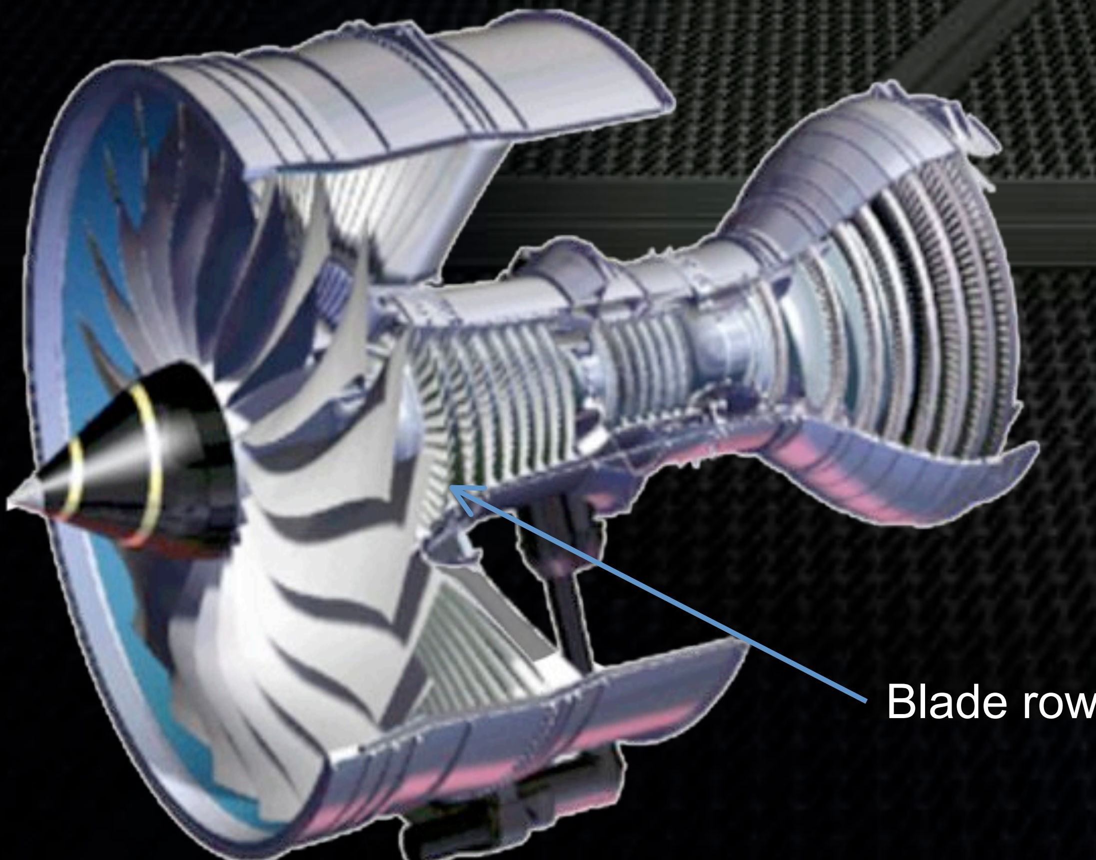

NVIDIA® TESLA™
Case study: CFD
Dr. Graham Pullan
University of Cambridge
NVIDIA
TESLA
PERSONAL
SUPERCOMPUTER

Outline

- CFD (for turbomachinery)
- A good fit for GPUs?
- Implementation
- Results

Turbomachinery

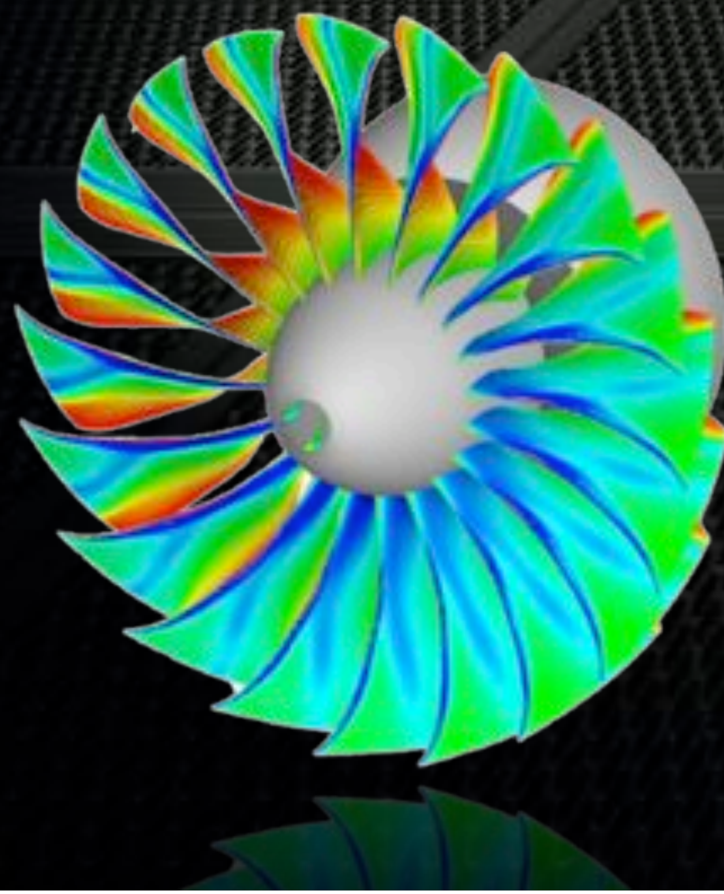


Thousands of blades

Arranged in rows

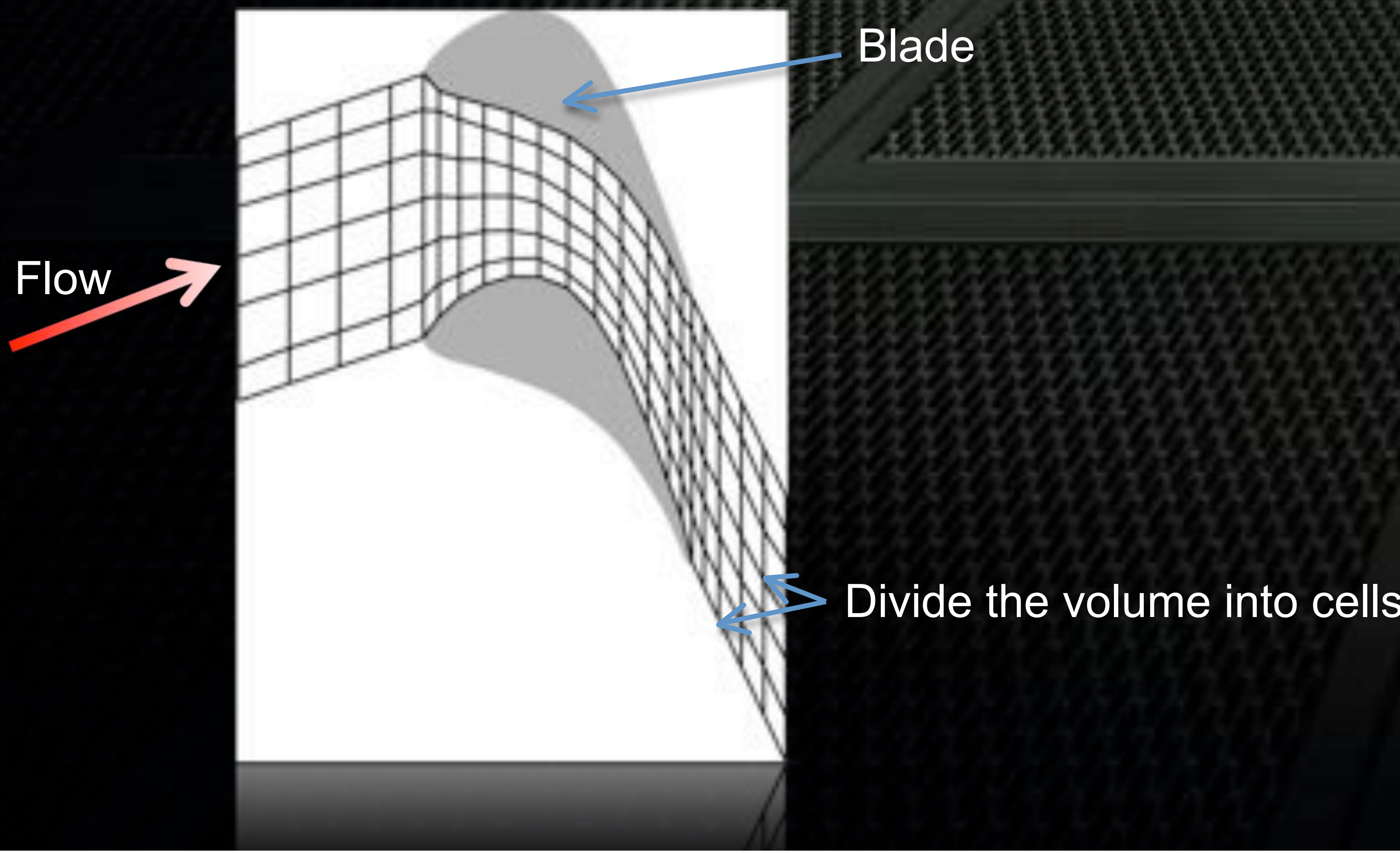
Each blade row has a bespoke blade profile designed with CFD

