Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative

SUPSI

Programming CUDA

Tiziano Leidi

Sviluppo di Applicazioni High-Performance
Bachelor in Ingegneria Informatica

C for CUDA

CUDA provides several additions to the C/C++ environment:

- C language extensions to enable heterogeneous programming
- mathematical functions of the C/C++ standard library (for both host and device)
- intrinsic functions (only supported on the device)
- CUDA runtime API and CUDA driver API
- Environment variables

C/C++ for the device code has some limitations compared to the full standard. For example it does not support run time type information (RTTI) and exception handling.

Example: Device Code

A simple kernel to add two vectors of integers:

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

CUDA Kernel

- The function type qualifier __global__ declares a function as being an executable kernel on the CUDA device.
- This function can only be called from the host.
- All kernels must be declared with a return type of void.

Example: Host Code

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                // host copies of a, b, c
    int *d_a, *d_b, *d_c;          // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c
    // and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Example: Host Code

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Function Type Qualifiers

The available function type qualifiers are:

__device__

- Executed on the device
- Callable from the device only

__global__

- Executed on the device
- Callable from the host only

__host__

- Executed on the host
- Callable from the host only

Calling the Kernel

- The host calls the kernel by specifying the name of the kernel plus an execution configuration:

```
Func<<< Dg, Db, Ns >>>(parameter);
```

- The configuration includes three things:
  - The number of threads in a group - the block geometry
  - The number of groups - the grid geometry
  - The number of bytes in shared memory that is dynamically allocated (optional) in addition to the statically allocated memory

dim3

- To pass the grid and block dimensions in a kernel invocation an integer vector type is used: dim3.
- dim3 has 3 elements x, y and z.
- In C code, dim3 can be initialized as:

```
dim3 grid = { 512, 512, 1 };
```

- In C++ code, dim3 can be initialized as:

```
dim3 grid( 512, 512, 1 );
```

- Not all the 3 elements need to be provided. Any element not provided during initialization is initialized to 1.

Calling the Kernel

Dimensions:

- *Dg* is of type dim3:
  Dg.x * *Dg.y* = number of blocks being launched;
- *Db* is of type dim3:
  Db.x * *Db.y* * *Db.z* = number of threads per block;
- *Ns* is of type size_t.

Built-in variables:

- *gridDim* is of type dim3 - dimensions of the grid.
- *blockIdx* is of type uint3 - block index within the grid.
- *blockDim* is of type dim3 - dimensions of the block.
- *threadIdx* is of type uint3 - thread index within the block.

