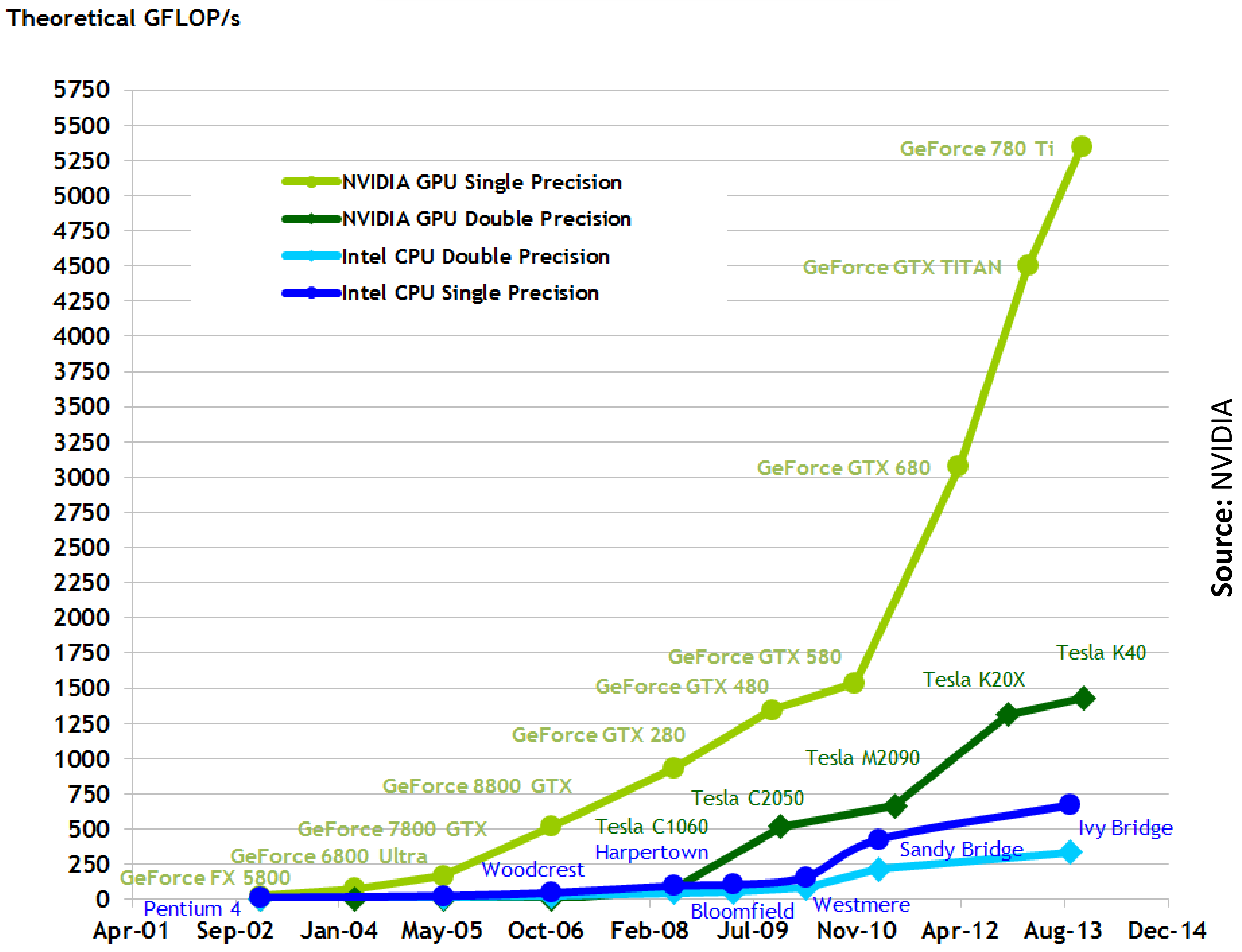
CSE 590: Special Topics Course (Supercomputing)

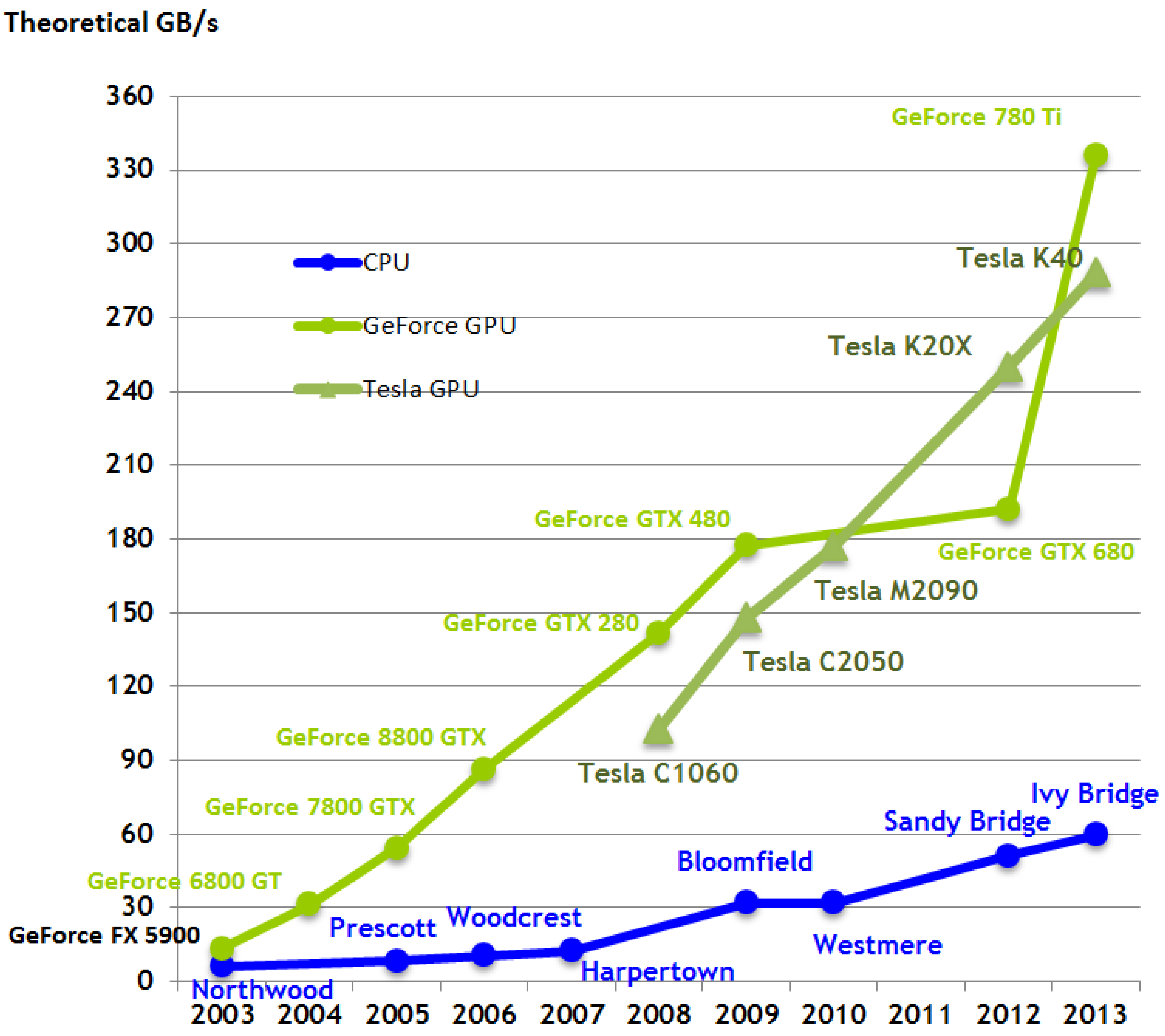
Lecture 9 (GPGPU Computing & CUDA)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2016

