



UNIVERSITY OF  
CAMBRIDGE

800 YEARS  
1209 - 2009

# Accelerating CFD with Graphics Hardware

Graham Pullan (Whittle Laboratory, Cambridge University)

16 March 2009

# Today

- Motivation
- CPUs and GPUs
- Programming NVIDIA GPUs with CUDA
- Application to turbomachinery CFD: Turbostream
- Conclusions





UNIVERSITY OF  
CAMBRIDGE

800 YEARS  
1209 ~ 2009

# Part 1: Motivation

# Turbomachinery



Thousands of blades

Arranged in rows

Each blade row has a  
bespoke blade profile  
designed with CFD

# Approximate compute requirements

“Steady” models (no wake/blade interaction, etc)

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

# Approximate compute requirements

“Steady” models (no wake/blade interaction, etc)

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” models (with wakes, etc)

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

# Aim

To produce an order of magnitude reduction in run-times for the same hardware cost





UNIVERSITY OF  
CAMBRIDGE

800 YEARS  
1209 ~ 2009

# Part 2: CPUs and GPUs



# Moore's Law

*“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue.”*

Gordon Moore (Intel), 1965



# Moore's Law

*“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue.”*

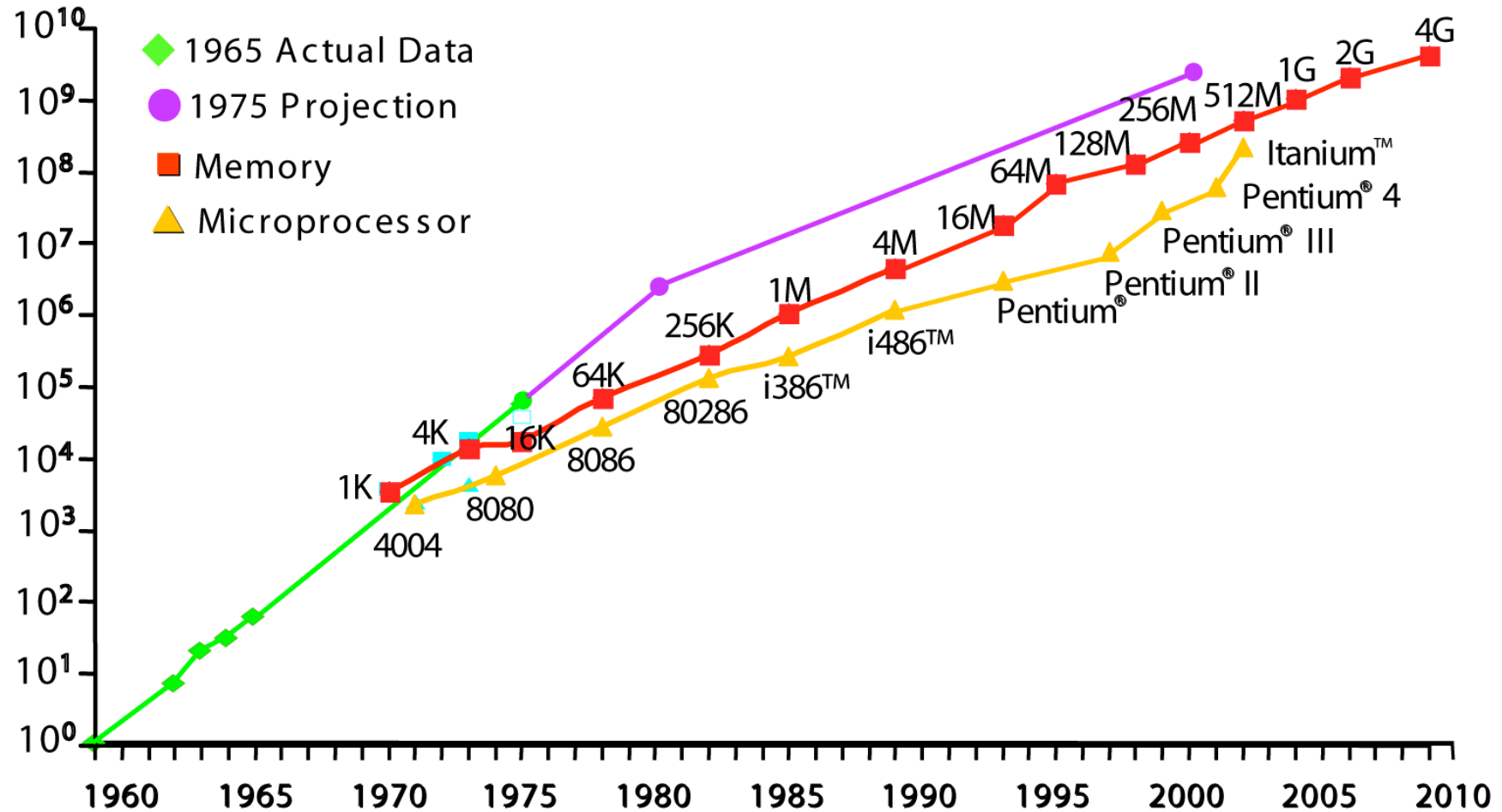
Gordon Moore (Intel), 1965

*“OK, maybe a factor of two every **two** years.”*

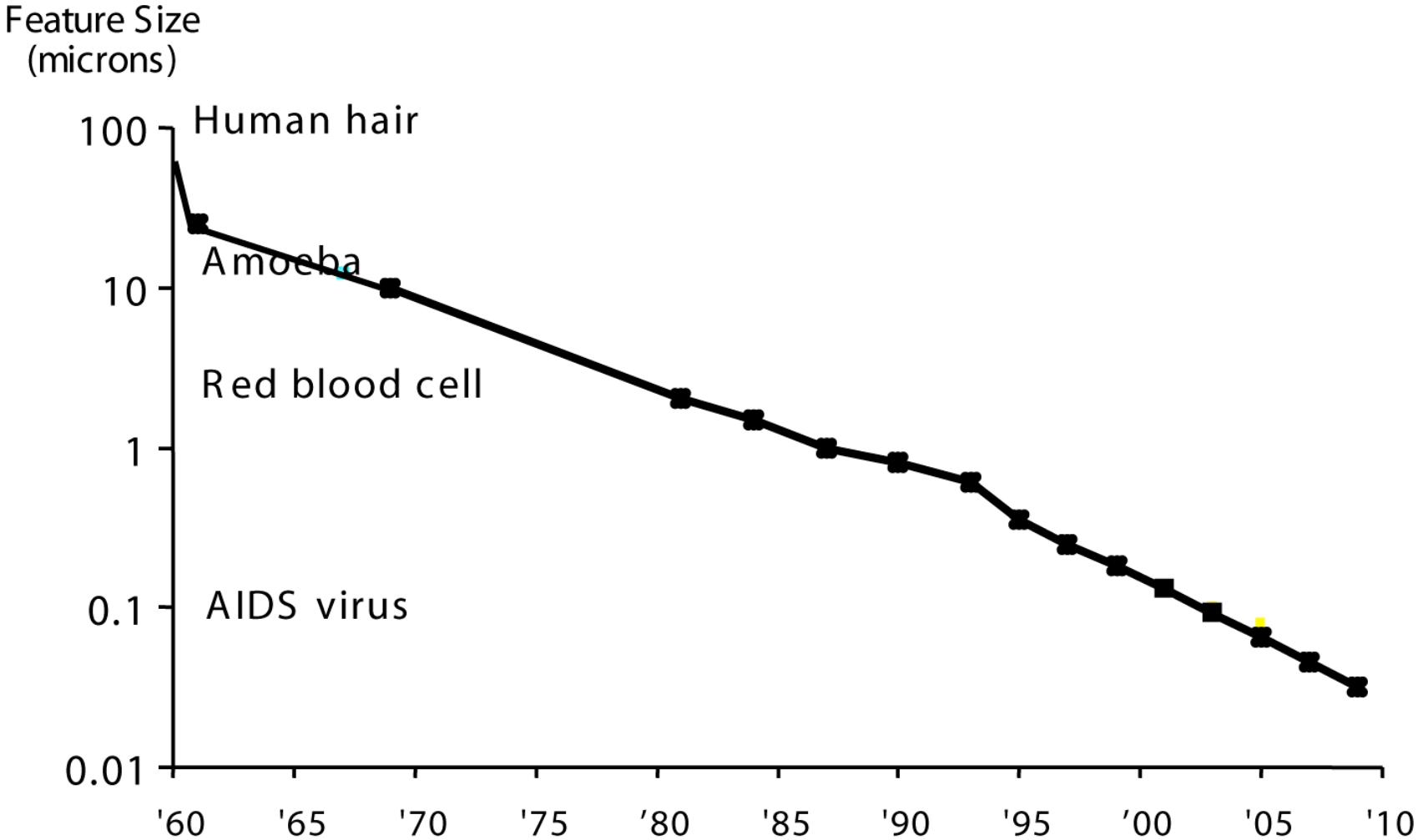
Gordon Moore (Intel), 1975 [paraphrased]

