TECHNISCHE UNIVERSITÄT WIEN
Fakultät für Informatik
Cyber-Physical Systems Research Unit

CUDA Programming
(Basics, Cuda Threads, Atomics)

Presenter: Ezio Bartocci

Outline

- How to write simple kernels and how to execute them ?
- How to allocate basic memory ?
- How to coordinate the execution ?

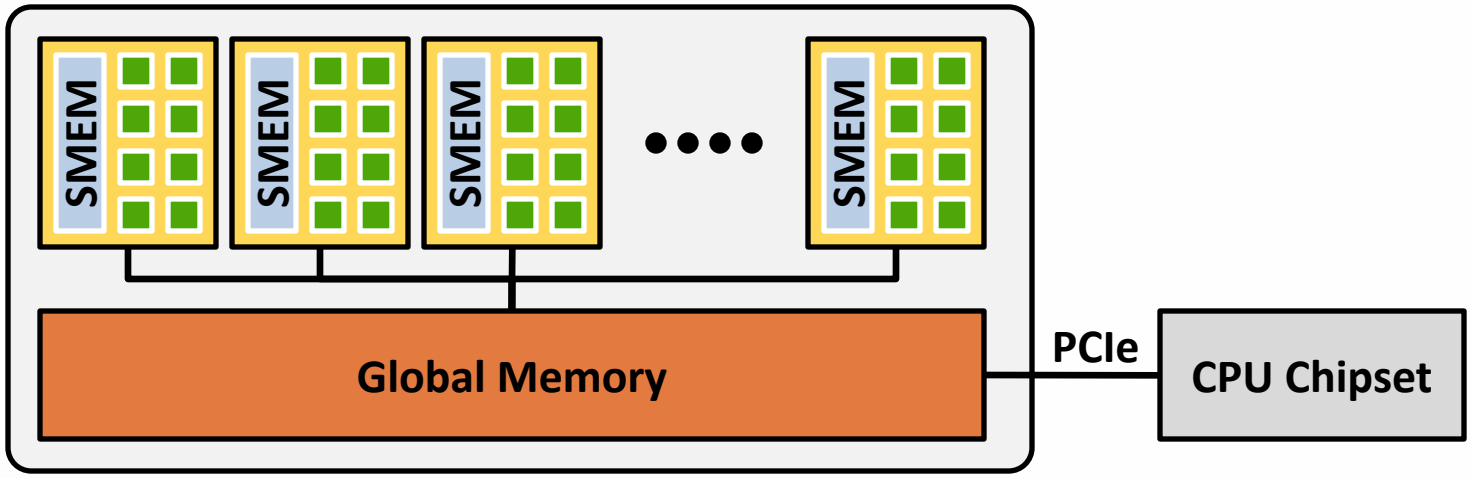
WRITING/EXECUTING A KERNEL

