Parallelism

Master 1 International

Andrea G. B. Tettamanzi
Université de Nice Sophia Antipolis
Département Informatique
andrea.tettamanzi@unice.fr

Lecture 5, Part b

Throughput-Oriented Architectures

Table of Contents

- Introduction
- FPGAs
- GP-GPUs
- NVIDIA GPU Architecture
- CUDA Programming Model
- Throughput-Oriented Programming

Throughput-Oriented Architectures

- The current trend in CPUs is to go from single-core to multi-core
- Aim: to deliver higher performance by exploiting modestly parallel workloads arising from the execution of
  - multiple independent programs
  - individual programs consisting of multiple parallel tasks
- A related architectural trend is the growing prominence of throughput-oriented microprocessor architectures.
  - Field-programmable gate arrays (FPGAs)
  - Sun's Niagara
  - NVIDIA's graphics processing units (GPUs)
- When executing (massively) parallel workloads, focus on maximizing total throughput at the cost of serial performance

Improving Total Throughput

- Trade-off in (parallel) computing:
  - Increase total throughput (i.e., amount of work done/unit time)
  - Decrease latency for a single task (i.e., time elapsed)
- Therefore, we have:
  - Latency-oriented designs: traditional CPUs
  - Throughput-oriented designs
- Applications: any problem where parallelism abounds
  - Real-time computer graphics (e.g., gaming, virtual reality)
  - Video processing
  - Medical-image analysis
  - Molecular dynamics, astrophysical simulation
  - Gene sequencing

