GPU Programming

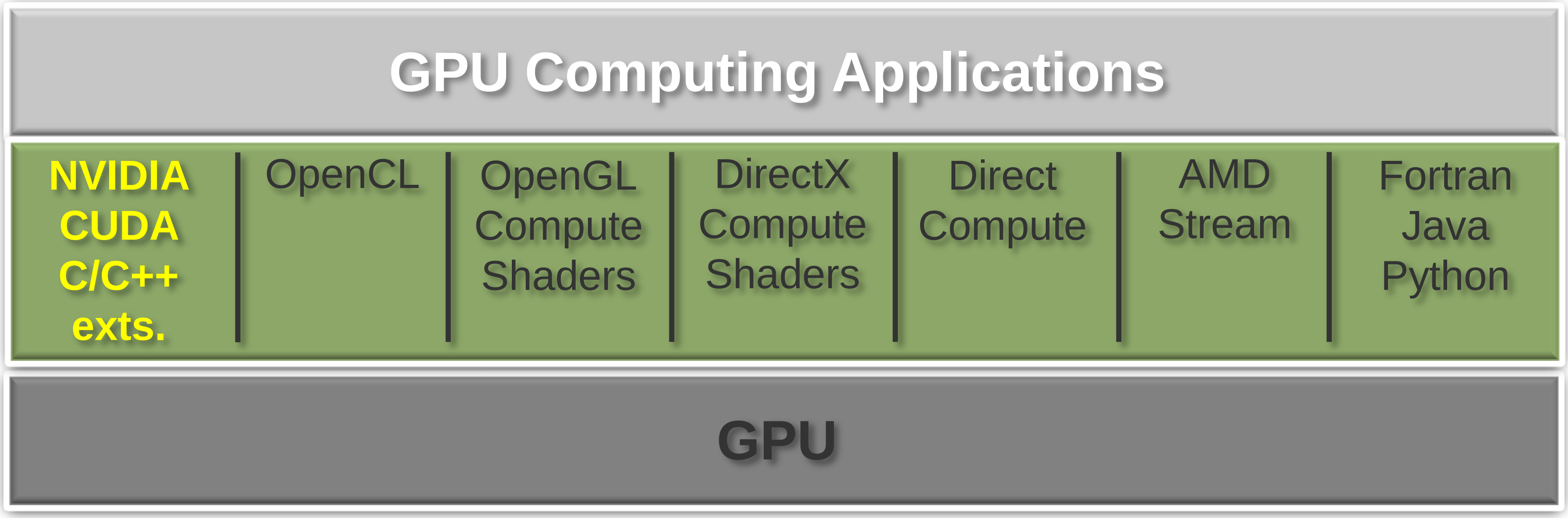
Rüdiger Westermann
Chair for Computer Graphics & Visualization

Overview

- Programming interfaces and support libraries
- The CUDA programming abstraction
- An in-depth CUDA example
- CUDA-OpenGL binding