CUDA Programming Model

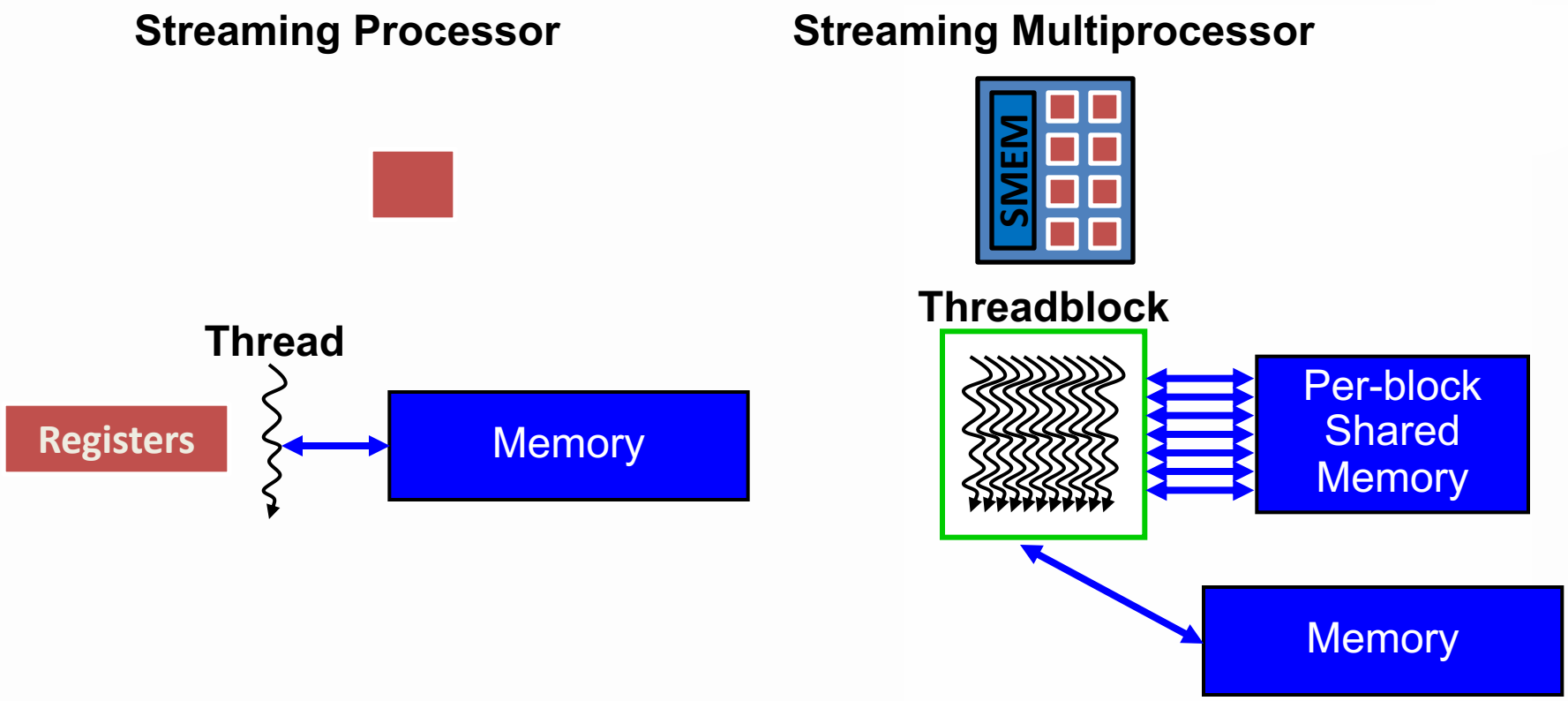
- **A Kernel is the code that is launched and executed on a device by multiple threads**

- **The kernel execution is hierarchical**
  - Threads are grouped into blocks
  - Blocks are grouped into grids
  - Each thread is free to execute a unique code path
  - The code has a built-in thread and block ID variables