# Was Moore right?

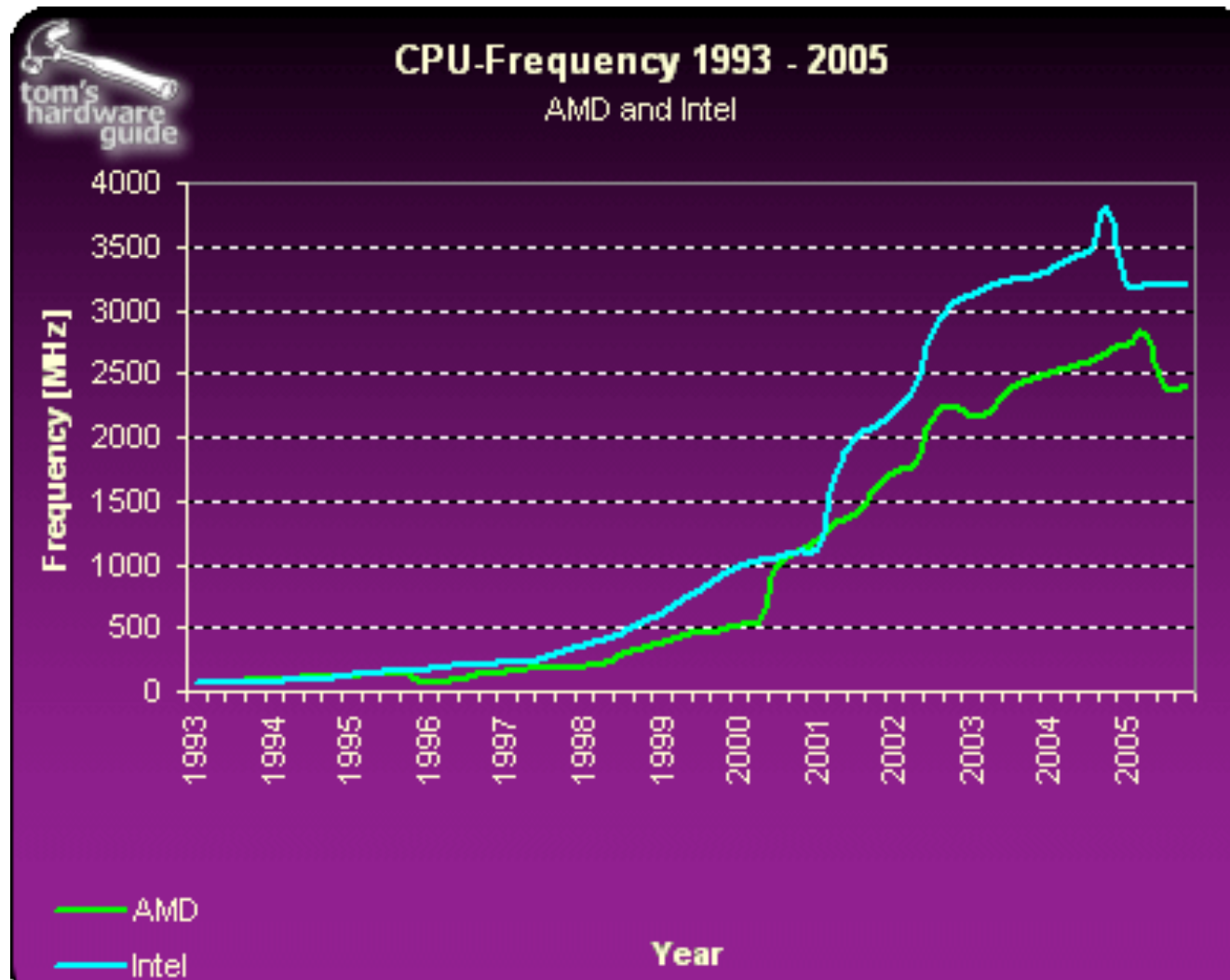
## Transistors



# Feature size



# Clock speed



# What to do with all these transistors?

# Parallel computing

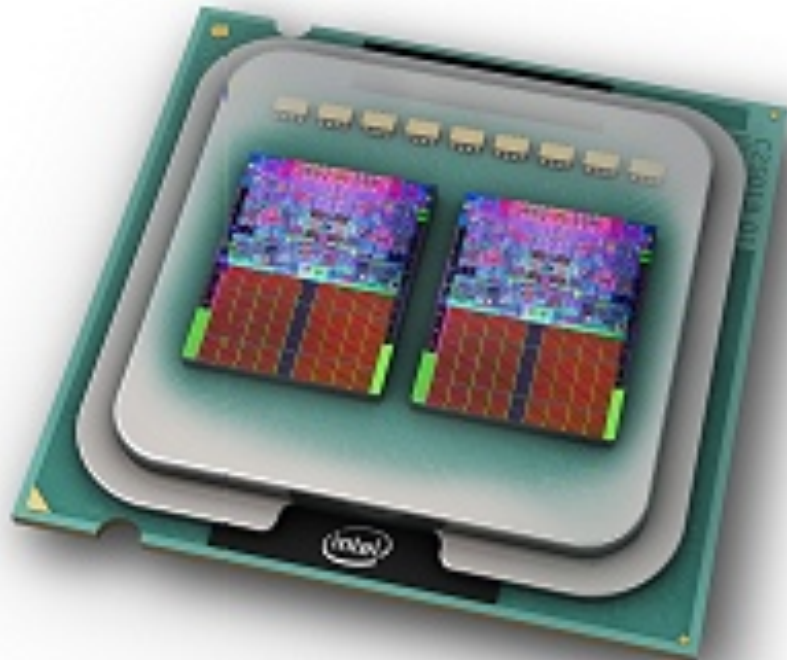
Multi-core chips are either:

- Instruction parallel  
(Multiple Instruction, Multiple Data) – MIMD

or

- Data parallel  
(Single Instruction, Multiple Data) – SIMD

# Today's commodity MIMD chips: CPUs



## Intel Core 2 Quad

- 4 cores
- 2.4 GHz
- 65nm features
- 582 million transistors
- 8MB on chip memory