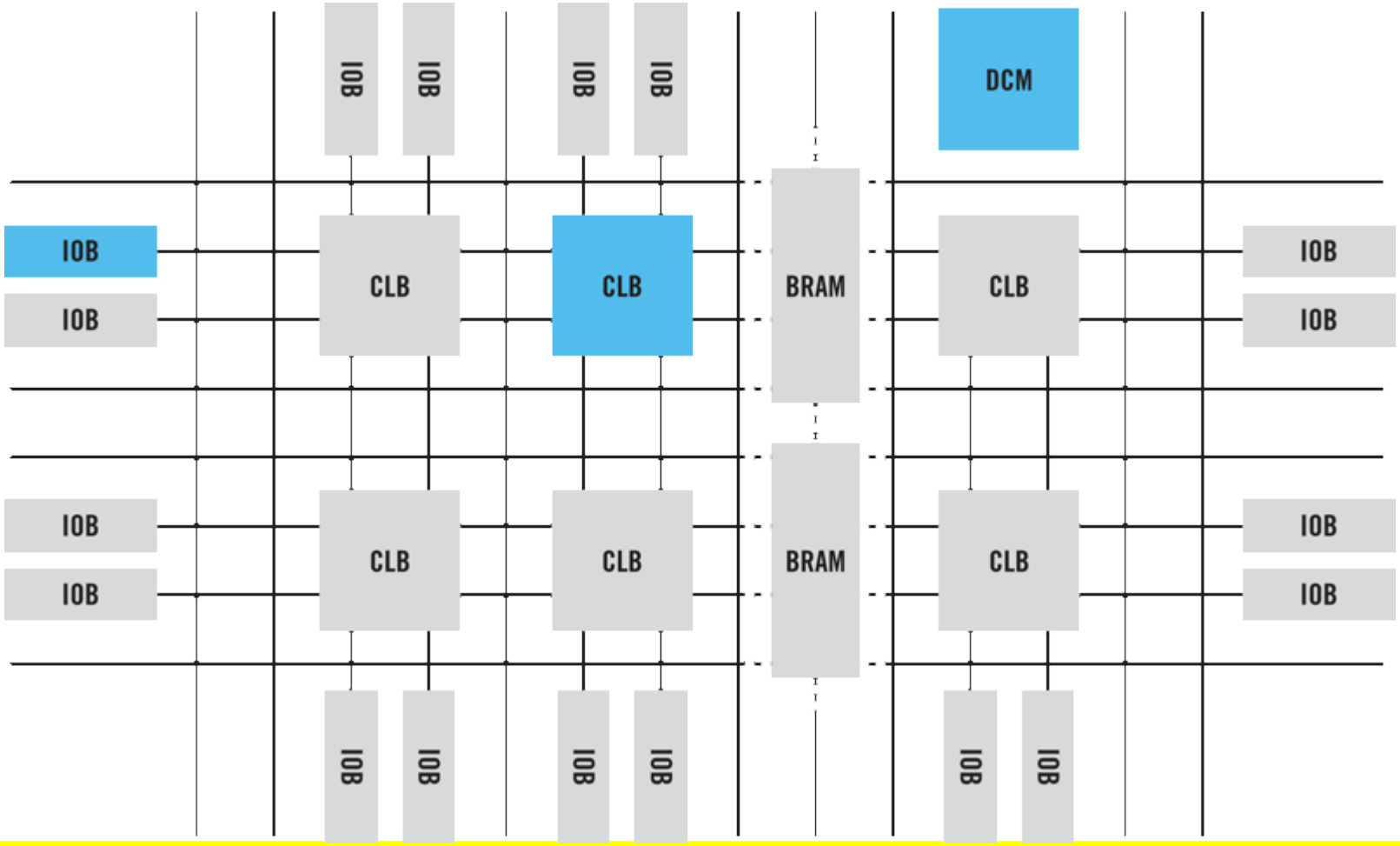
Field-Programmable Gate Arrays

- Two big players: Xilinx and Altera
- Semiconductor devices based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects
- Can be reprogrammed after manufacturing
- An alternative to Application-Specific Integrated Circuits (ASICs)
- Two types of FPGAs:
  - One-time programmable (OTP) FPGAs
  - SRAM-based FPGAs: can be reprogrammed at will – the dominant type

