Parallelism (PAR)

Programming with CUDA

Eduard Ayguadé and Josep R. Herrero

Computer Architecture Department
Universitat Politècnica de Catalunya

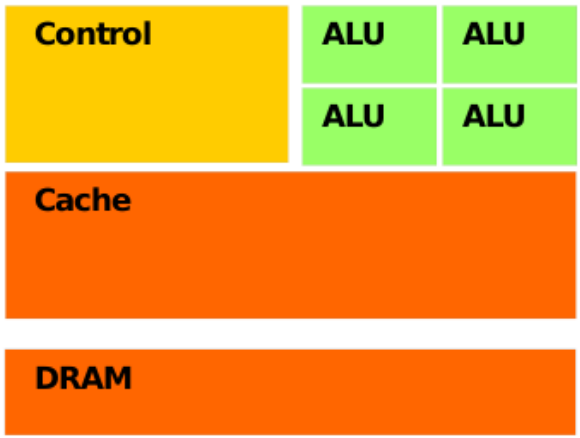
2012/13-Spring

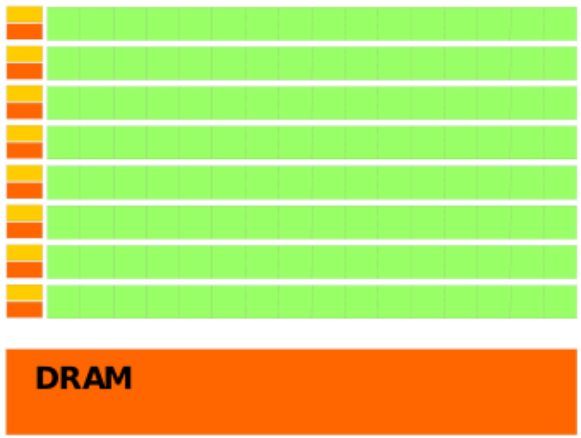
Outline

CPU vs. GPU architecture

CPUs are designed for general-purpose computing

- Sophisticated control to exploit ILP. ALU to exploit DLP. Multicores (independent control flow) for TLP
- Large caches to reduce impact of long latency memory accesses and to enable sharing in multicores

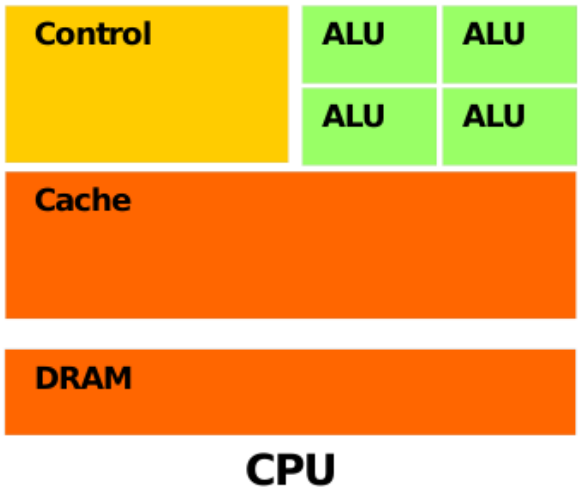


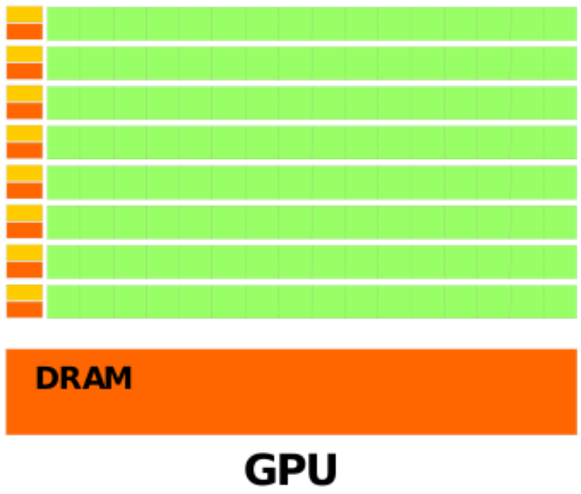


CPU vs. GPU architecture (cont.)

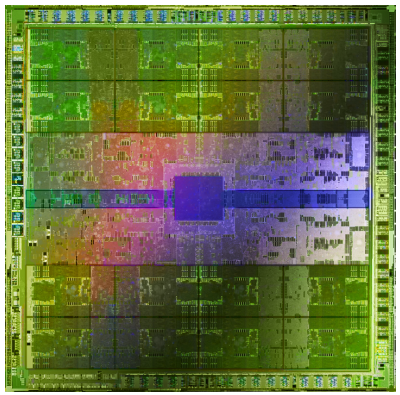
GPUs are specialized for highly parallel, compute-intensive computation

