Introduction into Multicore Programming

Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation
(RISC)

Short Introduction into GPU Programming in CUDA

Introduction

- GPUs were originally hardware blocks optimized for a small set of graphics operations.
- For speed up graphics related computation, the (very efficient) data parallelism was introduced on the GPUs.
- As it was demanded, GPUs became gradually more programmable for general purposes.
- In late 2006, NVIDIA introduced its CUDA architecture and tools to make data parallel computing on a GPU more straightforward.
- **CUDA** (an acronym for **Compute Unified Device Architecture**) is a parallel computing architecture developed by NVIDIA.

GPU Hardware

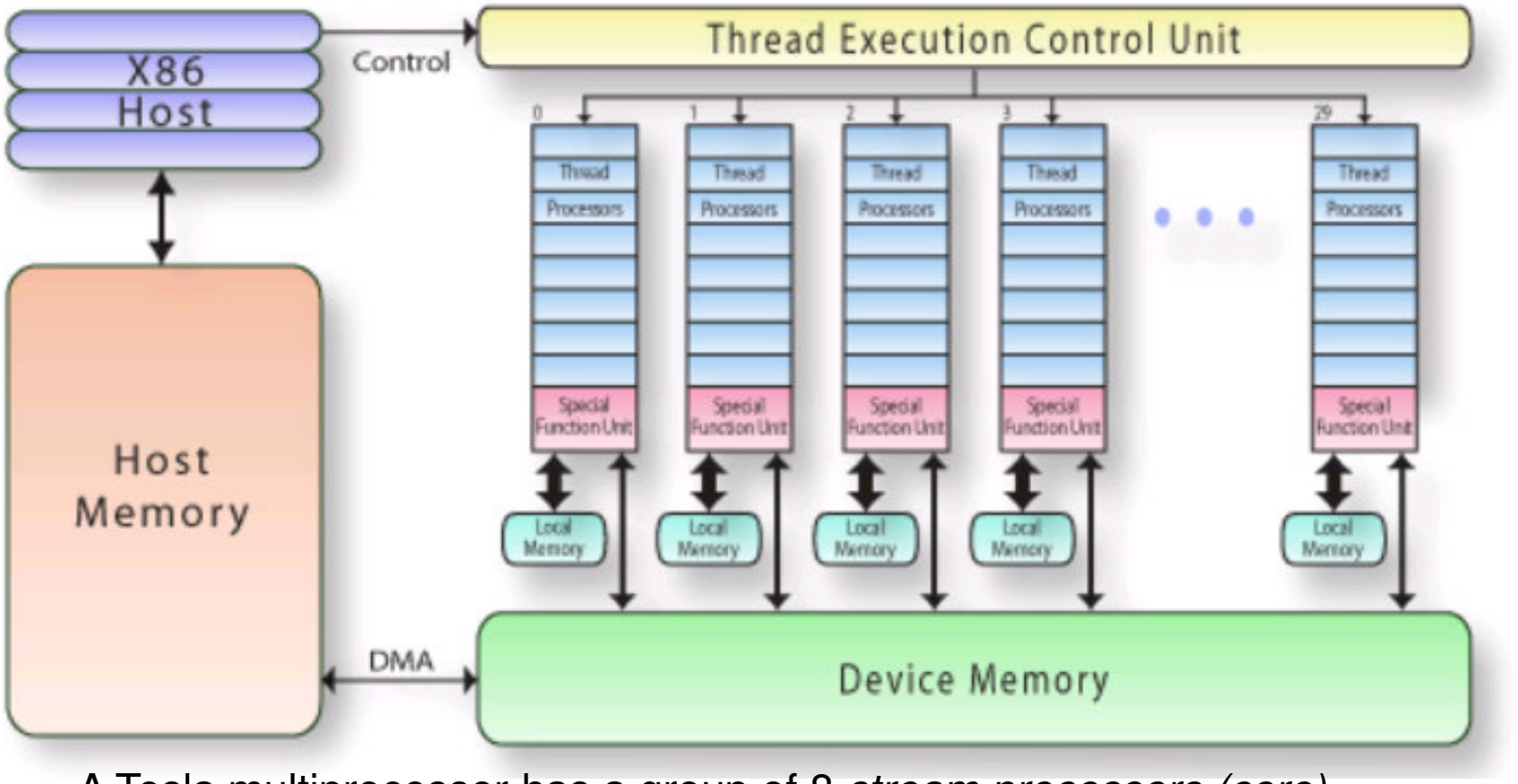
- A GPU is connected to a host through a high speed IO bus slot (typically a PCI-Express).
- The GPU has its **own device memory**, up to several gigabytes.
- Data is usually transferred between the GPU and host memories using programmed DMA.
- DMA can operate concurrently with both the host and GPU compute units (support for direct access to host memory from the GPU under certain restrictions).
- The device memory supports very **high data bandwidth** using a **wide data path**.