GPU vs CPU: FLOP/s

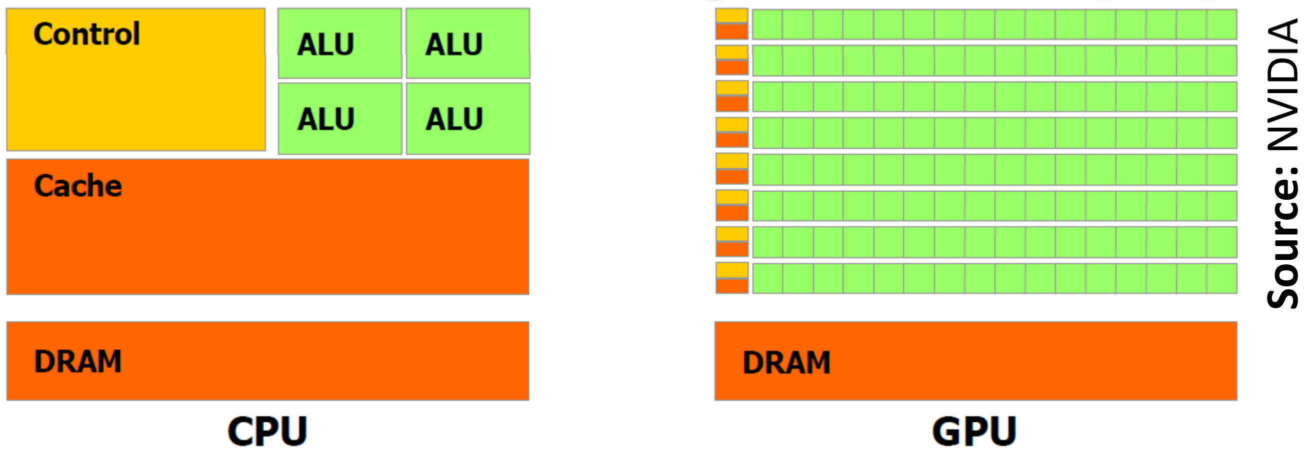


GPU vs CPU: Memory Bandwidth



Source: NVIDIA

GPU vs CPU: Design Philosophy



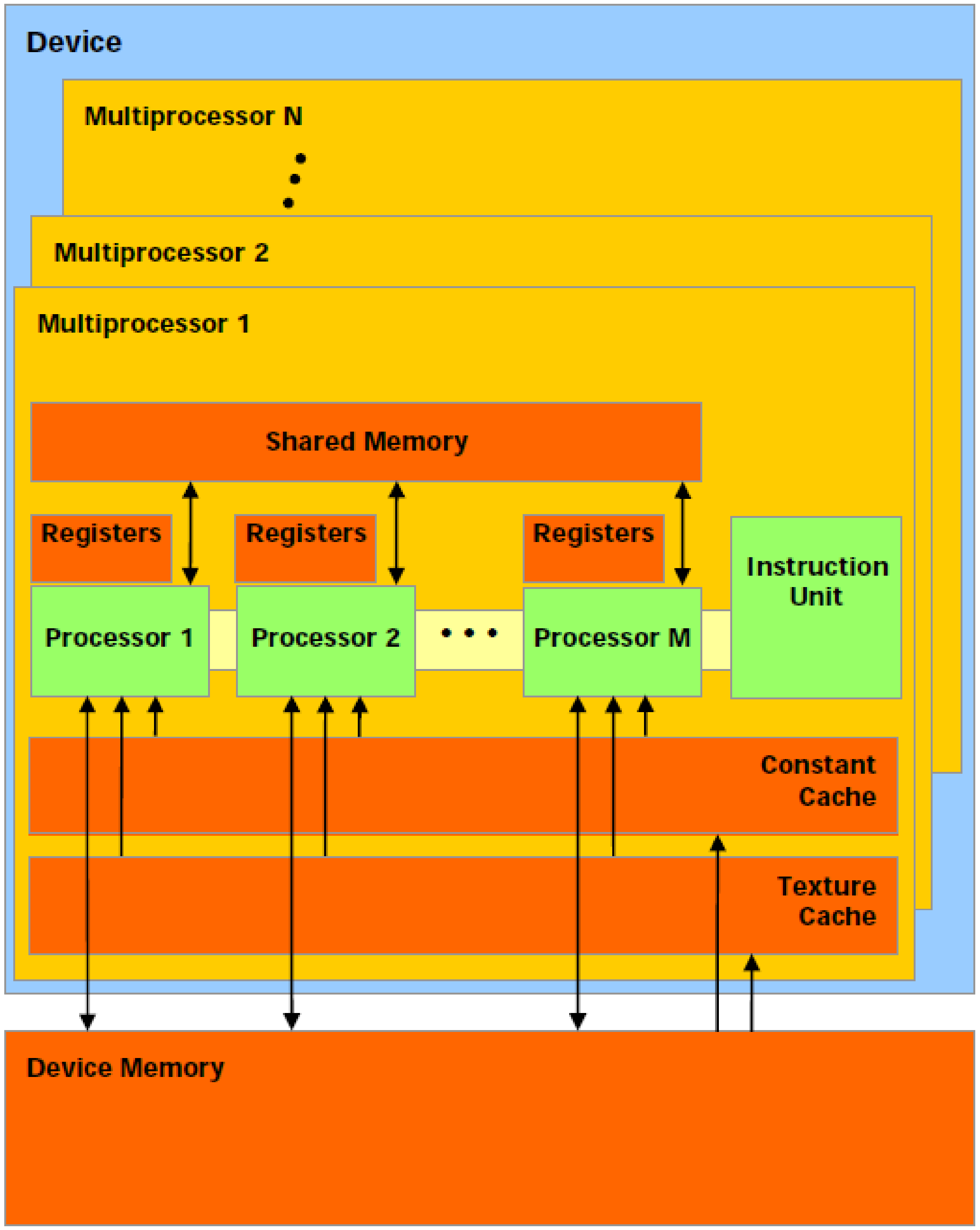
CPU’s are designed for *general purpose computations* requiring sophisticated control flow and caching mechanisms.

GPU’s are designed for *special purpose computations* with massive data-parallelism and high arithmetic intensity.

- Since the same program is executed for each data element there is a lower requirement of sophisticated flow control
- Because of high arithmetic intensity, the memory access latency can be hidden with calculations instead of big caches

So GPU’s can devote more transistors to data processing rather than data caching and flow control.