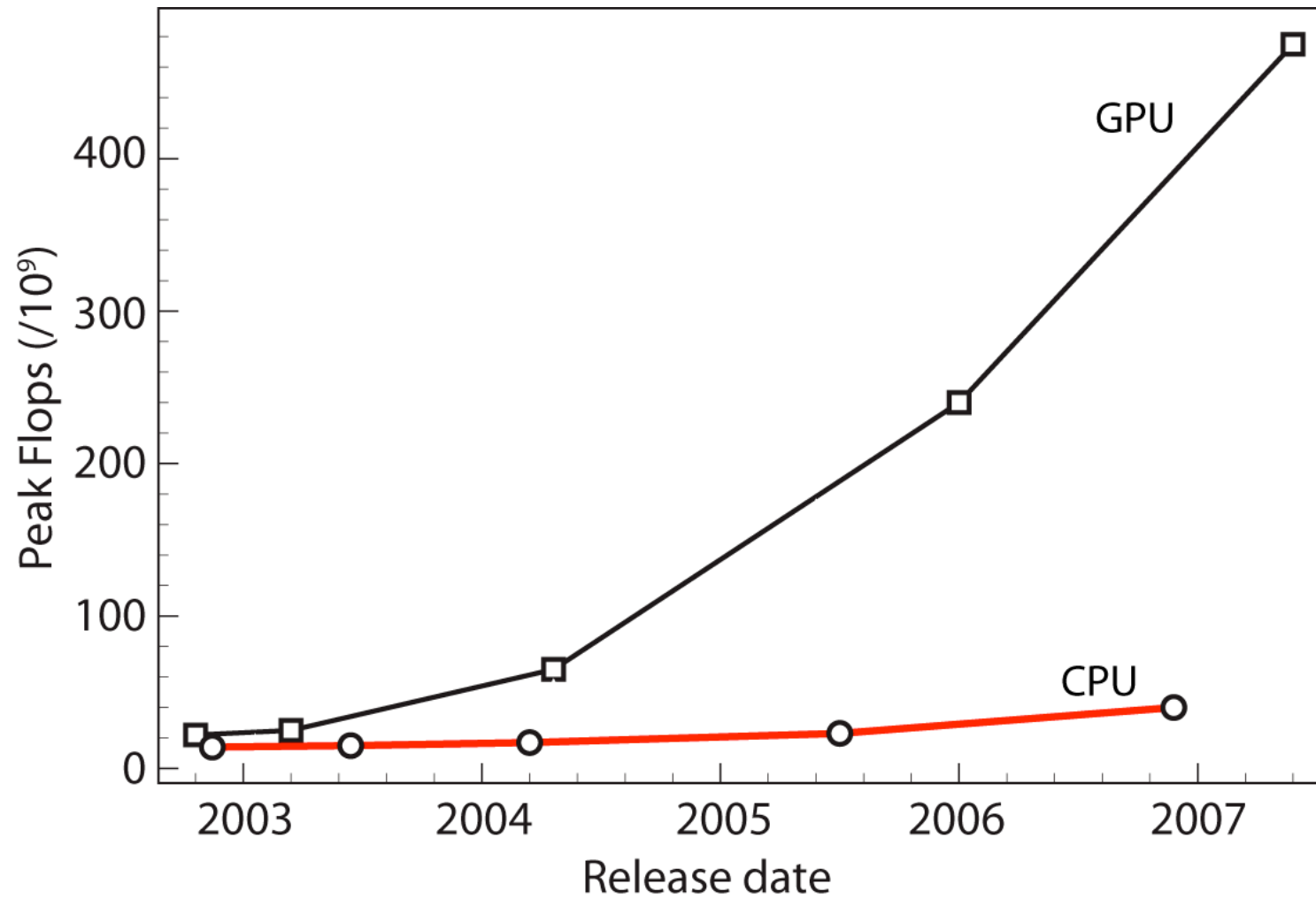
# Today's commodity SIMD chips: GPUs



## NVIDIA 8800 GTX

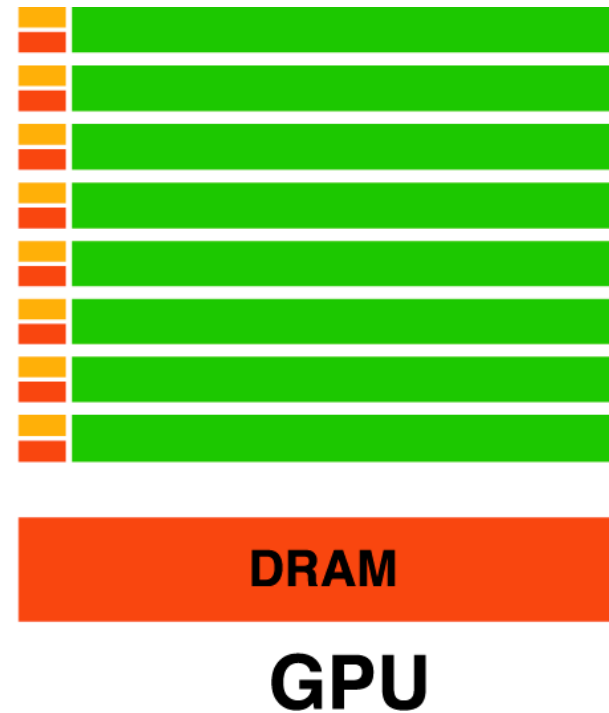