NVIDIA GPUs

- NVIDIA GPUs have a number of *multiprocessors*…
  …each of them executes in parallel with the others.
- each multiprocessor has a group of (8-16) *stream processors (core).*
- Each core can execute a sequential thread, …
- …but the cores execute in *SIMT (Single Instruction, Multiple Thread)* fashion;
- All cores in the same group execute the same instruction at the same time.
- The code is actually executed in groups of 32 threads, what NVIDIA calls a *warp (indivisible unit of processing*).
- There is also a small software-managed data cache attached to each multiprocessor, shared among the cores(*shared memory*).
  - This is a low-latency, high-bandwidth, indexable memory which runs essentially **at register speeds**.

NVIDIA Tesla Architecture

Karoly.Bosa@jku.at

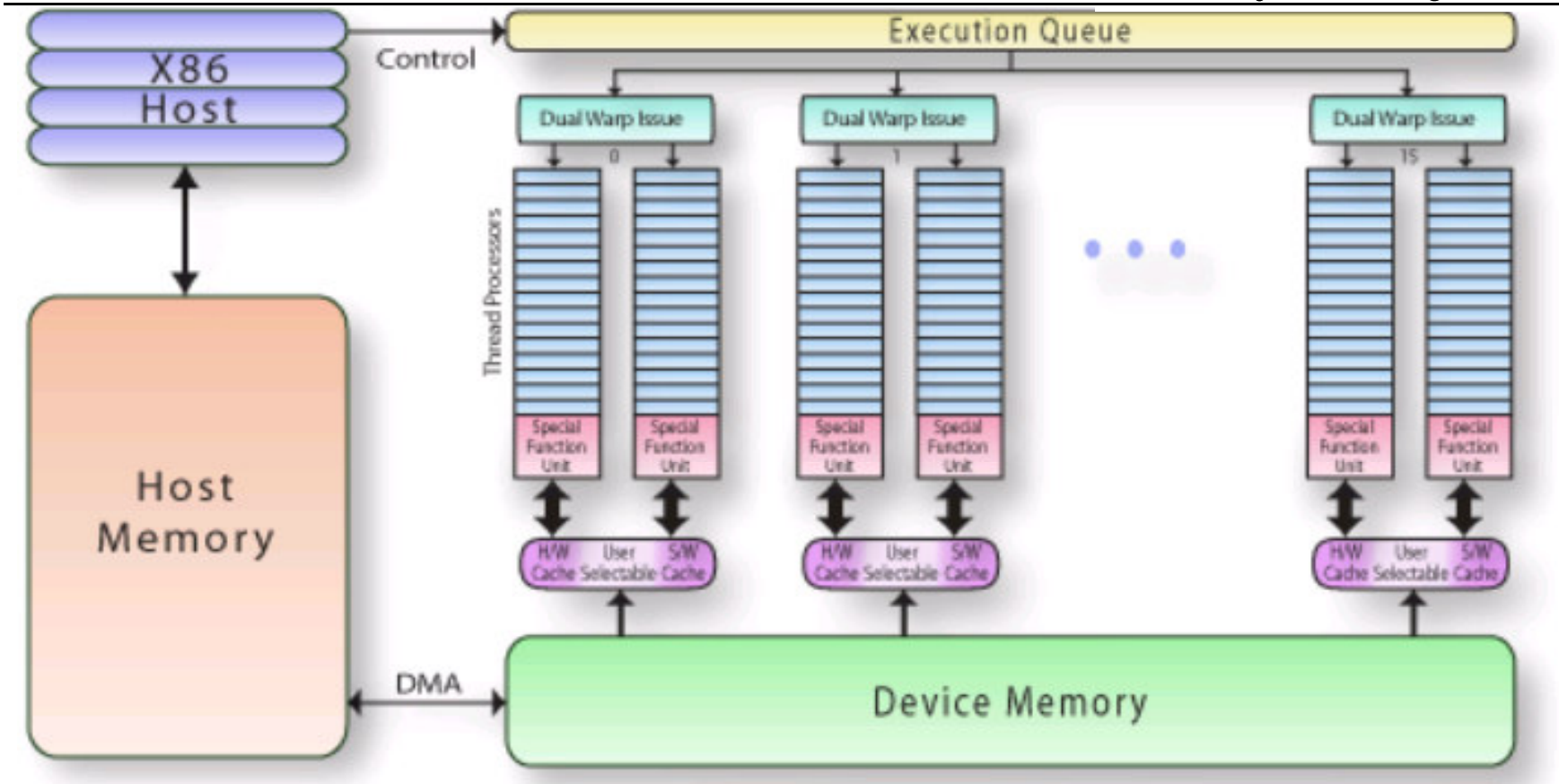


- A Tesla multiprocessor has a group of 8 *stream processors (core).*
- The 8 cores of a processor execute one instruction for an entire warp (32 threads) in four clock cycles.

- A Tesla supports up to 32 active warps on each multiprocessor.

NVIDIA Fermi Architecture

Karoly.Bosa@jku.at



- A Fermi multiprocessor has two groups of 16 stream processors.
- 2*16 cores to execute one instruction for each of two warps in two clock cycles.