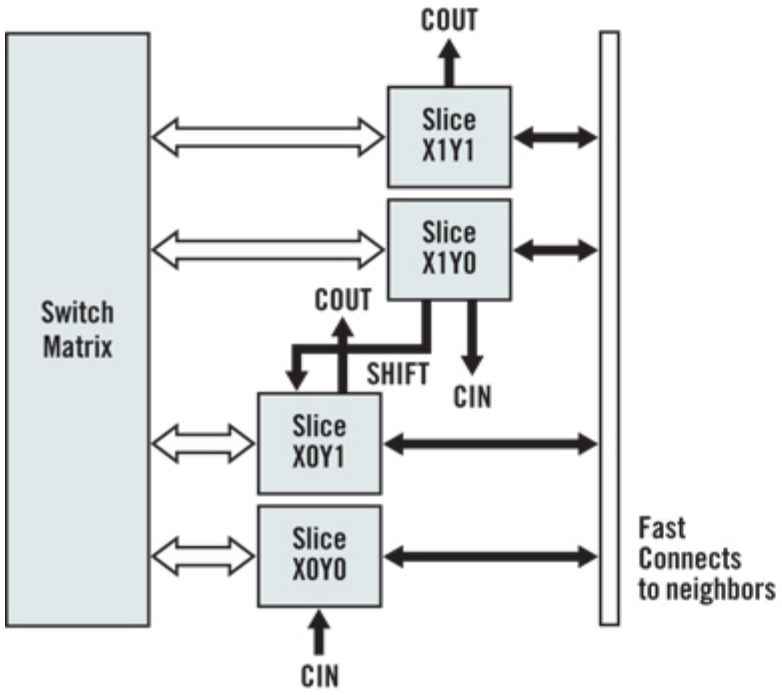
FPGA Applications

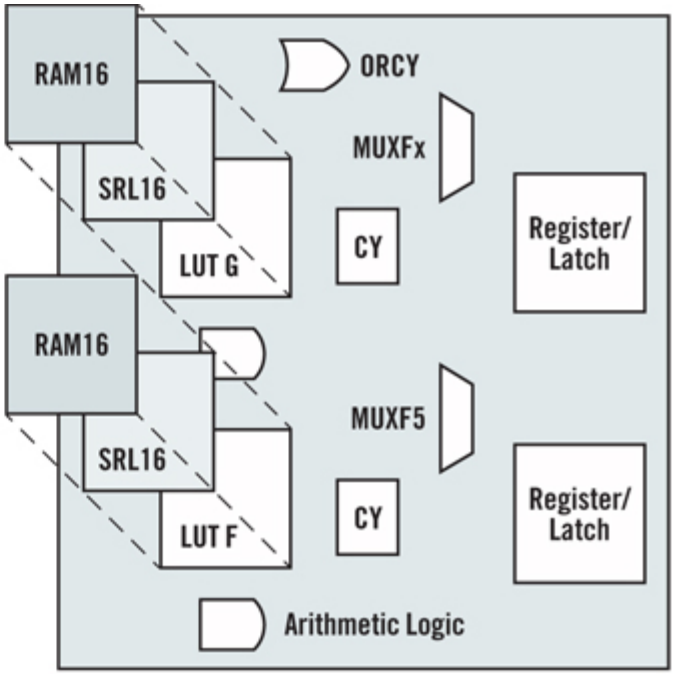
- Aerospace & Defense: image & signal proc., reconfiguration
- ASIC prototyping
- Audio, Video, Image processing, and Broadcast
- Automotive
- Industrial, scientific, and medical applications
- Consumer electronics
- Networking: high-badwidth, low latency devices for data centers
- Wired and wireless communication
- High-performance computing and data storage

FPGA Block Structure



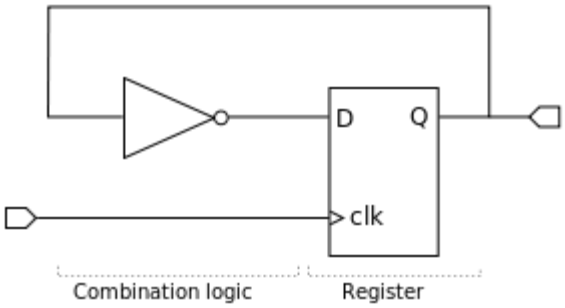
Configurable Logic Block





FPGA Programming

- Vendors provide integrated design suites (e.g., Xilinx ISE)
- FPGA “programs” take the form of register-transfer levels (RTLs)
- An RTL is a design abstraction which models a circuit in terms of
  - flow of data between registers
  - logical operations performed on those data
- The *de facto* standard for RTL is VHDL



```
D <= not Q;

process(clk)
begin
    if rising_edge(clk) then
        Q <= D;
    end if;
end process;
```