GPU programming

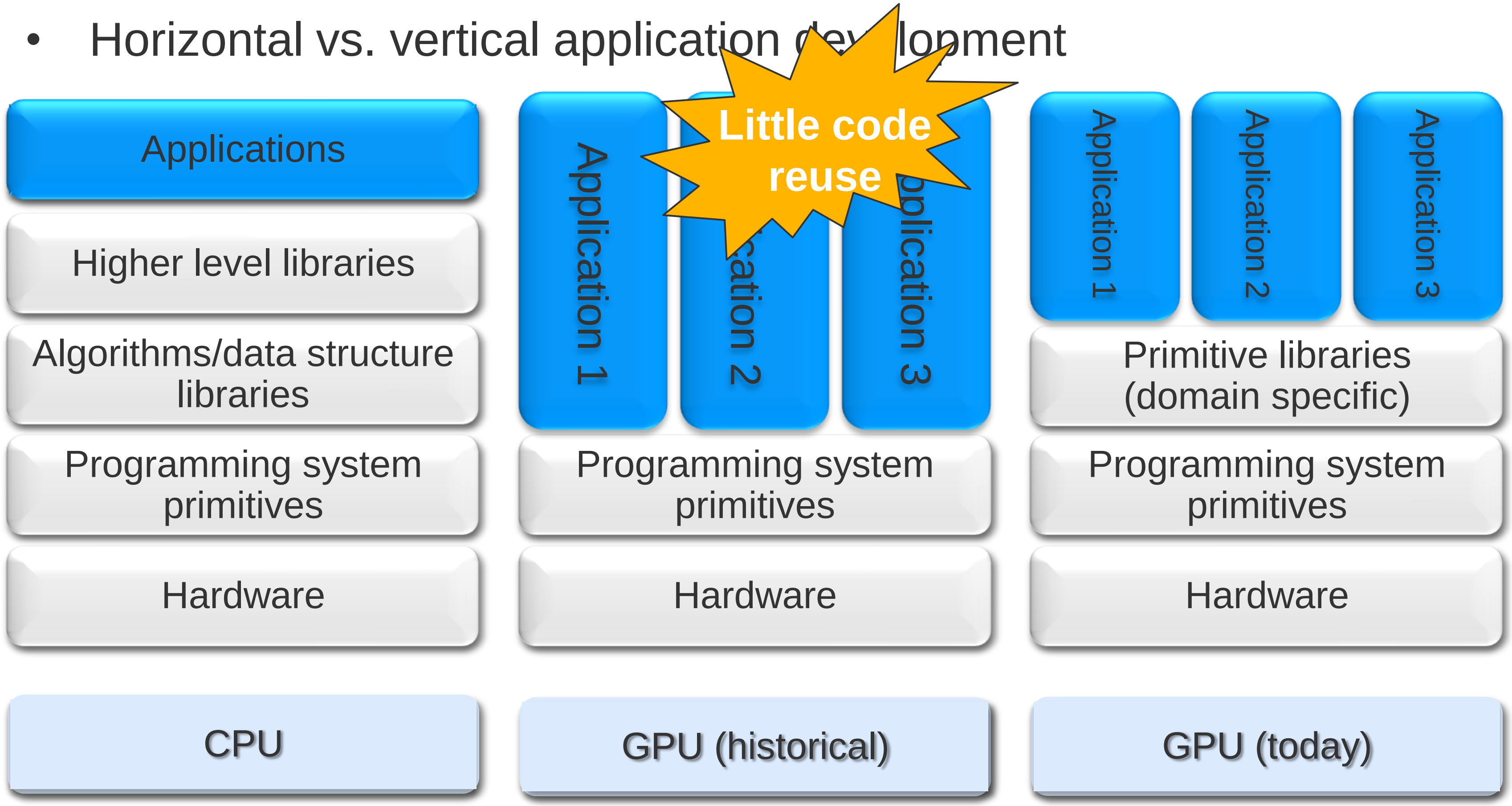
- How can we program the GPU?



