**Matrix Multiplication**

<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-c-runtime>

The CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory.

Device memory can be allocated either as *linear memory* or as *CUDA arrays*.

CUDA arrays are opaque memory layouts optimized for texture fetching. They are described in [Texture and Surface Memory](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#texture-and-surface-memory).

Linear memory exists on the device in a 40-bit address space, so separately allocated entities can reference one another via pointers, for example, in a binary tree.

Linear memory is typically allocated using cudaMalloc() and freed using cudaFree() and data transfer between host memory and device memory are typically done using cudaMemcpy(). In the vector addition code sample of [Kernels](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#kernels), the vectors need to be copied from host memory to device memory:

// Device code  
\_\_global\_\_ void VecAdd(float\* A, float\* B, float\* C, int N) {  
 int i = blockDim.x \* blockIdx.x + threadIdx.x;  
 if (i < N)  
 C[i] = A[i] + B[i];  
}

// Host code  
int main() {  
 int N = ...;  
 size\_t size = N \* sizeof(float);

// Allocate input vectors h\_A and h\_B in host memory  
 float\* h\_A = (float\*)malloc(size);  
 float\* h\_B = (float\*)malloc(size);

// Initialize input vectors  
 ...

// Allocate vectors in device memory  
 float\* d\_A; cudaMalloc(&d\_A, size);  
 float\* d\_B; cudaMalloc(&d\_B, size);  
 float\* d\_C; cudaMalloc(&d\_C, size);

// Copy vectors from host memory to device memory  
 cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);  
 cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);

// Invoke kernel  
 int threadsPerBlock = 256;  
 int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_B, d\_C, N);

// Copy result from device memory to host memory  
 // h\_C contains the result in host memory  
 cudaMemcpy(h\_C, d\_C, size, cudaMemcpyDeviceToHost);

// Free device memory  
 cudaFree(d\_A);  
 cudaFree(d\_B);  
 cudaFree(d\_C);

// Free host memory  
 ...  
}

Linear memory can also be allocated through cudaMallocPitch() and cudaMalloc3D(). These functions are recommended for allocations of 2D or 3D arrays as it makes sure that the allocation is appropriately padded to meet the alignment requirements described in [Device Memory Accesses](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses), therefore ensuring best performance when accessing the row addresses or performing copies between 2D arrays and other regions of device memory (using the cudaMemcpy2D() and cudaMemcpy3D() functions). The returned pitch (or stride) must be used to access array elements. The following code sample allocates a width x height 2D array of floating-point values and shows how to loop over the array elements in device code:

// Host code  
int width = 64, height = 64;  
float\* devPtr;  
size\_t pitch;  
cudaMallocPitch(&devPtr, &pitch, width \* sizeof(float), height);  
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code  
\_\_global\_\_ void MyKernel(float\* devPtr, size\_t pitch, int width, int height) {  
 for (int r = 0; r < height; ++r) {  
 float\* row = (float\*)((char\*)devPtr + r \* pitch);  
 for (int c = 0; c < width; ++c) {  
 float element = row[c];  
 }  
 }  
}

The following code sample allocates a width x height x depth 3D array of floating-point values and shows how to loop over the array elements in device code:

// Host code  
int width = 64, height = 64, depth = 64;  
cudaExtent extent = make\_cudaExtent(width \* sizeof(float), height, depth);  
cudaPitchedPtr devPitchedPtr;  
cudaMalloc3D(&devPitchedPtr, extent);

MyKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);

// Device code  
\_\_global\_\_ void MyKernel(cudaPitchedPtr devPitchedPtr,  
 int width, int height, int depth) {  
 char\* devPtr = devPitchedPtr.ptr;  
 size\_t pitch = devPitchedPtr.pitch;  
 size\_t slicePitch = pitch \* height;  
 for (int z = 0; z < depth; ++z) {  
 char\* slice = devPtr + z \* slicePitch;  
 for (int y = 0; y < height; ++y) {  
 float\* row = (float\*)(slice + y \* pitch);  
 for (int x = 0; x < width; ++x) {  
 float element = row[x];  
 }  
 }  
 }  
}

The reference manual lists all the various functions used to copy memory between linear memory allocated with cudaMalloc(), linear memory allocated with cudaMallocPitch() or cudaMalloc3D(), CUDA arrays, and memory allocated for variables declared in global or constant memory space.