Threads and Blocks

Thread 0

```
c[0]  = a[0] + b[0];
```

Thread 1

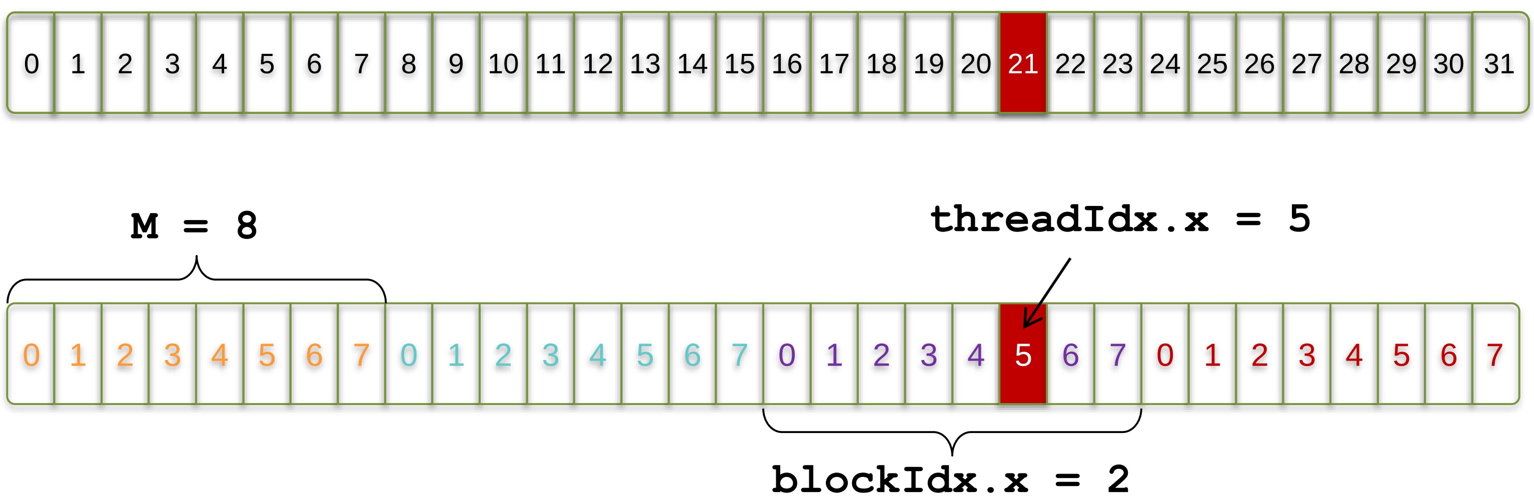
```
c[1]  = a[1] + b[1];