Architecture of a Modern GPU



Source: NVIDIA

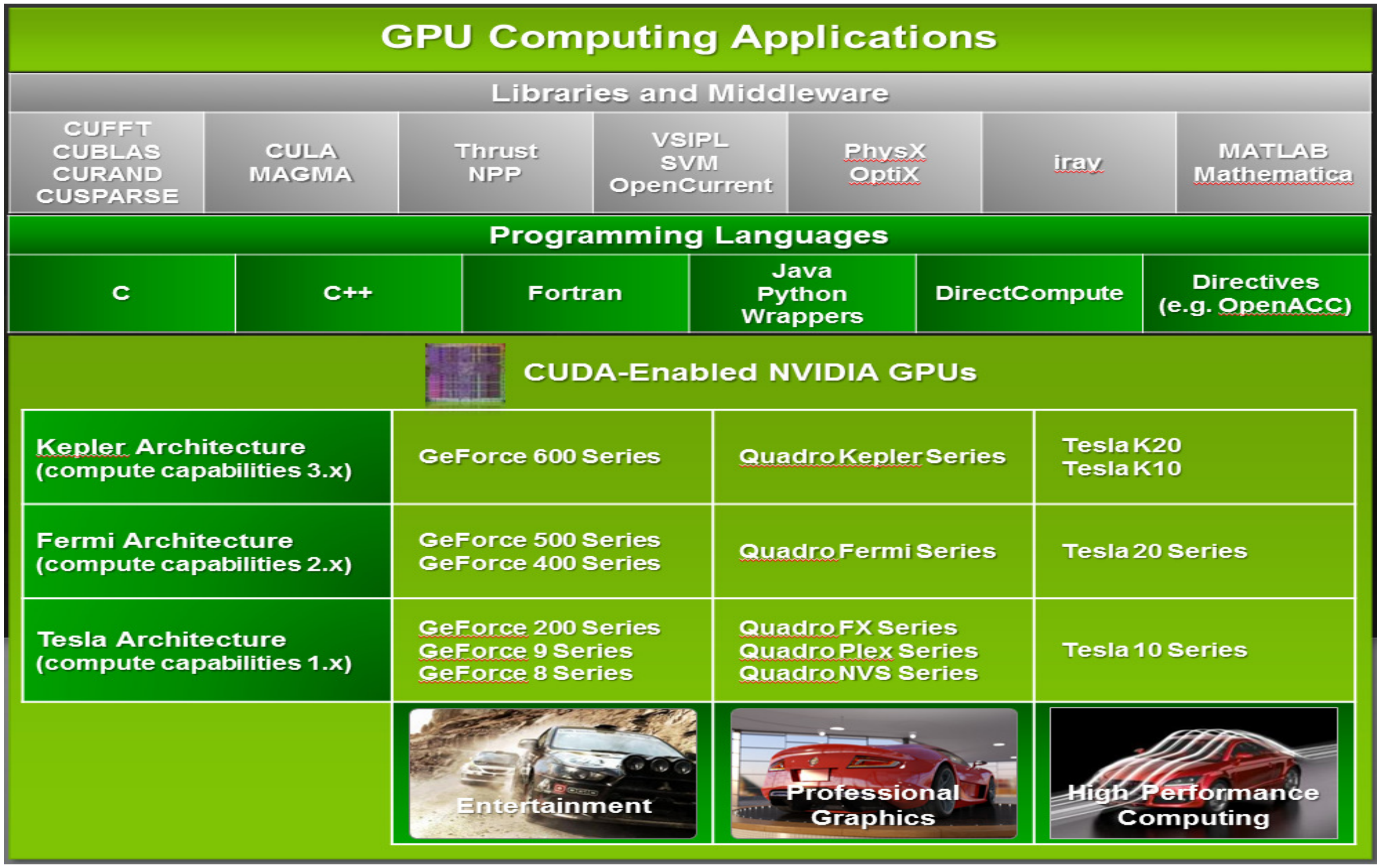
CUDA (Compute Unified Device Architecture)

A general purpose parallel computing architecture with

— a new parallel programming model, and

— instruction set architecture

that leverages the parallel compute engine in NVIDIA GPUs to solve data-parallel computations more efficiently than CPUs.



GPU Computing Applications
Libraries and Middleware
CUFFT CUBLAS CURAND CUSPARSE
CULA MAGMA
Thrust NPP
VSIPL SVM OpenCurrent
PhysX OptiX
iray
MATLAB Mathematica
Programming Languages
C
C++
Fortran
Java Python Wrappers
DirectCompute
Directives (e.g. OpenACC)
CUDA-Enabled NVIDIA GPUs
Kepler Architecture (compute capabilities 3.x)
GeForce 600 Series
Quadro Kepler Series
Tesla K20 Tesla K10
Fermi Architecture (compute capabilities 2.x)
GeForce 500 Series GeForce 400 Series
Quadro Fermi Series
Tesla 20 Series
Tesla Architecture (compute capabilities 1.x)
GeForce 200 Series GeForce 9 Series GeForce 8 Series
Quadro FX Series Quadro Plex Series Quadro NVS Series
Tesla 10 Series
Entertainment
Professional Graphics
High Performance Computing

Source: NVIDIA

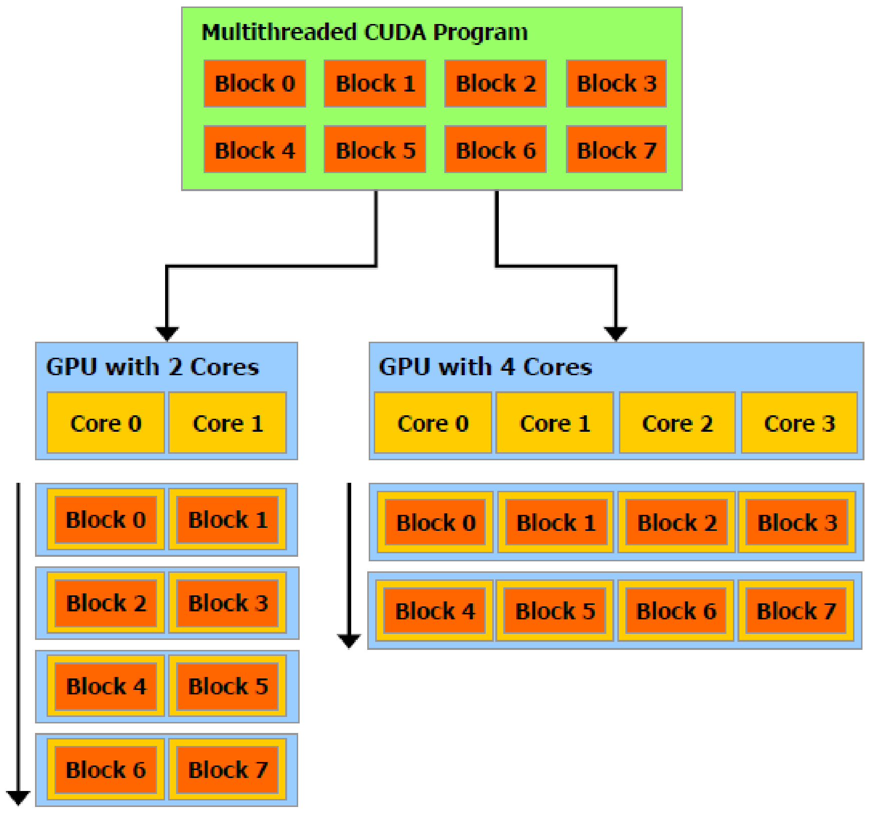
CUDA: a Scalable Programming Model

Three Key abstractions exposed as a minimal set of language extensions

- — A hierarchy of thread groups
- — Shared memories
- — Barrier synchronization

The programmer partitions

- — the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads
- — each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block



Source: NVIDIA

The thread blocks can be executed in any order — concurrently or sequentially — leading to automatic scalability.