CFD of a jet engine fan

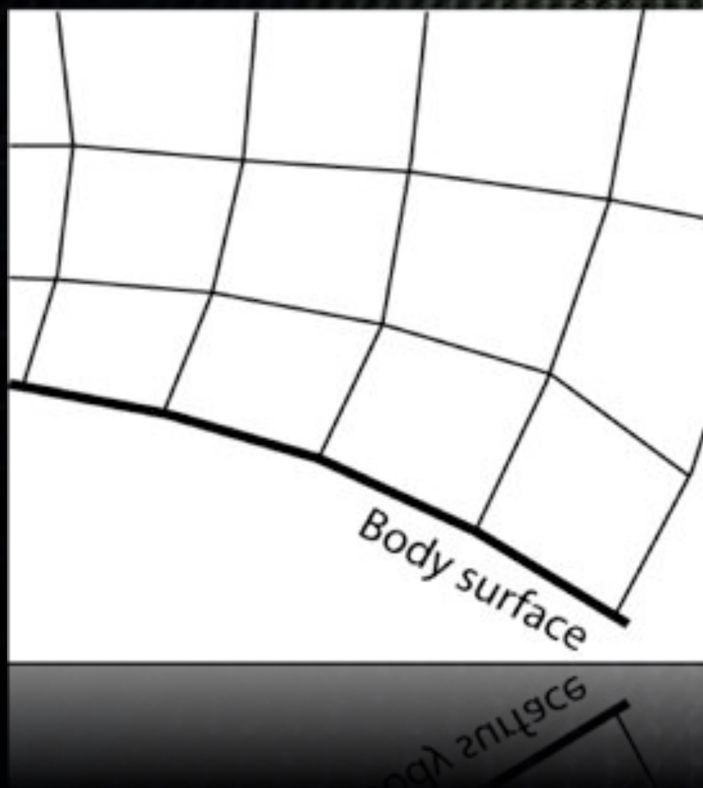


Blades coloured by pressure

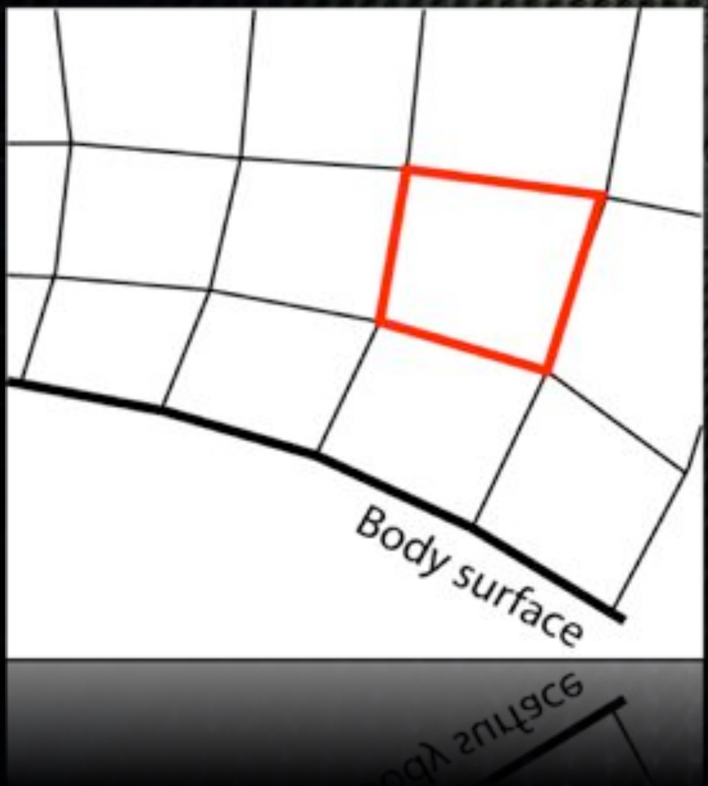
Introduction to CFD



Governing equations for each cell



Governing equations for each cell

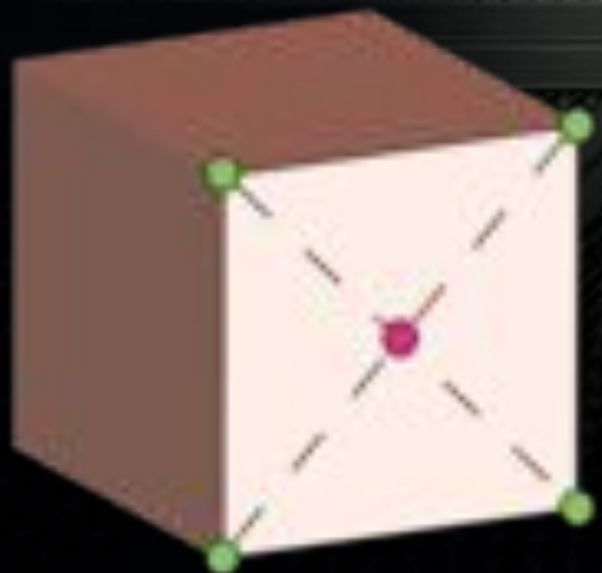


Conserve:

- Mass
- Momentum
- Energy

Example: mass conservation

- Evaluate mass fluxes on each face

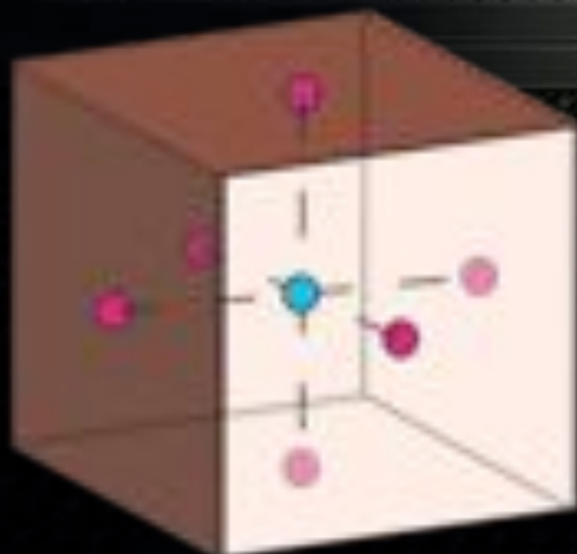


$$F_{mass} = \frac{A}{4}\sum \rho V_n$$

Example: mass conservation

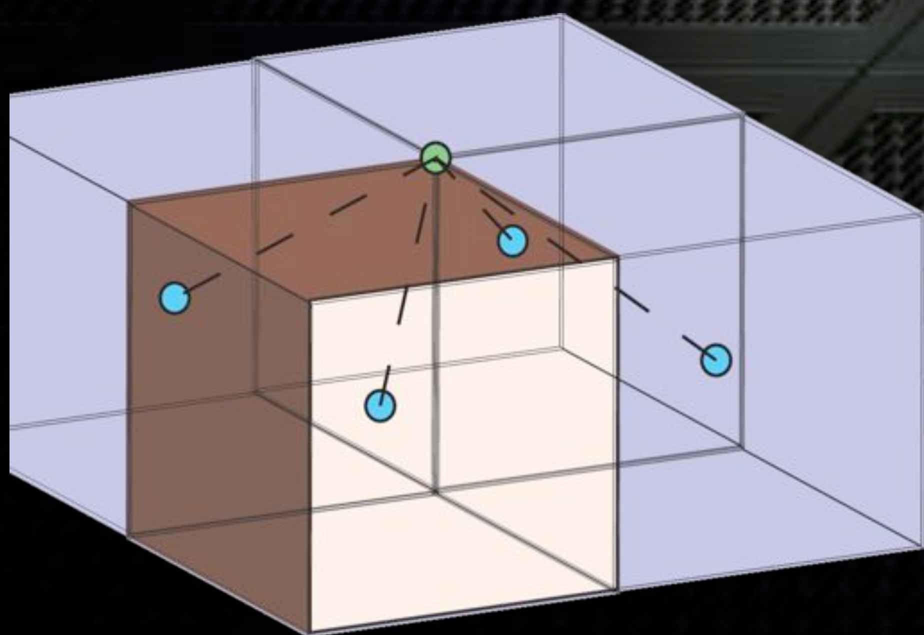
- Sum fluxes on faces to find density change in cell

$$\Delta\rho_{cell} = \frac{\Delta t}{\Delta vol}\sum F_{mass}$$



Example: mass conservation

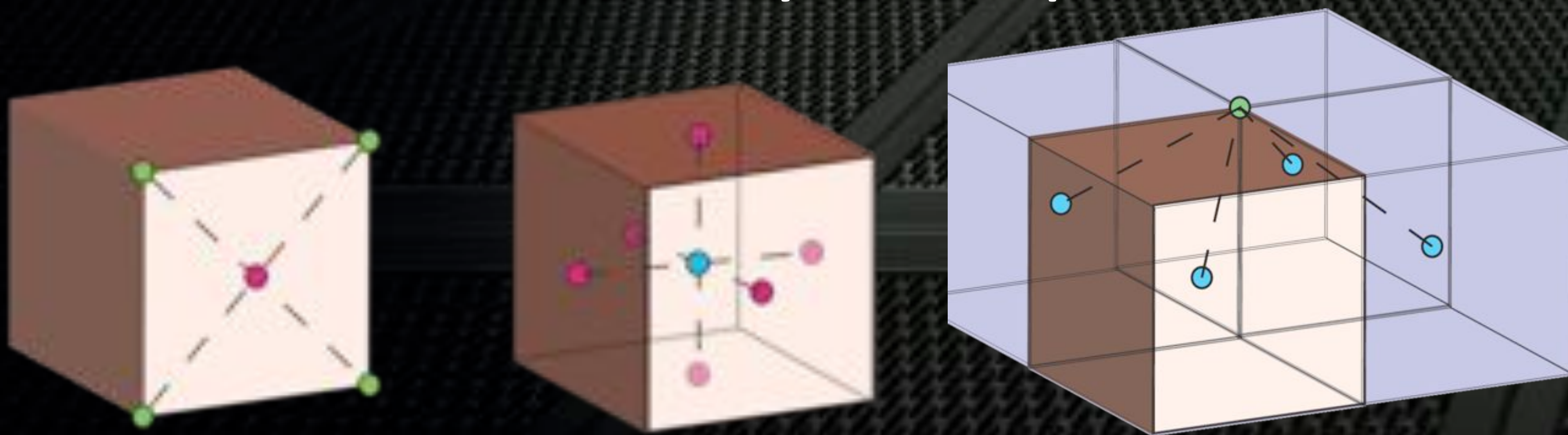
- Update density



$$\Delta\rho_{node} = \frac{1}{8}\sum\Delta\rho_{cell}$$

(only 4 of 8 surrounding cells shown)

Similarity of steps



Each step uses data from surrounding nodes – “stencil” operation

Similarity of equations

- For each equation (5 in all):
  - Set relevant flux (mass, momentum, energy)
  - Sum fluxes
  - Update nodes
  - (plus smoothing – also stencil
    boundary conditions – not stencil)

CPU run times (x86 machines)

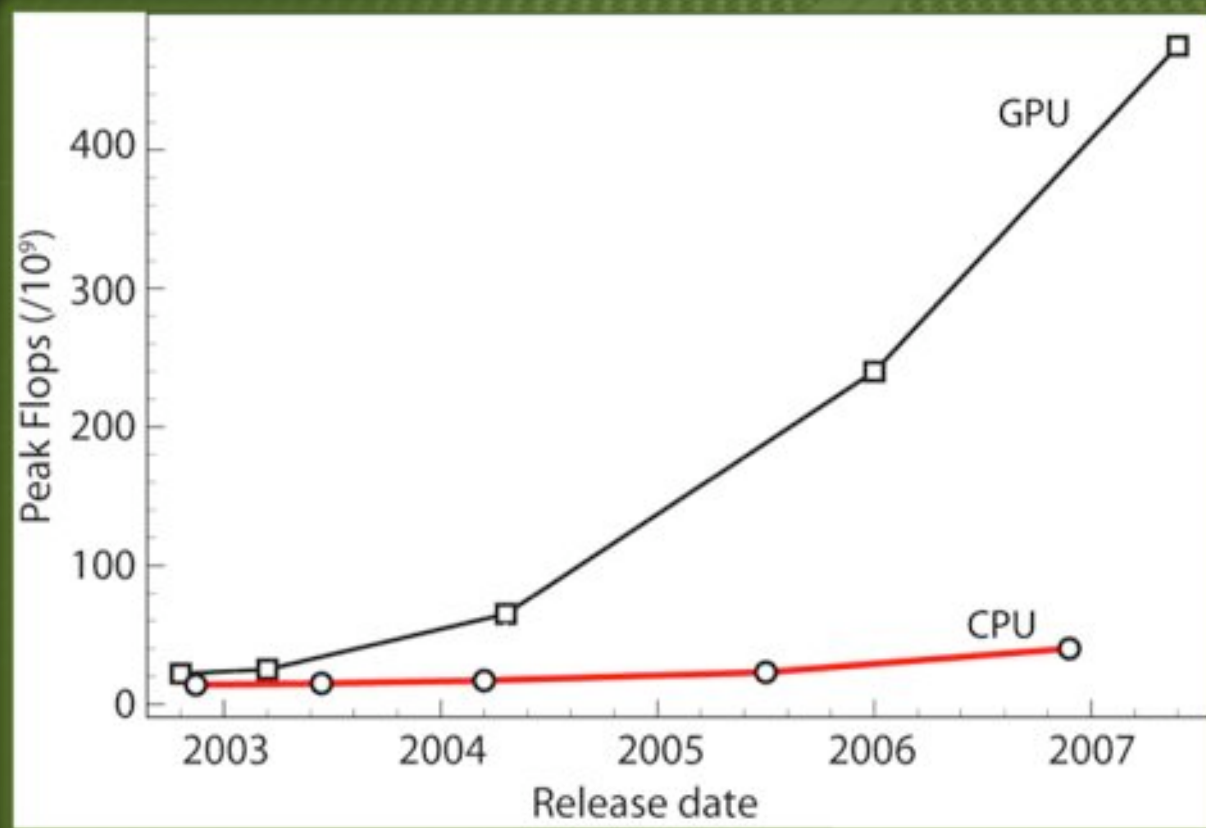
Steady approximation – one blade per row

1 blade	0.5 Mcells	1 CPU hour
1 stage (2 blades)	1.0 Mcells	3 CPU hours
1 component (5 stages)	5.0 Mcells	20 CPU hours

