UNIVERSITÀ di VERONA

Dipartimento di INFORMATICA

PARCO Parallel Computing Lab

NVIDIA CUDA®

Part I

Laurea magistrale in Ingegneria e Scienze Informatiche
Laurea Magistrale in Medical Bioinformatics

Nicola Bombieri – Federico Busato

Agenda

- What is CUDA?
- Calling a Device Function
- CUDA Concepts
- CUDA Keywords:
  - Built-I
- Memory Management
- Error handling
- CUDA Compiler Driver
- Timing the code
- Examples
- Exercises
- References
- Books

What is CUDA?

- **CUDA** (**C**ompute **U**nified **D**evice **A**rchitecture) is a parallel computing platform and programming model that makes using a GPU for general purpose computing
- CUDA is small extension of C/C++ language
- With CUDA you can accelerate your C/C++ code by moving the computationally intensive portions of your code to an NVIDIA GPU

Calling a Kernel Function

- A kernel is a function callable from the host and executed on the CUDA device
- The kernel is the heart of your CUDA code

```
__global__ void MyKernel (int* Array, int size, …) {
    …
}

int main() {
    …
    MyKernel<<<…>>>(Array, size, …);
    …
}
```

- A kernel is defined using the __global__ declaration specifier
- The number of CUDA threads that execute a kernel is specified using a new <<<…>>> execution configuration syntax
- The kernel **must** have return type **void**

CUDA Architecture Overview

Built-in Variables

- `threadIdx.x`

  Contains the thread index within the block (dimension X)

- `blockIdx.x`

  Contains the thread index within the grid (dimension X)

- `blockDim.x`

  Contains the dimension of the block (number of threads) (dimension X)

- `gridDim.x`

  Contains the dimension of the grid (number of blocks) (dimension X)

✓ Common case of threads indexing within the Kernel:

```
GlobalThreadIndex = blockIdx.x * blockDim.x + threadIdx.x
```

✓ Useful Kernel Function:

`printf(…)` // need synchronization (implit/explicit) after kernel

Common CUDA Code Steps

1) Initialize host data structures
2) Allocate device data structures `cudaMalloc`
3) Copy host data structures to device `cudaMemcpy`
4) Invoke kernel function `MyKernel<<<>>>`
5) Copy device result to host `cudaMemcpy`
6) Free host and device data structures `cudaFree`

   When the process terminates all allocated memory is released

7) Reset the device

   `cudaDeviceReset`

CUDA memory management (I)

Device Variables (symbols):

```
cudaMemcpyToSymbol(symbol, void* src, size_t size, offset=0)
```

- Copies SIZE <u>bytes</u> from the memory area pointed by SRC to the memory area pointed to by OFFSET <u>bytes</u> from the start of SYMBOL
- Pointer arithmetic is <u>not</u> valid on SYMBOL

```
cudaMemcpyFromSymbol(void* dest, symbol, size_t size, offset=0)
```

- Copies SIZE <u>bytes</u> from the memory area OFFSET <u>bytes</u> from the start of SYMBOL to the memory area pointed to by DEST
- Pointer arithmetic is <u>not</u> valid on SYMBOL

CUDA memory management Example (I)

```
__global__ void MyKernel (int* Array, int size, …) { …  }

int main() {
    …
    int* devArray;
    int size = 120;
    cudaMalloc(&devArray, size * sizeof(int));
    cudaMemcpy(devArray, hostArrayA, size * sizeof(int), cudaMemcpyHostToDevice);
    //valid: cudaMemcpy(devArray + 4, hostArrayA, …);
    MyKernel<<<…>>>(devArray, size, …);
    cudaMemcpy(hostArrayB, devArray, size * sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(devArray);
    cudaDeviceReset();
}
```

CUDA memory management Example (II)

```
__device__ sArray[128];

__global__ void MyKernel (…) {
    …
    sArray[4] = 14;
}

int main() {
    …
    cudaMemcpyToSymbol(sArray, hostArrayA, 4 * sizeof(int), 4);
    MyKernel<<<…>>>(…);
    cudaMemcpyFromSymbol(hostArrayB, sArray, 4 * sizeof(int), 4);
}
```