- A Fermi supports up to 48 active warps on each multiprocessor.

Compute Capability Number

- The *compute capability* of a device is defined by *a major revision number* and *a minor revision number:*
  - Devices with the same major revision number are of the same core architecture.
  - The minor revision number corresponds to an incremental improvement to the core architecture.
- For instance,
  - The major revision number of devices based on the Fermi architecture is 2.x. Prior devices are all of compute capability 1.x.
- You can check the compute capability of your GPU on **http://www.nvidia.com/object/cuda_gpus.html**
- You can find the description of all the compute capability in the *CUDA C Programming Guide* (http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

Programming Fundamentals

- GPUs are programmed as a sequence of *kernels* (a special kind of function).
- Typically, each kernel completes execution before the next kernel begins, with an implicit barrier synchronization between kernels (only one function can be executed on the device on the same time);
  - Fermi has some support for multiple, independent kernels to execute simultaneously.

Programming Fundamentals - Limitations

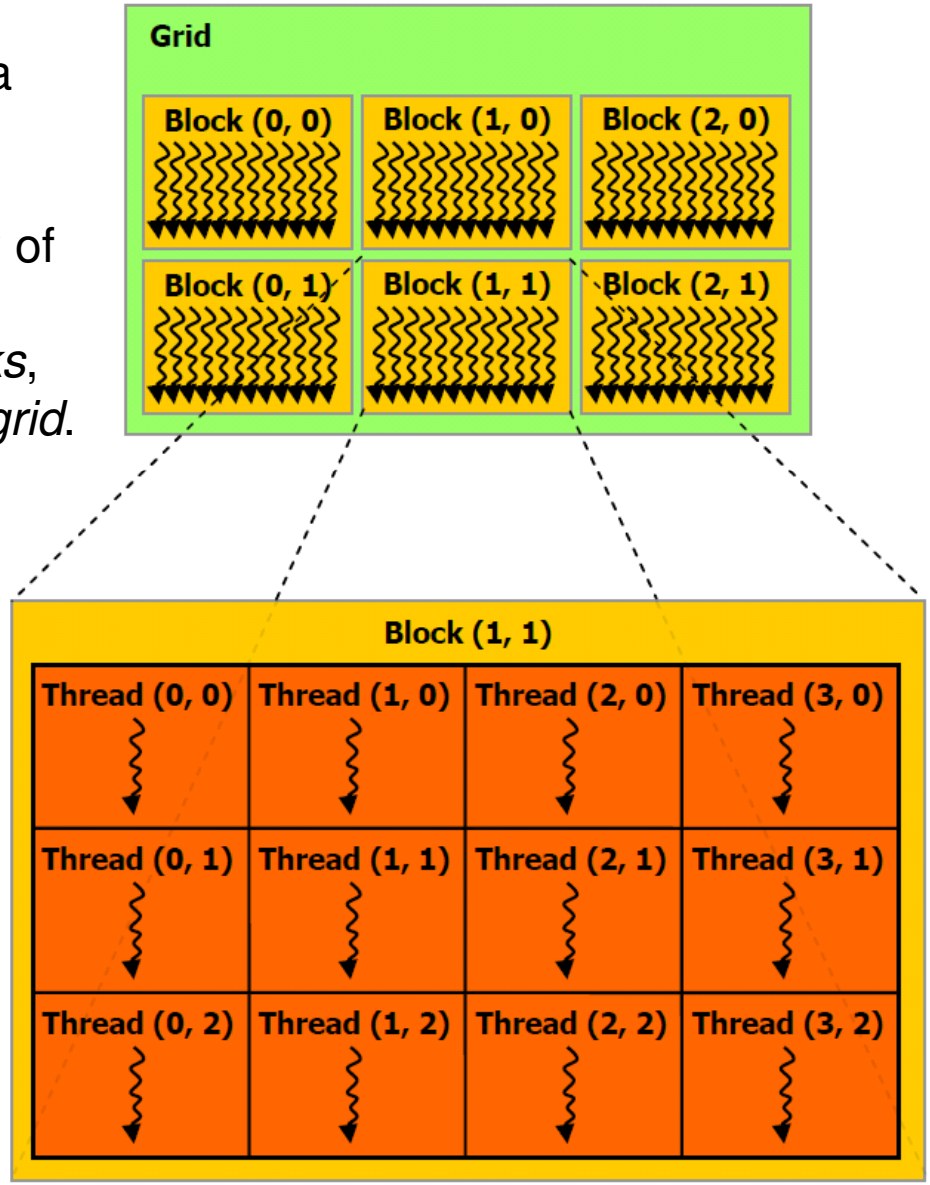
Karoly.Bosa@jku.at

- GPUs do not allow the multiprocessors to synchronize with each other.
- Threads can't spawn more threads;
- Threads on one multiprocessor can't interact to threads on another multiprocessor;
- There's no facility for a critical section among all the threads across the whole system.
- No recursion in device code
- No function pointers in device code

Introduction into CUDA

Karoly.Bosa@jku.at

- In parallel programming model of CUDA, the host program launches a sequence of kernels.
- A kernel is organized as a hierarchy of threads:
  - *Threads* are grouped into *blocks*,
  - and blocks are grouped into a *grid*.
- Each thread has a unique local index in its block,
- and each block has a unique index in the grid.
- Kernels can use these indices to compute array indices, for instance.