Unsteady approximation – all blades in row

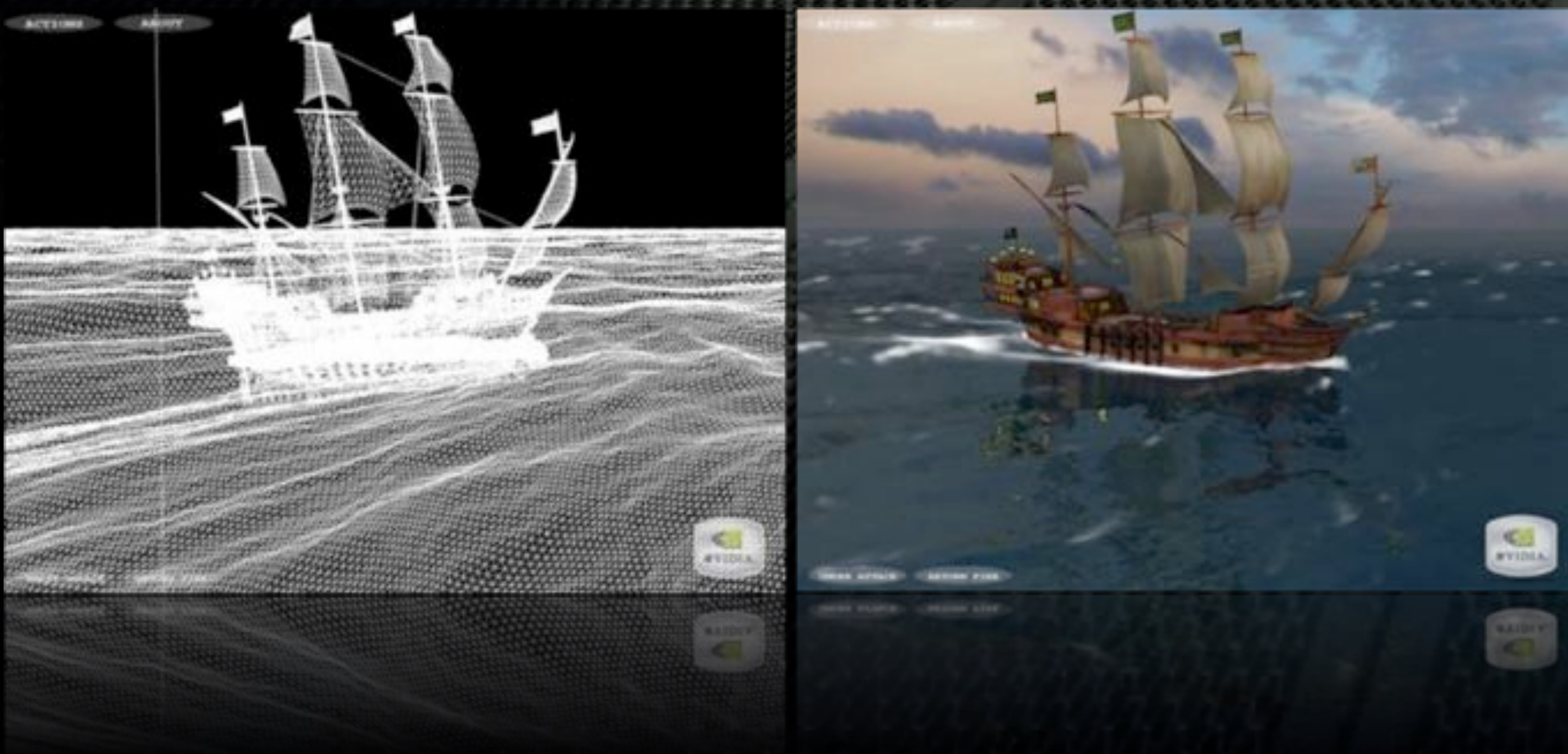
1 component (1000 blades)	500 Mcells	0.1 M CPU hours
Engine (4000 blades)	2 Gcells	1 M CPU hours

Peak FLOPs



GPU
CPU
Peak Flops (/10^9)
0
100
200
300
400
2003
2004
2005
2006
2007
Release date

The purpose of GPUs



Graphics and scientific computing

GPUs are designed to apply the
same ***shading function***
to many ***pixels*** simultaneously

Graphics and scientific computing

GPUs are designed to apply the

same *function*

to many *data* simultaneously

Are GPUs a good fit for CFD?

- Our CFD code is:
  - SIMD (same functions applied to all cells in domain)
  - Single precision
  - Large datasets (c 10M nodes) fit on one 4GB Tesla card
    - (bandwith on card is high c 102 GB/s
      much slower to/from card c 8 GB/s
      and steps in CFD are "memory bound")

CUDA

- Programming GPUs without the graphics abstraction
- Scalar variables (not graphics-type 4-vectors!)
- Extensions to C (not graphics APIs, eg OPENGL)

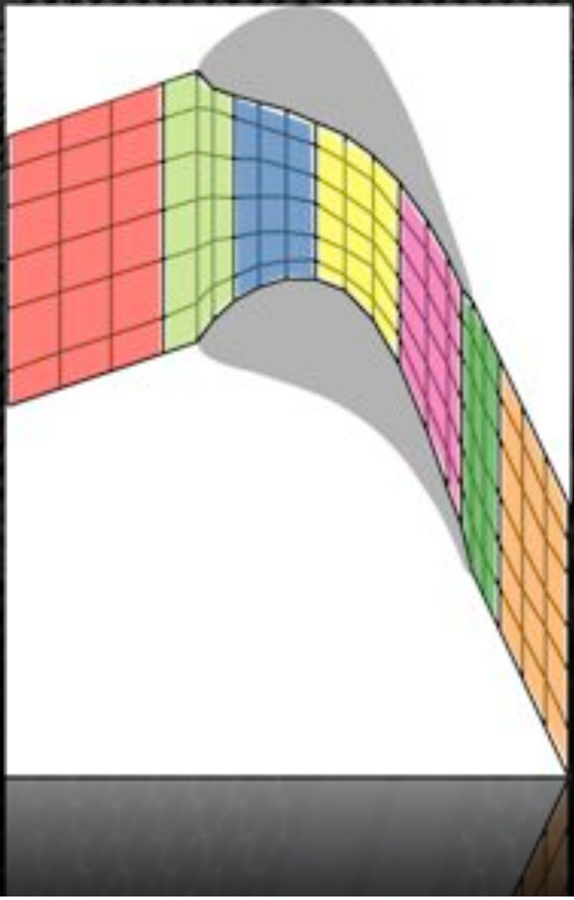
CUDA

- Programming GPUs without the graphics abstraction
- Scalar variables (not graphics-type 4-vectors!)
- Extensions to C (not graphics APIs, eg OPENGL)

- BUT – porting 15,000 lines of existing FORTRAN CFD code to CUDA still a lengthy task

Overall strategy

- Divide up domain
  - each sub-domain to a thread block
  - update nodes in sub-domain with most efficient stencil operation we can come up with!
  - update sub-domain boundaries (MPI if needed)



SBLOCK – stencil framework

- SBLOCK framework for stencil operations on structured grids:
  - Source-to-source compiler
    - Takes in high level kernel definitions
    - Produces optimised kernels in C or CUDA
- Allows new stencils to be implemented quickly
- Allows new stencil optimisation strategies to be deployed on all stencils (without typos!)

SBLOCK


Kernel definition
CUDA template
C template
GPU Kernel
CPU Kernel

Example SBLOCK definition

```
kind = "stencil"
bpin = ["a"]
bpout = ["b”]
lookup = ((1,0, 0), (0, 0, 0), (1,0, 0), (0, 1,0),
          (0, 1, 0), (0, 0, 1), (0, 0, 1))

calc = {"lvalue": "b",
        "rvalue": """sf1*a[0][0][0] +
                     sfd6*(a[1][0][0] + a[1][0][0] +
                          a[0][1][0] + a[0][1][0] +
                          a[0][0][1] + a[0][0][1])"""}
```

C implementation

```
void smooth(float sf, float *a, float *b)
{
   for (k=0; k < nk; k++) {
    for (j=0; j < nj; j++) {
      for (i=0; i < ni; i++) {
// compute indices i000, im100, etc (not shown) //
        b[i000] = sf1*a[i000] +
                    sfd6*(a[im100] + a[ip100] +
                             a[i0m10] + a[i0p10]
                        + a[i00m1] + a[i00p1]);
      }
    }
  }
}
```

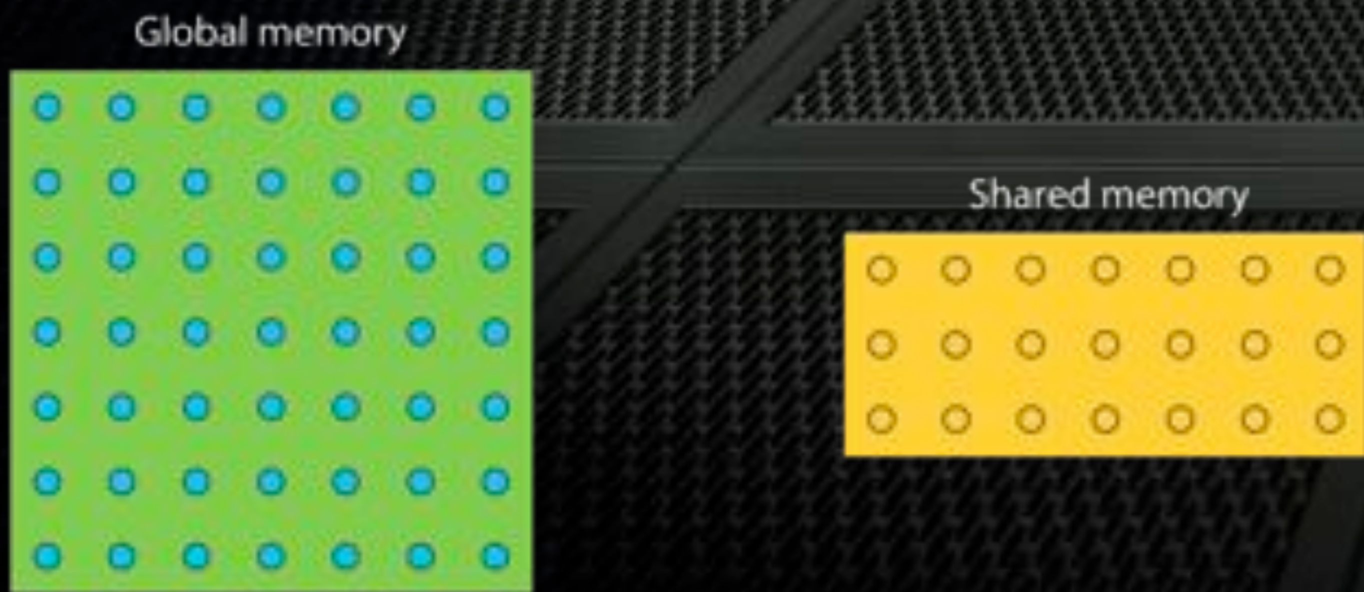
CUDA strategy (after Datta et al.)

