Cuda programming model (Repaso y Continuación)

Nancy Hitschfeld Kahler
(nancy@dcc.uchile.cl)

Departamento de Ciencias de la Computación
Universidad de Chile

(**Basada en CUDA C/C++ Basics**
Supercomputing 2011 Tutorial
Cyril Zeller, NVIDIA Corporation)

Contenido

- Computación heterogénea
- Programando en Cuda
- Ejemplos: organizando el espacio de cálculo

•

Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)

Host

Device

Heterogeneous Computing

device code

host code

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex - RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

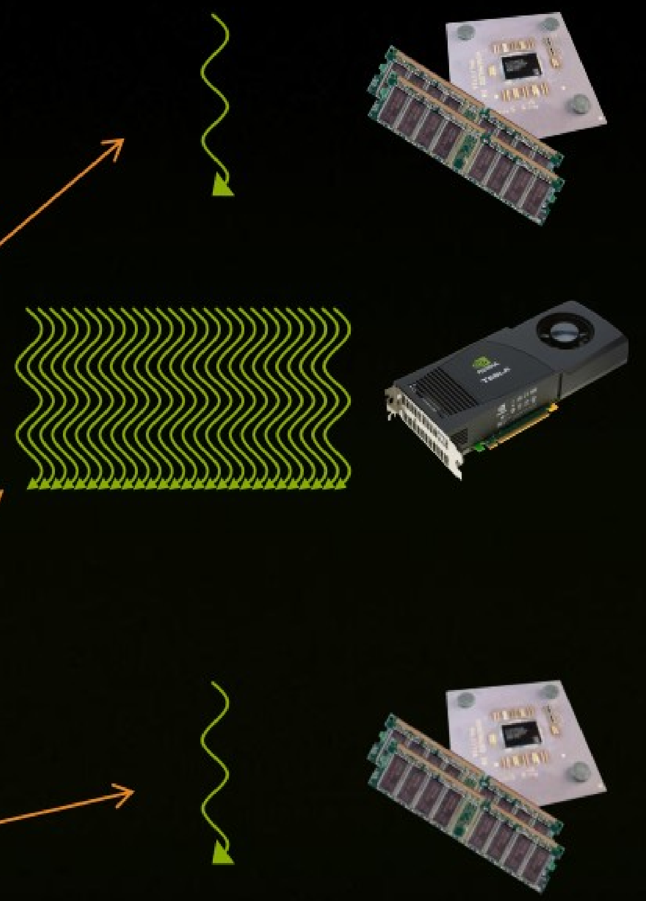
parallel function

serial function