Kernel Definition I.


```
// 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);
}
```

- CUDA C extends C by allowing the programmer to define C functions, called *kernels.*
- When a kernel is called, it is executed N times in parallel by N different CUDA threads.
- Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in ***threadIdx*** variable**.**

Kernel Definition II.


```
// 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);
}
```

- *threadIdx* is a 3-component/dimensional vector.
- Blocks are organized into a one-dimensional or two-dimensional *grid of thread blocks(see blockIdx and blockDim).*

Thread Blocks

- Threads in a single block will **always** be executed on a single multiprocessor, so they can
  - share the software data cache,
  - and **can synchronize and share data** with threads in the same block.
- A warp will always be a subset of threads from a single block.
- Threads in different blocks may be assigned to (depending on how the blocks are scheduled dynamically):
  - different multiprocessors concurrently,
  - to the same multiprocessor concurrently (using multithreading),
  - or may be assigned to the same or different multiprocessors at different times.
- Thread blocks are required to execute independently (in any order) from each other.

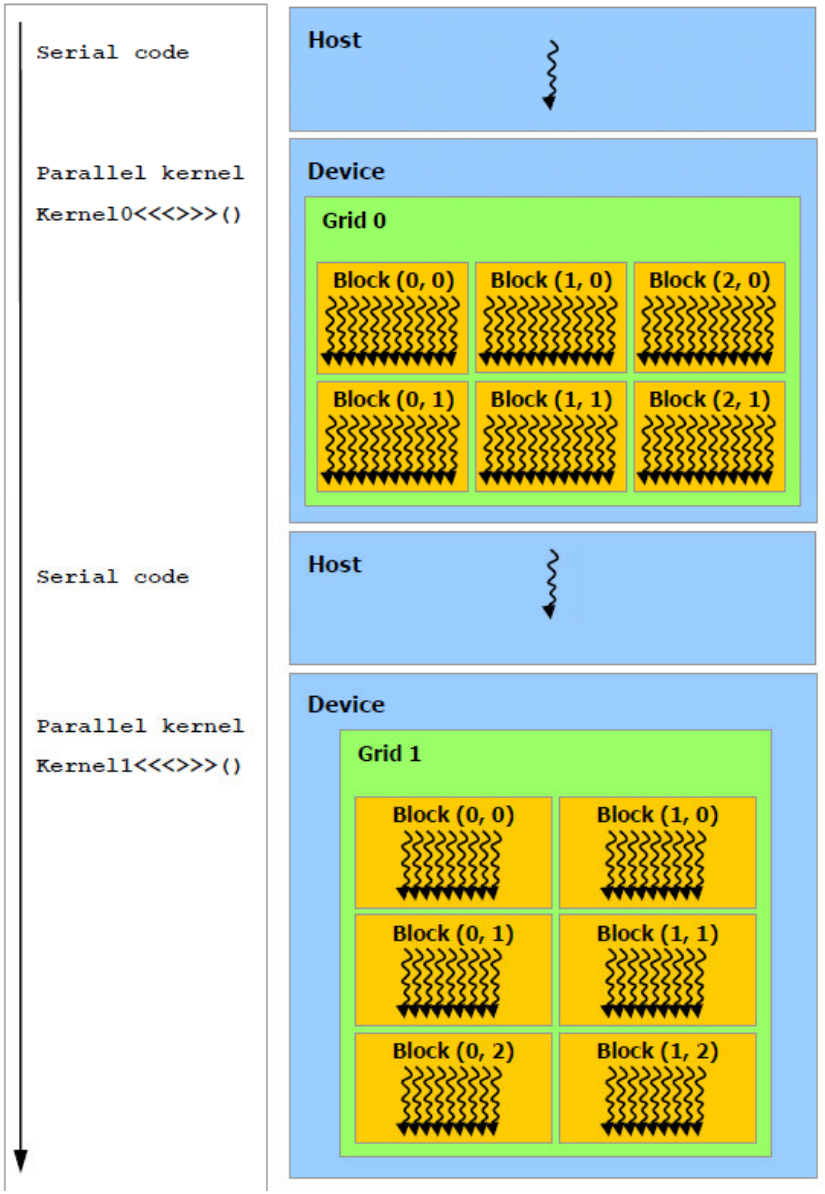
Typical Limitations of Thread Blocks

- There is a hard limit on the size of a thread block:
  - **512 threads** or 16 warps for Tesla,
  - 1024 threads or 32 warps for Fermi.
- Thread blocks are always created in warp units, so there is no point in trying to create a thread block of size that is not a multiple of 32 threads.
- All thread blocks in the whole grid will have the same size and shape.
- A Tesla multiprocessor can have 1024 threads simultaneously active, or 32 warps:
  - These can come from 2 thread blocks of 16 warps,
  - or 3 thread blocks of 10 warps,
  - or 4 thread blocks of 8 warps, and so on up to 8 blocks of 4 warps;
- Fermi can have 48 simultaneously active warps, equivalent to 1536 threads, from up to 8 thread blocks.
- (there is another hard limit, 8 thread blocks can simultaneously be active on a single multiprocessor in both architecture.)