- Each thread in a block reads sub-domain data from global device memory to SM shared memory (coalesced reads for maximum bandwidth)
- Synch threads
- Update nodes in sub-domain using shared memory and output result back to global memory

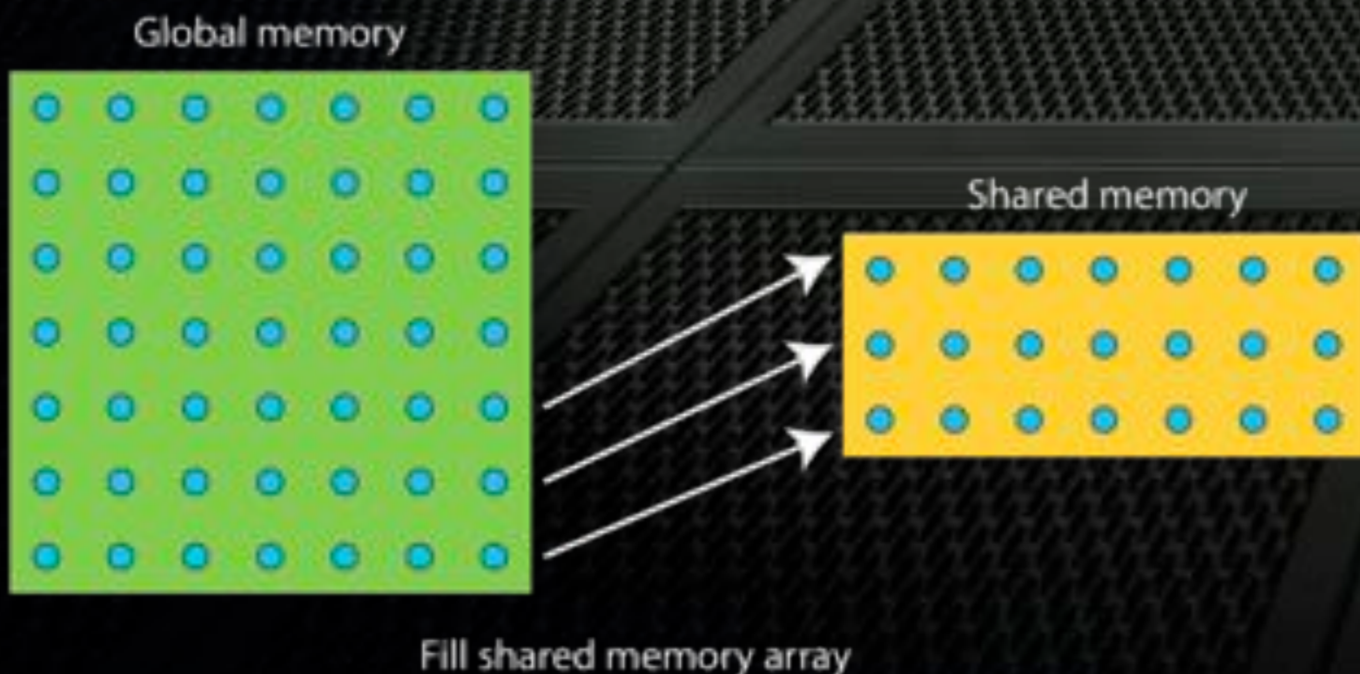
CUDA strategy (after Datta et al.)

- Each thread in a block reads sub-domain data from global device memory to SM shared memory (coalesced reads for maximum bandwidth)
- Synch threads
- Update nodes in sub-domain using shared memory and output result back to global memory
- But shared memory and max threads per block are limited, so best plan is to march through sub-domain plane-by-plane…