serial code

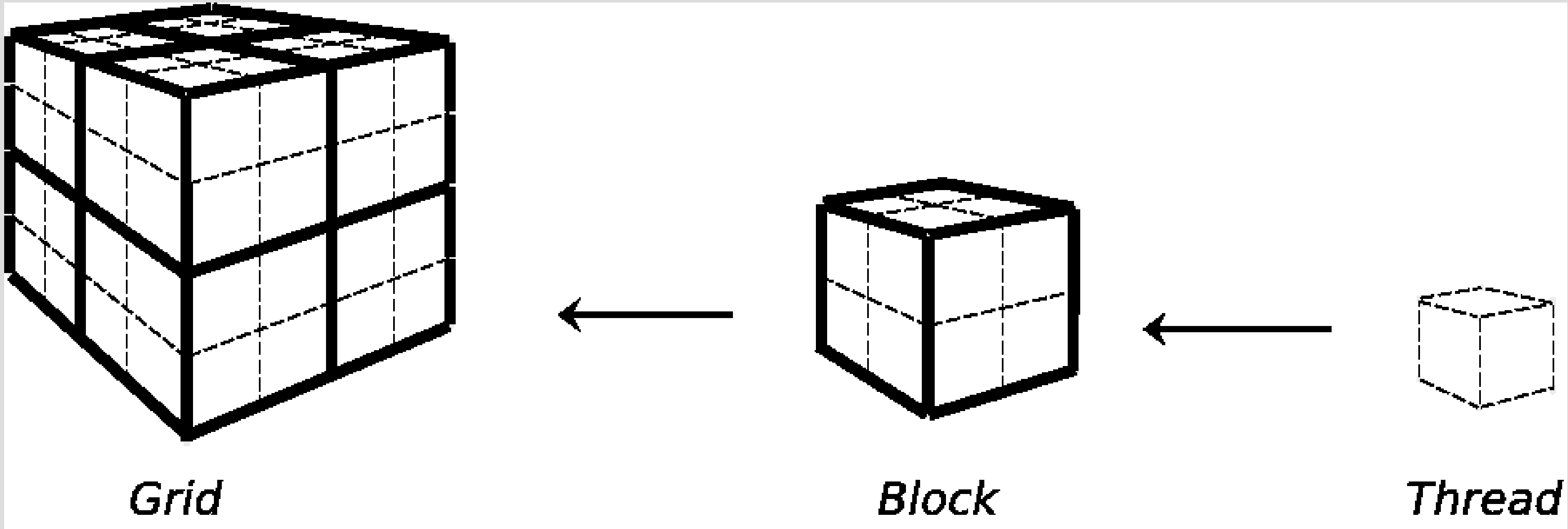
parallel code

serial code



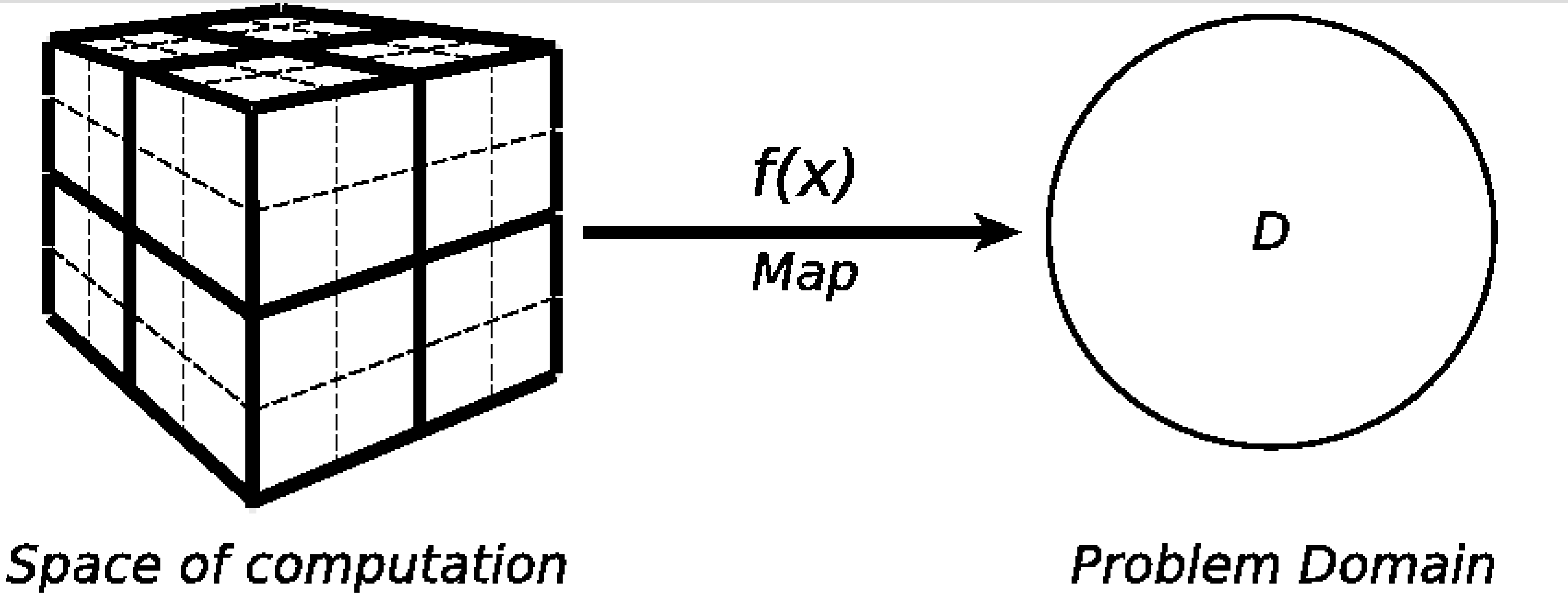
GPU mapping

- Space of computation structure



GPU mapping

- Space of computation structure



Programming in Cuda

- Code Adds two vectors A and B of size N and stores the result into vector C

```
// Kernel definition. Run on device

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

int main() // Run on CPU
{
    ...
    // Kernel invocation with 1 block and N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Programming in Cuda

- It is necessary to allocate memory
  - Host and device memory are separeted 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