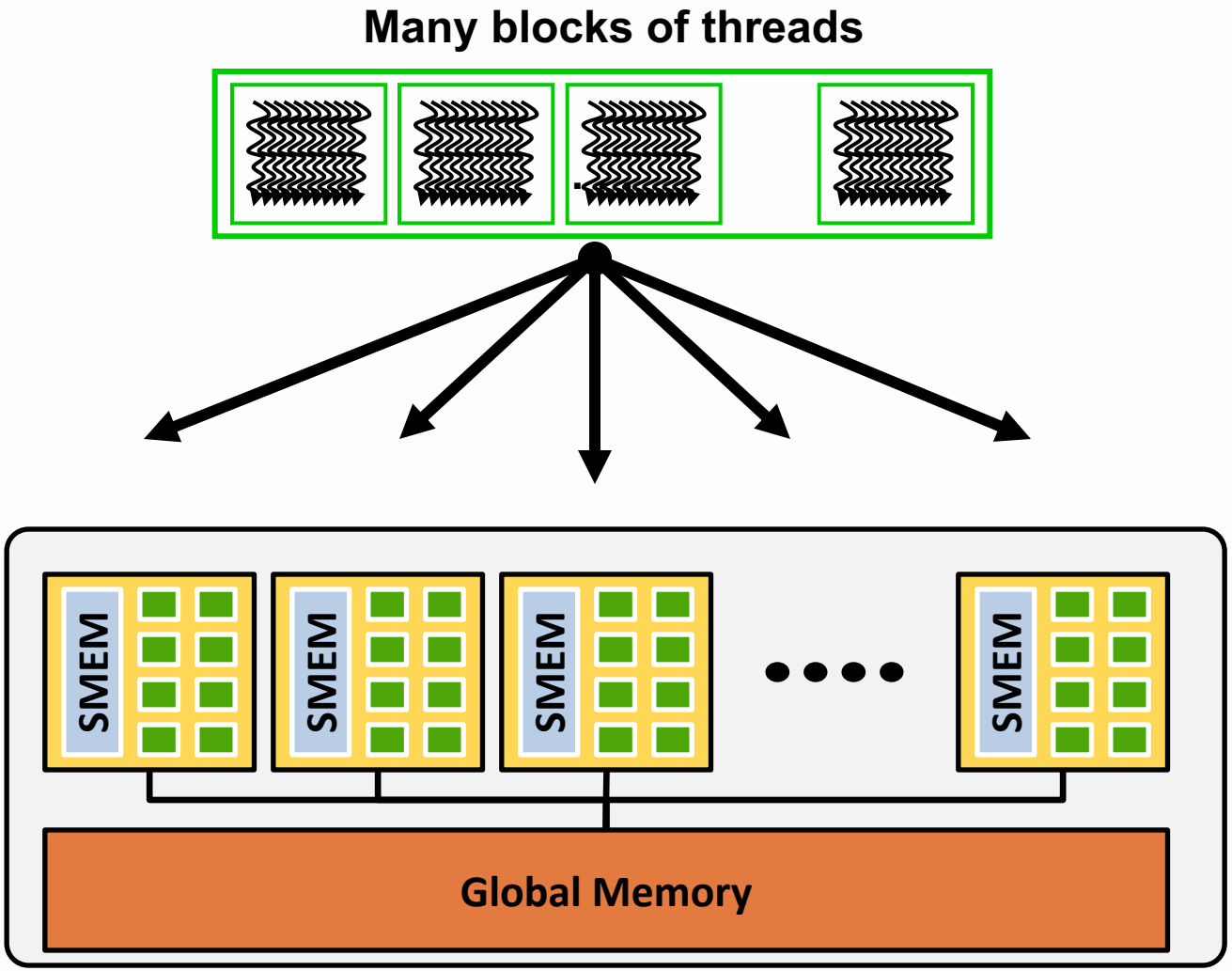
GPU Hardware Architecture - High level



Blocks of threads



Grid of blocks of threads

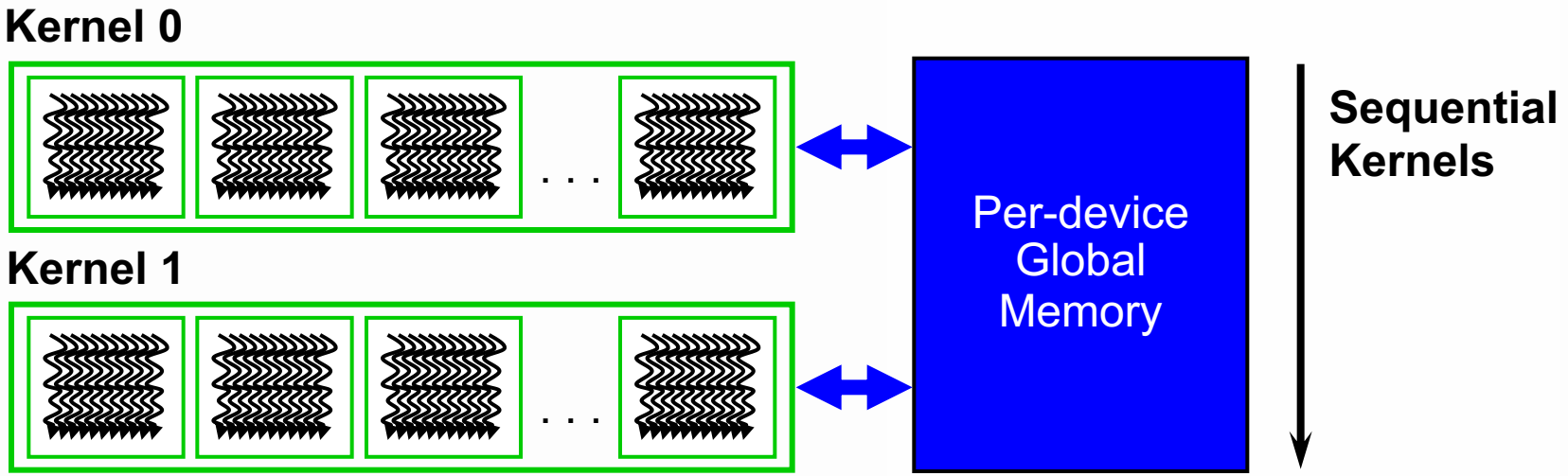


Thread Hierarchy

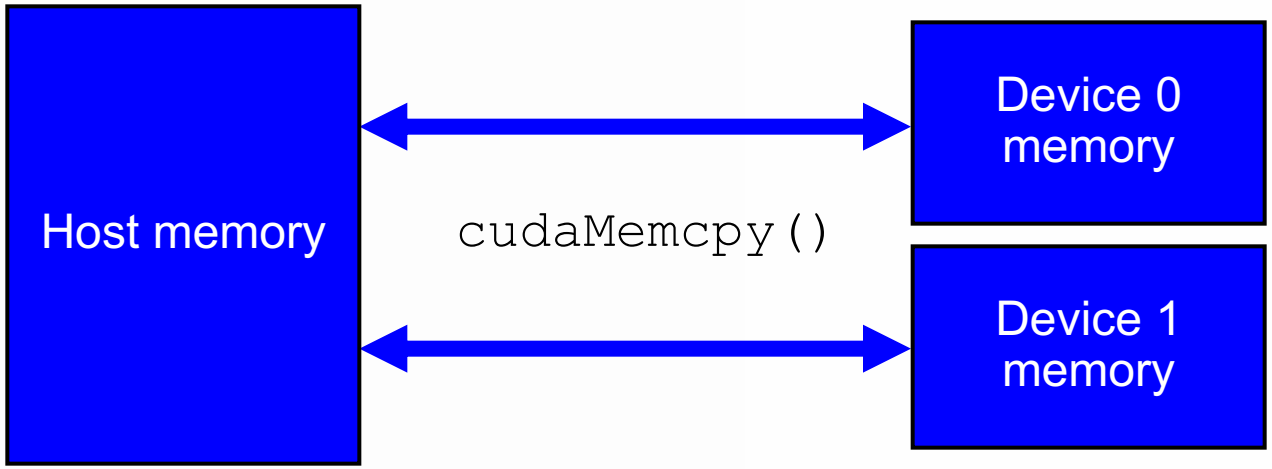
- **Threads launched for a parallel section are partitioned into thread blocks**

- **Thread block is a group of threads that can:**
  - Synchronize their execution
  - Communicate via shared memory

Memory Model



Memory Model



Example: Vector Addition Kernel

Device Code

```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Host code for vecAdd

```
// allocate and initialize host (CPU) memory
float *h_A = …,    *h_B = …; *h_C = …(empty)

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
   cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float),
   cudaMemcpyHostToDevice) );

// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

```
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// copy result back to host memory
cudaMemcpy( h_C, d_C, N * sizeof(float),
   cudaMemcpyDeviceToHost) );

// do something with the result…

// free device (GPU) memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Different Kernels

```
__global__ void kernelA( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 5;
}
```

Output: 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

```
__global__ void kernelB( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernelC( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Code executed on GPU

- **C/C++ with some restrictions:**
  - No variable number of arguments
  - No static variables
  - No recursion

- **Functions has special qualifiers:**
  - **__global__** : called only by CPU, execute in GPU must return void
  - **__device__** : called/execute in GPU functions, but not from the CPU
  - **__host__** : can be called from CPU
  - **__host__** and **__device__** qualifiers can be combined
  - sample use: overloading operators

Memory Spaces

- **Separate memory spaces between GPU/CPU**
  - The data is moved between GPU and CPU through the PCIe
  - We need special functions to allocate memory into GPU

- **Pointers**
  - Pointers contains addresses for CPU and GPU that are indistinguishable
  - Not correct deferencing memory addresses in GPU using CPU functions can crash the system

Allocation / Deallocation of GPU Memory

- **Host (CPU) manages device (GPU) memory:**
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

cudaMemcpy

- cudaMemcpy(void *dst, void *src, size_t nbytes,
  enum cudaMemcpyKind direction);
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - Does not start copying until previous CUDA calls complete

- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

- Non-blocking copies are also available

Code Example 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
```

Code Example 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
```