```

Thread 2

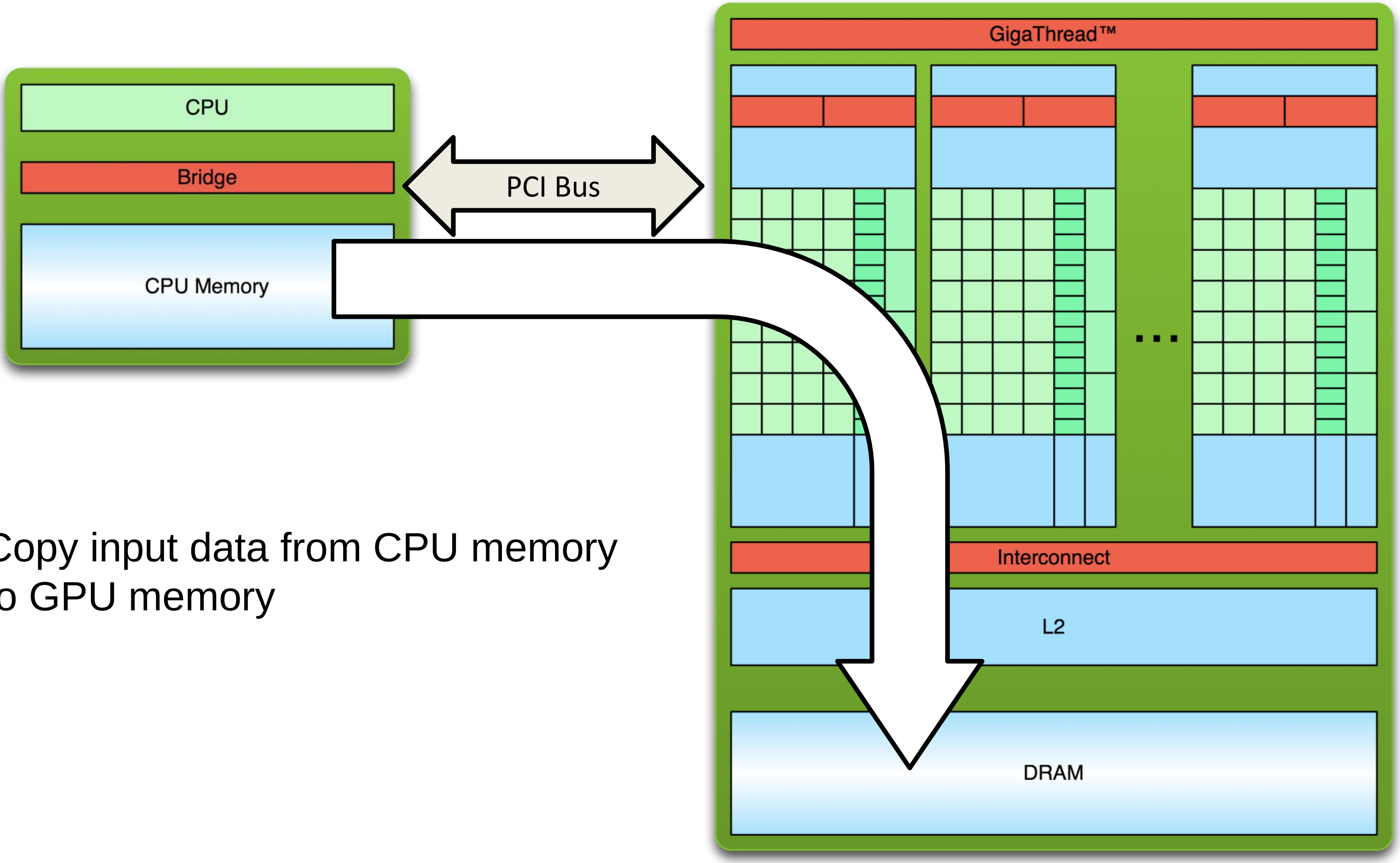
```
c[2]  = a[2] + b[2];
```

Thread 3

```
c[3]  = a[3] + b[3];
```