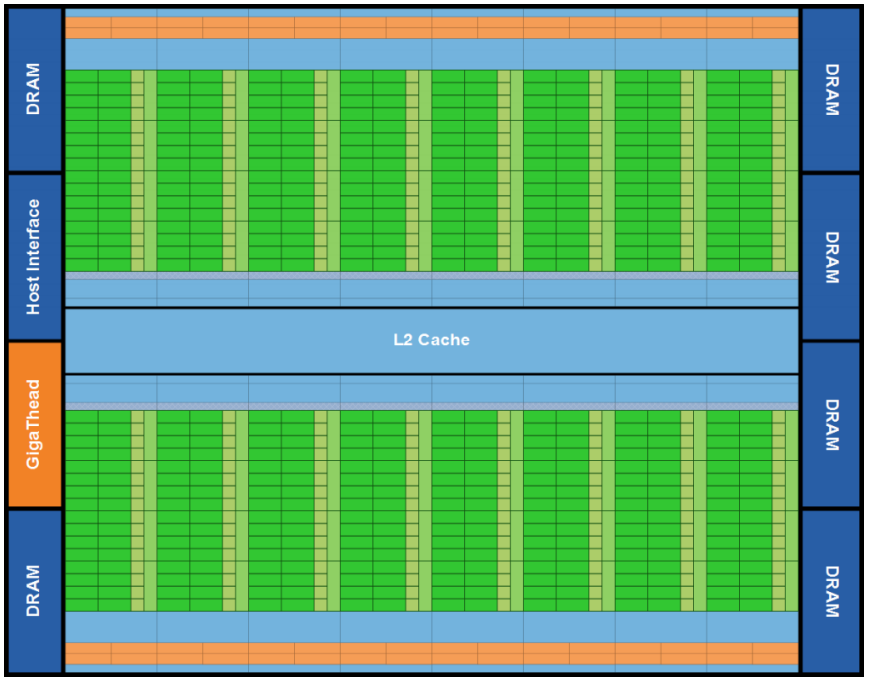
- Simple control: same computation on all compute units
- Massive number of threads to reduce impact of long latency memory accesses





NVIDIA GPU architecture

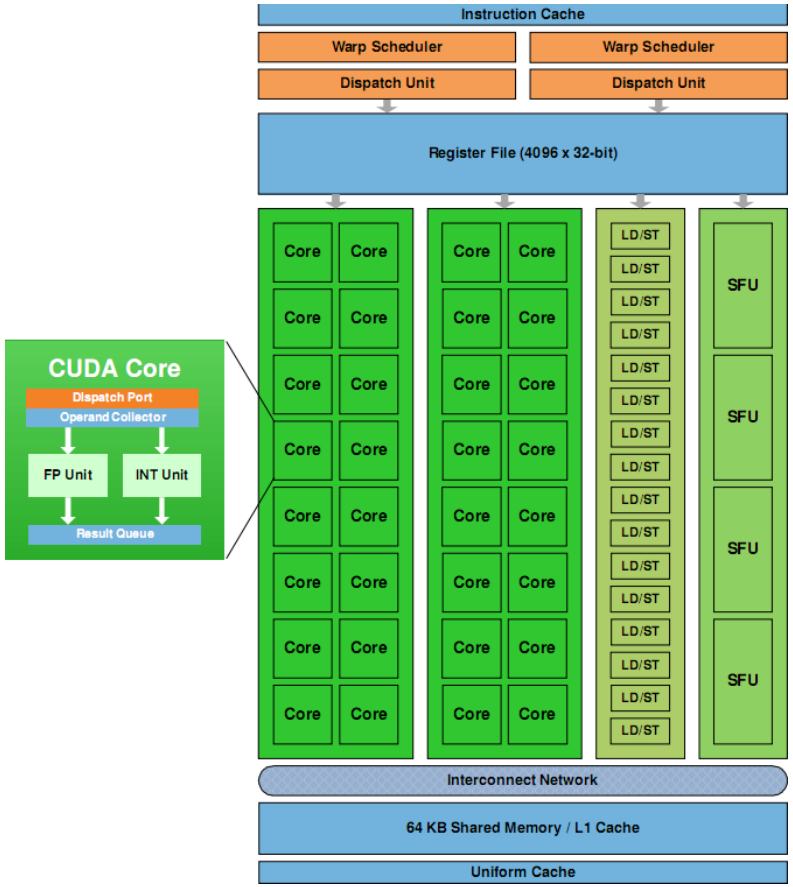




NVIDIA Fermi: 16 streaming multiprocessors (SM) interconnected via a 768k L2 cache crossbar

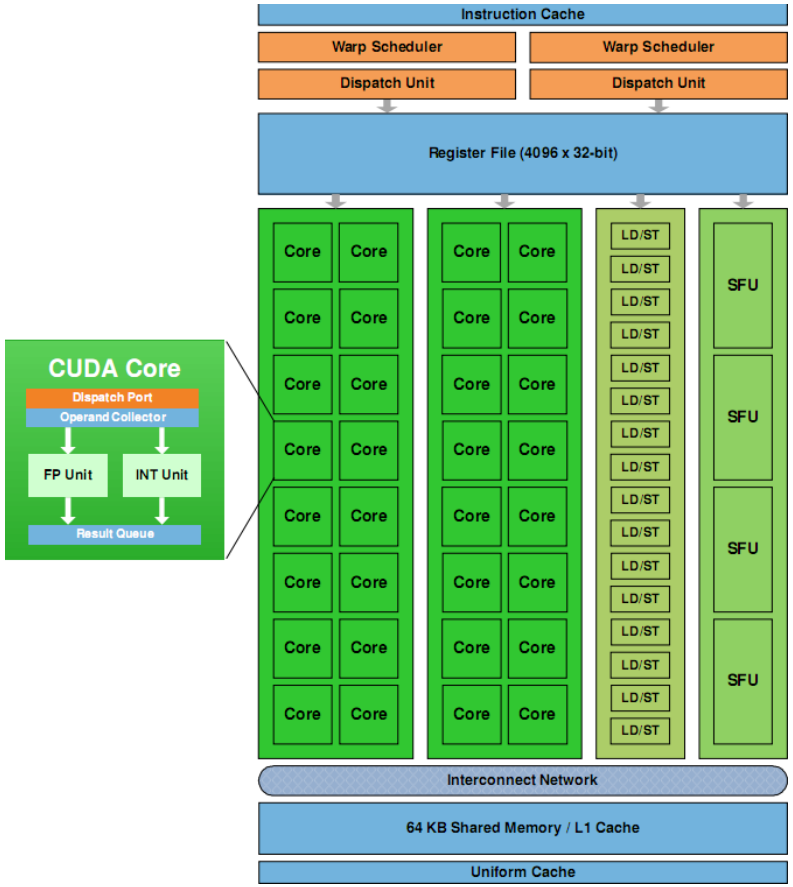
NVIDIA GPU architecture (cont.)