CUDA Error Handling (I)

- Every CUDA call (except kernel launches) return an error code of type cudaError_t
  - No error = "cudaSuccess"
  - otherwise an error code

```
cudaError_t err = cudaMalloc( &fooPtr, -1 );
if ( cudaSuccess != err )
    printf("Error! : %d\n", err);
```

- CUDA kernel invocations <u>do not return</u> any value. Error from a CUDA kernel call can be checked after its execution by calling cudaGetLastError()

```
fooKernel<<< x, y >>>(); // Kernel call
cudaDeviceSynchonize();
cudaError_t err = cudaGetLastError();
. . .
```

Important!!: Need to Synchronize Host and Device

- A human-readable description of the error can be obtained from

char *cudaGetErrorString(cudaError_t code);

CUDA Error Handling (II)

Elegant solutions:

```
#define cudaError( msg )    getLastCudaError(msg, __FILE__, __LINE__)
#define CudaSafeCall( err ) cudaSafeCall(err, __FILE__, __LINE__ )

inline void getLastCudaError(const char *errorMessage,
                             const char *file, const int line) {
    cudaDeviceSynchronize();
    cudaError_t err = cudaGetLastError();
    if (cudaSuccess != err) {
        std::cerr << file << " (" << line << ") : getLastCudaError()
        CUDA error : " << errorMessage << " : (" << (int) err << ") "
                  << cudaGetErrorString(err) << std::endl << std:: endl;

        std::exit(EXIT_FAILURE);
    }
}

inline void cudaSafeCall(cudaError err, const char *file,
                         const int line) {
    __getLastCudaError("", file, line);
}
```

Alternative timing of your Code (I)

Events:

```
cudaEvent_t startTimeCuda, stopTimeCuda;
cudaEventCreate(&startTimeCuda);
cudaEventCreate(&stopTimeCuda);

cudaEventRecord(startTimeCuda, 0); //0 is the default stream

…code…       //also host code and cuda functions
Kernel<<<>>>();  //one or more
…code…

cudaEventRecord(stopTimeCuda,0);
cudaEventSynchronize(stopTimeCuda);
float msTime;
cudaEventElapsedTime(&msTime, startTimeCuda, stopTimeCuda);
```

- Compute the elapsed time in milliseconds between two recorded events

Alternative timing of your Code (II)

Host Timer

```
// host timer init
// host timer start

…code…
Kernel<<<>>>();
…code…

cudaDeviceSynchronize();
// host timer stop
```

- Explicit synchronization barrier cudaDeviceSynchronize() is required to block CPU execution until all previously issued commands on the device have completed
- Without this barrier, this code would measure the kernel launch time and not the kernel execution time

IMPORTANT: the execution is non-deterministic ->Sometimes it is better to repeat many times the computation and than computes the average time to get a better approximation of the execution time.

CUDA Compiler Driver (I)

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
  - It automatically invokes all the necessary tools and compilers like cudacc, g++, ...
- Any executable with CUDA code requires the CUDA <u>runtime</u> library (`cudart`)
- This library must be in the standard library path or the environment variable LD_LIBRARY_PATH must contain the path to this library
  - Common case:

```
export LD_LIBRARY_PATH=/usr/local/cuda/<lib/lib64>:$LD_LIBRARY_PATH
export PATH=/usr/local/cuda/bin:$PATH
```

- To compile a CUDA program:

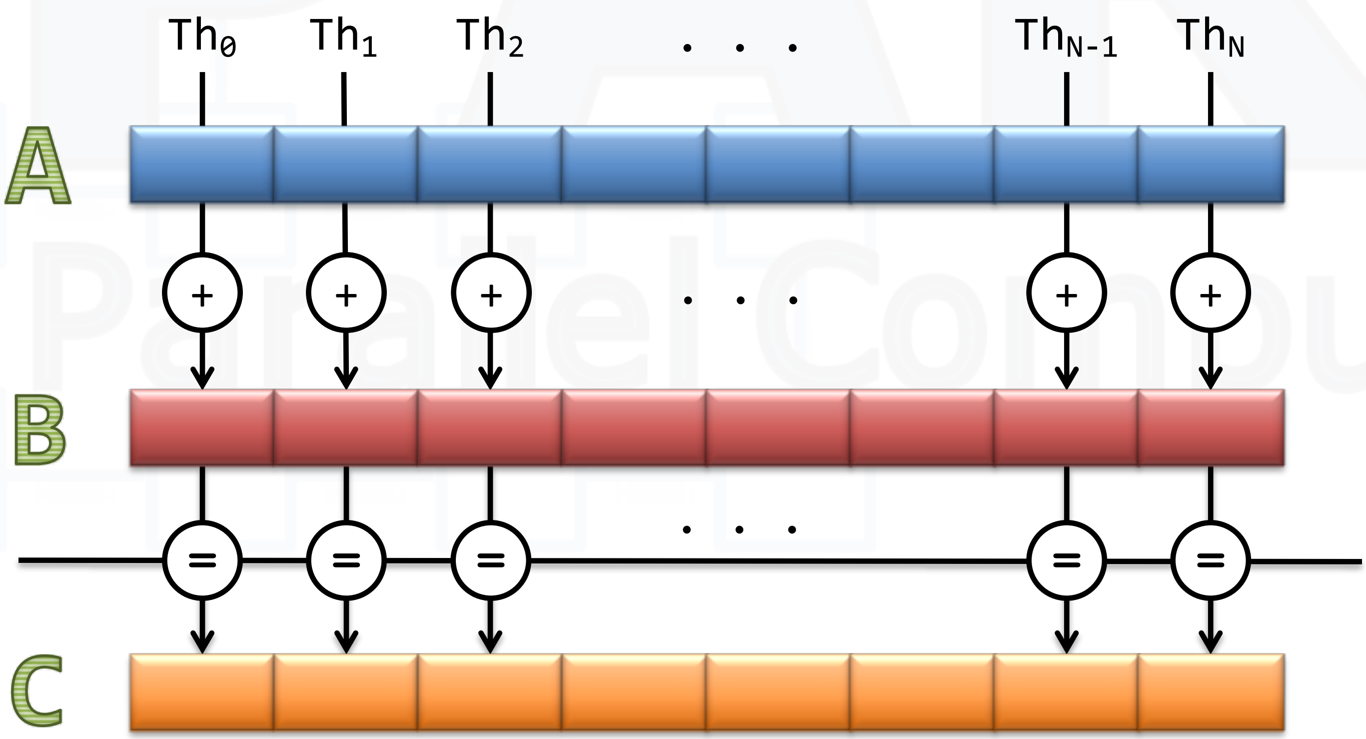
```
nvcc source.cu -o out.x
```

Some parameters:

-std=c++11 -arch=sm_62

Examples

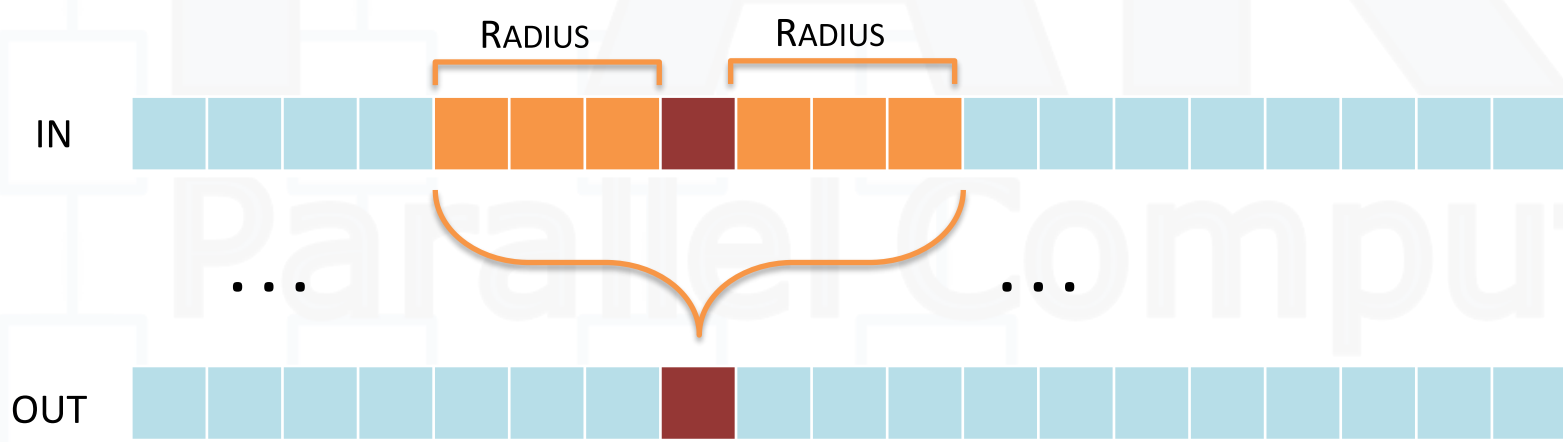
- Hello World
- Vector addition:



Exercise

1D Stencil (I)

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
  - If radius is 3, then each output element is the sum of 7 input elements

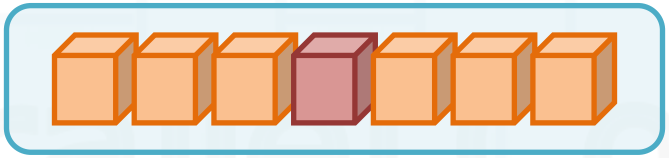


- Fundamental to many algorithms Standard discretization methods, interpolation, convolution, filtering

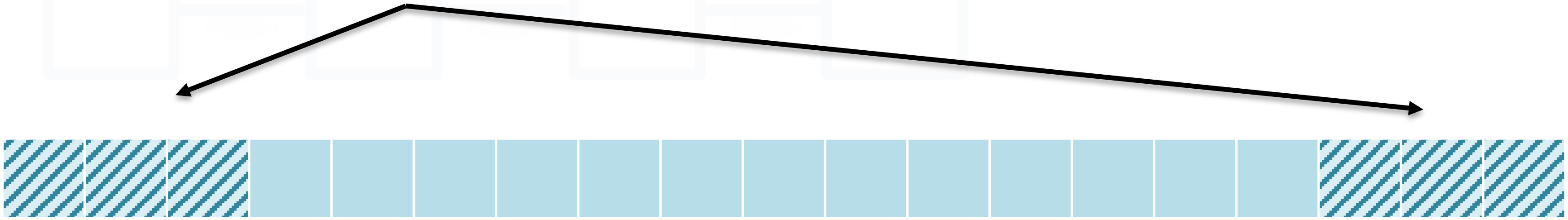
Exercise
1D Stencil (II)

- Each thread processes one output element
- With radius 3, each input element is read seven times
  - Input elements are read several times