- We will focus on C/CUDA and NVIDIA GPUs in this tutorial

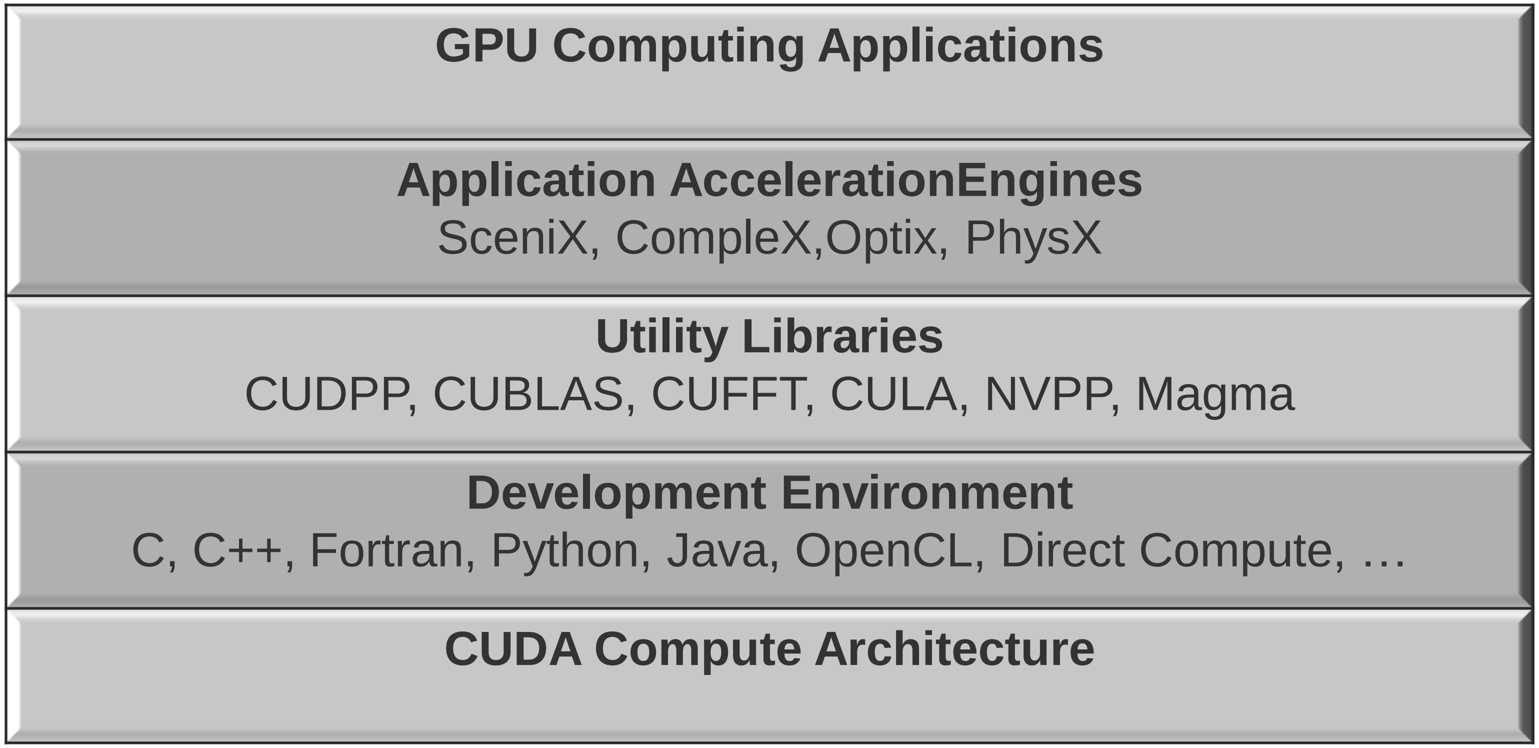
GPU programming

- Horizontal vs. vertical application development



GPU programming

- Software libraries to support GPU/CUDA programming


GPU Computing Applications
Application AccelerationEngines
SceniX, CompleX,Optix, PhysX
Utility Libraries
CUDPP, CUBLAS, CUFFT, CULA, NVPP, Magma
Development Environment
C, C++, Fortran, Python, Java, OpenCL, Direct Compute, …
CUDA Compute Architecture

GPU programming

- A few software libraries to support GPU/CUDA programming
  - CUDPP – a library of high-performance parallel primitives for GPUs
    - Provides scan-primitives, stream compaction, radix sort, sparse matrix-vector multiply, random number generation
  - CUBLAS
    - **B**asic **L**inear **A**lgebra **S**ubprograms like vector, vector/matrix, and matrix/matrix operations (subset of BLAS 1/2/3)
  - CUFFT (http://developer.download.nvidia.com)
    - 1D, 2D, and 3D transforms of complex and real-valued data
    - Transform sizes (2D/3D) up to 16384 in any dimension
  - MAGMA - http://icl.cs.utk.edu/magma/
    - Dense linear algebra on heterogeneous CPU+GPU systems
  - CULA by EM Photonics (http://www.culatools.com/)
    - Emulates LAPACK on GPUs
    - LU, QR and singular value decomposition, least squares