- Each SM has 32 cores and 4096 registers, sharing 64 KB (48K L1 and 16K shared memory or viceversa)
- Each core has one FP unit and one integer unit

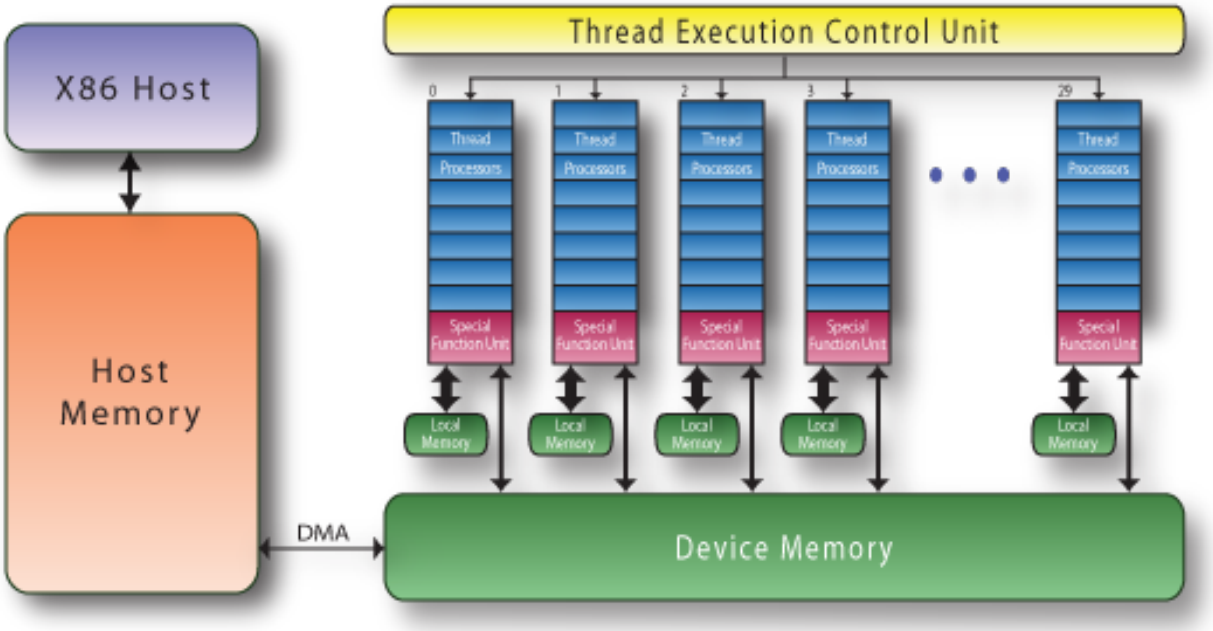


NVIDIA GPU architecture (cont.)

- Each SM has 16 load/store units, allowing sixteen threads per clock to calculate memory addresses
- Each SM also has 4 Special Function Units (SFU) that execute complex instructions (sin, cosine, reciprocal, and square root)



