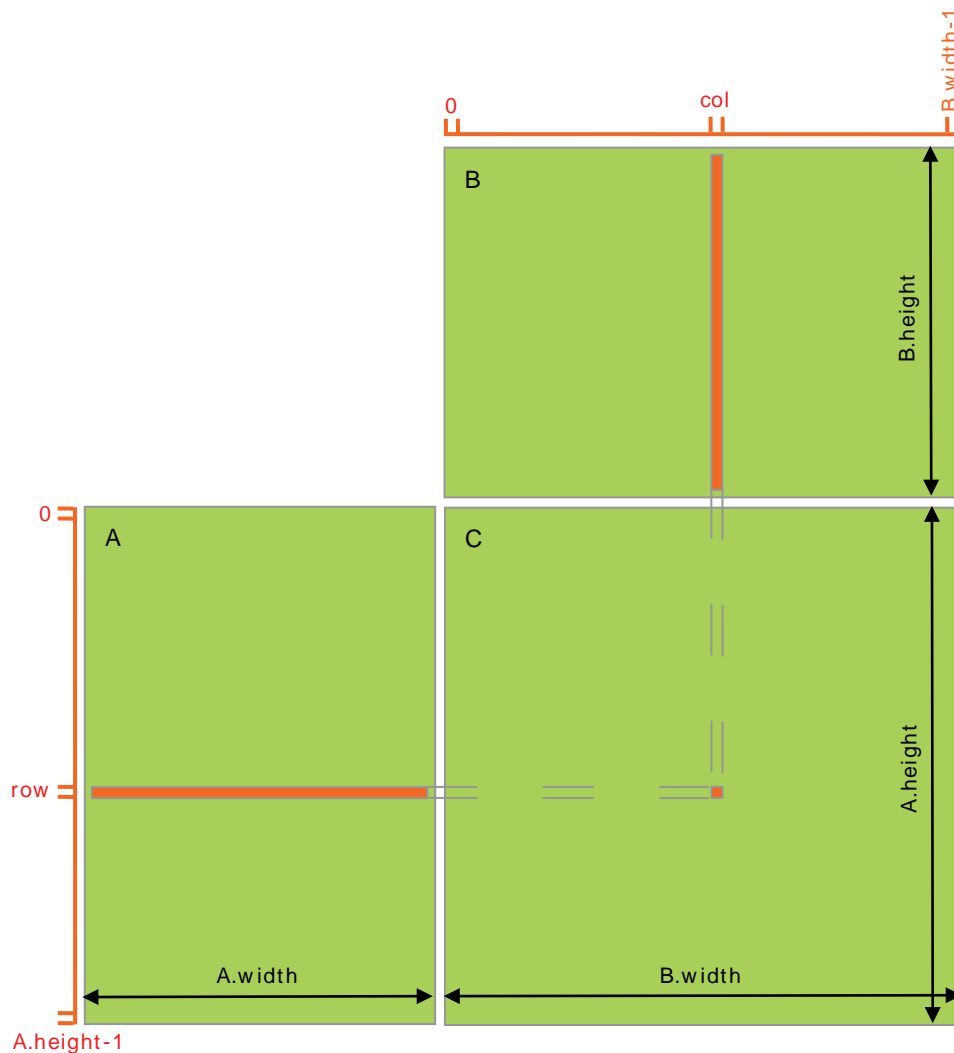
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
```

```

{
    // 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;
}

```

Figure 7. Matrix Multiplication without Shared Memory



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 C_{sub} of C and each thread within the block is responsible for computing one element of C_{sub} . As illustrated in [Figure 8](#), C_{sub} 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 C_{sub} , and the sub-matrix of B of dimension $(block_size, A.width)$ that has the

same column indices as C_{sub} . 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 C_{sub} 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)
{
    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
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 8. Matrix Multiplication with Shared Memory