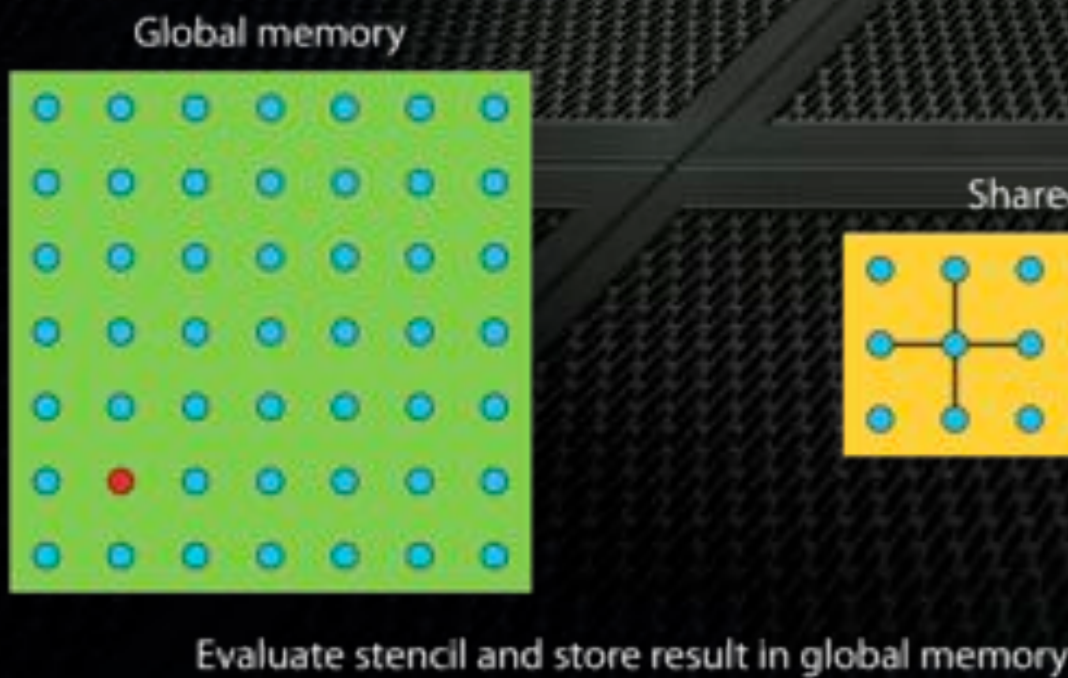
CUDA strategy

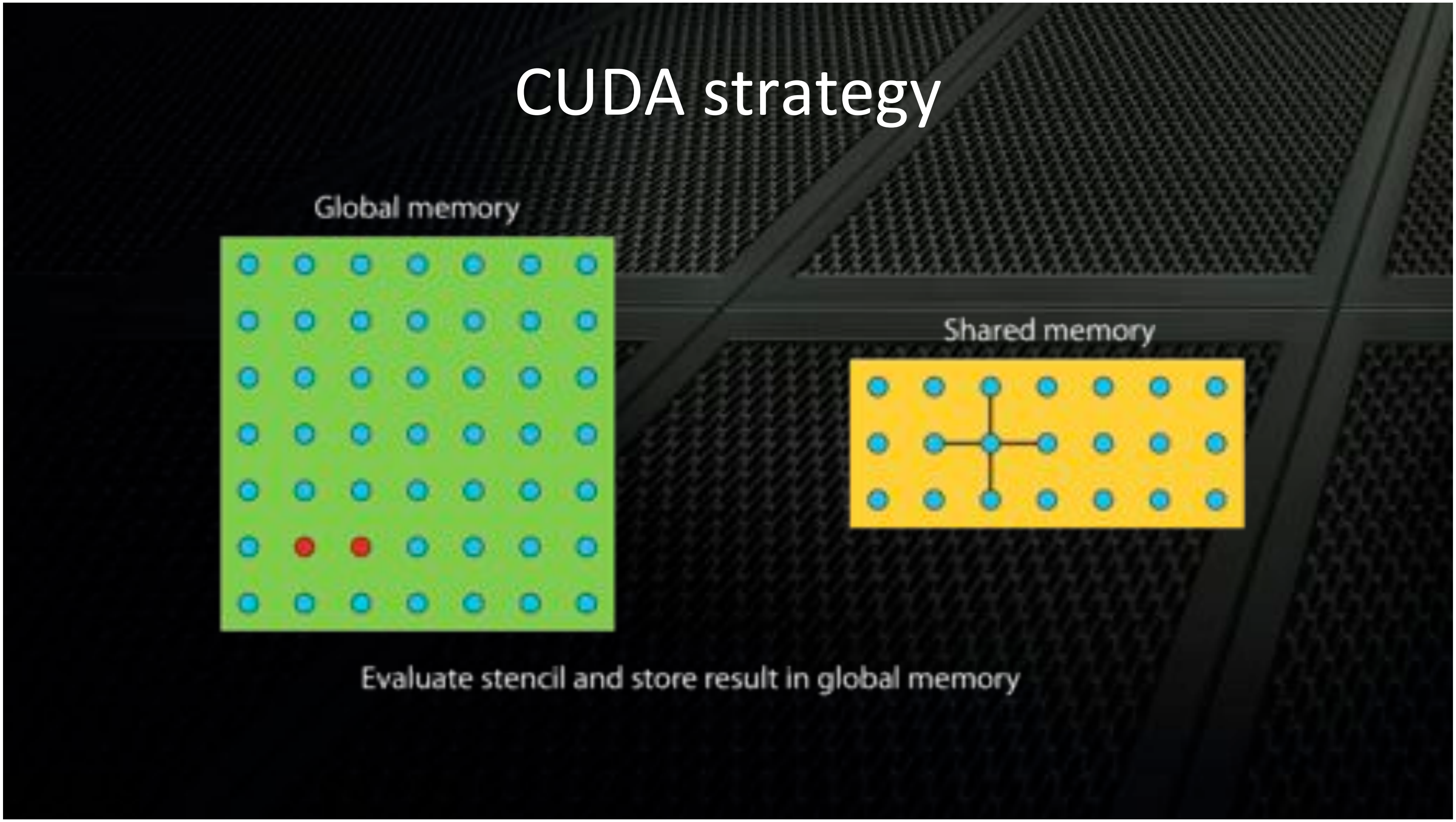


CUDA strategy

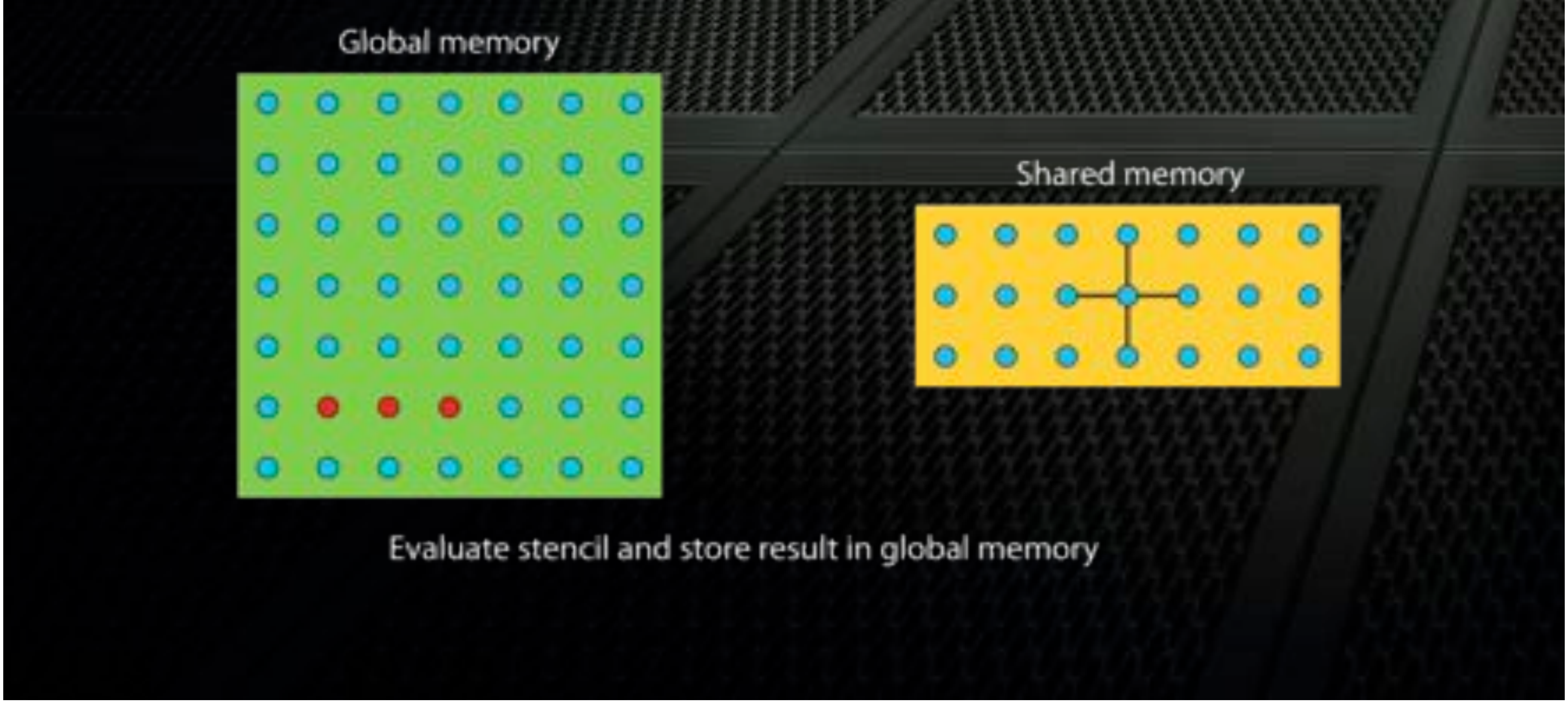


CUDA strategy

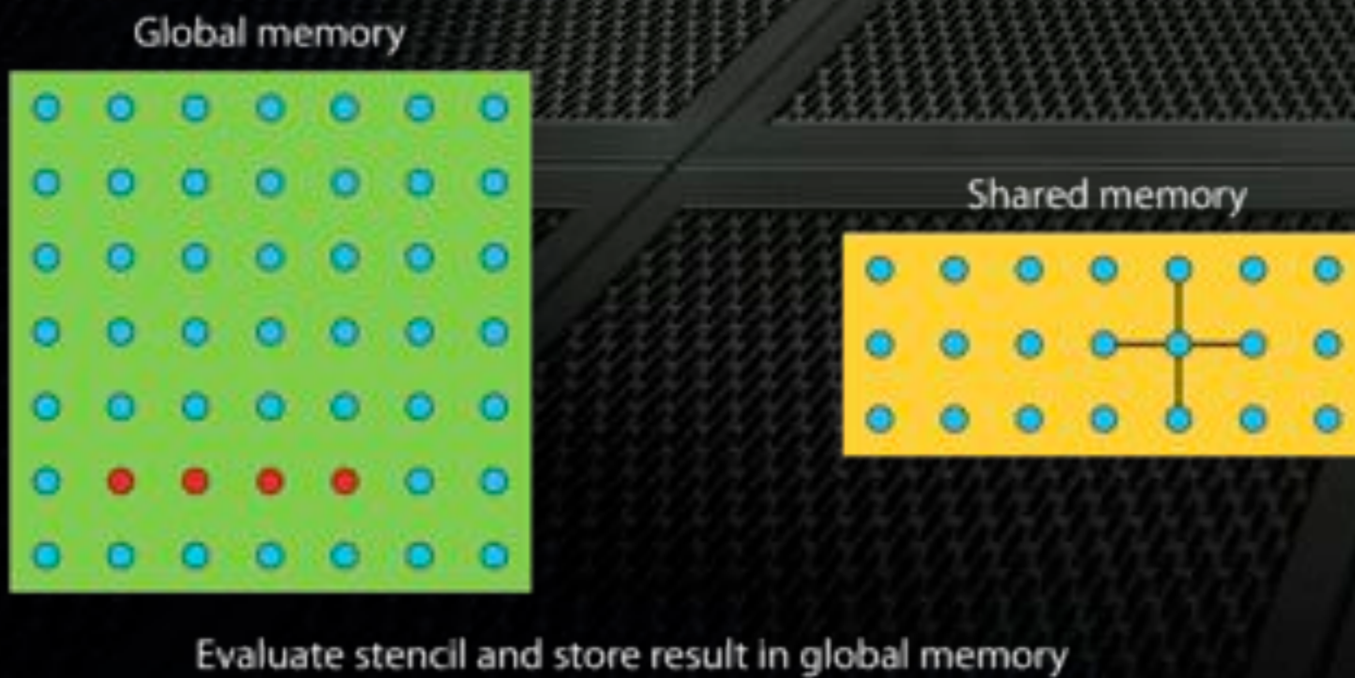



CUDA strategy
Global memory
Shared memory
Evaluate stencil and store result in global memory