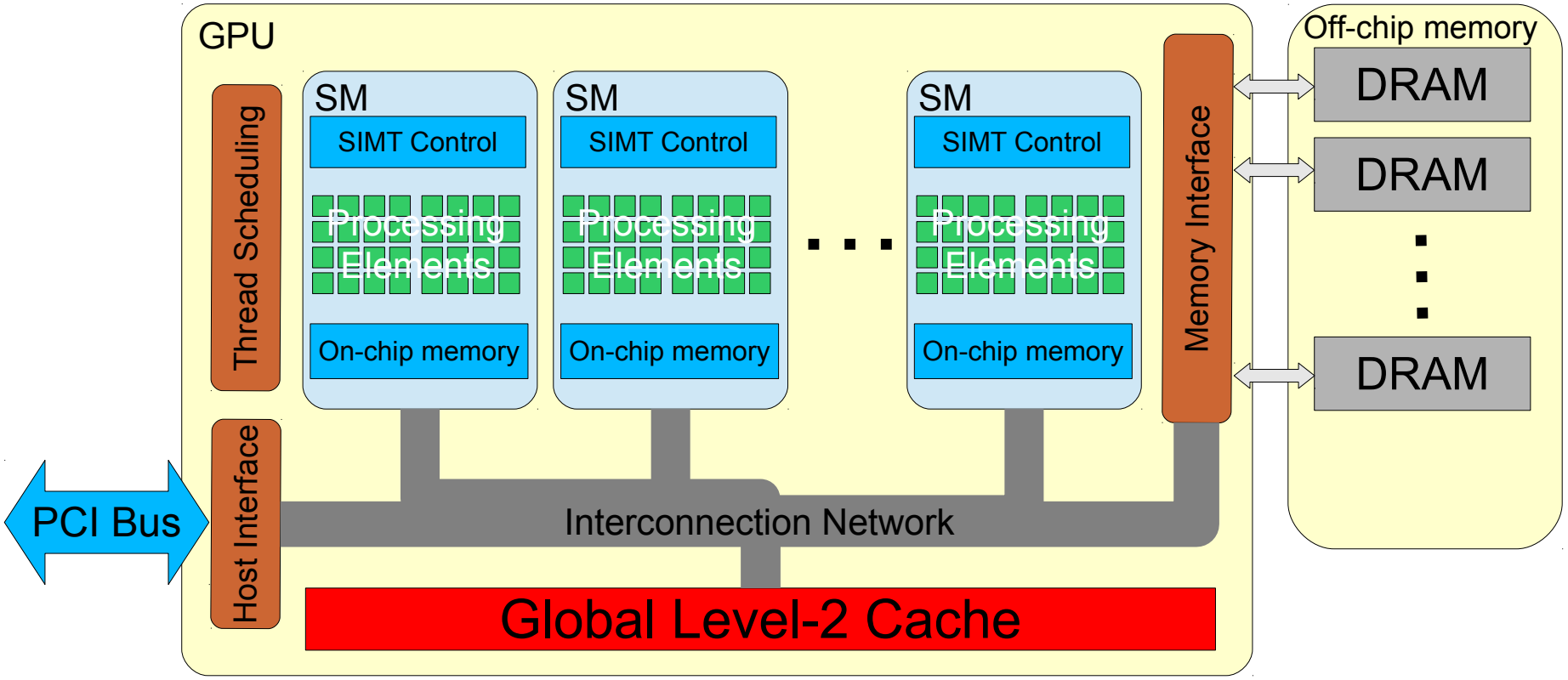
GP-GPUs

- Graphics Processing Units (GPUs), aka graphics accelerators:
  - The leading exemplars of modern throughput-oriented architectures
  - Produced in high volumes, their price is accessible
- General-Purpose GPUs (GP-GPUs)
  - Primarily designed for graphics processing
  - Suitable for other tasks as well
  - Example: NVIDIA Tesla GPUs

NVIDIA GPU Architecture

- Beginning with the G80 processor released in late 2006, NVIDIA GPUs support the CUDA architecture for parallel computing
- Array of processors, called “streaming multiprocessors” (SM)
- Each SM
  - Supports on the order of 1000 co-resident threads
  - Is equipped with a large register file
  - Contains many scalar processing elements that execute the instructions issued by the running threads
  - Contains high-bandwidth, low-latency on-chip shared memory
  - Provides direct read/write access to off-chip DRAM
- Single-instruction, multiple-thread (SIMT) execution model
  - Threads executed in groups (“warps”), in a SIMD fashion