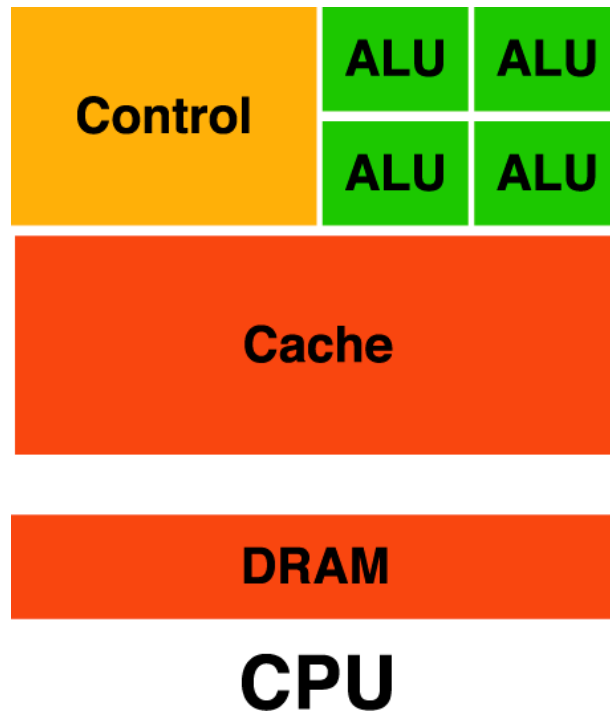
- 128 cores
- 1.35 GHz
- 90nm features
- 681 million transistors
- 768MB on board memory