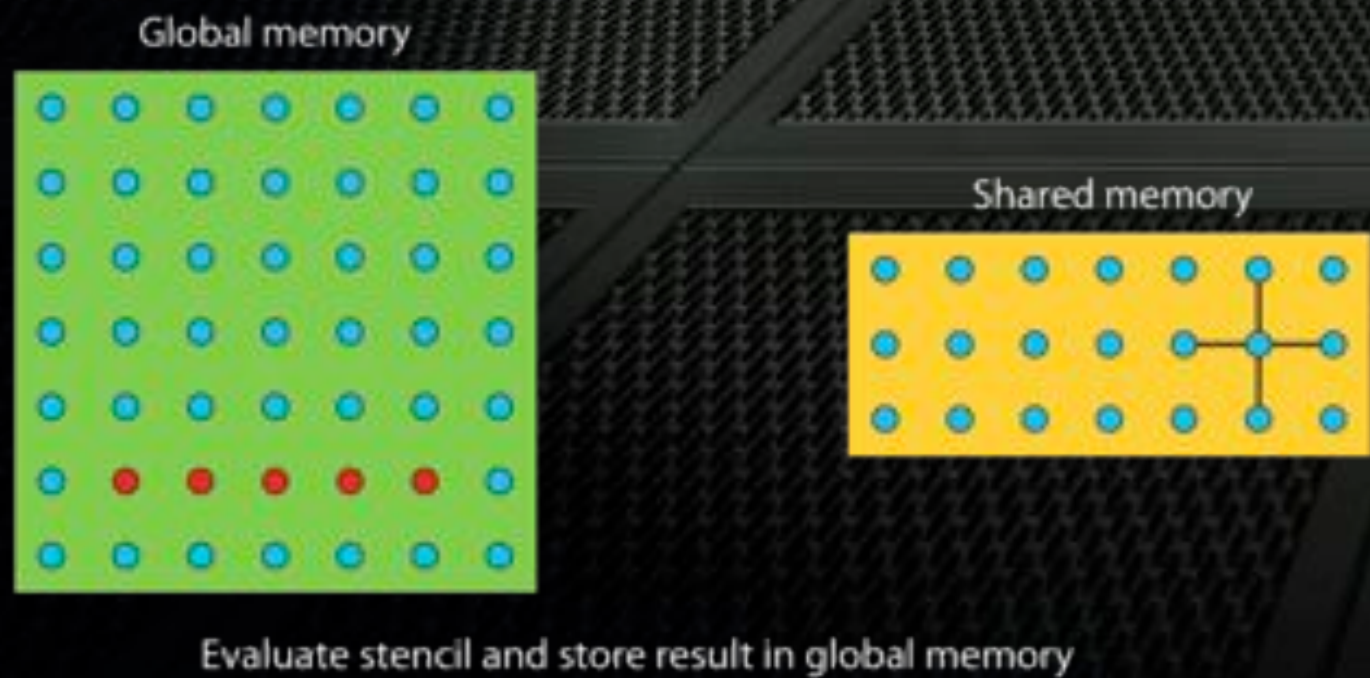
CUDA strategy



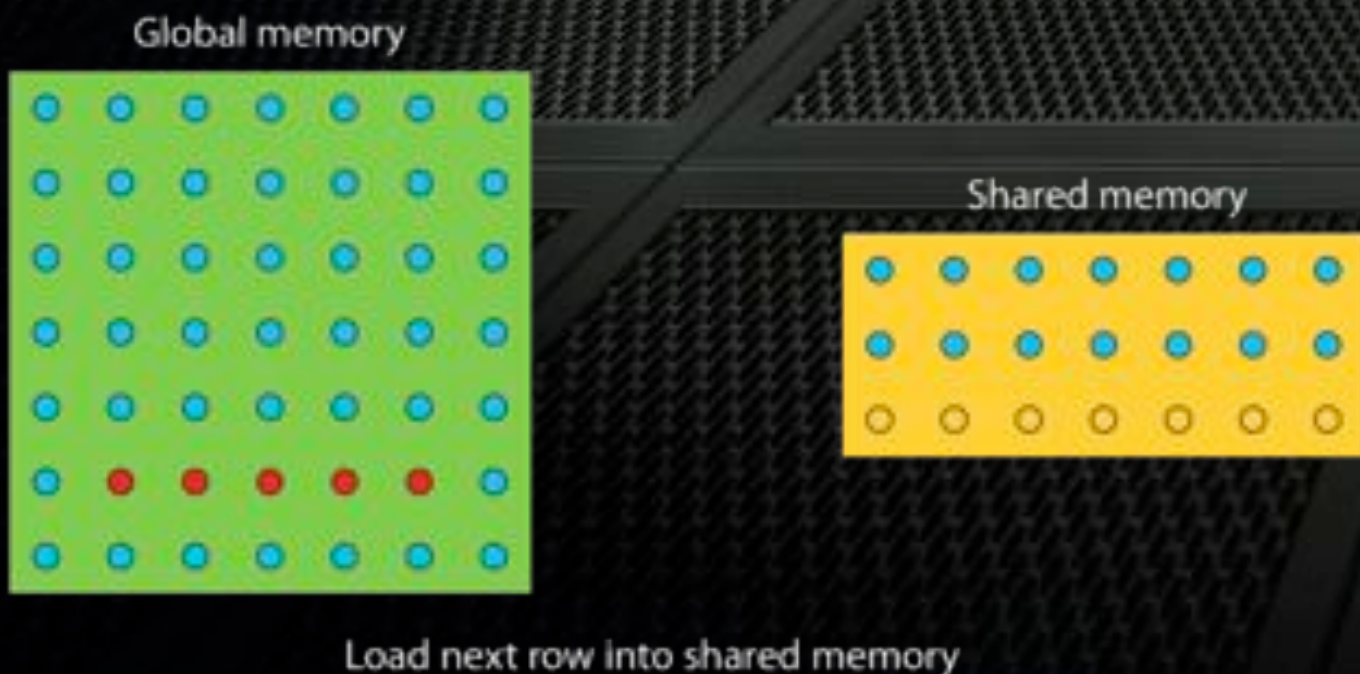
CUDA strategy



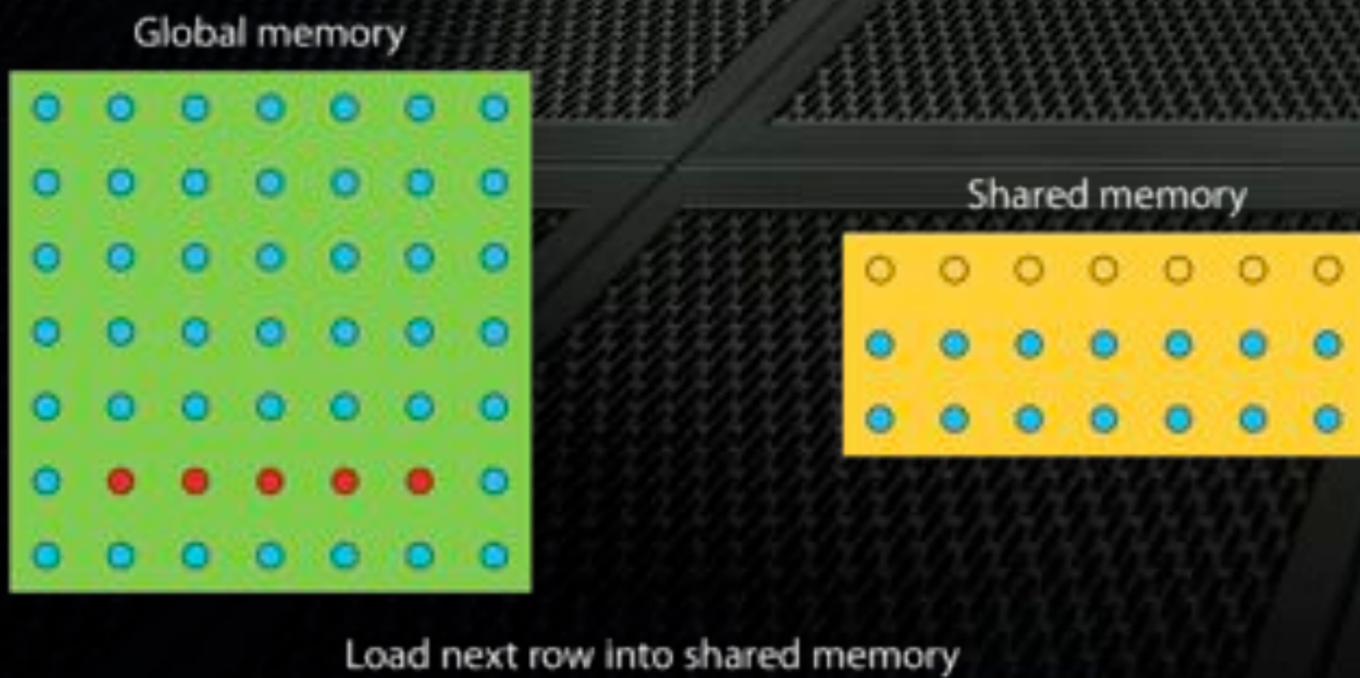
CUDA strategy



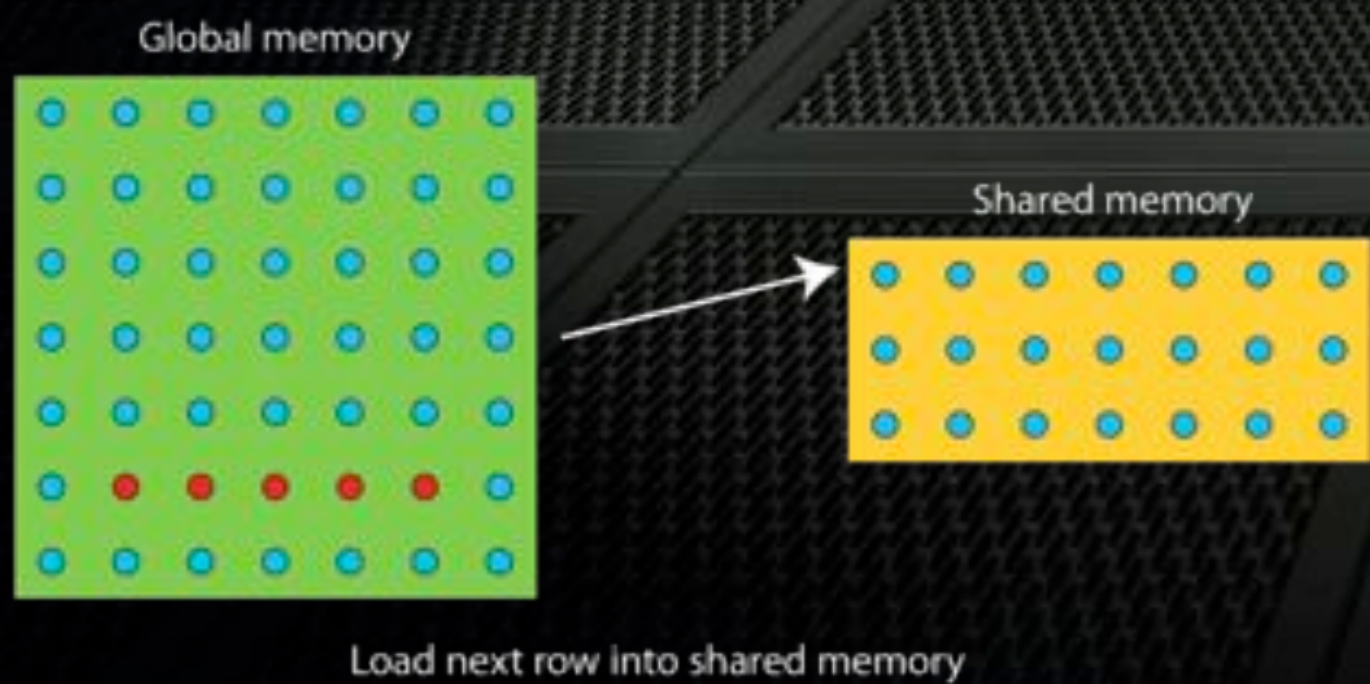
CUDA strategy



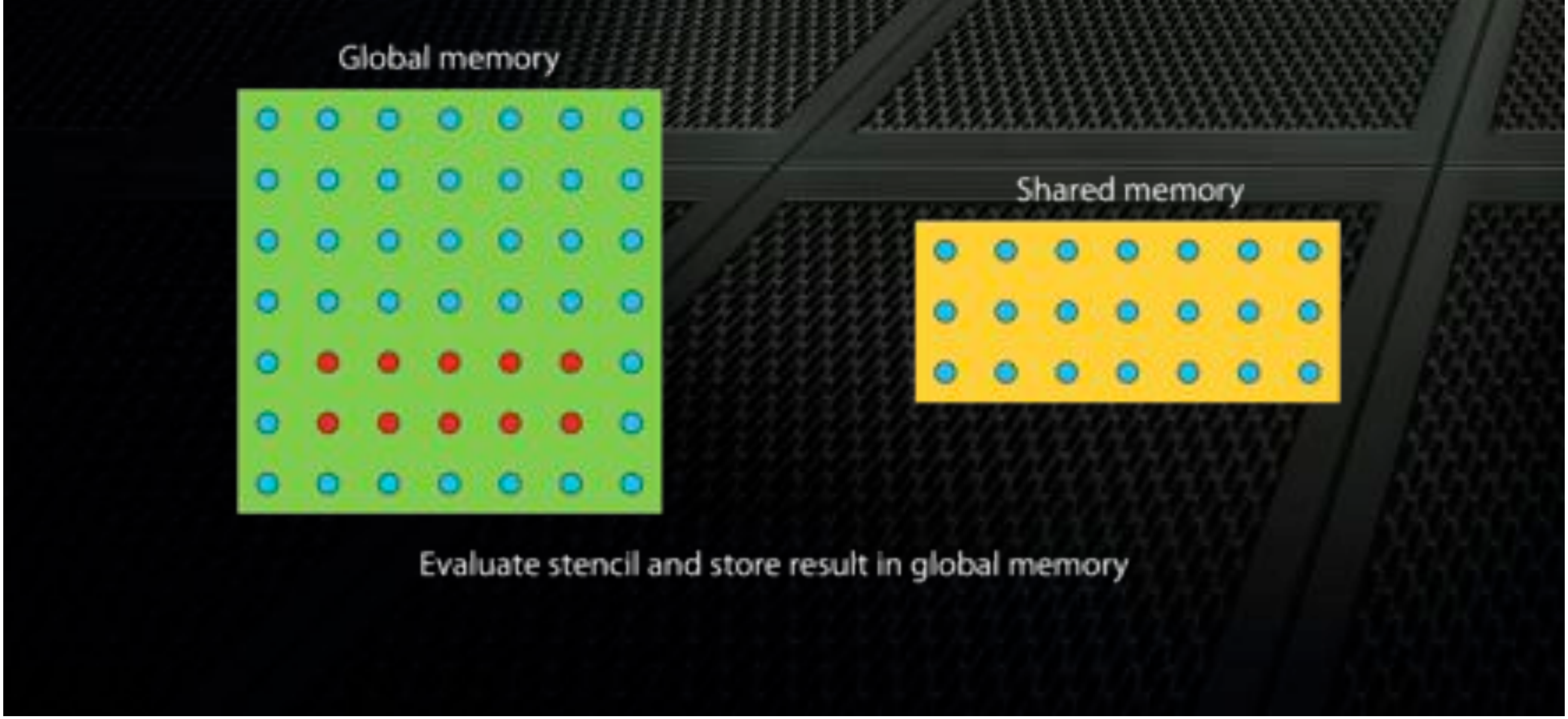
CUDA strategy



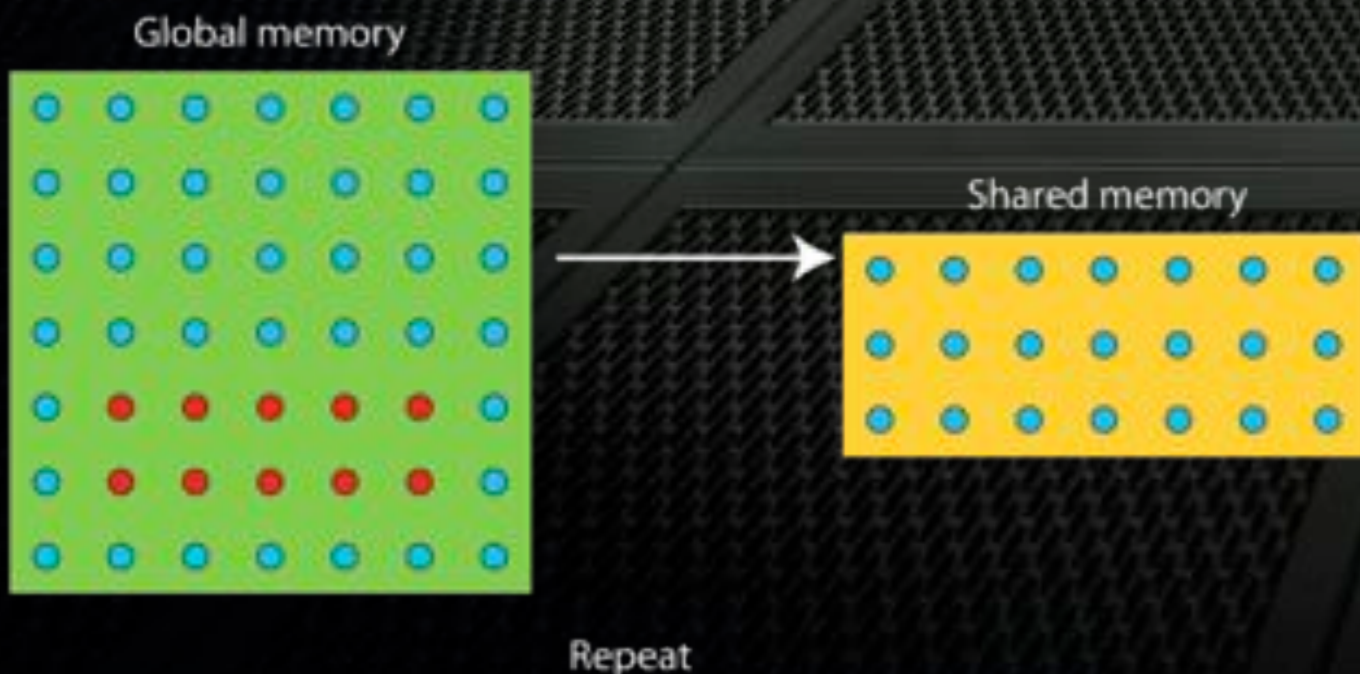
CUDA strategy



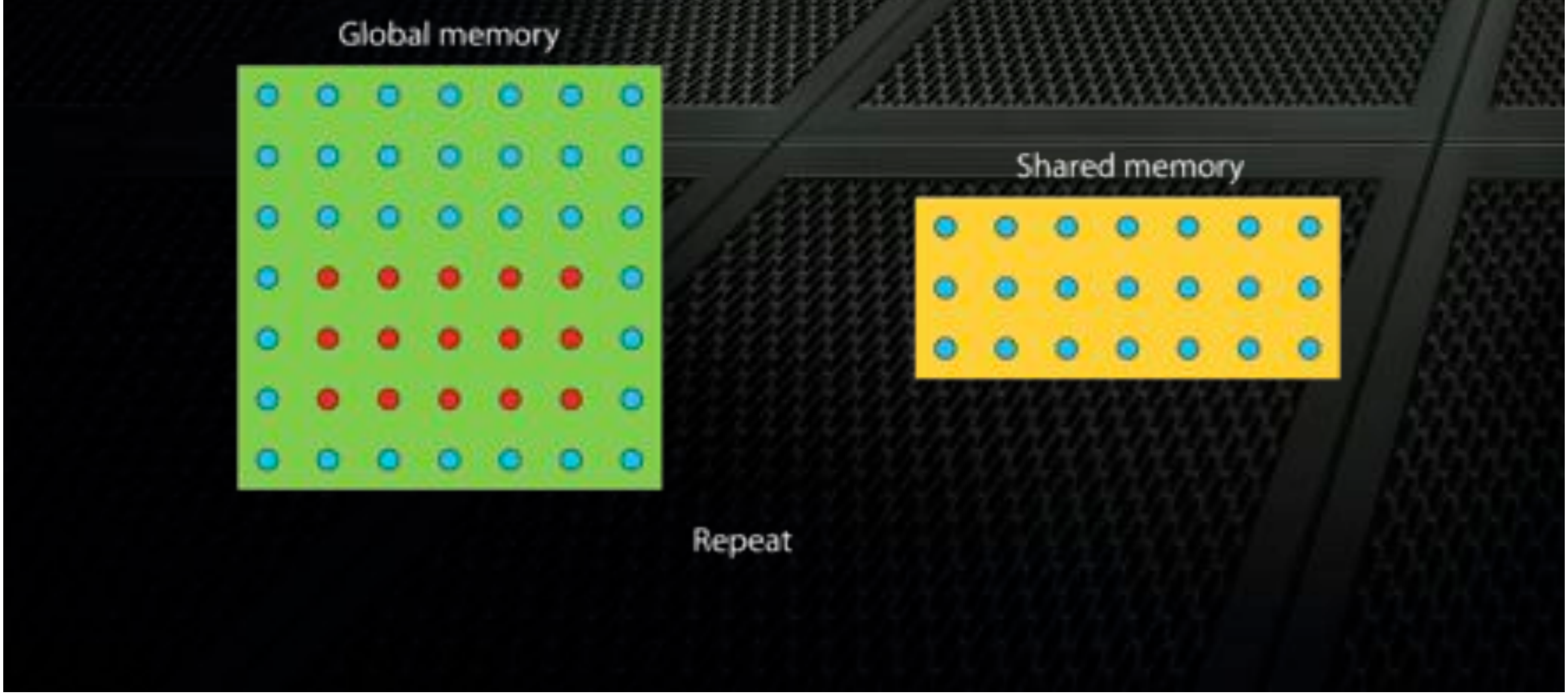
CUDA strategy