https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Programming in Cuda

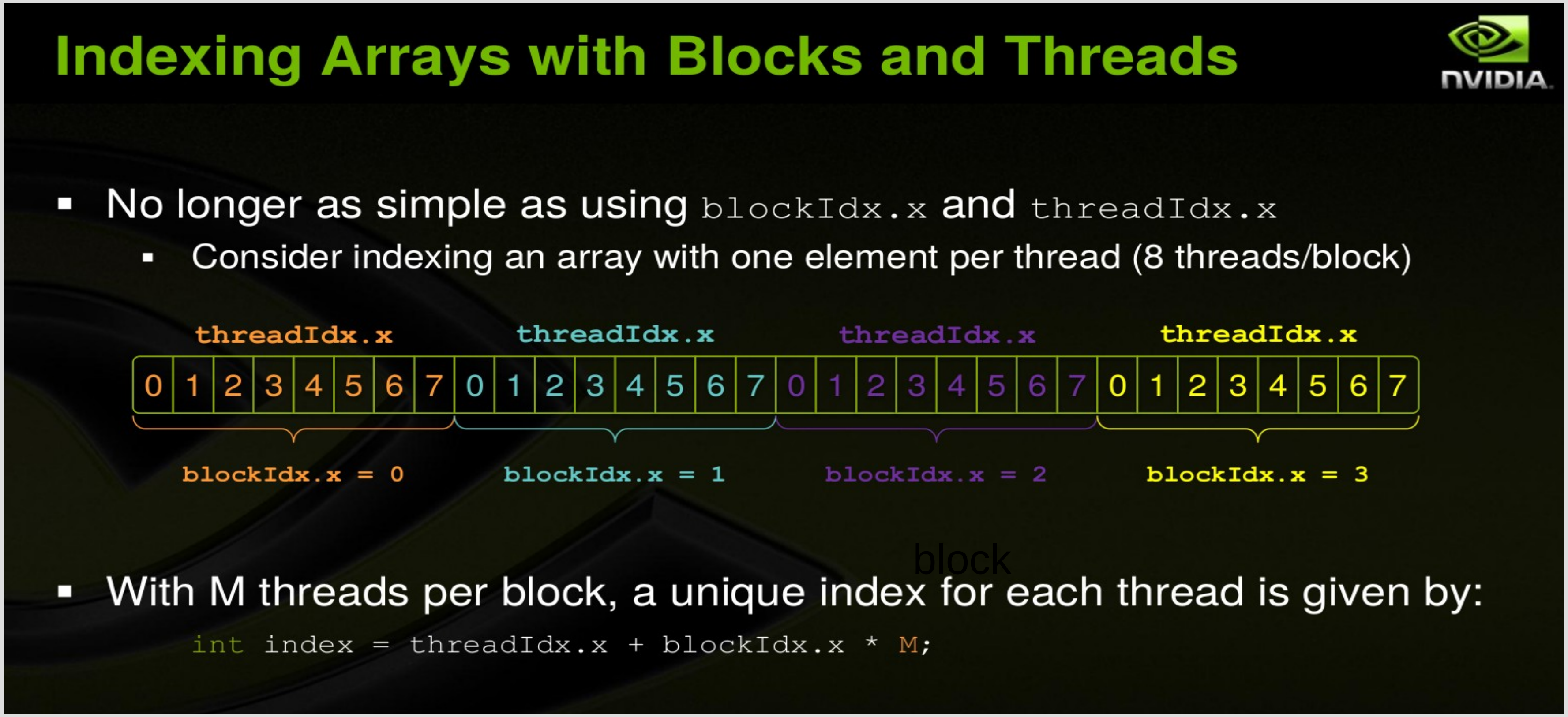
```
#define N 512
int main(void) {
    int *a, *b, *c;     int *d_a, *d_b, *d_c;
    int size = N*sizeof(int)
    // Alloc space for device  for a, b y c
    cudaMalloc((void **)&d_a, siza );
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
    //
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size); random_ints(c, N);
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    add<<<1,N>>>(d_a, d_b, d_c);
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

- With M threads per block, a unique index for each thread is given by:

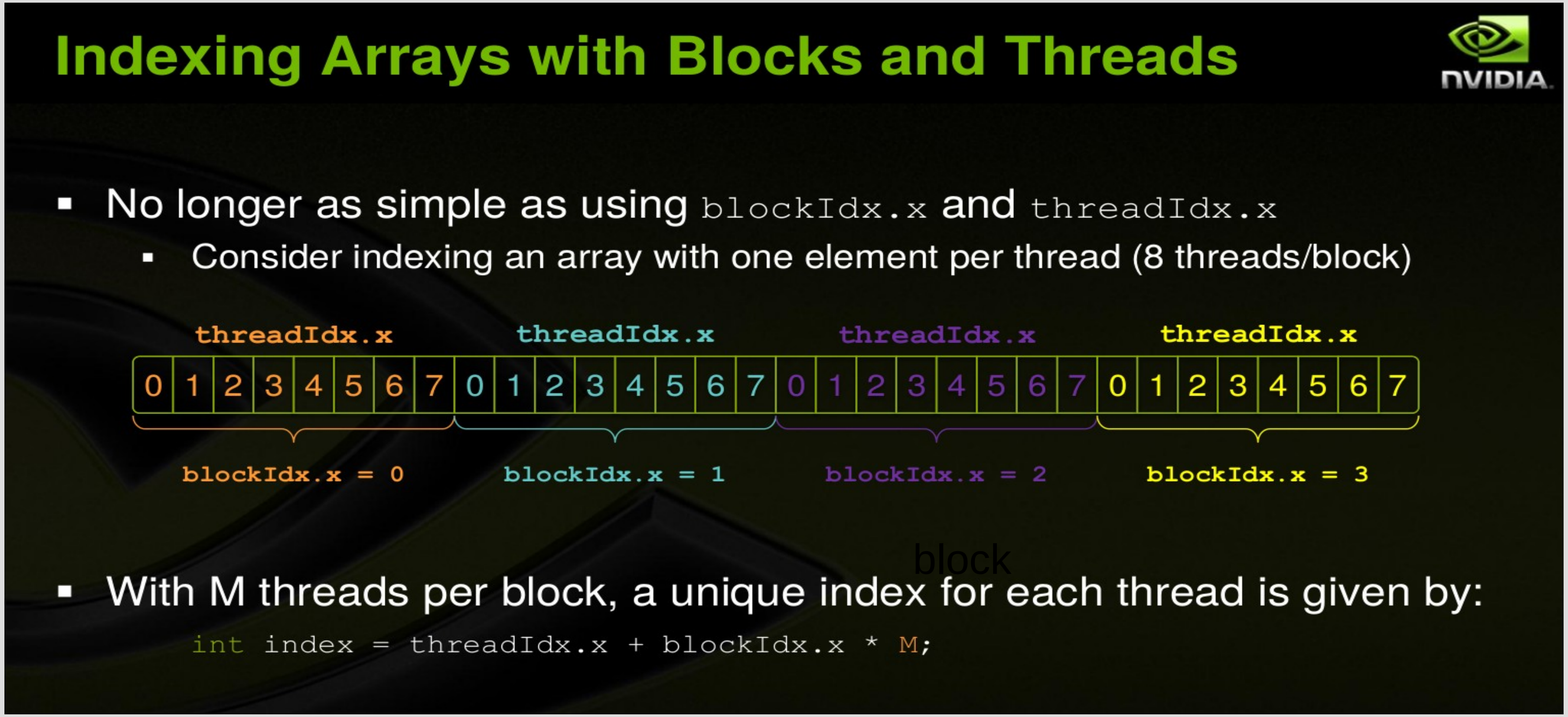
```
int index = threadIdx.x + blockIdx.x * M;
```



¿Qué thread se encarga del index=21?

Nota: Usar en vez de M → blockDim.x

int index = threadId.x + blockId.x*blockDim.x



¿como cambia la llamada a add? (SI N es multiplo de blockDim.x o THREADS_PER_BLOCK)

add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

En el kernel:
int index = threadIdx.x + blockId.x*blockDim.x

Qué problemas aparecen con el uso de bloques?

```
__global__ void add(int *a, int *b, int *c, int n) {

    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
       c[index] = a[index] + b[index];
}
```

Cómo se actualiza la llamada a add?

```
    add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

Programming in Cuda

- ¿Por qué usar bloques y threads?
  - A diferencia de los bloques, los threads proveen mecanismos eficientes para :
    - Comunicación
    - Sincronizacion
    - Para ejemplo ir a:

      "**Cooperating threads**" **slide 46 en... sc11-cuda-c-basics.pdf**)