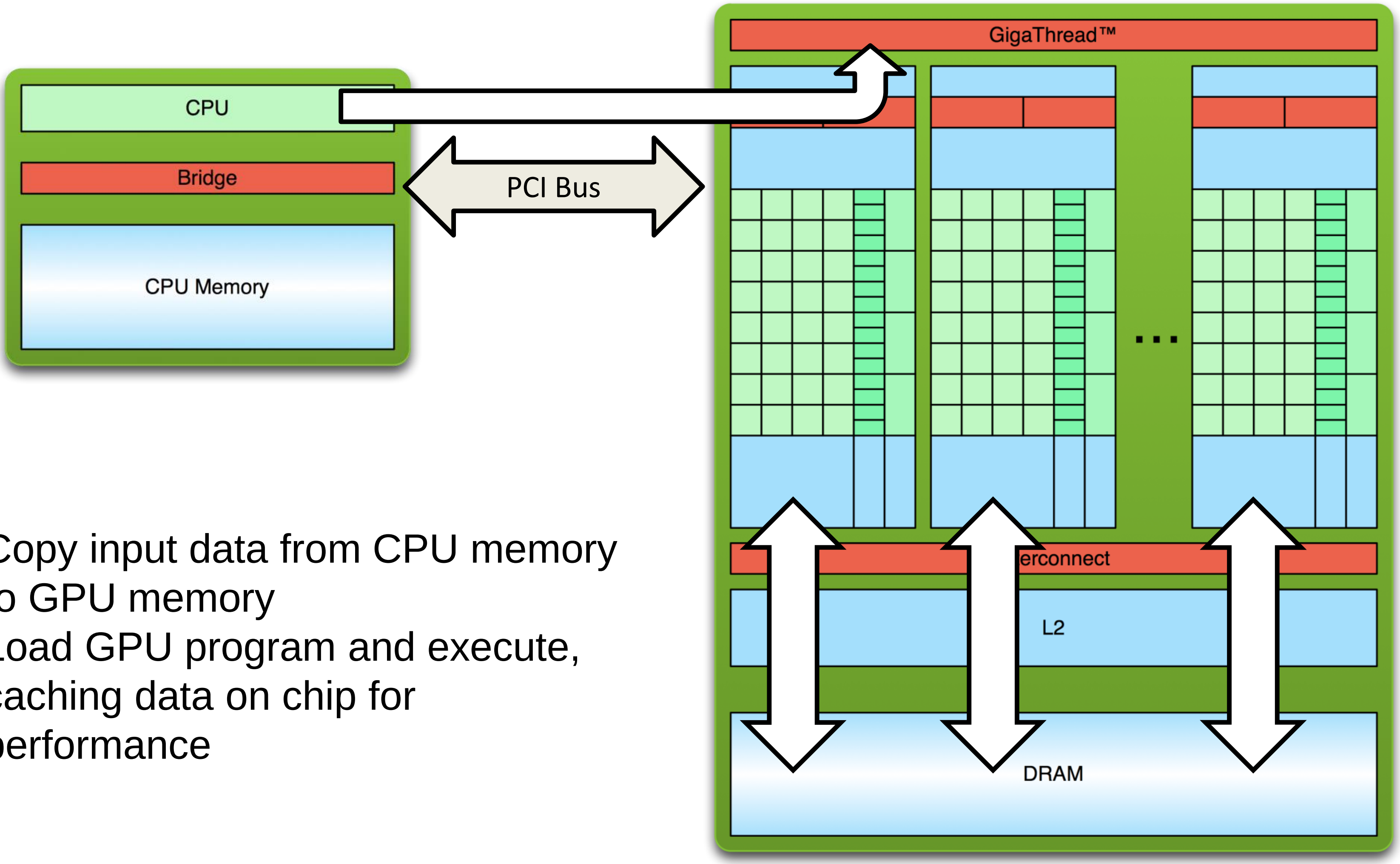


Processing Flow



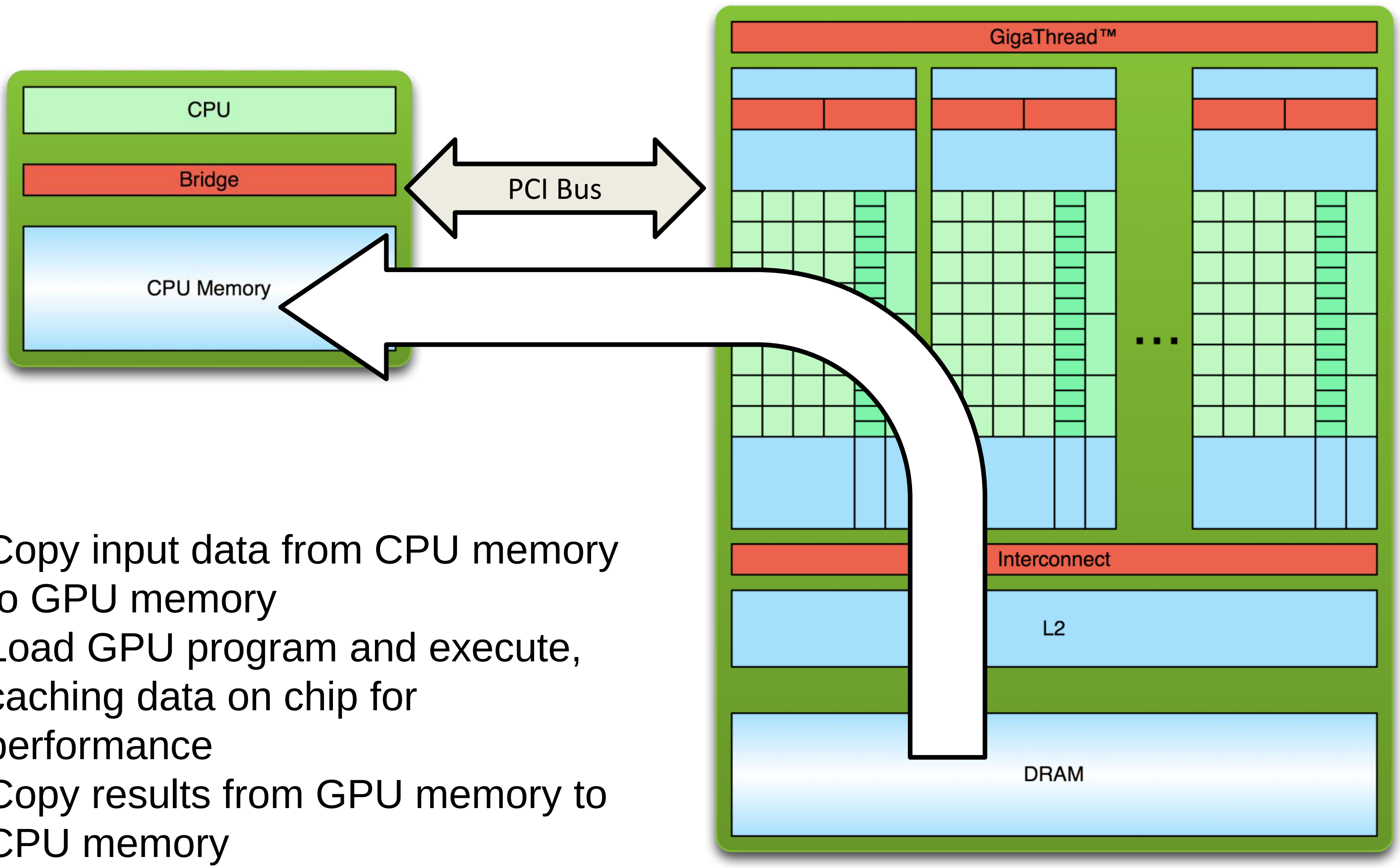
1. Copy input data from CPU memory to GPU memory

Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Processing Flow



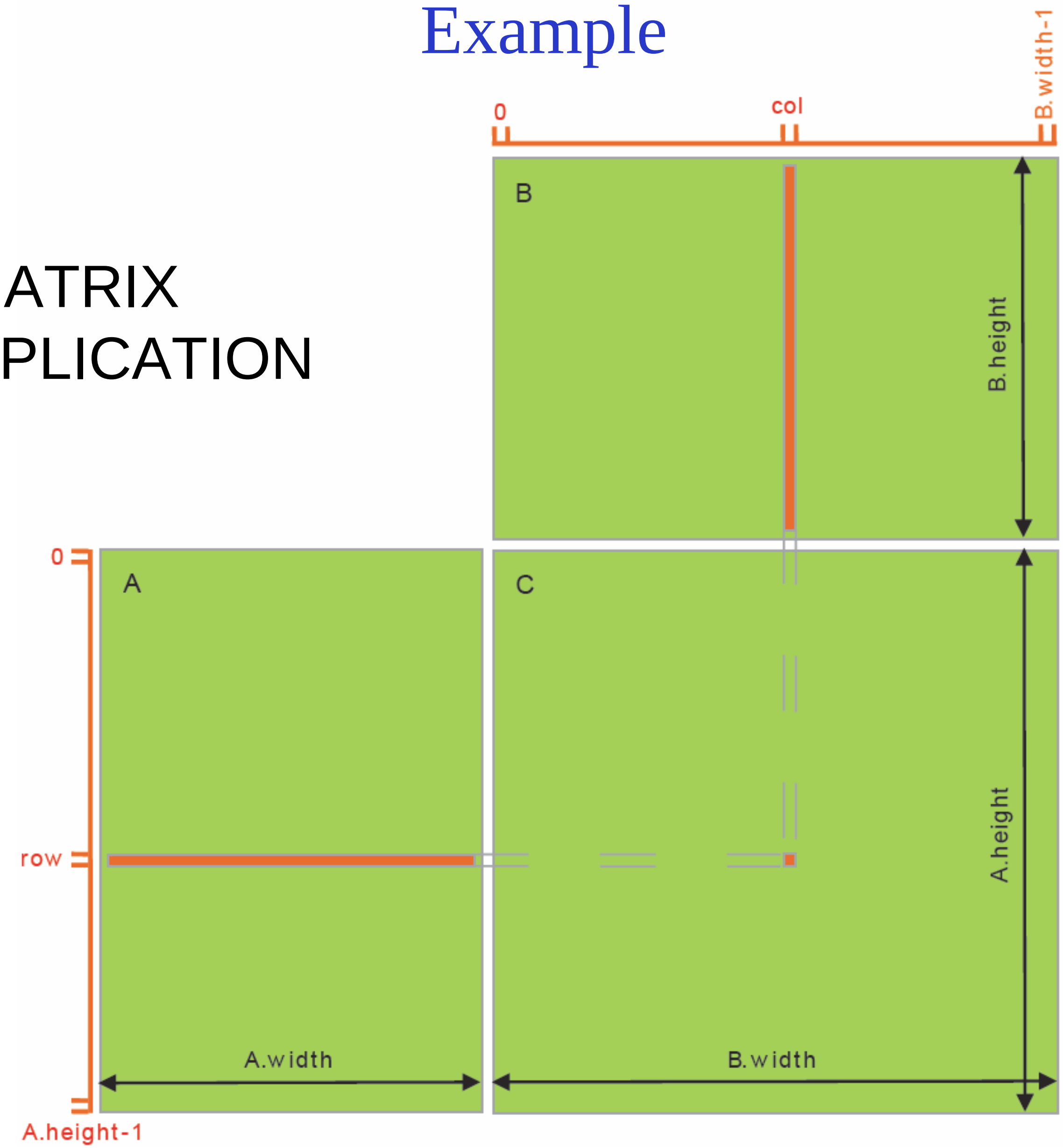
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Compilation