Heterogeneous Programming

Karoly.Bosa@jku.at

- CUDA threads execute on a physically separate *device:*
  - *When the kernels execute on a GPU,*
  - *the rest of the C program executes on a CPU.*
- The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces (*host memory and device memory).*
- Device memory can be allocated either as *linear memory or as CUDA arrays.*



Asynchronous Execution between Host and Device

- In order to facilitate concurrent execution between host and device, some function calls are **asynchronous**(!).

- This means control is returned to the host thread before the device has completed the requested task. These asynchronous calls are:
  - **Kernel launches**;
  - Device-device memory copies;
  - Host-device memory copies of a memory block of 64 KB or less;
  - Memory copies performed by functions that are suffixed with async;
  - Memory set function calls.

Device Memory Example

Karoly.Bosa@jku.at

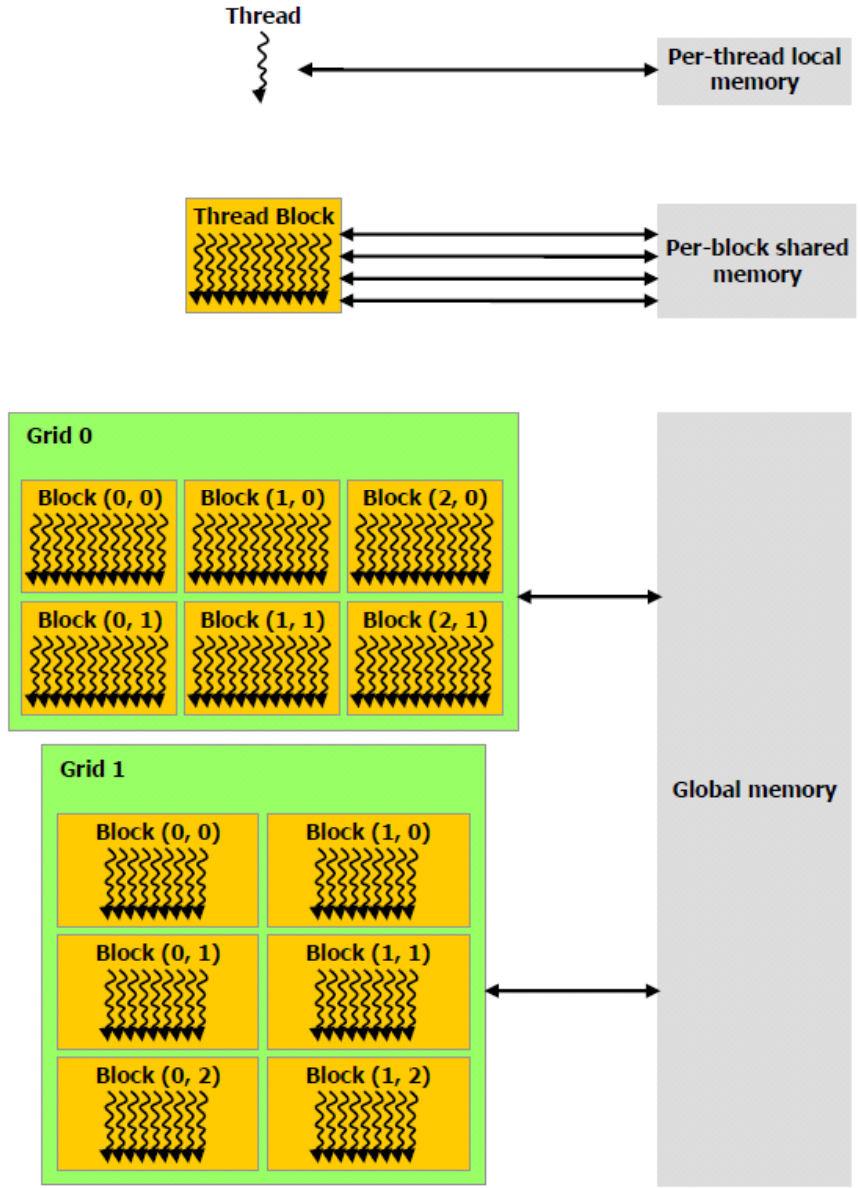
```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// 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
    ...
}
```

- cudaMalloc()
- using cudaFree()
- cudaMemcpy()