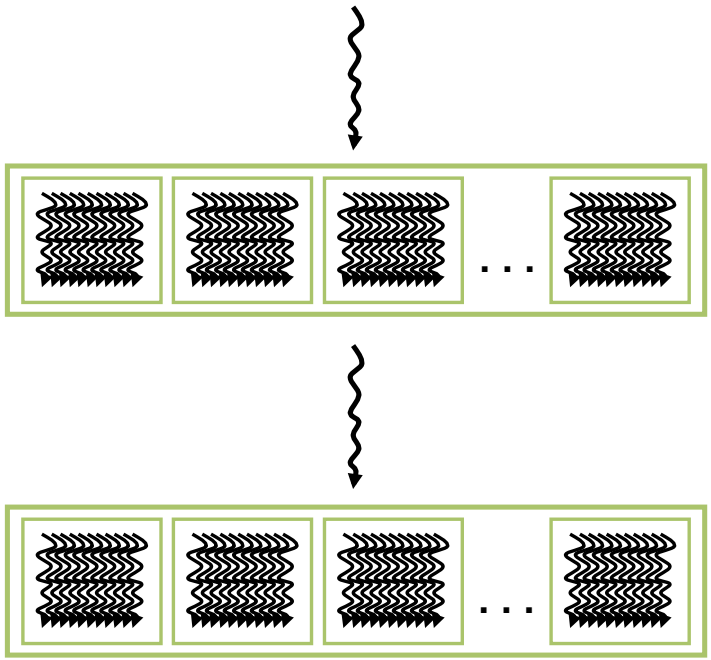
CPU-GPU block diagram

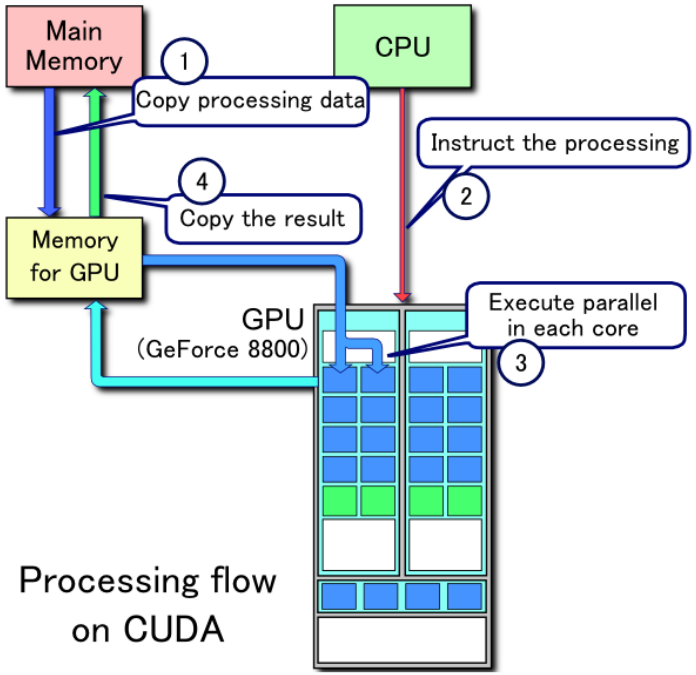


Separate address spaces between CPU (host memory) and GPU (device memory). Communication using IO commands and DMA memory transfers (e.g. PCI Express)

CPU-GPU execution flow

CPUs for sequential or modestly parallel parts, GPUs for highly parallel parts (kernels in CUDA)





Outline

Device management

- Application can query and select GPUs
  - `cudaGetDeviceCount(int *count)`
  - `cudaSetDevice(int device)`
  - `cudaGetDeviceProperties(cudaDeviceProp *prop, int device)`
  - `cudaGetDevice(int *device)`
- Device sharing
  - Multiple threads can share a device
  - A single thread can manage multiple devices

Offloading kernel execution

- Kernel: function that runs on the device and is called from host
- Definition using keyword `__global__`

```
__global__ void helloworld(void) {
    }
```

Offloading kernel execution (cont.)

- Kernel offloading for execution on GPU device

```
...
helloworld<<<1,1>>>();
...
```

Triple angle brackets mark a "kernel launch" (call from host code to device code)

  - We will cover the parameters inside the brackets in a few slides

- The kernel launch is asynchronous (i.e. the host does not wait for the termination of the kernel execution, control returns to the CPU immediately)

Offloading kernel execution (cont.)

- Host may need to wait before continuing execution
- The execution of `cudaThreadSynchronize()`[1] blocks the CPU until all preceding CUDA kernels have completed

```
__global__ void helloworld(void) {
}

int main(void) {
     helloworld<<<1,1>>>();
     cudaThreadSynchronize();
     printf("Hello World!\n");
     return 0;
}
```

[1] Deprecated, to be replaced with `cudaDeviceSynchronize()`

Offloading to multi–GPU nodes

- Any host thread can access all GPUs in the node (if more than one available) using the `cudaSetDevice` call

```
__global__ void helloworld(void) {
}

int main(void) {
      int numDevs;
      cudaGetDeviceCount(&numDevs);

      for (int d = 0; d < numDevs; d++) {
            cudaSetDevice(d);
            helloWorld<<<1,1>>>();
      }

      for (int d = 0; d < numDevs; d++) {
            cudaSetDevice(d);
            cudaThreadSynchronize();
      }

      return 0;
}
```

Laboratory examples (1)

1. Compile and run `deviceQuery.cu` in `deviceQuery` directory. Observe how many devices are available in the node and their characteristics
2. Compile and run `hello1.cu` in `helloworld` directory. Observe the definition of multiple kernels and their invocation in the devices available. Trace using `Extrae` to observe kernel execution in multiple devices

Outline

Parallel kernel

- How to make use of the multiple streaming multiprocessors (SM) and threads inside each SM?
- Kernel replication using the arguments in the kernel launch
  - Replication in different SM

```
helloWorld<<<N,1>>>();
```

  - Replication in different threads inside one SM

```
helloWorld<<<1,N>>>();
```

Parallel kernel

- Each kernel instance can be univoquely identified, which can be used to make kernel instances cooperate in the resolution of a common problem

```
__global__ void helloWorld(char* str) {
     int idx = blockIdx.x;

     str[idx] += idx;
}

int main (int argc, char *argv[]) {
     ...
     char str[] = "Hello World!";
     for(i = 0; i < 12; i++)
          str[i] -= i;
     ...

     helloWorld<<<12, 1>>>(d_str);

     ...
}
```

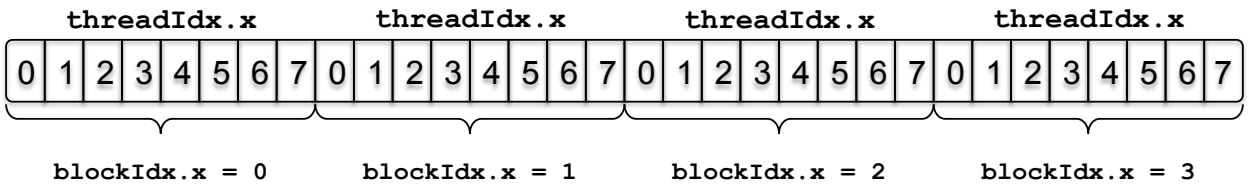
Parallel kernel

- Similarly if replication occurs at the threads level

```
__global__ void helloWorld(char* str) {
     int idx = threadIdx.x;

     str[idx] += idx;
}

int main (int argc, char *argv[]) {
     ...
     char str[] = "Hello World!";
     for(i = 0; i < 12; i++)
          str[i] -= i;
     ...

     helloWorld<<<1, 12>>>(d_str);

     ...
}
```