Thread Computation



NOTE: handle left and right bounds



Exercise

Matrix Multiplication

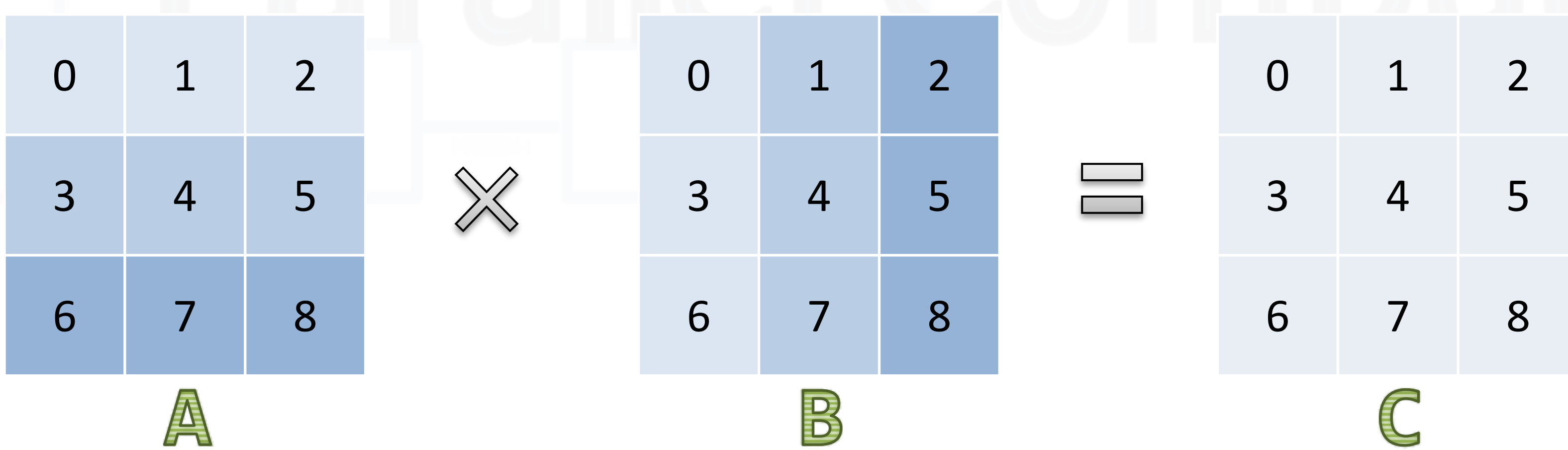
Simplest Version:

- The matrix must be linearized:

```
A[i][j] = A[i * rows + j]
```

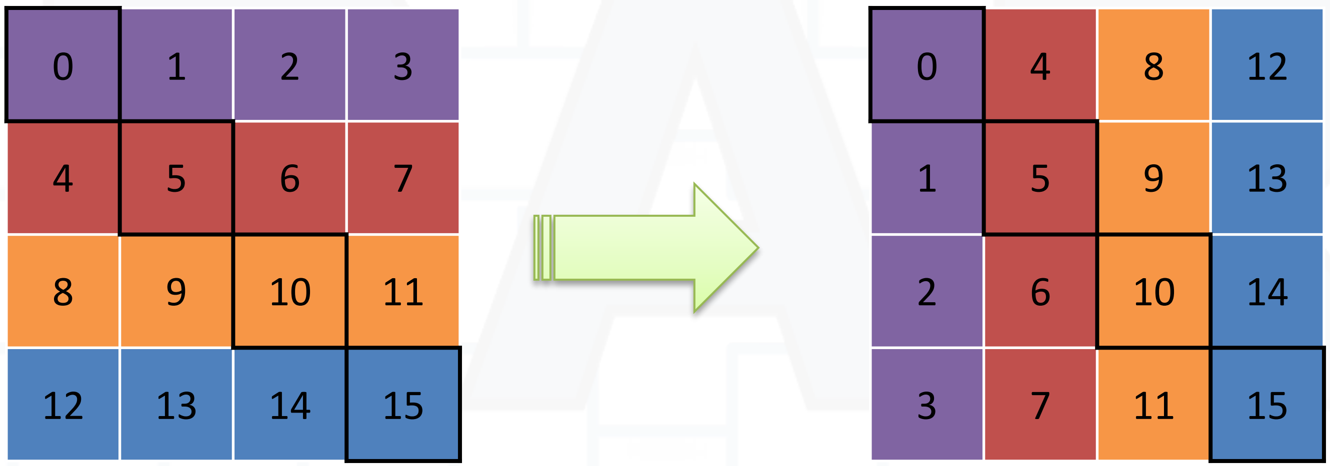
- Set up 2D Grid and 2D Blocks
- Each thread takes one row and one column
- Block size divide N, for each dimensions
- Requires N x N threads. e.g. N = 100 -> 10'000 Threads

What is the speedup against the sequential version for large values?