NVIDIA GPU Architecture



CUDA Programming Model

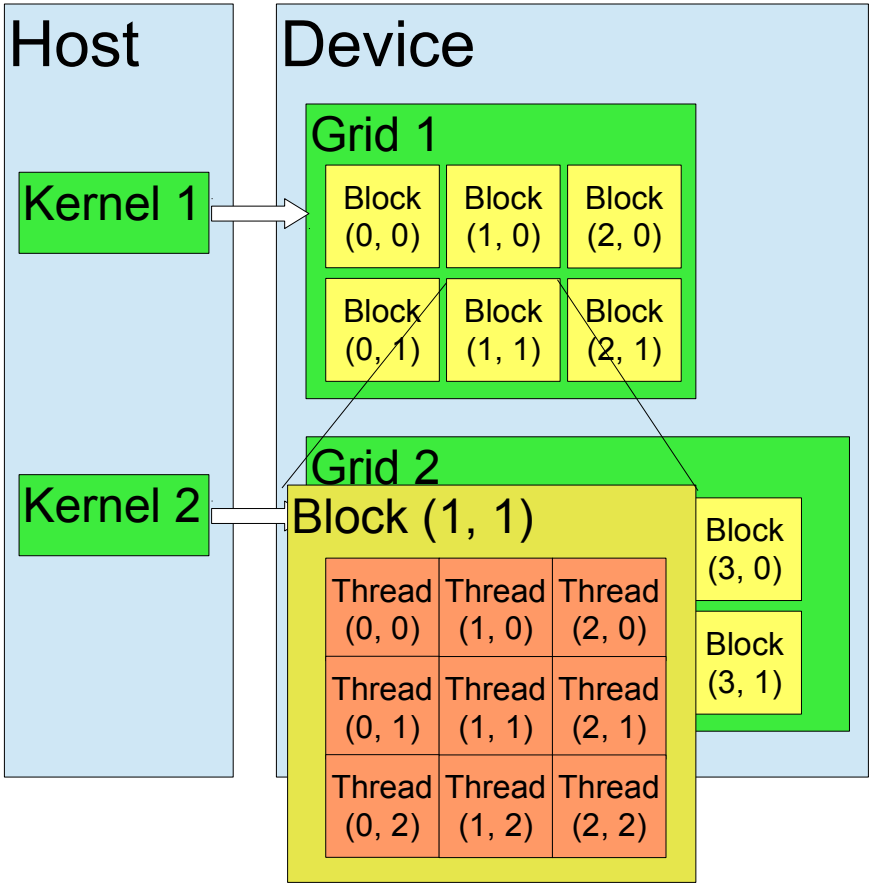
- Minimalist set of abstractions for parallel programming on massively multithreaded architectures
- A CUDA program is organized into
  - one or more threads executing on a host processor (CPU)
  - one or more parallel kernels that can be executed by the host thread(s) on a parallel device (GPU)
- Kernels run a sequential program across a set of parallel threads.
- The programmer specifies for each kernel launch:
  - the number of blocks
  - The number of threads per block
- CUDA kernels are thus similar in style to a blocked form of the familiar SPMD paradigm, but somewhat more flexible

CUDA Threads

- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

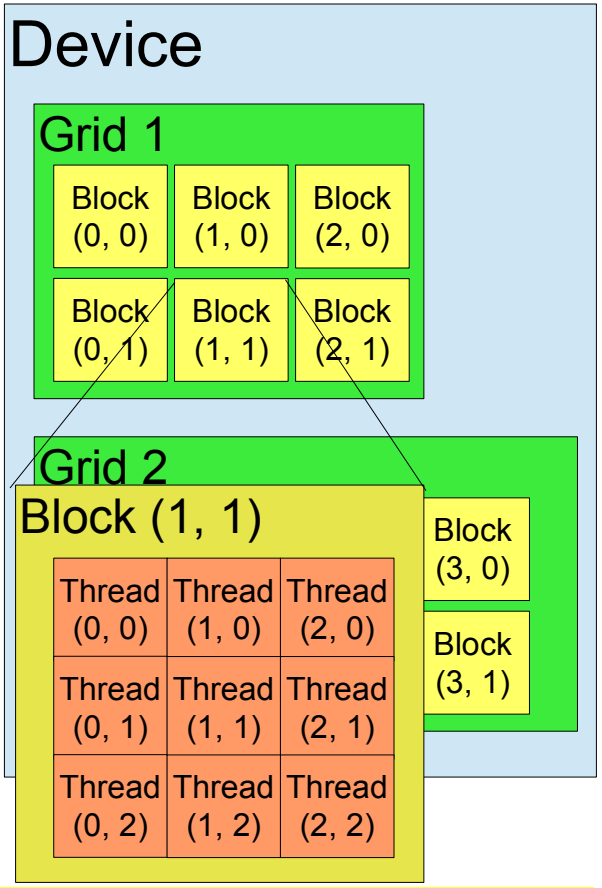
Thread Blocks and the Grid

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate!