Memory Hierarchy

Karoly.Bosa@jku.at

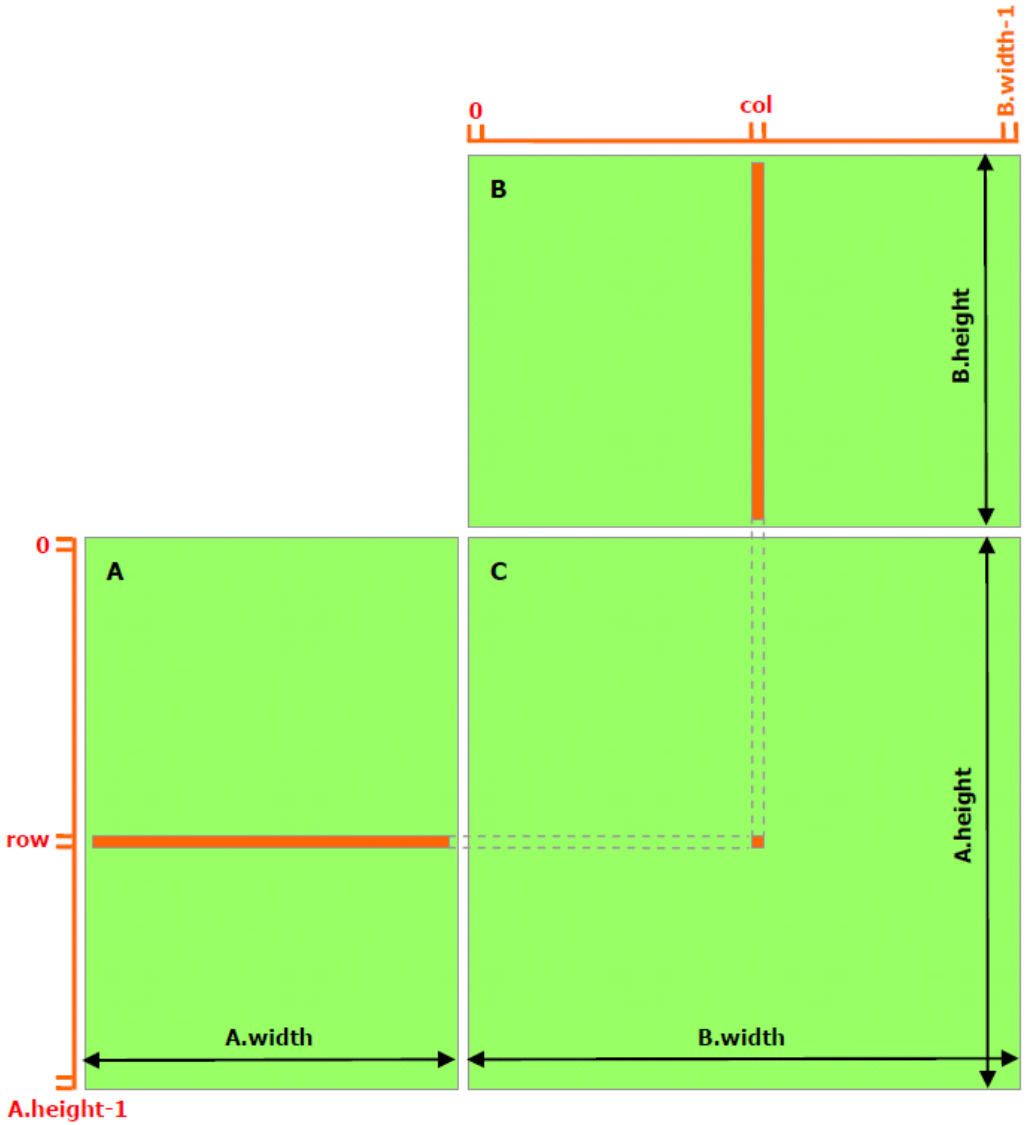
- CUDA threads may access to multiple memory spaces:
  - Each thread has private local memory.
  - Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block.
  - All threads have access to the same global memory (persistent across kernel launches by the same application).



Shared Memory

- Threads within a block can cooperate
  - by sharing data through some *shared memory and*
  - *by synchronizing their execution to coordinate memory accesses.*
- *For this, one can specify synchronization points in the kernel by calling the* ***__syncthreads()*** function;
- __syncthreads() acts as 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 (much like an L1 cache).