CUDA – Compute Unified Device Architecture

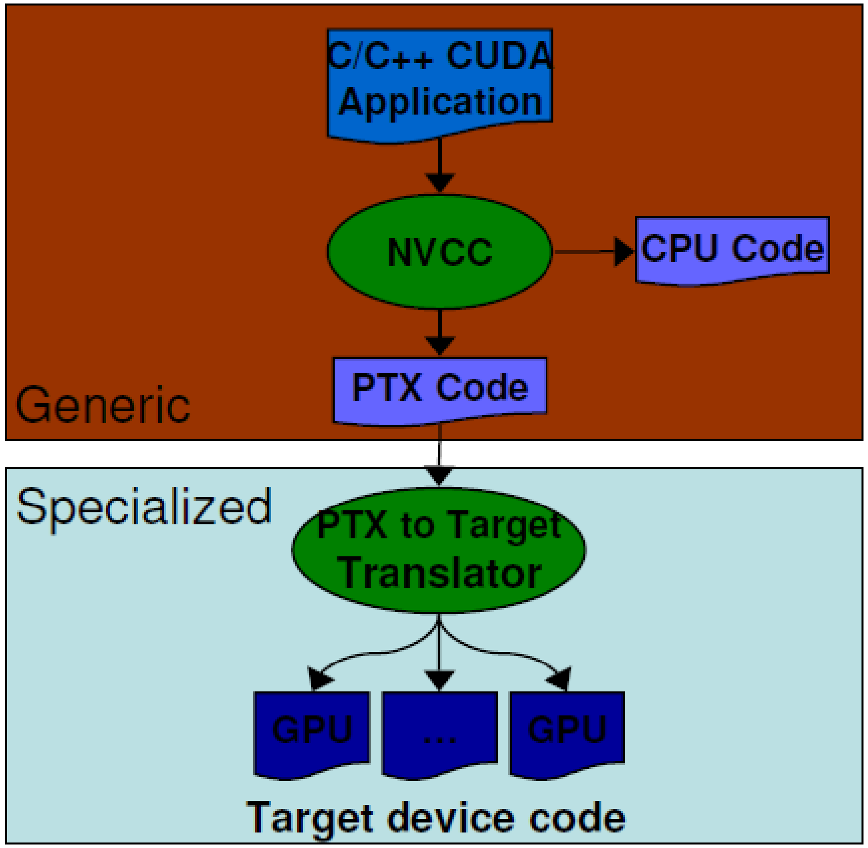
- CUDA parallel programming abstraction
  - Parallel computing architecture and programming model
  - Unified hardware and software specification for parallel computing
- Hardware multithreading
- General purpose programming model
  - User launches batches of threads on the GPU (application controlled SIMD program structure)
  - Fully general load/store memory model
  - Simple extension to standard C
- Not a graphics API
  - But graphics API interoperability is possible
  - “Buffer exchange” between CUDA and graphics API

CUDA – Compute Unified Device Architecture

- CUDA is designed to fully exploit the SIMD execution paradigm underlying the GPU design
  - Parallel kernels composed of many threads
  - All threads execute the same sequential program
  - Threads are grouped into thread blocks
  - Cooperation within a block via fast, shared memory and hardware synchronization barriers
  - Blocks are virtualized multiprocessors
  - All blocks must be independent, implicit barrier between kernel launches
  - Support by runtime API
    - cudaMalloc(), cudaMemcpy(), cudaFree()
    - cudaGetLastError()
    - …