# CPUs vs GPUs

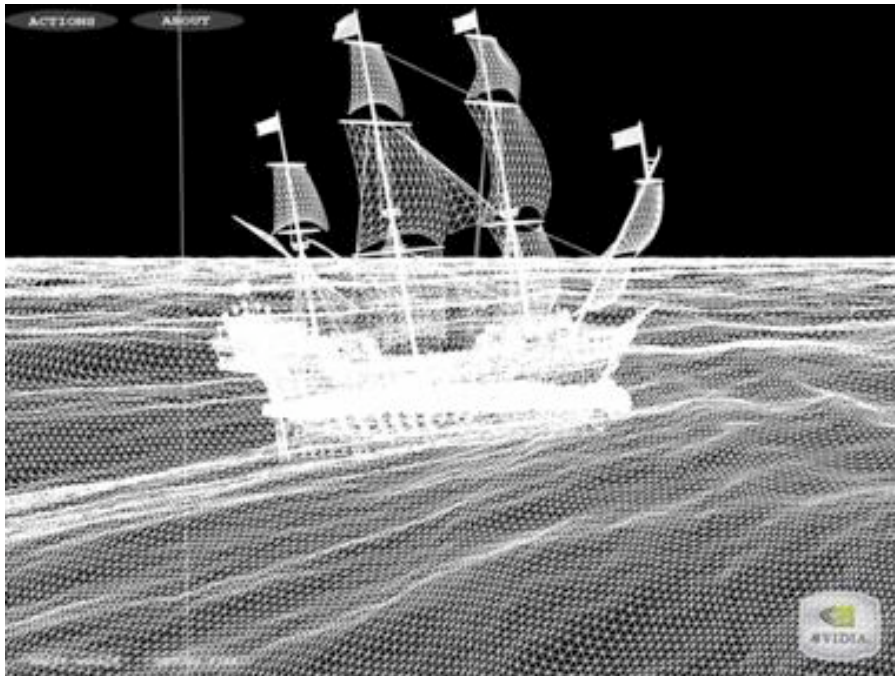


# CPUs vs GPUs