Indexing a vector with blocks and threads

Consider indexing an array with one element per thread and M threads per block (e.g. 8)

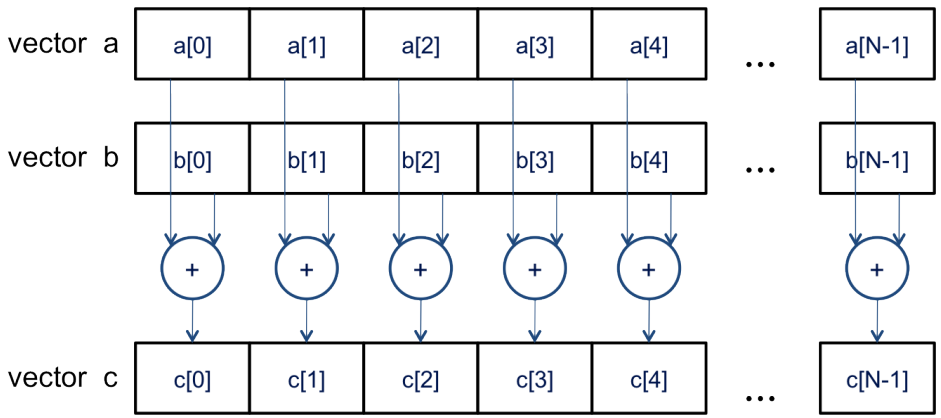


Then a unique index for each thread is given by

- `int index = threadIdx.x + blockIdx.x * M;`

Example: vector addition (sequential code)

```
// Compute vector sum C = A+B
void vecAdd(float* a, float* b, float* c, int n) {
  for (i = 0, i < n, i++)
    c[i] = a[i] + b[i];
}

int main() {
   float a[N], b[N], c[N];
   // initialize a and b
    vecAdd(a, b, c, N);
   // display the results
   return 0;
}
```



Example: vector addition (kernel definition and invocation)

Kernel code on device

```
// Each kernel invocation performs one pair-wise addition
__global__ void vecAddKernel(float* a_d, float* b_d, float* c_d, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    c_d[i] = a_d[i] + b_d[i];
}
```

Kernel invocation on host

```
__host__ int vectAdd(float* a, float* b, float* c, int n) {
    ...
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256), 256>>>(a, b, c, n);
    ...
}
```

Function declarations in CUDA

Functions in CUDA programs can be:

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

- A kernel function must return void

Example: vector addition (kernel code revisited)

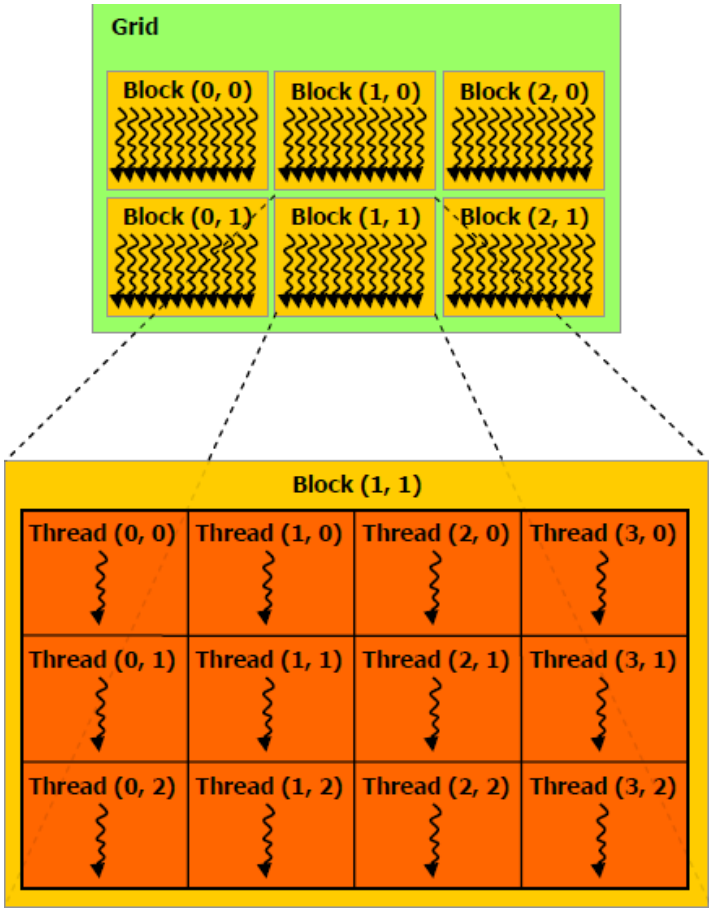
What if `blockDim.x` does not divide `N`?

```
// Each thread performs one pair-wise addition
__global__ void vecAddKernel(float* a_d, float* b_d, float* c_d, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i<n)
        c_d[i] = a_d[i] + b_d[i];
}
```

Guarded execution to ensure that we are not going out from the n elements

Identifying threads inside a grid

- In general, the GPU offers a **grid** of threads, divided into blocks (**thread blocks**), and each block is further divided into **threads**
- The **grid** of **thread blocks** can actually be partitioned into 1, 2 or 3 dimensions