The following code sample illustrates various ways of accessing global variables via the runtime API:

\_\_constant\_\_ float constData[256];  
float data[256];  
cudaMemcpyToSymbol(constData, data, sizeof(data));  
cudaMemcpyFromSymbol(data, constData, sizeof(data));

\_\_device\_\_ float devData;  
float value = 3.14f;  
cudaMemcpyToSymbol(devData, &value, sizeof(float));

\_\_device\_\_ float\* devPointer;  
float\* ptr;  
cudaMalloc(&ptr, 256 \* sizeof(float));  
cudaMemcpyToSymbol(devPointer, &ptr, sizeof(ptr));

cudaGetSymbolAddress() is used to retrieve the address pointing to the memory allocated for a variable declared in global memory space. The size of the allocated memory is obtained through cudaGetSymbolSize().

**[3.2.3. Shared Memory](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html" \l "shared-memory)**

As detailed in [Variable Memory Space Specifiers](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#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](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy) and detailed in [Shared Memory](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory). Any opportunity to replace global memory accesses by shared memory accesses should therefore be exploited 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 9](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory__matrix-multiplication-no-shared-memory). *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);

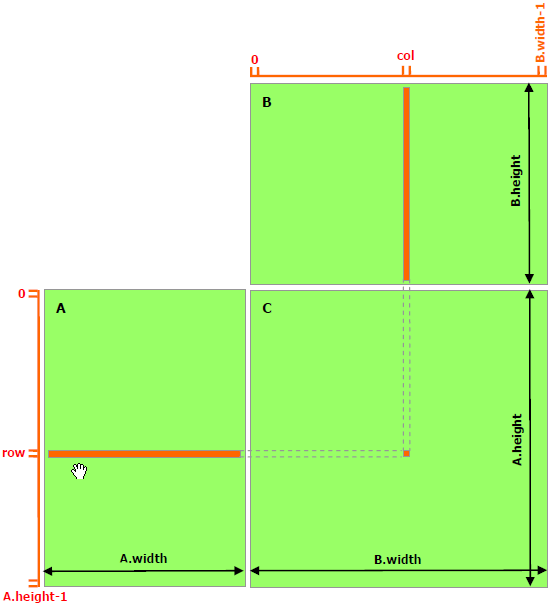
// Allocate C in device memory  
 Matrix d\_C;  
 d\_C.width = C.width; d\_C.height = C.height;  
 size = C.width \* C.height \* sizeof(float);  
 cudaMalloc(&d\_C.elements, size);

// 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, Cd.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 9. 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 *Csub* of *C* and each thread within the block is responsible for computing one element of *Csub*. As illustrated in [Figure 10](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory__matrix-multiplication-shared-memory), *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\_\_](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-function-specifier) 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);

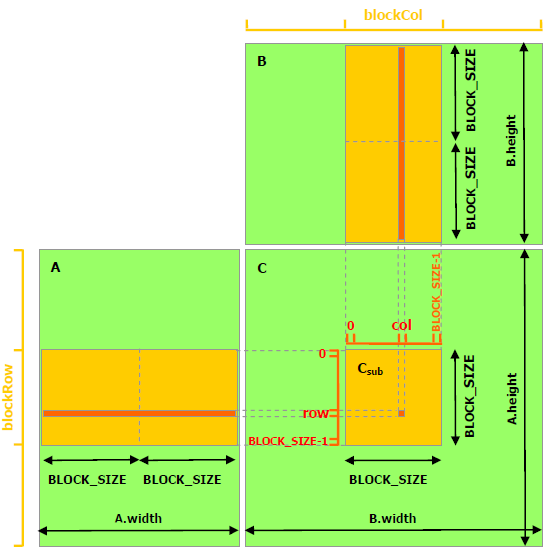
// Shared memory used to store Asub and Bsub respectively  
 \_\_shared\_\_ float As[BLOCK\_SIZE][BLOCK\_SIZE];  
 \_\_shared\_\_ float Bs[BLOCK\_SIZE][BLOCK\_SIZE];

// 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 10. Matrix Multiplication with Shared Memory*  
  


Read more at: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#ixzz58ruAuTTf>   
Follow us: [@GPUComputing on Twitter](http://ec.tynt.com/b/rw?id=aBENEGgL0r44W6acwqm_6r&u=GPUComputing) | [NVIDIA on Facebook](http://ec.tynt.com/b/rf?id=aBENEGgL0r44W6acwqm_6r&u=NVIDIA)