CUDA – compiling CUDA for GPUs

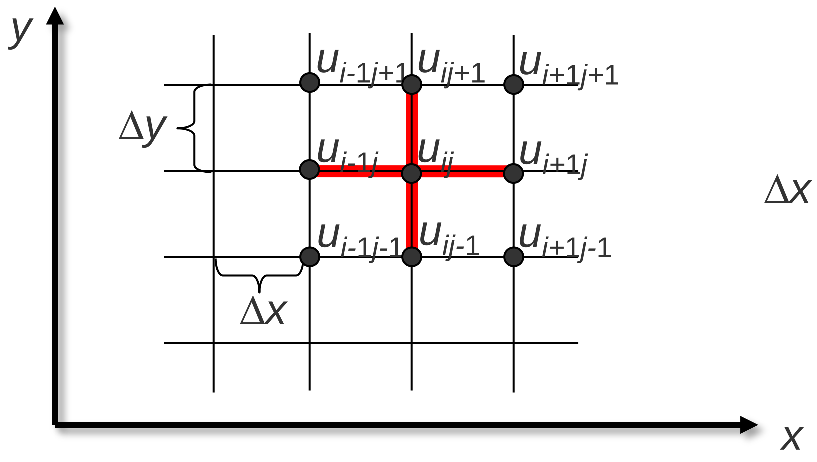
- NVCC compiles into CPU and **P**arallel **T**hread e**X**ecution code



CUDA programming example

- Evaluation of a finite-difference stencil on a 2D uniform grid

$$
\begin{aligned}
u_{ij} &= u_{i+1\,j} - 2u_{ij} + u_{i-1\,j} + u_{ij+1} - 2u_{ij} + u_{ij-1} \\
&= u_{i+1\,j} + u_{i-1\,j} + u_{ij+1} + u_{ij-1} - 4u_{ij}
\end{aligned}
$$



$\Delta x = \Delta y = 1$

CUDA programming example

- The sequential CPU code for computing the finite-difference stencil

```
int main(float *u, int Nx, Ny) {

   // pointer to CPU (host) memory
   float *out;

   // allocate array on host
   out = (float *)malloc(sizeof(float)*Nx*Ny);

   // compute stencil
   for (int j=1; j<Ny-1; j++) {
      for (int i=1; i<Nx-1; i++) {
         out[i + j * Nx] =
                  u[i+1 + j * Nx] +
                  u[i-1 + j * Nx] +
                  u[i + (j+1) * Nx] +
                  u[i + (j-1) * Nx] –
                  4 * u[i + j * Nx];

   // copy result to input array and free memory
   memcopy(out, u, sizeof(float)*Nx*Ny);
   free(out);
}
```

CUDA programming example

- The parallel GPU code for computing the finite-difference stencil
  Requirements:
  - Allocation of memory on the GPU device and moving data to/from that memory
  - A kernel – a function callable from the host (CPU) and executed on the device by many threads in parallel
  - An execution configuration to specify the number and grouping of parallel threads used to execute the kernel
  - Performing parallel computations based on a subdivision of the problem domain, i.e., a means to specify which part of the domain a thread has to work on

CUDA programming example

- CUDA setup code for the stencil computation

Returns a pointer to a linear memory segment in global device memory

```
#include <stdio.h>
#include <cuda.h>

int main(float *u, int Nx, Ny) {

   // pointers to GPU (device) memory
   float *u_d, *out;

   // allocate arrays on the GPUdevice
   cudaMalloc((void **) &u_d, sizeof(float)*Nx*Ny);
   cudaMalloc((void **) &out, sizeof(float)*Nx*Ny);

   •
   •
   •
```