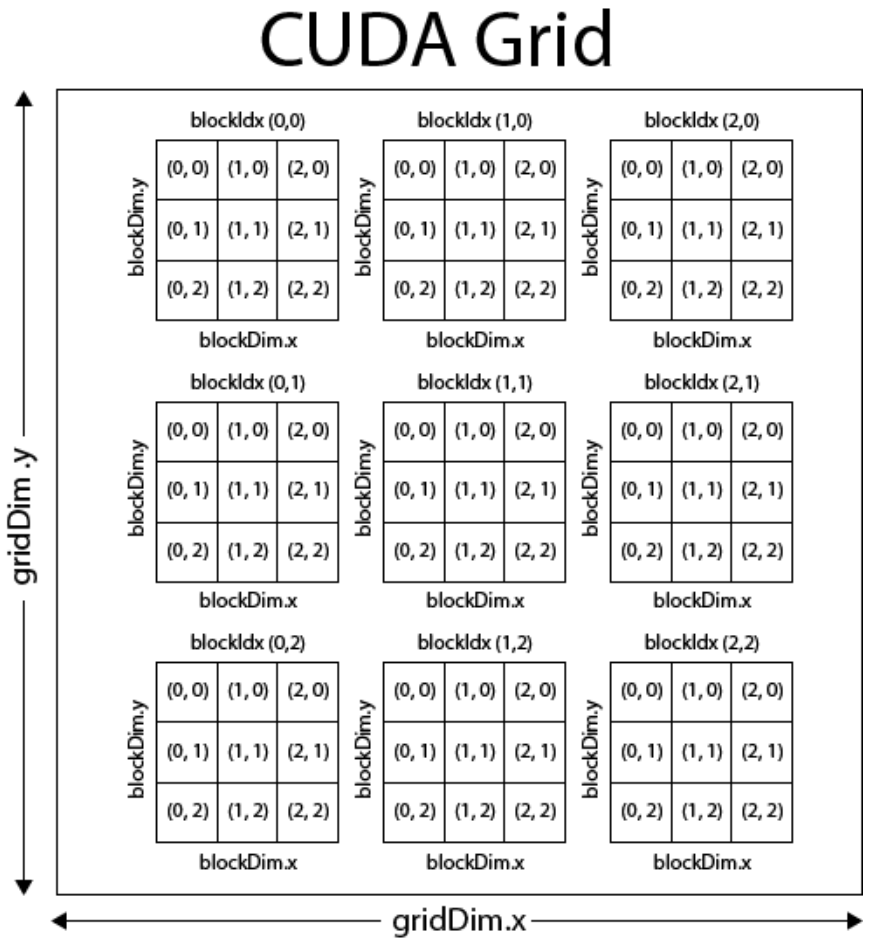
Identifying threads inside a grid

Each thread uses indices (1D, 2D or 3D) to decide what to do and data to work on (data decomposition)

- `blockIdx`
- `threadIdx`

defined inside:

- a grid with `gridDim` blocks,
- each block with `blockDim` threads



dim3 datatype

Data type to define variables to hold block and grid dimensionalities

```
__host__ int vectAdd(float* a, float* b, float* c, int n) {
    ...
    // Run ceil(n/256) blocks of 256 threads each
    dim3 DimGrid(ceil(n/256), 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernnel<<<DimGrid,DimBlock>>>(a, b, c, n);
    ...
}
```

Thread scheduling and execution

- Threads in a block are divided in 32-thread warps (scheduling units in SM)
- Example: if 3 blocks assigned to an SM, each with 256 threads, how many warps are there in an SM?
  - Each block is divided into $256/32 = 8$ warps
  - There are $8 * 3 = 24$ warps
- At any point in time, only one of the 24 warps will be selected for instruction fetch and execution (multithreading)

Laboratory examples (2)

1. Compile, run and trace `hello2.cu` in `helloworld` directory. Change the kernel definition and invocation so that multiple blocks (and one thread per block) or multiple threads inside a single block are used. Check that the result is always the one expected. Observe in trace data movement between host and device or vice-versa
2. Complete the kernel code for `VectorAdd.cu` in `Add` directory assuming it is invoked on both blocks and threads
3. Complete the kernel code for `MatrixAdd.cu` in `Add` directory assuming it is invoked on both two dimensional blocks and threads