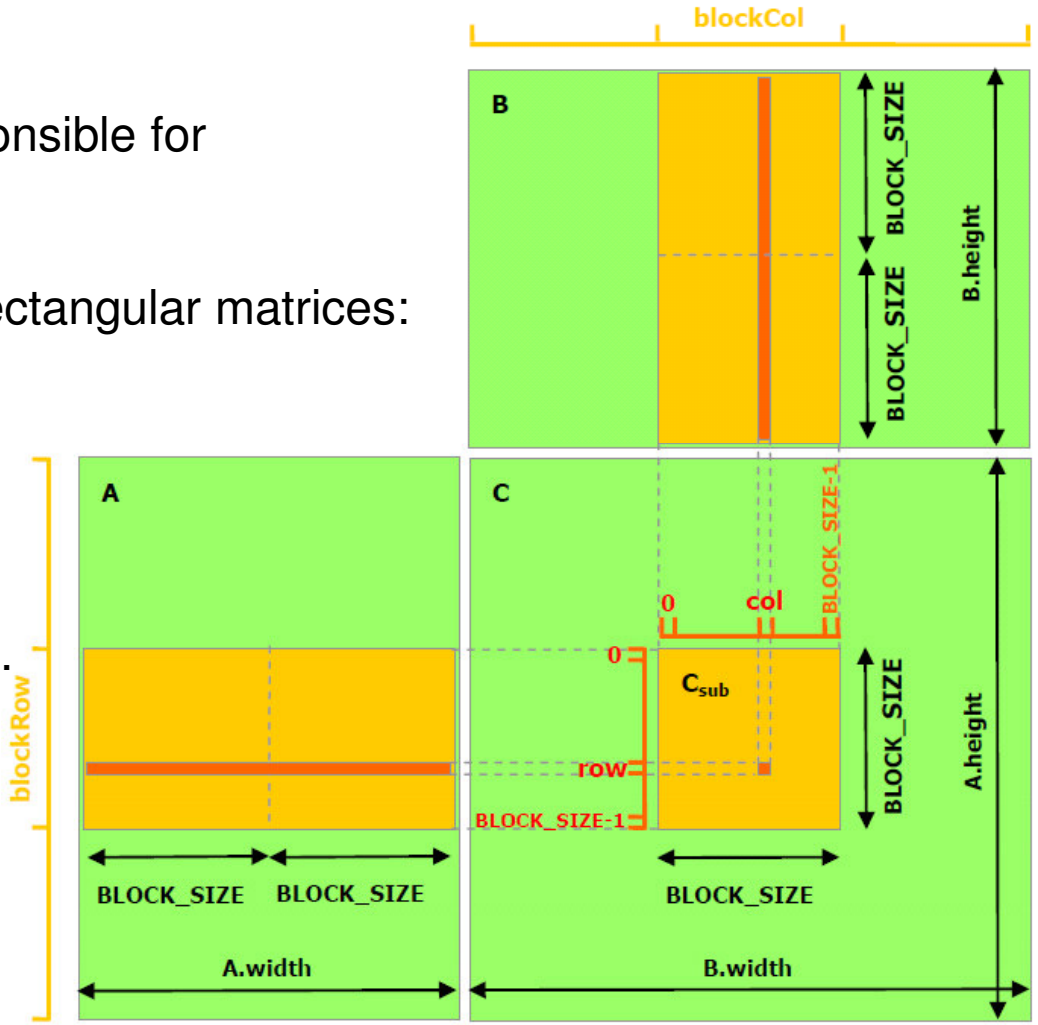
Example: Matrix Multiplication without Shared Memory



Example: Matrix Multiplication with Shared Memory

Karoly.Bosa@jku.at

- 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}.
- C_{sub} is equal to the product of two rectangular matrices:
  - The sub-matrix of A of dimension (A.width, block_size)
  - and the sub-matrix of B of dimension (block_size, A.width).
- Additional step I required: in every turn two corresponding square matrices must be loaded from global memory to shared memory,
- but this is done by concurrent threads (overhead is negligible).



Shared Memory Example: Definitions

Karoly.Bosa@jku.at

```
// 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;

// Thread block size
#define BLOCK_SIZE 16
```

- **Field *stride* is used in submatrices to query the width of the original matrix (to calculate the addresses of the first elements of rows).**

Shared Memory Example: Device Functions

Karoly.Bosa@jku.at

```
// 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;
}
```

Shared Memory Example: Host Code, Part I.

Karoly.Bosa@jku.at

```
// 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);
```

Shared Memory Example: Host Code, Part II.

Karoly.Bosa@jku.at

```
    // 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);
}
```

Shared Memory Example: Kernel, Part I.

Karoly.Bosa@jku.at

```
// 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
```

Shared Memory Example: Kernel, Part II.

Karoly.Bosa@jku.at

```
    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);
}
```

Using More than one GPU I.