CUDA programming example

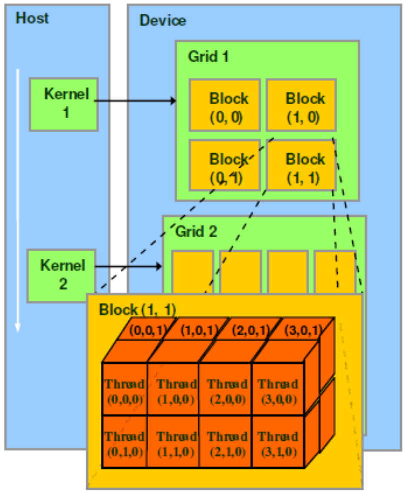
- Moving data between host and device memory.

```
    . . .
    // send data from host to device: u to u_d
    cudaMemcpy(u_d, u, sizeof(float)*Nx*Ny, cudaMemcpyHostToDevice);


    // here:
    // CUDA configuration and execution of the parallel thread program


    // retrieve data from device: out to u
    cudaMemcpy(u, out, sizeof(float)*Nx*Ny, cudaMemcpyDeviceToHost);

    // cleanup
    cudaFree(u_d); cudaFree(out);

} // end of main
```

Copies content of CPU array to device array, and vice versa

CUDA programming example

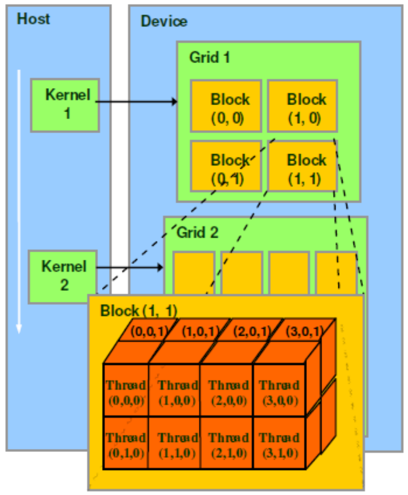
- CUDA configuration and execution of the parallel thread program
  - Remember: threads are grouped into blocks and blocks are structured into grids
    - Blocks and grids can have multiple dimensions
    - Threads in one block are executed on the sam SM, whereas blocks can run on different SMs



  - The block/grid abstraction allows balancing the trade-offs between running many **independent** threads in parallel, and running blocks of threads that are **synchronized** and can cooperate with each other
  - The execution configuration can be requested by a thread via variables:
    - **blockIdx**: the thread's block index within the grid
    - **threadIdx**: the thread's index within the block
    - **blockDim**: the number of threads in the thread's block

CUDA programming example

- CUDA configuration and execution of the parallel thread program
  - Remember: threads are grouped in blocks and blocks are structured in grids
  - Blocks and grids can have multiple dimensions



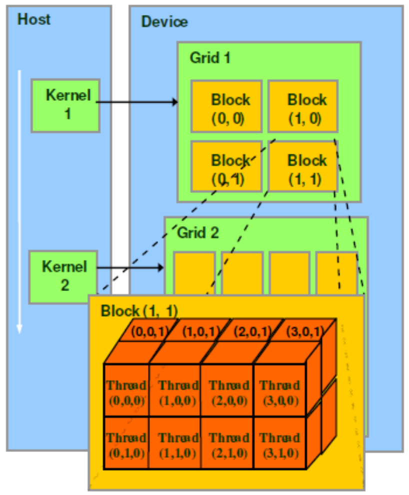
```
// compute (2D) execution configuration
const int BLOCKSIZEX = 8;   // number of threads per block along dim 1
const int BLOCKSIZEY = 8;   // number of threads per block along dim 2


int nBlocksX = Nx / BLOCKSIZEX -2; // how many blocks along dim 1
int nBlocksY = Ny / BLOCKSIZEY -2; // how many blocks along dim 2

dim3 dimBlock(BLOCKSIZEX, BLOCKSIZEY);  // set values
dim3 dimGrid(nBlocksX, nBlocksY);

// call computeStencilOnDevice kernel
computeStencilOnDevice <<< dimGrid, dimBlock >>> (u_d, out, Nx, Ny);
```