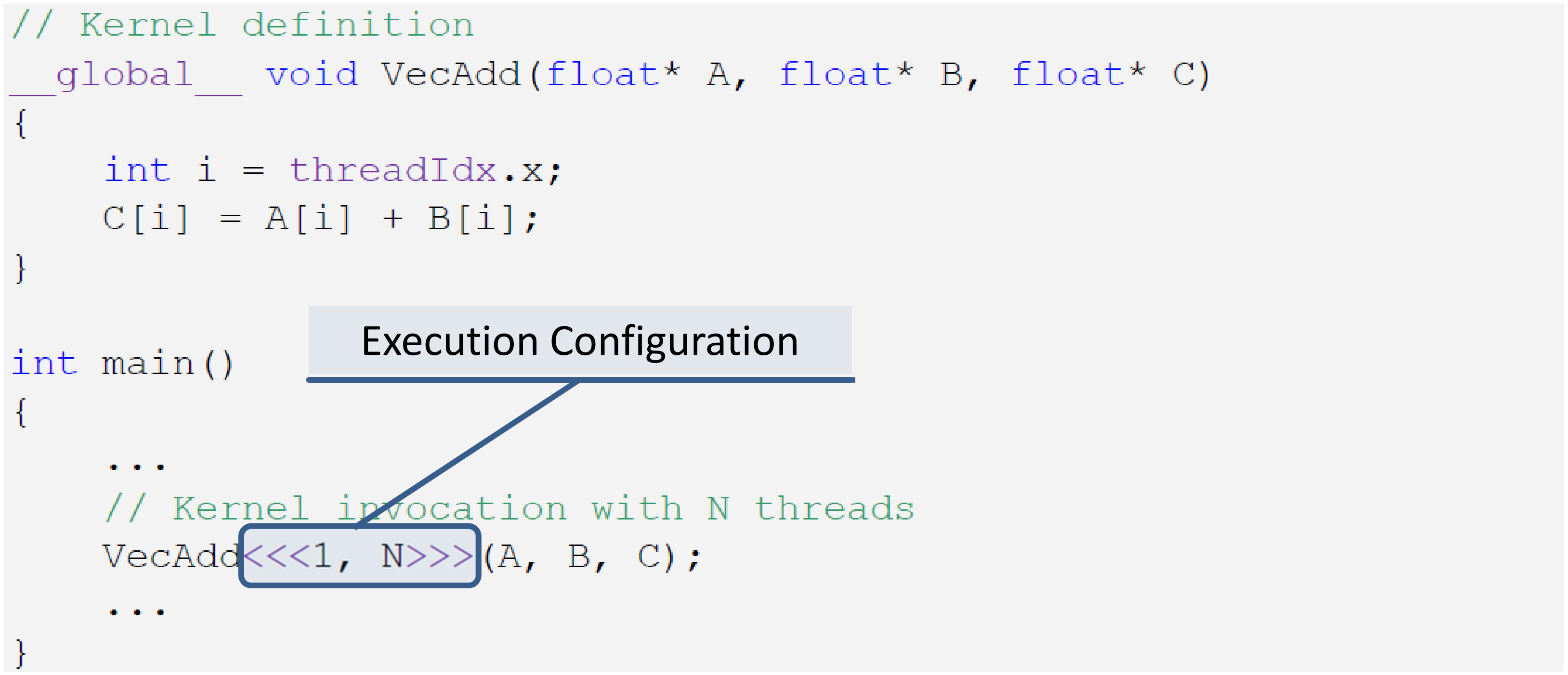
Differences between CPU and CUDA Threads

- CUDA threads are extremely lightweight compared to CPU threads
    - Only a few cycles to create
    - Instant switching
- CUDA runs thousands of threads while CPU’s run only a few

CUDA Extensions to C Functional Declarations

	Executed on the:	Only callable from the:
__device__ float DeviceFunc( )	device	device
__global__ void KernelFunc( )	device	host
__host__ float HostFunc( )	host	host

Kernel Functions



```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Source: NVIDIA

— Called from host (CPU)

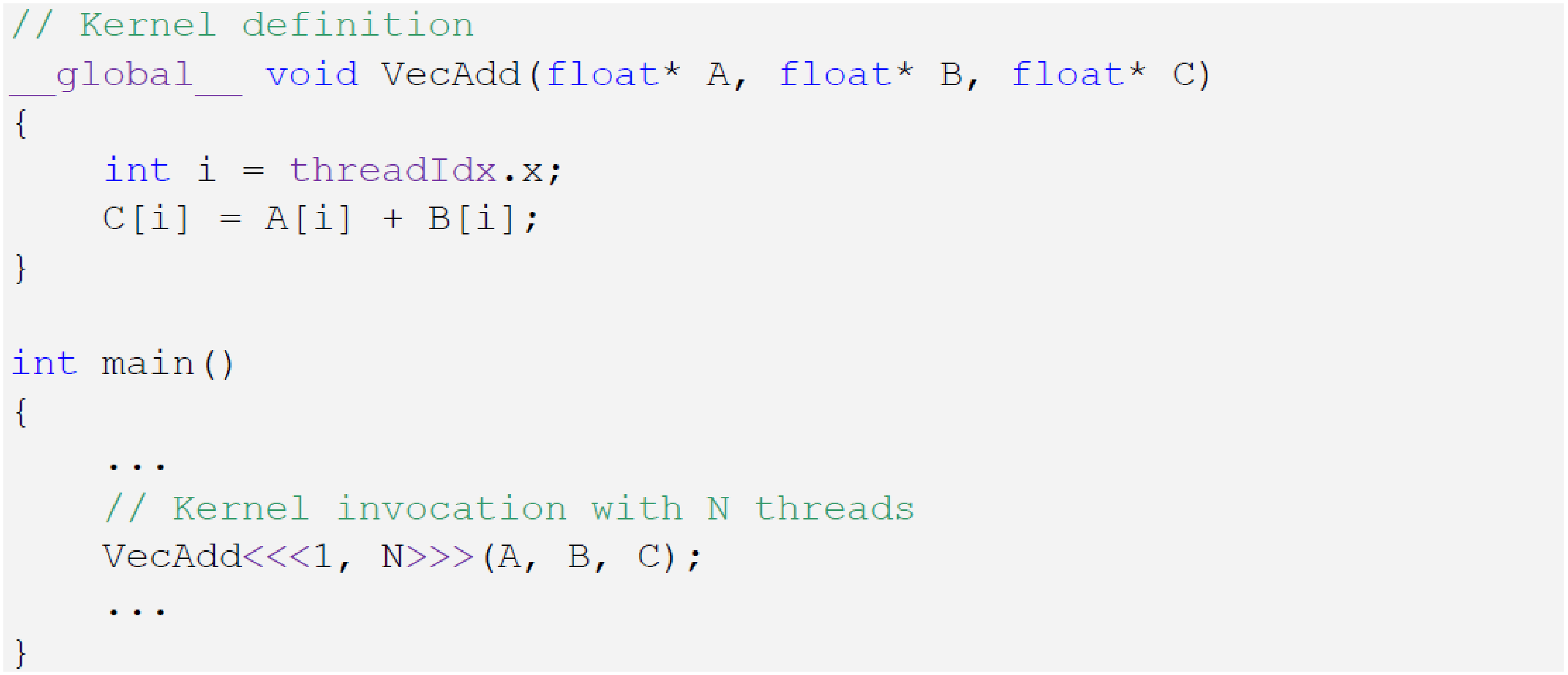
— Executed on device (GPU)

— Only one kernel runs at a time (for compute capability < 2.0)

— All running threads execute the same kernel (except above)

— All kernel launches are asynchronous (control returns to the CPU immediately)

Kernel Functions (Restrictions)