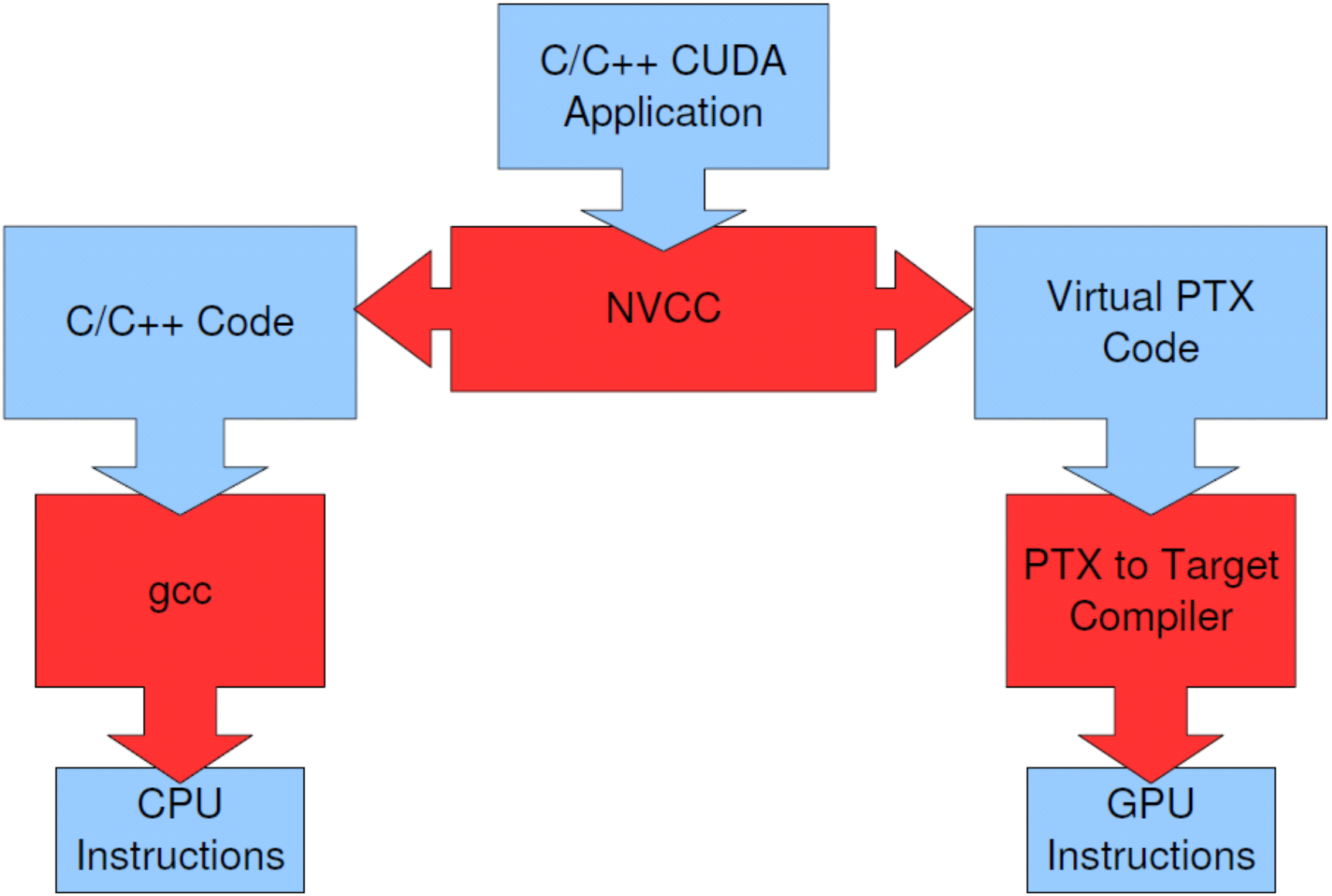
- A host system can have multiple GPU devices.
- **But** a host thread (e.g.: POSIX thread) can execute device code on only one device at any given time.
- **However** multiple host threads can execute device code:
  - on the same device or
  - on multiple devices.
- GPU devices can be enumerated, their properties can be queried, and one of them can be selected for kernel executions.

Using More than one GPU II.


```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
    if (device == 0) {
        if (deviceProp.major == 9999 && deviceProp.minor == 9999)
            printf("There is no device supporting CUDA.\n");
        else if (deviceCount == 1)
            printf("There is 1 device supporting CUDA\n");
        else
            printf("There are %d devices supporting CUDA\n",
                   deviceCount);
    }
}
```

- By default, the GPU device associated to the host thread is implicitly selected as device 0 (as soon as a GPU related function is called).
- Any other device can be selected by calling *cudaSetDevice()* first.
- Once a device has been selected, either implicitly or explicitly, the calling *cudaThreadExit()* must be called before another device selection.

Compiling a CUDA Program



Trying out all these Things

- If you have CUDA capable GPU:
  - Download and install from the http://developer.nvidia.com/object/cuda_archive.html
    - Developer Drivers
    - CUDA Toolkit
    - CUDA SDK
      - It includes sample programs in source form.
      - To compile them issue command **make** in the corresponding directory.
  - Remark: You cannot use CUDA under VMware (even your host has an NVIDIA GPU), because the access to host GPU is not transparent (VMware always uses a generic video driver).
- If you do not have CUDA capable GPU:
  - You should download the **CUDA version 2.3** instead of the newest one (this is the last version which contains the CUDA emulator):
    - To compile sample programs issue command **make emu=1** in the corresponding directory.

Debugging (Linux)

- Until CUDA 2.3, debugging CUDA program was possible only by using the emulator.
- In later version you could debug codes on the GPUs (with a compute capability of 1.1 or later).
- Limitation: X11 cannot be running on the GPU that is used for debugging(!).
- Compilation**: nvcc -g -G ....**
- Debugger tool: **cuda-dbg** (an extension to the standard gdb).
- Detailed Documentation: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/cuda-gdb.pdf