Outline

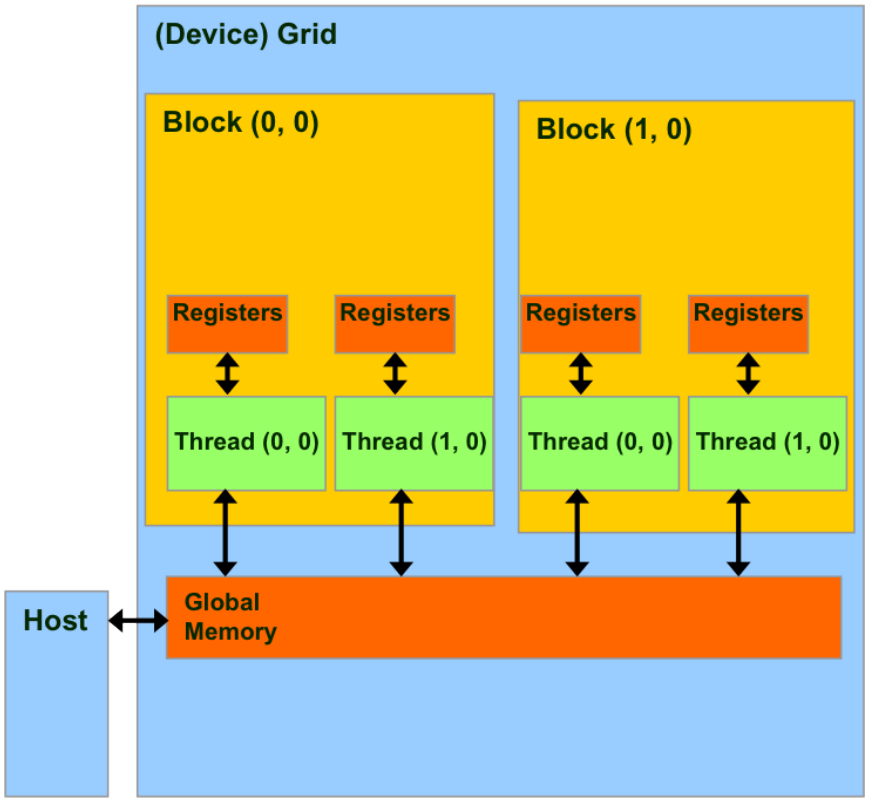
Partial view of the device memory

Device code can

- read and write **thread registers**
- read and write **grid global memory**

Host code can

- Allocate data in **grid global memory**
- Transfer data between **host** and **grid global** memories



Basic device memory management

Global memory

- Contents visible to all threads
- Variables in grid global memory declared using `__device__` attribute

```
#define N 1000

__device__ int A[N]; // declared outside function and kernel bodies.
                     // Lifetime: application

__global__ kernel() {
     int tid = blockId.x * blockDim.x + threadId.x;

     A[tid]++;
}
```

- Also dynamically allocatable using `cudaMalloc` and `cudaFree`

Basic device memory management

`cudaMalloc`

- Allocates object in the device global memory
- Two parameters: pointer to the allocated object, size of allocated object in bytes

`cudaFree`

- Frees object from device global memory
- Parameter: pointer to object

Example: vector addition (host code) (1)

```
int main() {
   float a[N], b[N], c[N];       // vectors in host memory
   float *dev_a, *dev_b, *dev_c; // pointers to vectors dynamically allocated in global memory

   // Allocate a, b and c on the device
   cudaMalloc( &dev_a, N * sizeof(float) ) ;
   cudaMalloc( &dev_b, N * sizeof(float) ) ;
   cudaMalloc( &dev_c, N * sizeof(float) ) ;

   ...

   // kernel invocation
   dim3 DimGrid(ceil(N/256), 1, 1);
   dim3 DimBlock(256, 1, 1);
   vecAddKernel<<<DimGrid,DimBlock>>>(dev_a, dev_b, dev_c, N);

   ...

   // Free the memory allocated on device
   cudaFree( dev_a );
   cudaFree( dev_b );
   cudaFree( dev_c );

   return 0;
}
```

Basic device memory management (cont.)

Memory data transfer

- `cudaMemcpy(...)`
  - Synchronous
    - Blocks the CPU until the copy is complete
    - Copy begins when all preceding CUDA calls have completed
  - 4 parameters: pointer to destination, pointer to source, number of bytes to be copied and type of transfer
- `cudaMemcpyAsync(...)`
  - Asynchronous, does not block the CPU, allowing the overlap of data transfer and computation
  - Additional `stream` parameter that if not `0` can be used to synchronize

Example: vector addition (host code) (2)

```
int main() {
   float a[N], b[N], c[N];       // vectors in host memory
   float *dev_a, *dev_b, *dev_c; // pointers to vectors dynamically allocated in global memory

   cudaMalloc( &dev_a, N * sizeof(float) ) ;
   cudaMalloc( &dev_b, N * sizeof(float) ) ;
   cudaMalloc( &dev_c, N * sizeof(float) ) ;

   // Initialize a and b in host memory (same as sequential)

   // Copy a and b from host to device memory
   cudaMemcpy( dev_a, a, N * sizeof(float), cudaMemcpyHostToDevice );
   cudaMemcpy( dev_b, b, N * sizeof(float), cudaMemcpyHostToDevice );

   dim3 DimGrid(ceil(N/256), 1, 1);
   dim3 DimBlock(256, 1, 1);
   vecAddKernel<<<DimGrid,DimBlock>>>(dev_a, dev_b, dev_c, N);

   // Copy back c from device to host
    cudaMemcpy( c, dev_c, N * sizeof(float), cudaMemcpyDeviceToHost );

   // Display the results

   cudaFree( dev_a ); cudaFree( dev_b ); cudaFree( dev_c );

   return 0;
}
```

Sharing data between GPUs

Options:

- Explicit copies via host
- Peer-to-peer memory access: Direct copy from pointer on one device to pointer on another device
  `cudaMemcpyPeer( void *dst, int dstDevice, void *src, int srcDevice, size_t count )`
- Zero-copy shared host array: direct device access to host memory (not covered in this course)

Laboratory examples (3)

1. Compile, trace and run `VectorAdd.cu` in `Add` directory. Run with different numbers of blocks and threads. Do you observe any difference?
2. Complete the host code, compile and run `MatrixAdd.cu` in `Add` directory

Outline

Threads vs. blocks

Unlike blocks, threads have mechanisms to:

- Synchronize: the instruction `__syncthreads()` forces all warps (i.e. all threads in a block) to wait until the rest have reached the same point
- Communicate via memory: let's take a look at a more complete view of the device memory ...