- CUDA C language programs have the suffix ".cu".
- Can be compiled directly to executable or to object code.
- Can be later linked to objects created by different compilers (e.g. integrated into visual studio).

Example

MATRIX MULTIPLICATION



Example

```
// 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) {
```

Example

```
// 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);
```

Example

```
    // 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);
}
```

Example

```
// 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;
}
```

Memory Management

CUDA provides a simple API for handling device memory:

- cudaMalloc(), cudaFree(), cudaMemcpy()
- similar to the C equivalents malloc(), free(), memcpy()

Host and device memory are separate entities.

Device pointers point to GPU memory:

- May be passed to/from host code
- May not be dereferenced in host code

Host pointers point to CPU memory:

- May be passed to/from device code
- May not be dereferenced in device code

Variable Type Qualifiers

The available variable type qualifiers are:

__device__

- Resides in global memory space
- Has the lifetime of an application
- Is accessible from all the threads within the grid and from the host through the runtime library

__constant__

- Resides in constant (global) memory space
- Has the lifetime of an application
- Is accessible from all the threads within the grid and from the host through the runtime library

Variable Type Qualifiers

__shared__

- Resides in the shared memory space of a thread block
- Has the lifetime of the block
- Is only accessible from all the threads within the block

Register Memory

Registers are:

- The fastest form of memory on the GPU
- Reside on the multi-core
- Only accessible by the thread
- Have the lifetime of the thread

Shared Memory

Shared memory:

- Can be as fast as a registers when there are no bank conflicts or when reading from the same address
- Accessible by any thread of the block from which it was created
- Has the lifetime of the block

Global Memory

Global memory:

- Potentially 150x slower than register or shared memory
- Watch out for uncoalesced reads and writes
- Accessible from either the host or device
- Has the lifetime of the application

Local Memory

Local memory:

- A potential performance bottleneck
- Resides in global memory
- Can be 150x slower than register or shared memory
- Only accessible by the thread
- Has the lifetime of the thread

Used automatically by CUDA for:

- register overflow
- arrays addressed by parameters in register memory
- possibly misaligned global memory reads

Constant Memory

Constant memory:

- Cached access to global memory
- Read-only
- Limited to 64kb for the application
- Shared by all threads
- Shared by instruction cache
- Cache is 8kb