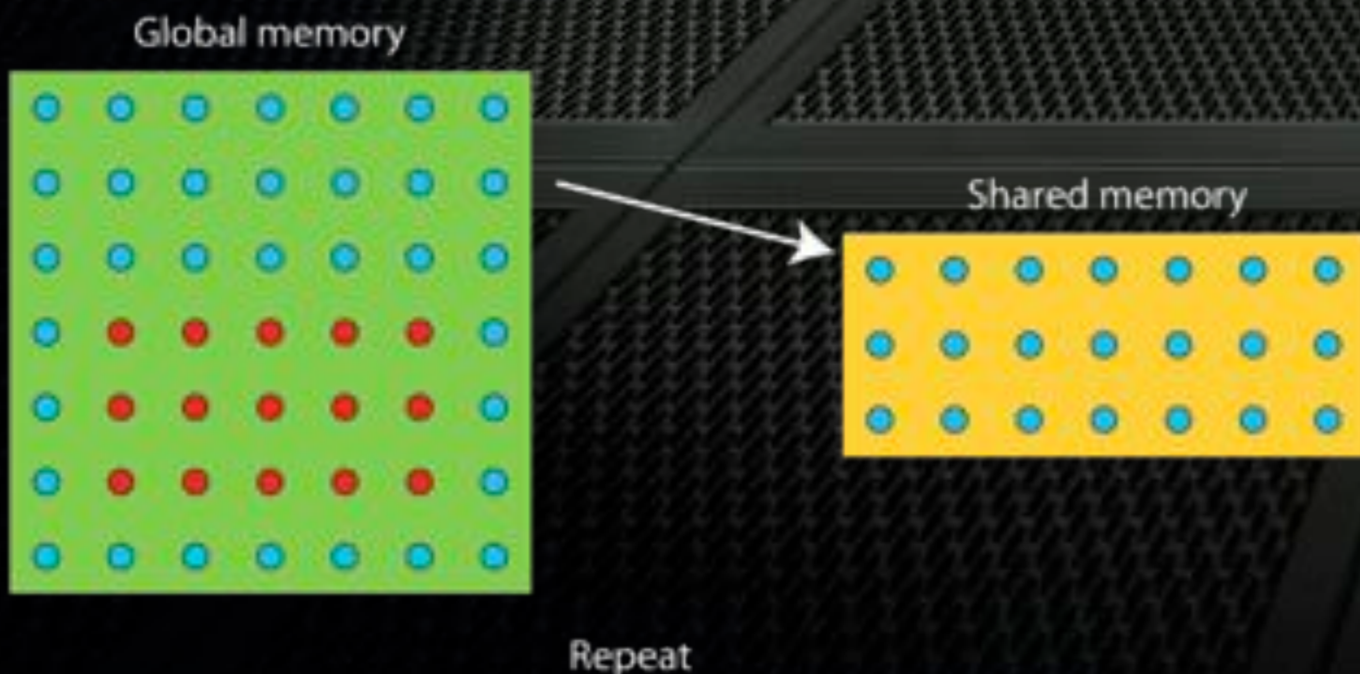
CUDA strategy



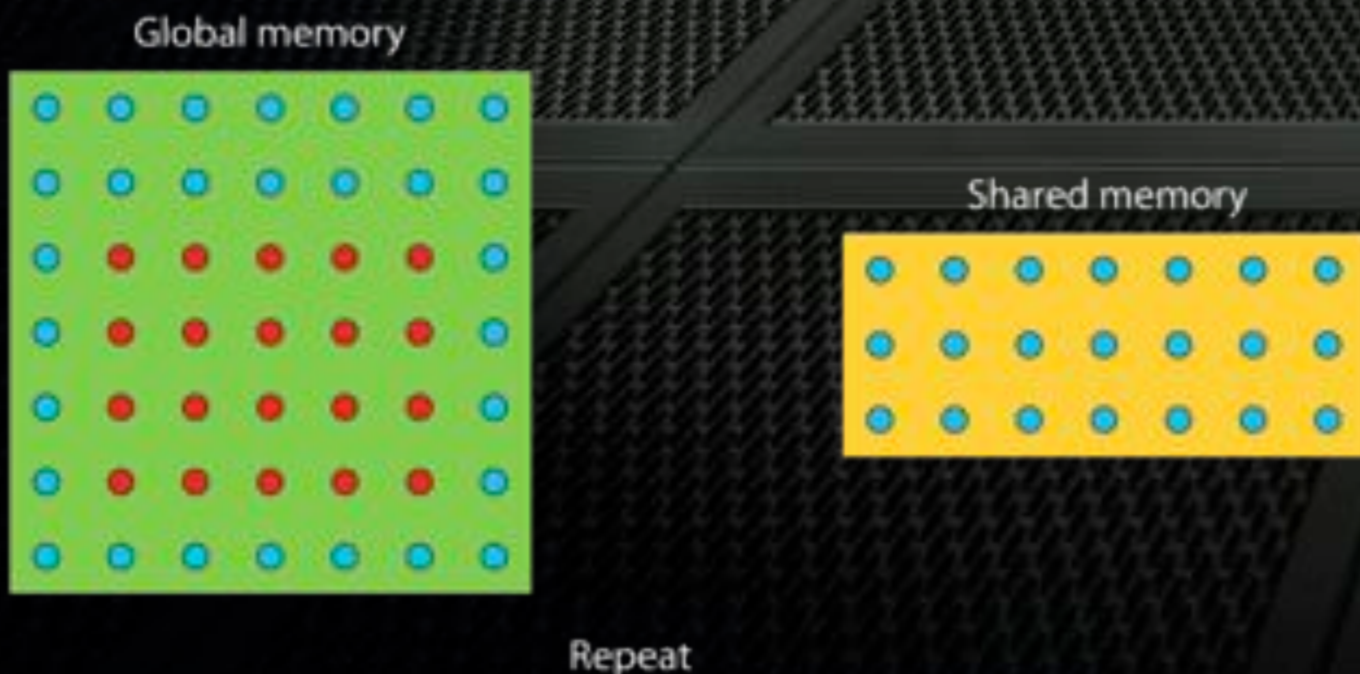
CUDA strategy



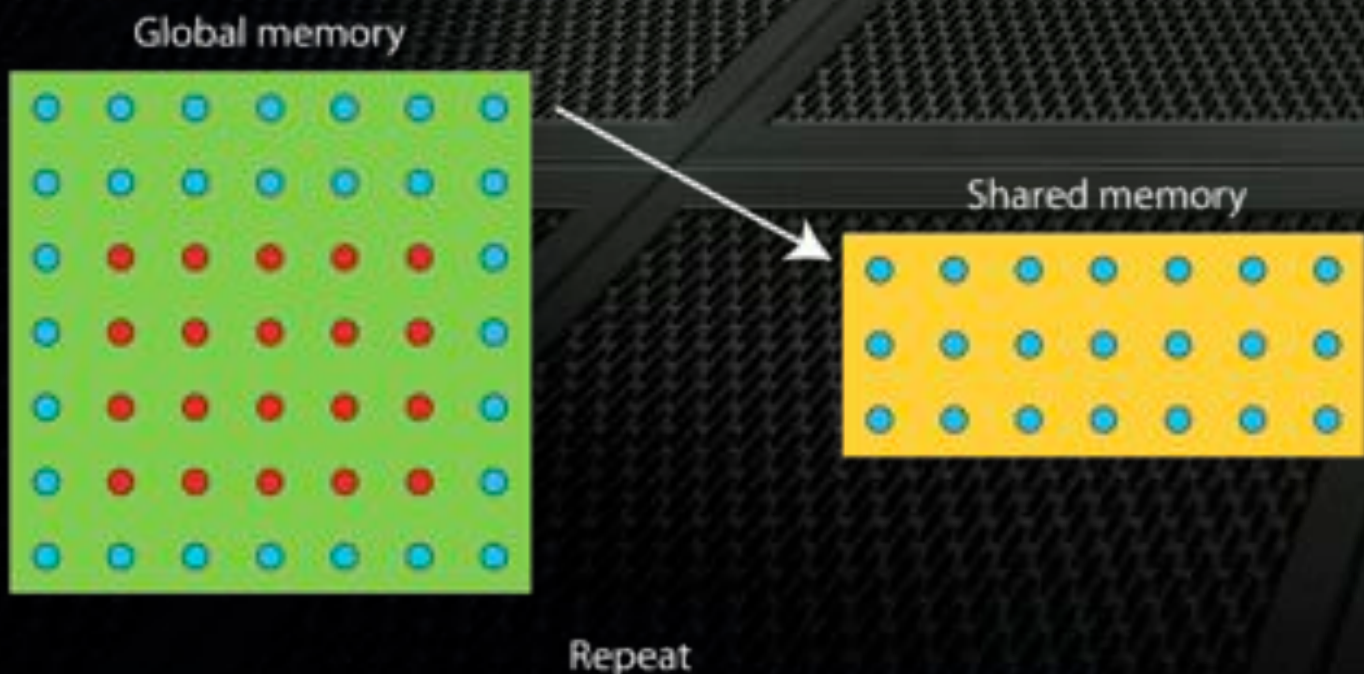
CUDA strategy



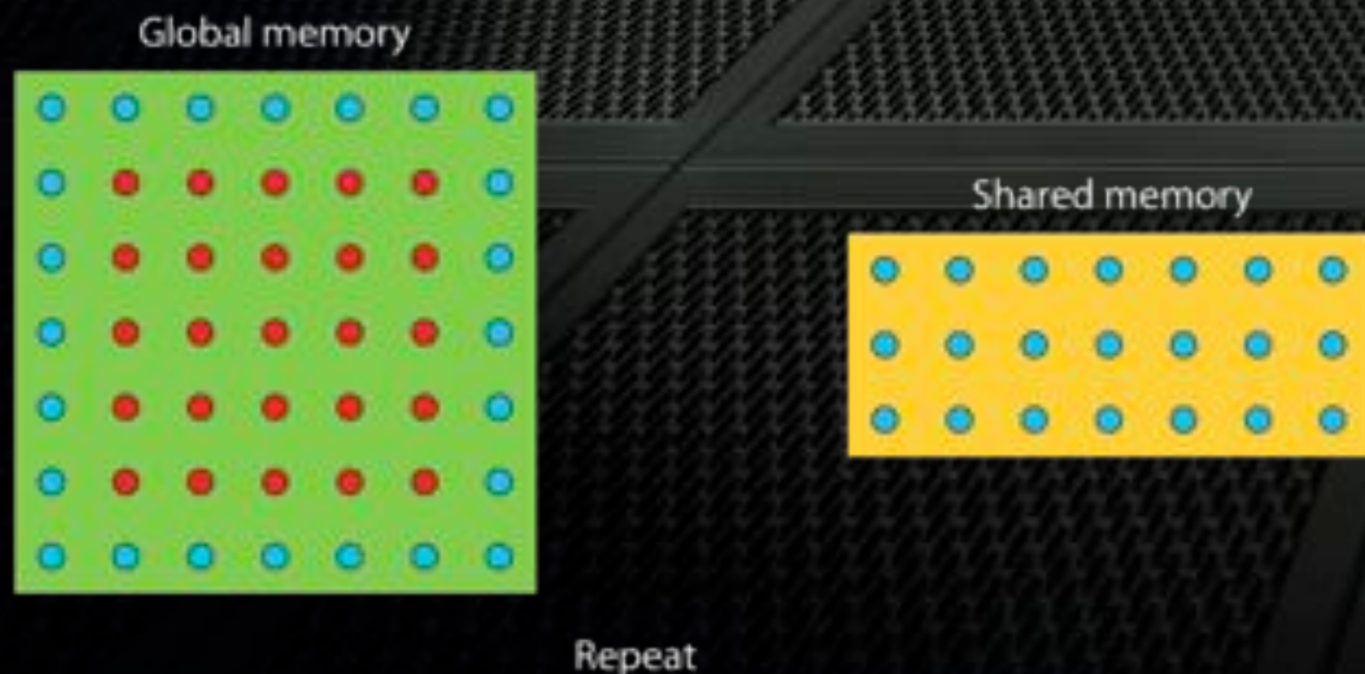
CUDA strategy



CUDA strategy



CUDA strategy



CUDA example

```
__global__ void smooth_kernel(float sf, float *a_d, float *b_d)
{
__shared__ float a[16][5][3];  // shared memory array
```