```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Source: NVIDIA

— Must return *void*

— Variable number of arguments (i.e., *varargs*) not allowed

— No static variables

— No access to host memory

— Must be non-recursive

Thread Hierarchy: Thread Index

Threads can be identified using a 1, 2 or 3 dimensional *thread index* forming a 1, 2 or 3 dimensional *thread block*.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Source: NVIDIA

Thread Hierarchy: Block Index

Blocks can be identified using a 1, 2 or 3 dimensional *block index* forming a 1, 2 or 3 dimensional *grid*.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Source: NVIDIA

Thread Hierarchy: Grids, Blocks and Threads

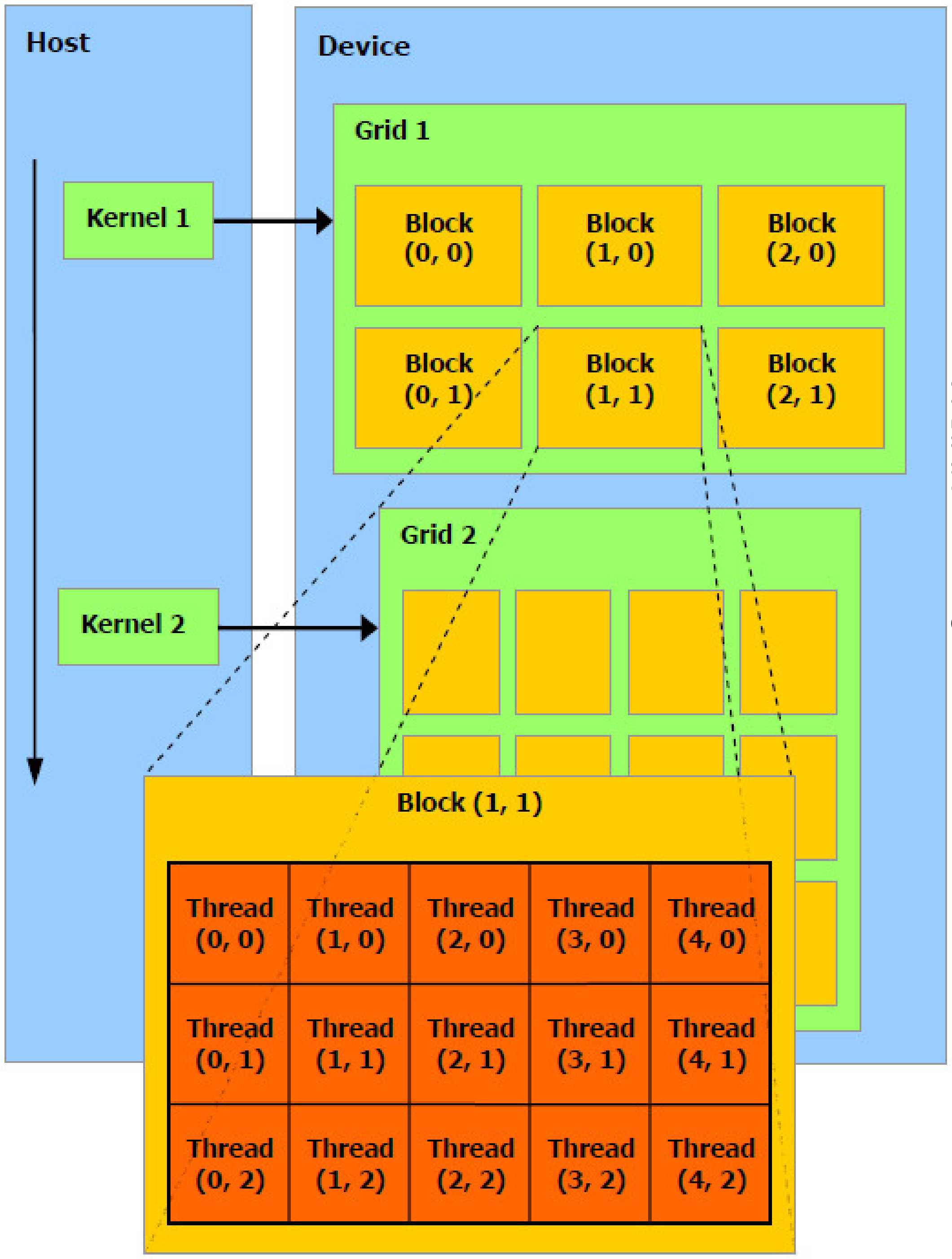
All __device__ and __global__ functions have access to the following built-in device variables

— ***dim3 gridDim***: dimenions of the grid in blocks

— ***dim3 blockDim***: dimenions of a block in threads

— ***dim3 blockIdx***: block index within the grid

— ***dim3 threadIdx***: thread index within a block



Source: NVIDIA

CUDA Memory Model

Registers

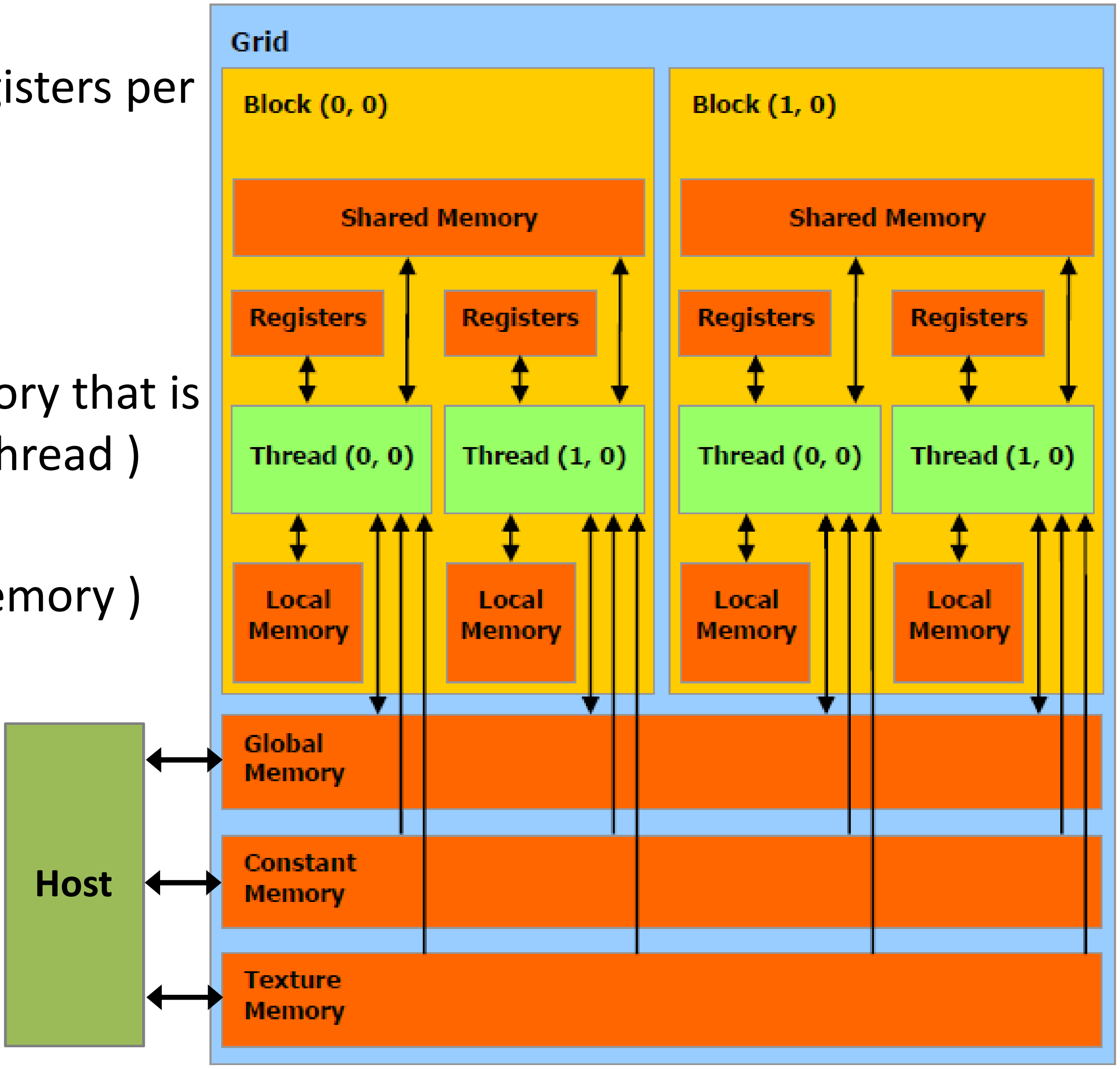
- Very large number of registers per processor (thread)
- Instant access