Calls kernel with input arguments using the specific thread configuration

CUDA programming example

- The kernel – the program that is executed by the threads on the SPs

```
__global__ void computeStencilOnDevice(float *u, float *out, int Nx, Ny) {

   // the thread in the current block
   int tX = threadIdx.x;
   int tY = threadIdx.y;

   // the index of the first thread in the current block
   // i.e., the base of the subgrid in the global grid
   int baseX = blockIdx.x * (blockSizeX - 2) + 1;
   int baseY = blockIdx.y * (blockSizeY - 2) + 1;

   // the grid index at which the numerical stencil
   // is to be evaluated by the thread
   int i = baseX + tX;
   int j = baseY + tY;

   . . .
```



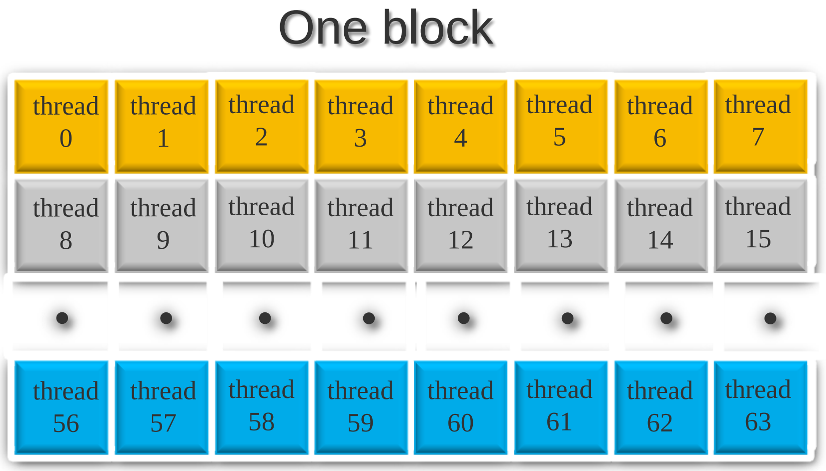
CUDA programming example

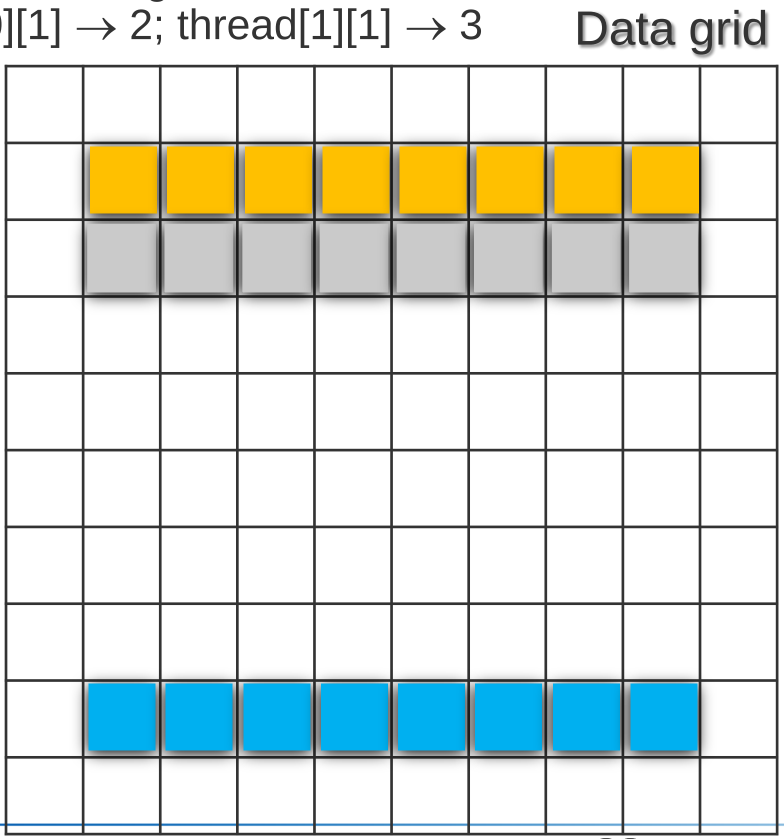
- The kernel – the program that is executed by the threads on the SPs

```
    ...
    // allocate fast shared memory (lifetime of a block) to cache the working set
    // reduces the number of device memory reads by a factor of 4
    __shared__ float u_sh[BLOCKSIZEX][BLOCKSIZEY];

    // fill the shared memory with the working set from the global device array
    u_sh[tX][tY] = u[i + j * Nx];

    __syncthreads(); // all threads wait at this barrier for all others

    if(tX > 0 && tX < BLOCKSIZEX-1 && tY > 0 && tY < BLOCKSIZEY-1) {

          // compute the stencil on the data in shared memory and write to out array
          out[i + j * Nx] = u_sh[tX+1][tY] + u_sh[tX-1][tY] +
                            u_sh[tX][tY+1] + u_sh[tX][tY-1] - 4.0 * u_sh[tX][tY];
    }
} // end of computeStencilOnDevice
```