CUDA example

```
__global__ void smooth_kernel(float sf, float *a_d, float *b_d)
{
__shared__ float a[16][5][3];  // shared memory array
a[i][j][0] = a_d[i0m10];       // fetch first three planes
a[i][j][1] = a_d[i000];
a[i][j][2] = a_d[i0p10];
__syncthreads();               // make sure planes are loaded
```

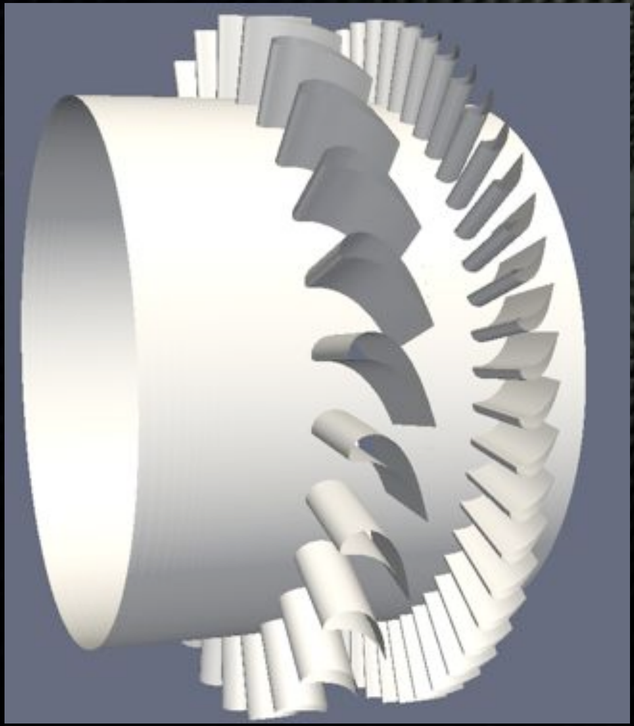
CUDA example

```
__global__ void smooth_kernel(float sf, float *a_d, float *b_d)
{
__shared__ float a[16][5][3];  // shared memory array
a[i][j][0] = a_d[i0m10];       // fetch first three planes
a[i][j][1] = a_d[i000];
a[i][j][2] = a_d[i0p10];
__syncthreads();               // make sure planes are loaded
// compute the stencil: //
b_d[i000]  = sf1*a[i][j][1] +
           + sfd6*(a[i-1][j][1] + a[i+1][j][1]
                 + a[i][j][0]   + a[i][j][2]
                 + a[i][j-1][1] + a[i][j+1][1])
// load next "k" plane and repeat //
```

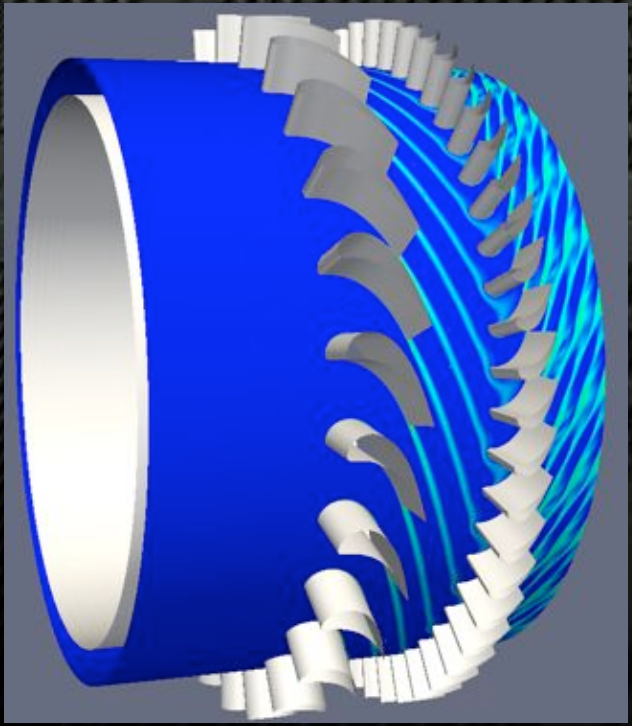
Turbostream

- CUDA port of existing FORTRAN code (TBLOCK)
- 15,000 lines FORTRAN
- 5,000 lines kernel definitions -> 30,000 lines of CUDA
- Runs on CPU or multiple GPUs
- 20x speedup on Tesla C1060 as compared to all cores of a modern Intel core2 quad.

Turbostream

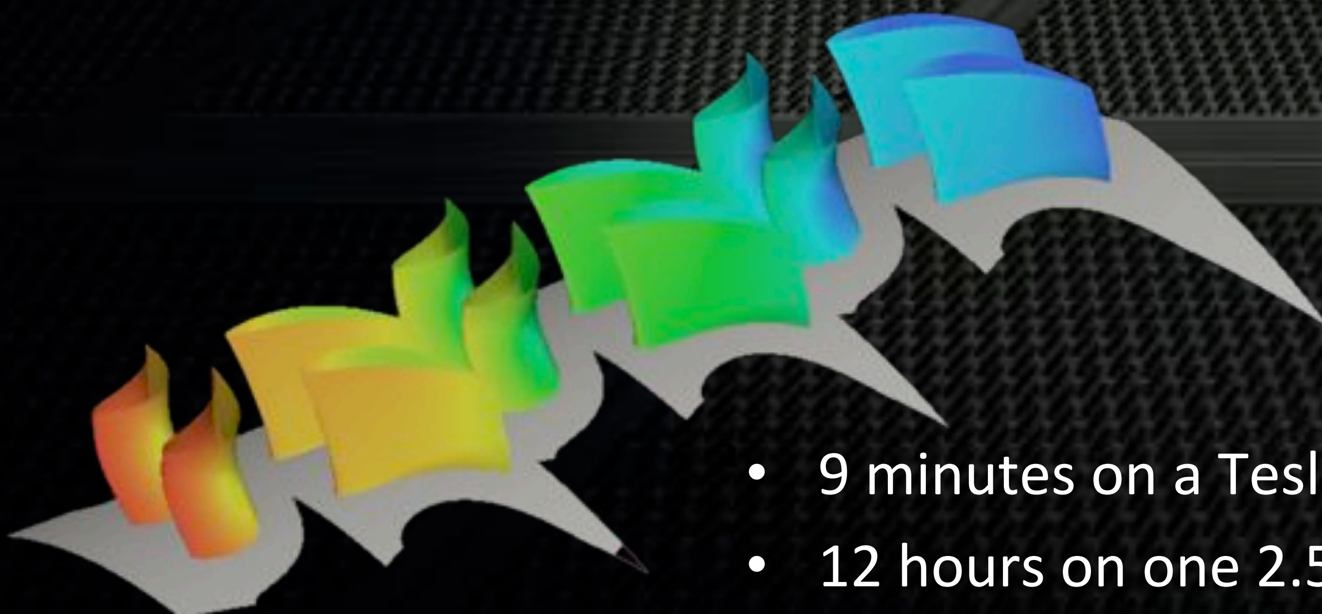


Turbine geometry



Flow solution

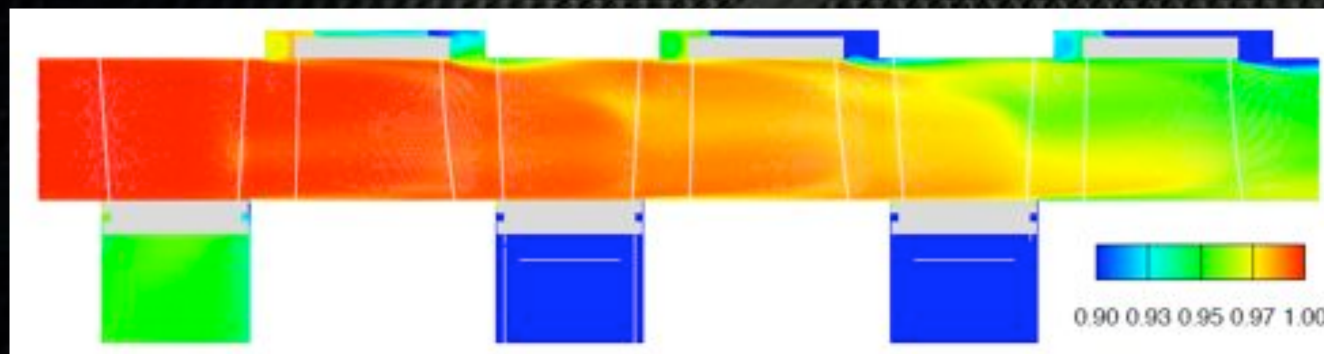
Turbostream



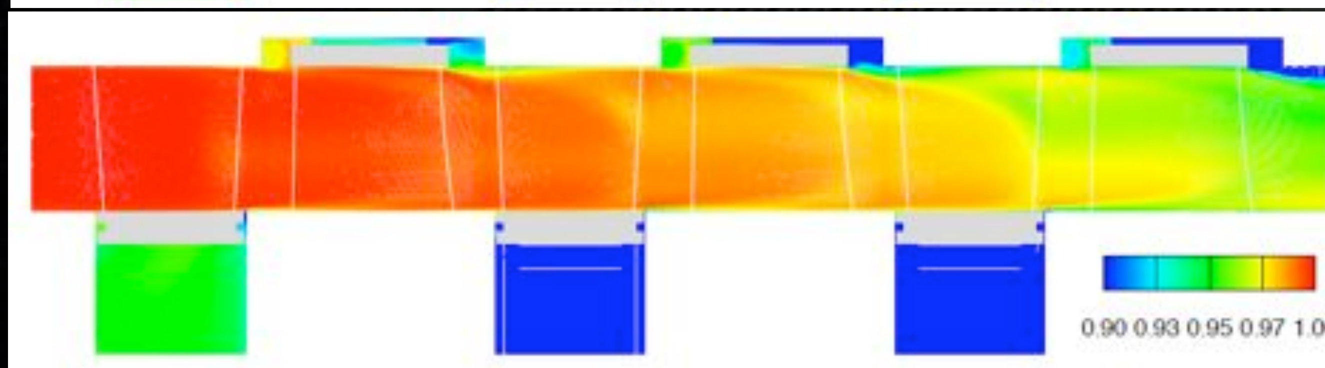
- 9 minutes on a Tesla S870 (4 GPUs)
- 12 hours on one 2.5GHz CPU core

FORTRAN & CUDA comparison

Fortran



CUDA



Impact of GPU accelerated CFD

- Tesla Personal Supercomputer enables
  - Full turbine in 10 minutes (not 12 hours)
  - One blade (for design) in 2 minutes
- Tesla cluster enables
  - Interactive design of blades for first time
  - Use of higher accuracy methods at early stage in design process

Summary

- Many science applications fit the SIMD model used in GPUs
- CUDA enables science developers to access to NVIDIA GPUs without cumbersome graphics APIs
- Existing codes have to be analysed and re-coded to best fit the many-core architecture
- The speedups are such that this can be worth doing
- For our application, the step-change in capability is revolutionary

More information

www.many-core.group.cam.ac.uk