Local Memory

- A portion of global memory that is private to a processor (thread)
- Used for register spills
- Slow (same as global memory)

Shared Memory

- A small (e.g., 16 KB) block of memory shared by all processors (threads) in a multi -processor (block)
- Divided into Several memory banks
- As fast as registers w/o bank conflicts



Source: NVIDIA

CUDA Memory Model

Global Memory

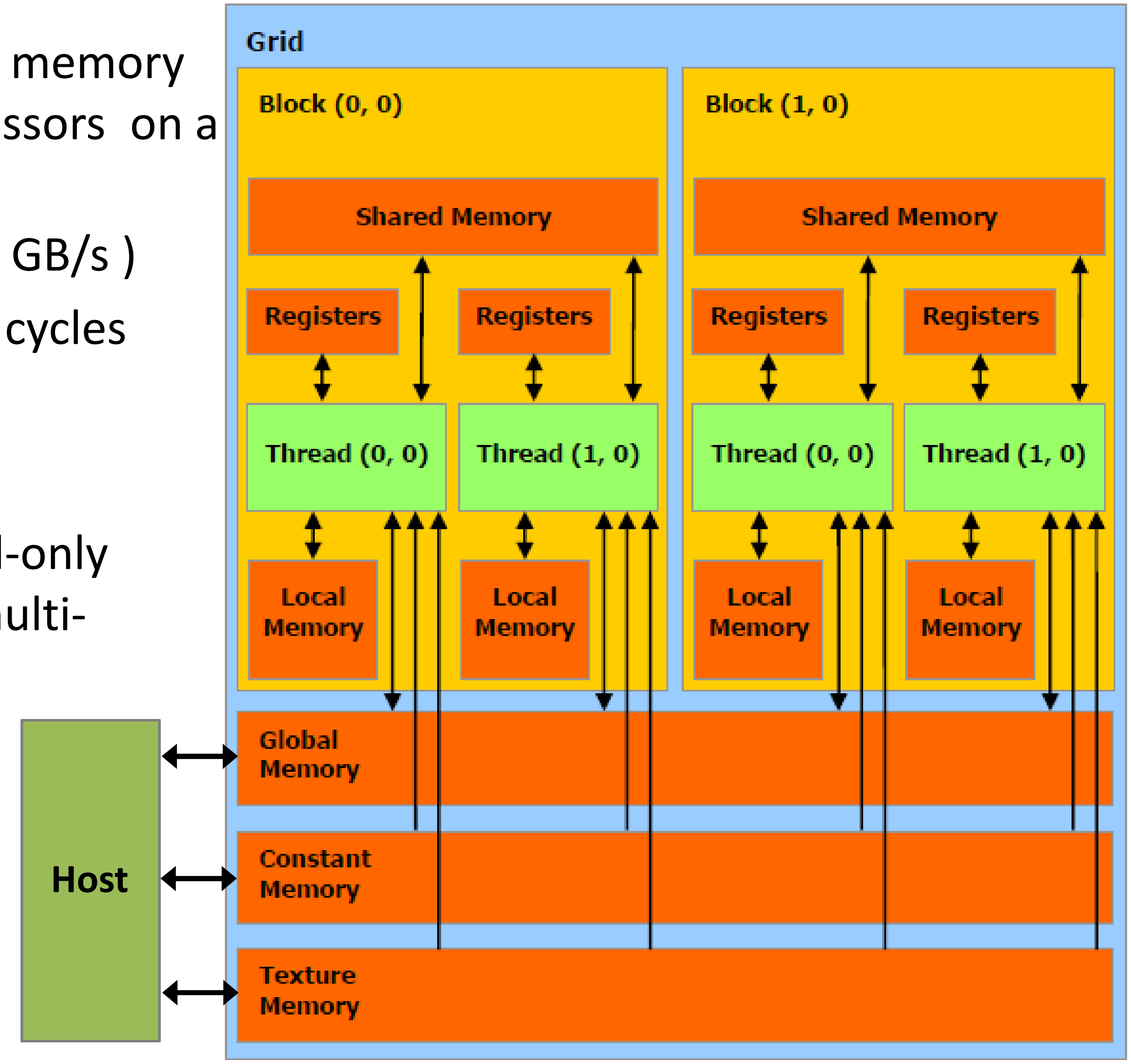
- A large block (in GB) of memory shared by all multiprocessors on a GPU
- High bandwidth (> 100 GB/s)
- Slow (several 100 clock cycles when not cached)

Constant Memory

- Small (e.g., 64 KB) read-only memory shared by all multi-processors
- Cached (per multi -processor)
- Slow (several 100 clock cycles on cache miss)

Texture Memory

- Similar to constant memory
- Reads can be samplings (e.g., nearest point of interpolation)