Exercises

Matrix Transpose



Simplest Version:

- Each thread takes one cell
- The kernel produce on output the transpose matrix
- Requires N x N threads. e.g.   N = 100 -> 10'000 Threads

References

- CUDA C Programming Guide

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

- CUDA Runtime API

  http://docs.nvidia.com/cuda/cuda-runtime-api/

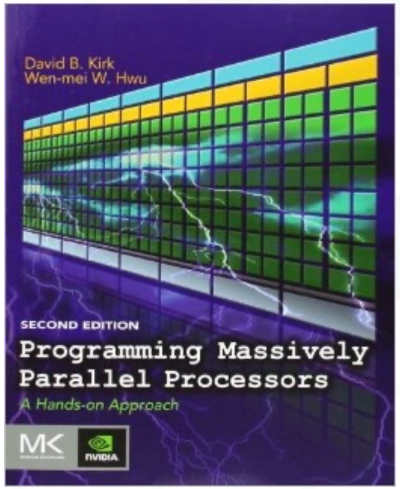
  http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/modules.html

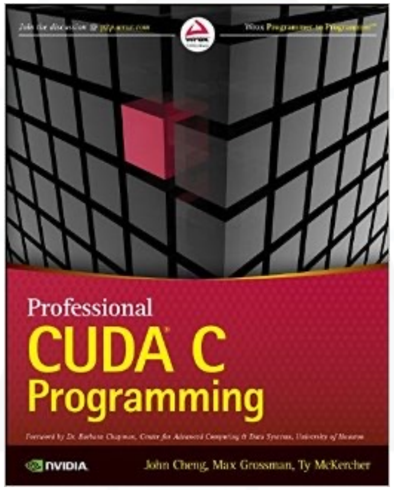
  (Memory Management)

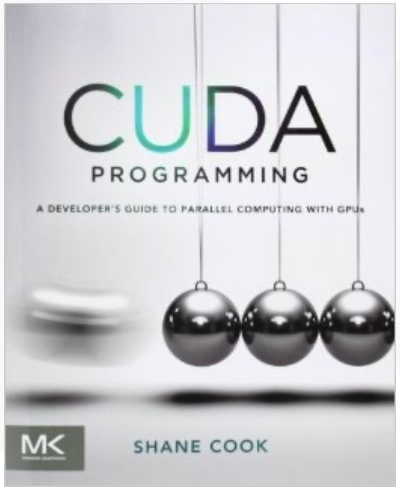
- CUDA Math API

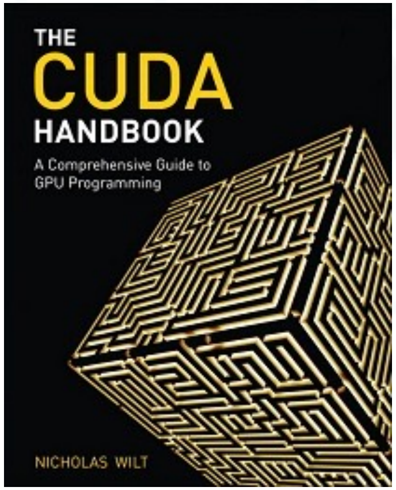
  http://docs.nvidia.com/cuda/cuda-math-api/

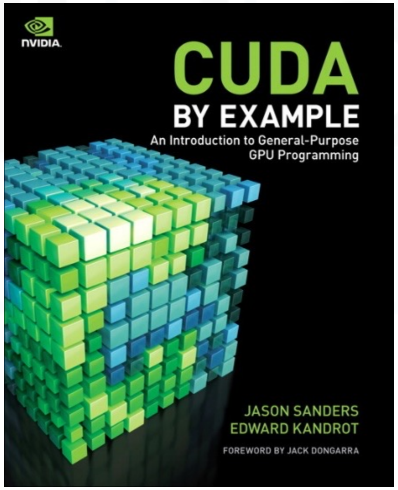
CUDA Programming Books

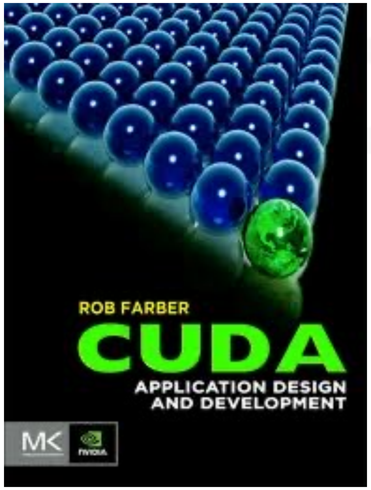












CUDA Advanced Books