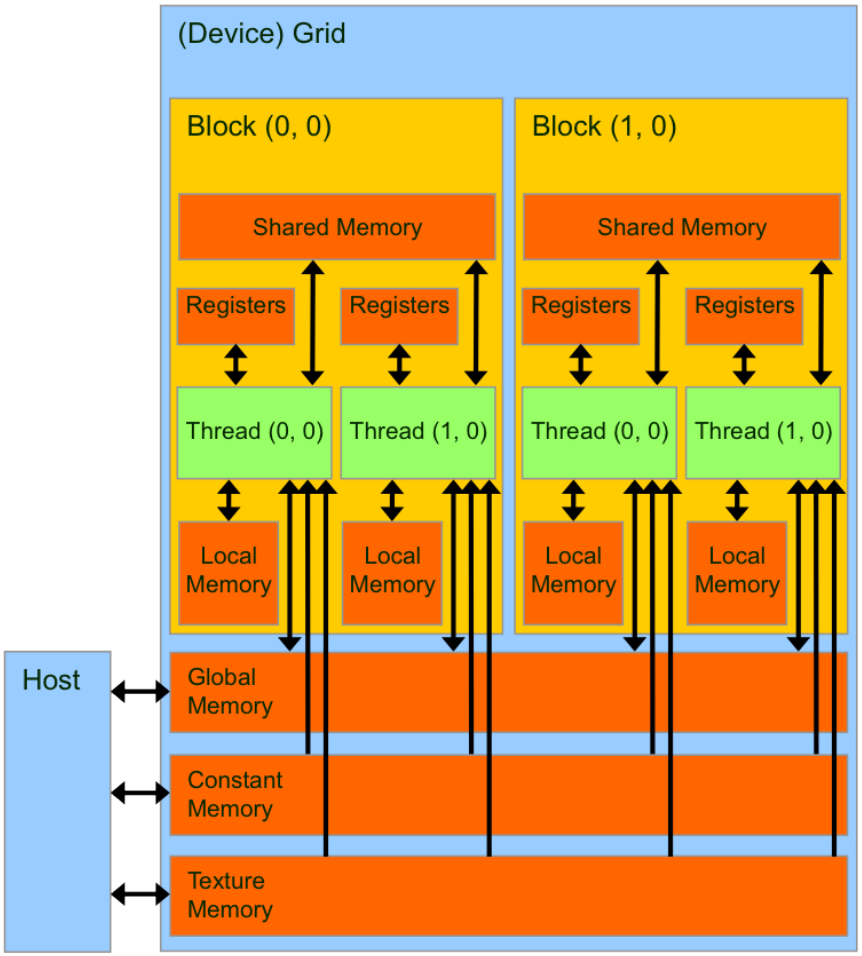
A more complete view of the device memory

Each thread can:

- R/W per-thread registers and local memory
- R/W per-block shared memory (100 times faster than global)
- R/W per-grid global memory
- Read only per-grid constant and texture memories

Host code can:

- R/W global, constant, and texture memories



A more complete view of the device memory

Shared memory

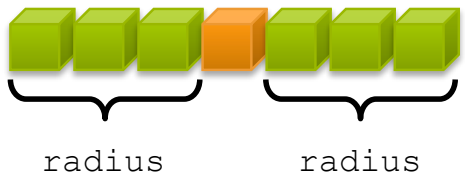
- Within a block, used to share data among threads
- Allocated per block, data is not visible to threads in other blocks
- Declared using `__shared__`

```
#define N 1000

__global__ kernel() {
     __shared__ int A[N];  // declared inside kernel bodies.
                           // Scope: block. Lifetime: kernel call
     int tid = threadId.x;

     A[tid]++;
     ...
}
```

Example: 1D stencil

Consider applying a stencil to a 1D array of elements

- Each output element is the sum of input elements within a $radius$

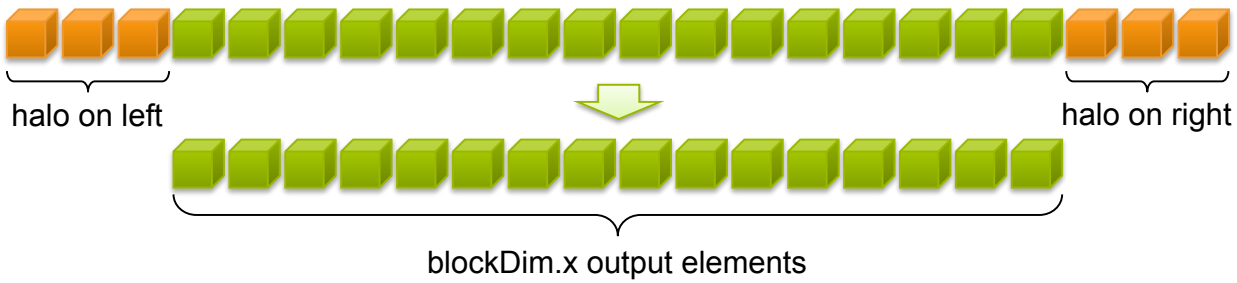


- Each thread processes one output element
- Each input element is read $2 * radius + 1$ times

Example: 1D stencil (cont.)

Cache data in shared memory

- Read (`blockDim.x + 2 * radius`) input elements from global memory to shared memory
- Compute `blockDim.x` output elements
- Write `blockDim.x` output elements to global memory



Laboratory examples (3)

1. Edit `Stencil.cu` in `Stencil` directory. Look at the two implementations of the kernel and try to understand how to make use of shared memory. Complete the kernels
2. Compile, run and trace `Stencil.cu` in `Stencil` directory, which invokes the kernel that directly accesses to global memory and the kernel that makes use of shared memory. Compare performance results of execution on CPU and GPU (global and shared memory)

Parallelism (PAR)

Programming with CUDA

Eduard Ayguadé and Josep R. Herrero

Computer Architecture Department
Universitat Politècnica de Catalunya

2012/13-Spring