Source: NVIDIA

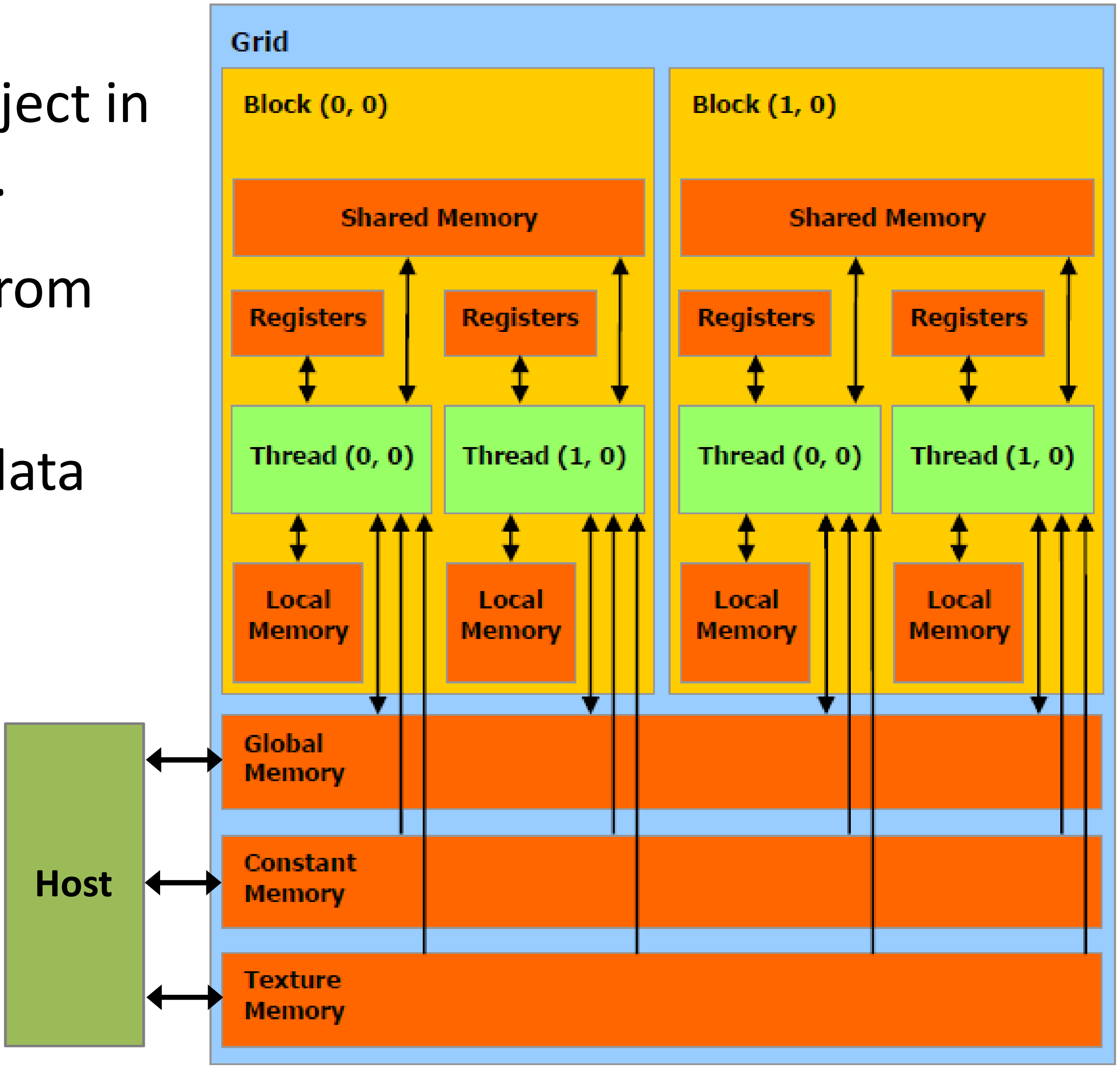
CUDA Memory Model

cudaMalloc( ): allocates object in the devices global memory.

cudaFree( ): frees objects from device global memory.

cudaMemcpy( ): memory data transfer:

- — host to host
- — host to device
- — device to host
- — device to device



Source: NVIDIA

Synchronization

For the following tasks control is returned to the host before the device completes the task

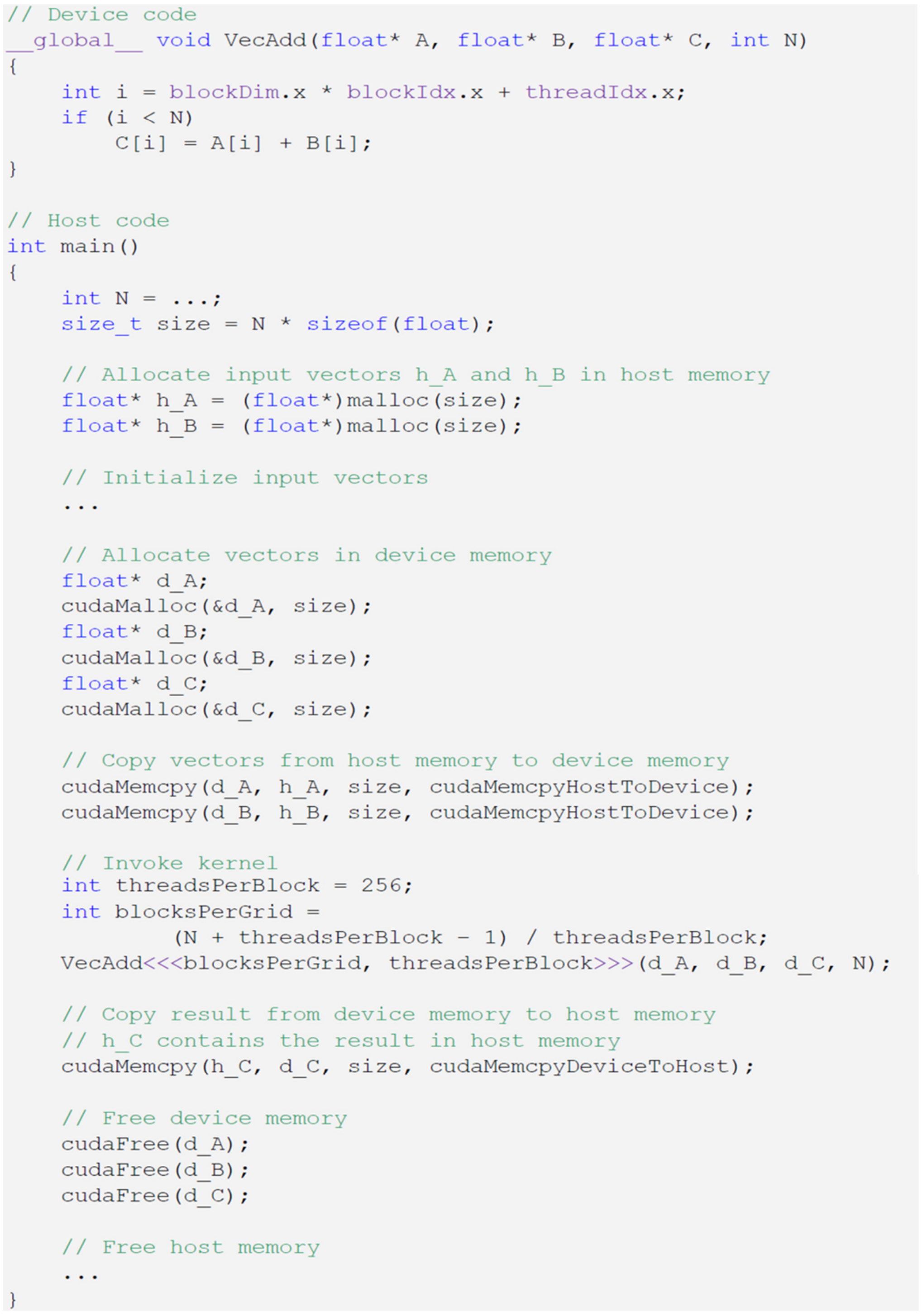
- — Kernel launches
- — Memory copies between two addresses on the same device
- — Memory copies of size 64KB or less from host to device
- — Memory copies by functions suffixed with *Async*
- — Memory set function calls

However, kernel launches and cudaMemcpy can start only after all previous CUDA calls have completed.

cudaDeviceSynchronize( ): blocks until the device has completed all previously requested tasks

__syncthreads( ): synchronize all threads in a block

Example: CUDA Memory Functions



```
// 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
    ...
}
```