Thread Blocks and the Grid

- Threads and blocks have IDs
  - So each thread can decide what data to work on
- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes



Memory Spaces

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can read/write global, constant, and texture memory (stored in DRAM)

Execution Model

- Kernels are launched in grids
  - One kernel executes at a time
- A block executes on one multiprocessor
  - Does not migrate
- Several blocks can execute concurrently on one multiprocessor
  - Control limitations:
    - At most 8 concurrent blocks per SM
    - At most 768 concurrent threads per SM
  - Number is limited further by SM resources
    - Register file is partitioned among the threads
    - Shared memory is partitioned among the blocks

CUDA Programming Model in C

- Functions are declared either
  - As a kernel entry point, using the __global__ modifier
    - May be called from the host only, executes on the device
    - Must return void
  - As a normal C function, using the __host__ modifier
    - May be called from the host only, executes on the host
  - As a kernel-only function, using the __device__ modifier
    - May be called from the device only, executes on device
- A few restrictions for functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

CUDA Programming Model in C

- The host program launches kernels using the function-call-like syntax fn_name**<<<**B, T**>>>**(arguments)
  - B: number of blocks
  - T: number of threads per block
- Kernels may use a set of special variables:
  - gridDim.x, gridDim.y: the dimension of the MS grid
  - blockIdx.x, blockIdx.y: the 2D index of the thread block
  - threadIdx.x, threadIdx.y, threadIdx.z: 3D index of thread
  - blockDim.x, blockDim.y, blockDim.z: dimension of the block

CUDA Programming Model in C

- Shared variables are declared using the __shared__ modifier
  - Scope and lifetime: a thread block
- Global variables are declared using the __global__ modifier
  - Scope: a grid; lifetime: the application
- Constants are declared using the __constant__ modifier
  - Scope: a grid; lifetime: the application
- Automatic variables without any qualifier reside in registers
  - Except for large structures that reside in local memory
- Pointers can point to memory in either global or shared memory:
  - Global memory: allocated in the host and passed to the kernel, obtained as the address of a global variable
  - Shared memory: statically allocated during the call

CUDA Programming Model in C

- Explicit GPU memory allocation
  - cudaMalloc(), cudaFree()
- Memory copy from host to device, etc.
  - cudaMemcpy(), cudaMemcpy2D(), …
- Runtime library split into:
  - A common component providing built-in vector types and a subset of the C runtime library supported in both host and device codes
  - A host component to control and access one or more devices from the host
  - A device component providing device-specific functions

CUDA Programming: An example

```
__global__ void increment(float *x, int n)
{
    // Each thread will process 1 element, which
    // is determined from the thread's index.
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )
        x[i] = x[i] + 1;
}
__host__ void parallel_increment(float *x, int n)
{
    // Launch increment() kernel with 1 thread
    // per element, grouped into ⌈n/256⌉ blocks
    // of 256 threads each.
    increment<<<ceil(n/256), 256>>>(x, n);
}
```

Throughput-Oriented Programming

- Scalability should be the programmer's central concern
- Techniques suitable for 4 parallel threads may be completely unsuitable for 4000 parallel threads
- Must expose substantial amounts of fine-grained parallelism, fulfilling the expectations of the architecture
- The maximal parallelism is often the best performer
- Example: sparse matrix-vector multiplication y = Ax
  - Moderate parallelism: 1 thread per matrix row
  - Intermediate parallelism: several threads per row
  - Maximal parallelism: 1 thread for each non-zero element
- Calculation is much cheaper than memory transfer
  - Preferable to locally recompute values than to store them

Thank you for your attention