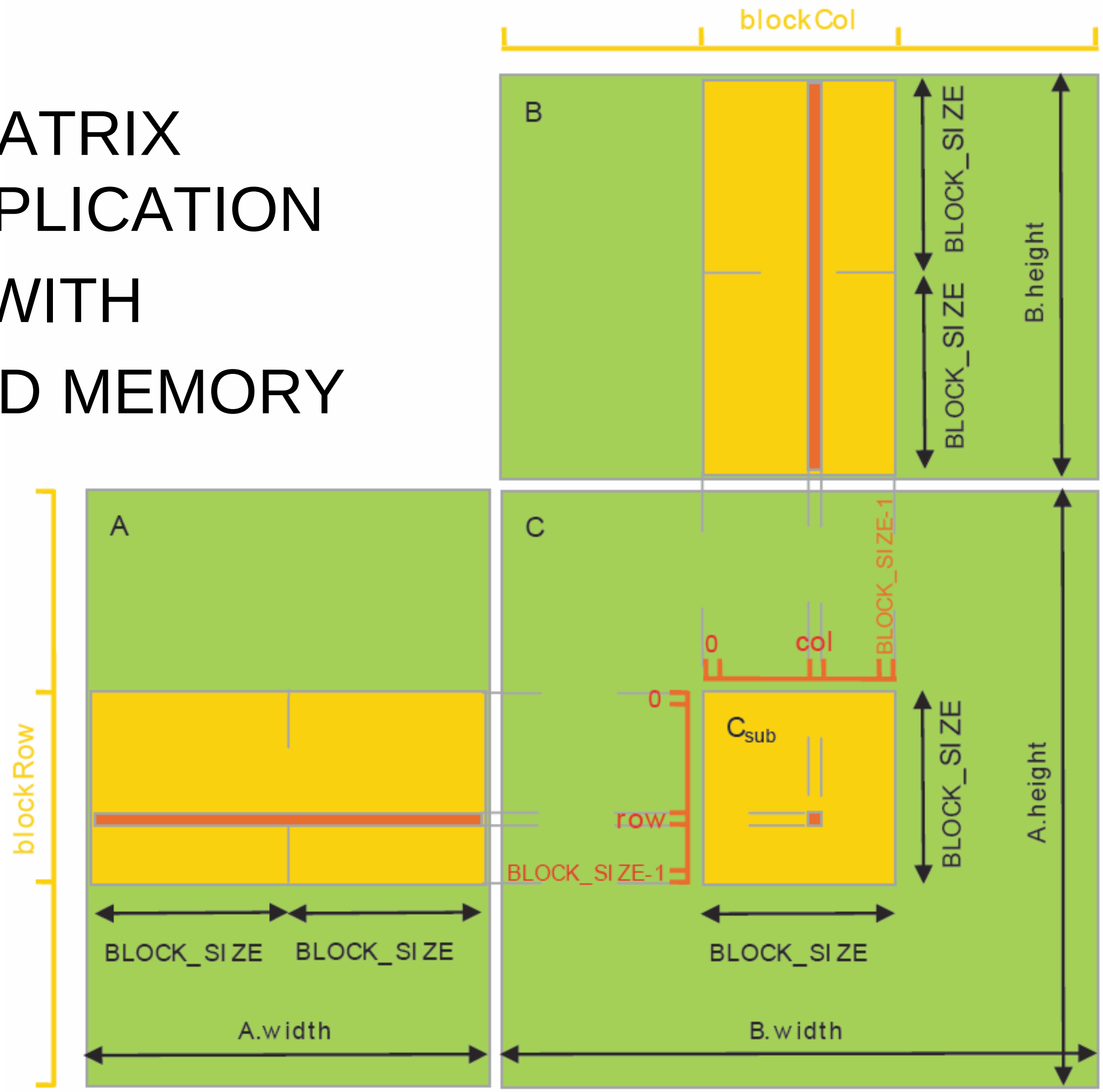
Texture Memory

Texture memory:

- Cached access to global memory
- Read-only
- L1 shared by multiple processors (depending on
- generation), L2 by all threads
- Optimized for 2D access
- L1 cache is 6kb to 8kb (depending on processor)

Example

MATRIX
MULTIPLICATION
WITH
SHARED MEMORY



B.width = stride

Example

```
// 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;
}
```

Larghezza della matrice di partenza, necessaria per andare a capo anche nel caso delle sottomatrici

Example

```
 // 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);
```

Example

```
// 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);
```

Example

```
    // 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);
}
```

Example

```
// 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;
```

Example

```
// 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();
```

Example

```
    // 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);
}
```

Syncthreads

- Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses.
- Synchronization points are specified in the kernel by calling *__syncthreads()*: a barrier at which all threads in the block must wait before any is allowed to proceed.
- For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core and *__syncthreads()* is expected to be lightweight.