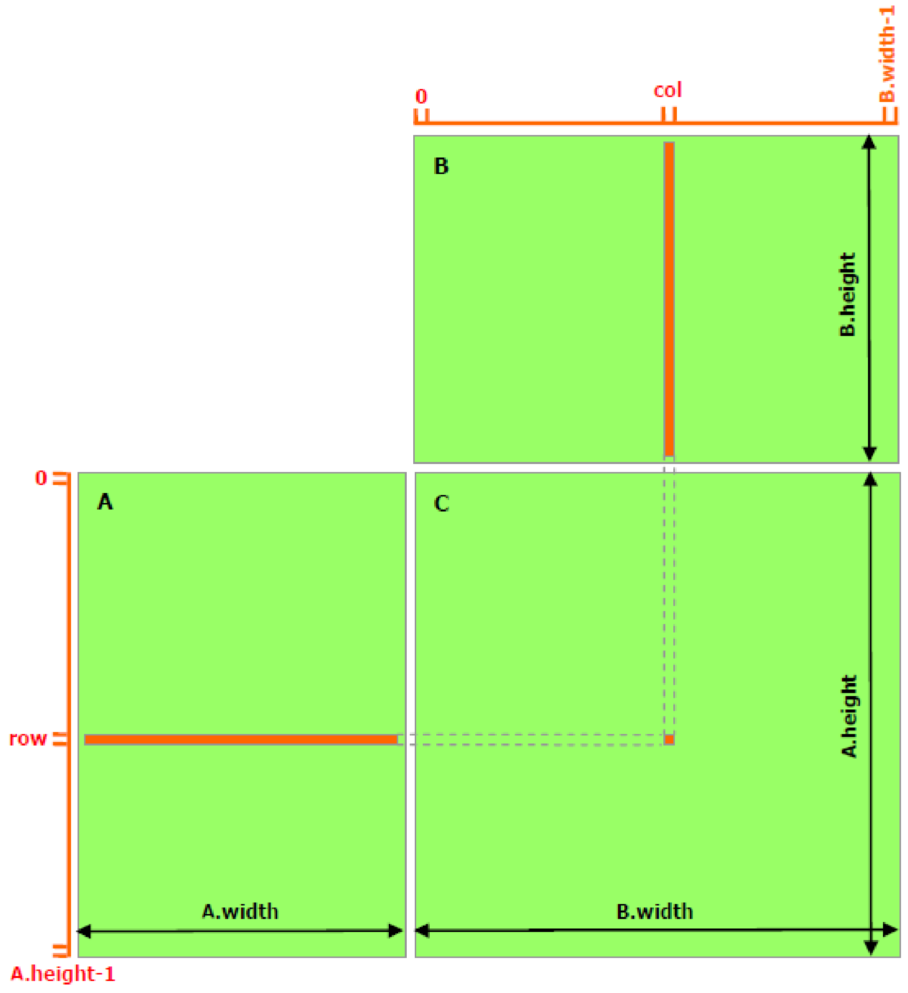
Source: NVIDIA

CUDA Variable Type Qualifiers

	Memory	Scope	Lifetime
automatic variables other than arrays	register	thread	kernel
automatic array variables	local	thread	kernel
__device__	global	grid	application
__shared__	shared	block	kernel
__constant__	constant	grid	application

Matrix Multiplication w/o Shared Memory

```
// 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);
}
```

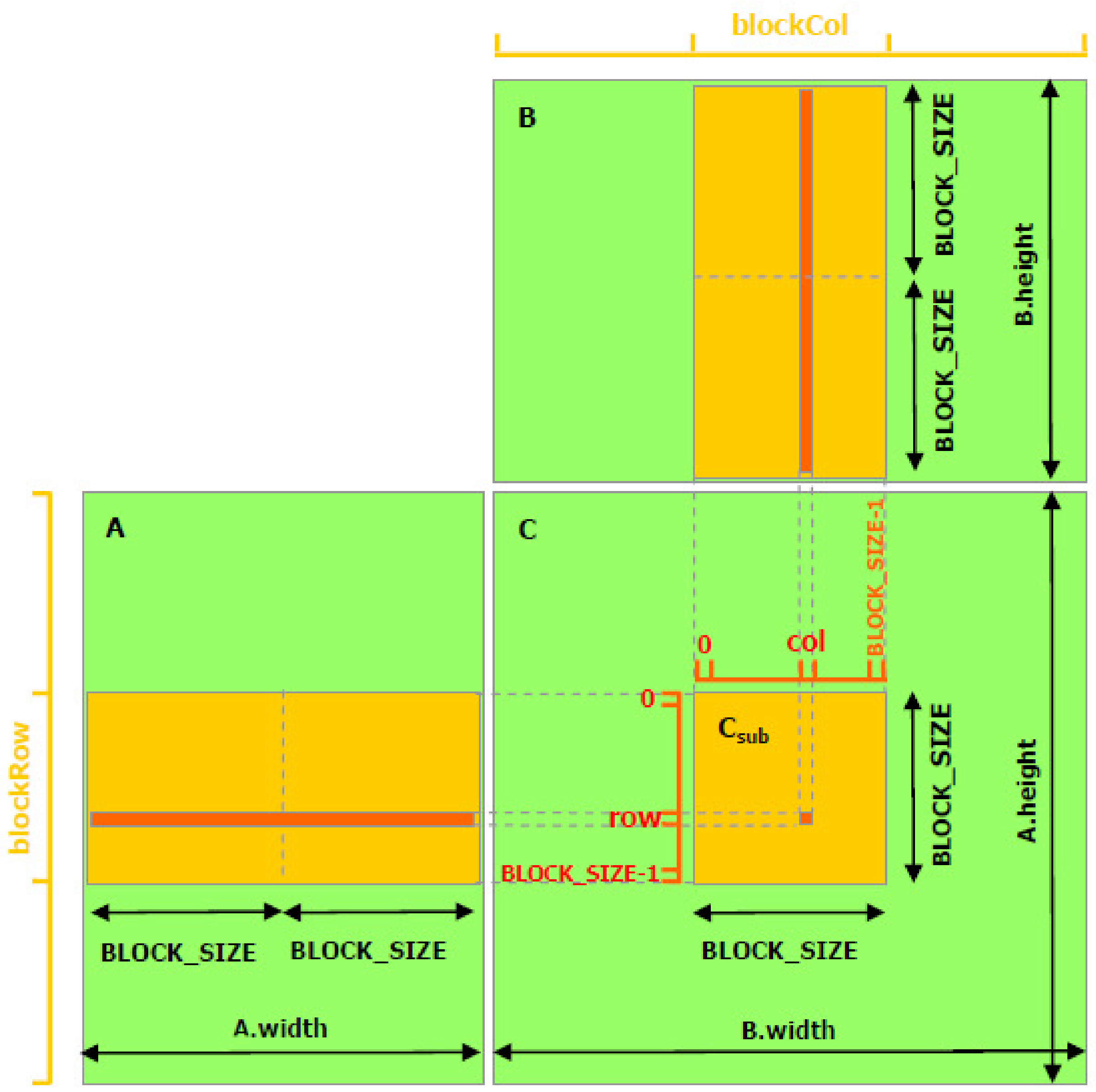


Source: NVIDIA

```
// 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;
}
```

Source: NVIDIA

Matrix Multiplication with Shared Memory



Source: NVIDIA

Matrix Multiplication with Shared Memory

```
// 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);
```

Source: NVIDIA

Matrix Multiplication with Shared Memory

```
    // 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);
}
```

Source: NVIDIA

Some Optimization Tips

— Increase data parallelism

— Keep resource usage (e.g., registers, shared memory) low enough to allow multiple warps per multiprocessor

— Increase arithmetic intensity

— Recompute on device to avoid costly host to device data transfers

— Use the fast shared memory more than the slow global memory

— Increase coalesced accesses to global memory

— Avoid bank conflicts in shared memory

— Improve spatial locality for cached memory

— One large data transfer is much faster than many small transfers