CUDA programming example

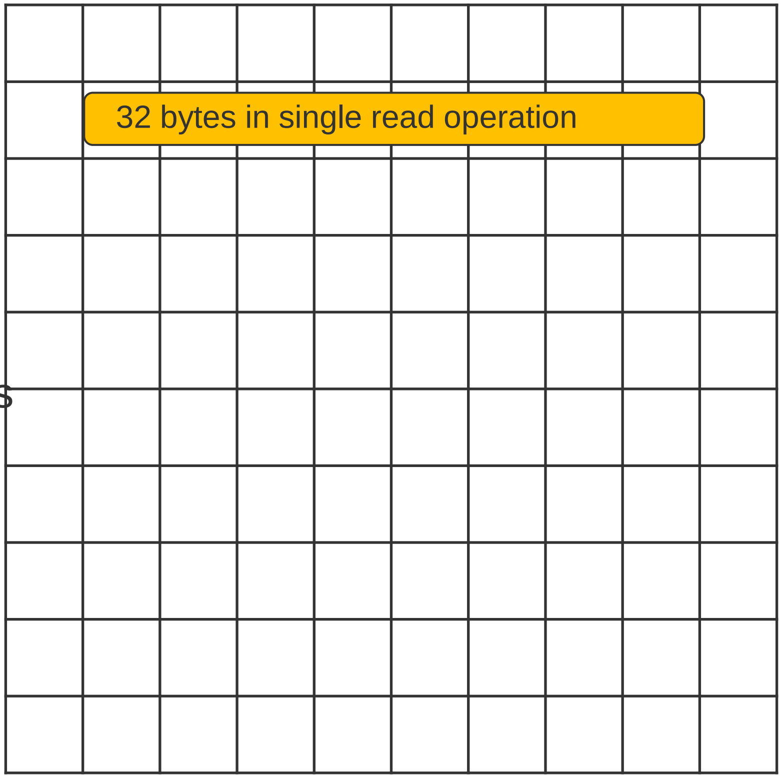
- Parallel memory transactions
  - Assume threads are ordered in row-major order, e.g., in a 2x2 block: thread[0][0] → 0; thread[1][0] → 1; thread[0][1] → 2; thread[1][1] → 3
  - The device data is laid out in this way, too
  - Per-thread memory reads due to `u_sh[tX][tY] = u[i + j * Nx];` should look like this:





CUDA programming example

- Non-coalesce parallel memory transactions
  - Every single read from the device memory reads buckets of at least 32 bytes
  - One single thread – even though only requesting 4 bytes – triggers the movement of 32 bytes when reading one float
  - This results in 8 x 32 bytes to be read overall
  - Moreover, since all threads read from the same memory bank, the read operations are serialized
  - In our case, since all requested data lies in succession, all 8 single reads are coalesce into one memory transaction of 8 x 4 = 32 bytes


32 bytes in single read operation