Code Example 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

cudaMemset( d_a, 0, num_bytes );
cudaMemcpy( h_a, d_a, num_bytes,
cudaMemcpyDeviceToHost );
```

Code Example 1

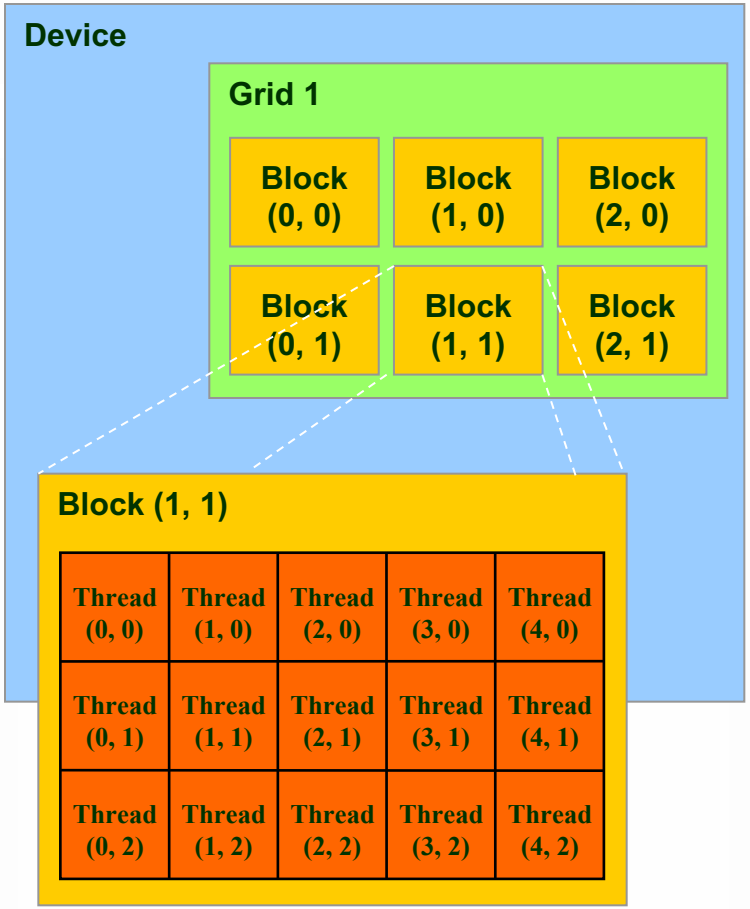
```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);
    int *d_a=0, *h_a=0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");
    free( h_a );
    cudaFree( d_a );
    return 0;
}
```

IDs and Dimensions

- **Threads:**
  - 3D IDs, unique within a block
- **Blocks:**
  - 2D IDs, unique within a grid
- **Dimensions set at launch**
  - Can be unique for each grid
- **Built-in variables:**
  - threadIdx, blockIdx
  - blockDim, gridDim



Kernel and 2D Indexing

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
```

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
```

```
int main(){
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x  = dimx / block.x;
    grid.y  = dimy / block.y;

    kernel<<<grid, block>>>( d_a, dimx, dimy );

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int row=0; row<dimy; row++)
    {
        for(int col=0; col<dimx; col++)
            printf("%d ", h_a[row*dimx+col] );
        printf("\n");
    }

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Thread blocks

- **No assumptions about interleaving can be made**
  - threads can be executed in any order
  - we cannot assume that the execution of threads are completed without pre-emption
  - threads may run sequentially or concurrently
- **Different blocks of threads are completely independent and cannot be synchronized**
- **The independence of different threads blocks enables to scale the computation**

HOW TO COORDINATE THE THREADS' EXECUTION ?

Global Communication

- The writes of all the threads complete before the next kernel starts
- Decomposing the computation requires to take this into consideration

```
step1<<<grid1,blk1>>>(...);
// The system ensures that all
// writes from step1 complete.
step2<<<grid2,blk2>>>(...);
```

Race Conditions

```
threadId:0

vector[0] = 10;
...
x = vector[0];
```

```
threadId:817

vector[0] += 1;
...
x = vector[0];
```

- What is the value of `x` in thread 0 and in thread 817?

Atomics

- When the atomic operations are used, only a single thread has the access to a piece of memory while the others need to wait
- Atomic means the the function cannot be divided or interrupt
- Race condition is avoided
- Many atomic instructions are available:

  Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor

Atomics

```
// Example of using Atomics
// Computing an histogram of colors
__global__ void histogram(int* color, int* buckets)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int c = colors[i];
  atomicAdd(&buckets[c], 1);
}
```

- Atomics instructions might be slower and in the worst case

Example: Global Min/Max (Naive)

```
__global__
void global_max(int* values, int* gl_max)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int val = values[i];
  atomicMax(gl_max,val);
}
```

Example: Global Min/Max (Better)

```
__global__
void global_max(int* values, int* max,
                int *regional_maxes,
                int num_regions)
{
  // i and val as before …
  int region = i % num_regions;
  if(atomicMax(&reg_max[region],val) < val)
  {
    atomicMax(max,val);
  }
}
```

Important Tips

- Single value causes serial bottleneck
- Create hierarchy of values for more parallelism

Summary

- We have showed how to develop simple kernels
- We showed how to coordinate the execution using atomics
- It is important to plan how to decompose the computation.