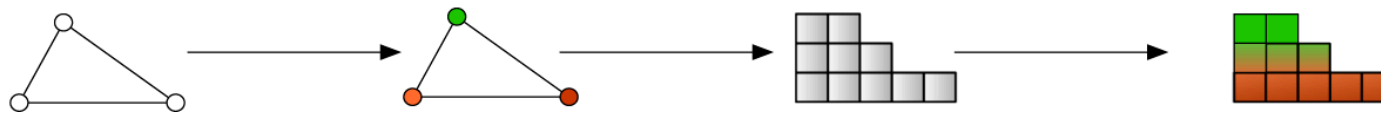
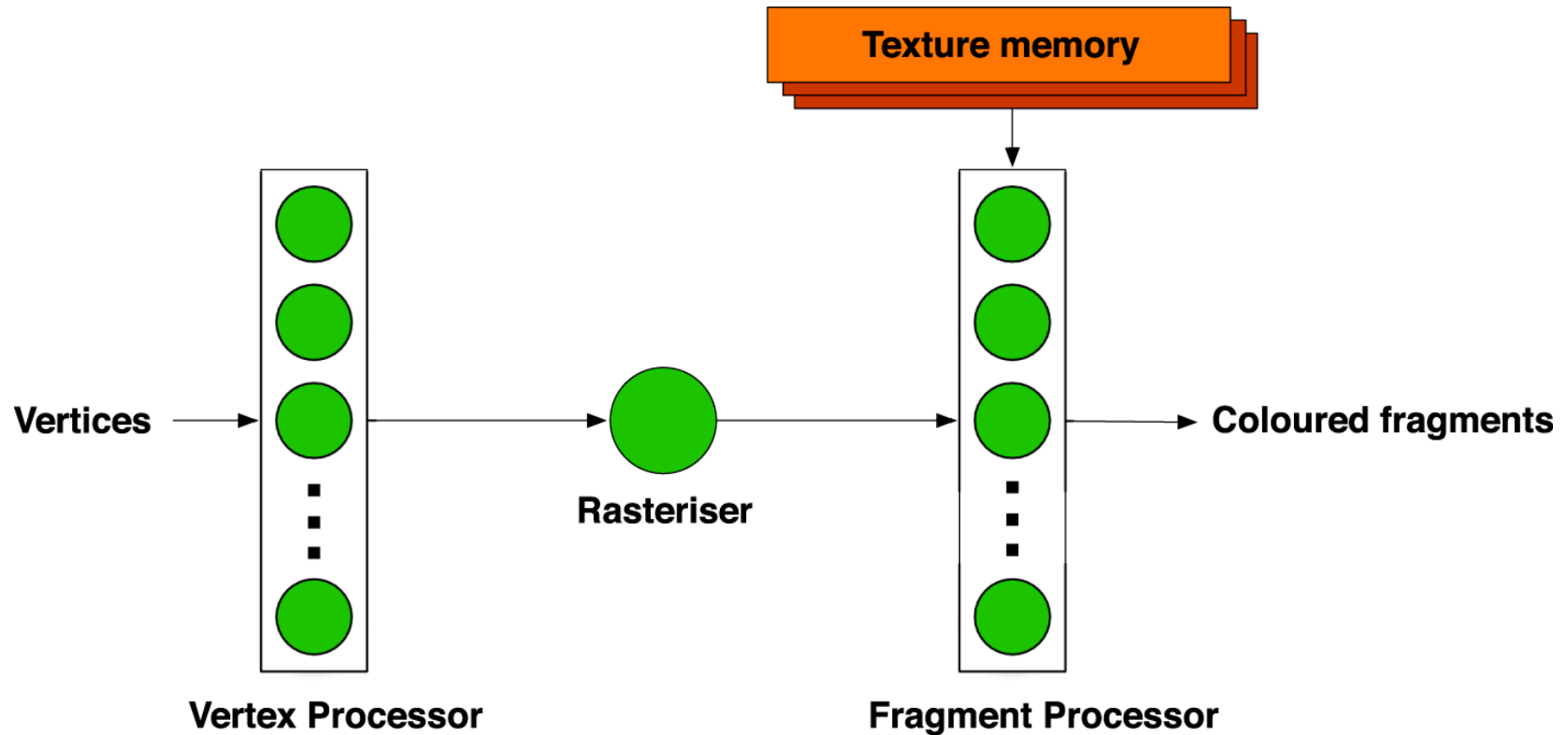
Transistor usage:



# Graphics pipeline



# (Traditional) graphics pipeline



# GPUs and scientific computing

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

# GPUs and scientific computing

GPUs are designed to apply the  
same *function*

to many *data* simultaneously

This is what most scientific computing needs!



UNIVERSITY OF  
CAMBRIDGE

800 YEARS  
1209 ~ 2009

# Part 3: Programming GPUs with CUDA



# 3 Generations of GPGPU (Owens, 2008)



## 3 Generations of GPGPU (Owens, 2008)

- Making it work at all:
  - Primitive functionality and tools (graphics)
  - Comparisons with CPU not rigorous

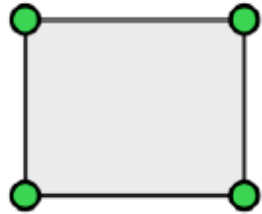
## 3 Generations of GPGPU (Owens, 2008)

- Making it work at all:
  - Primitive functionality and tools (graphics)
  - Comparisons with CPU not rigorous
- Making it work better:
  - Easier to use (higher level)
  - Understanding of how best to do it

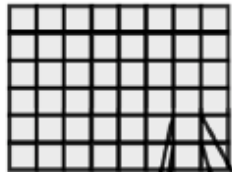
# 3 Generations of GPGPU (Owens, 2008)

- Making it work at all:
  - Primitive functionality and tools (graphics)
  - Comparisons with CPU not rigorous
- Making it work better:
  - Easier to use (higher level)
  - Understanding of how best to do it
- Doing it right:
  - Stable, portable, modular building blocks

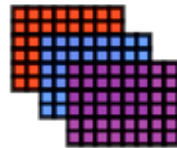
# GPU – Programming for graphics



Application specifies geometry – GPU rasterizes



Each fragment is shaded (SIMD)



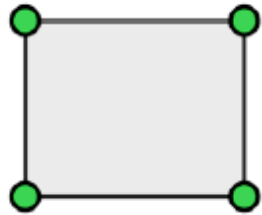
Shading can use values from memory (textures)



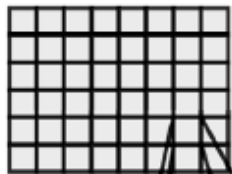
Image can be stored for re-use



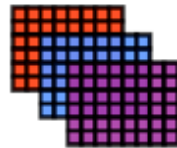
# GPGPU programming (“old-school”)



Draw a quad



Run a SIMD program over each fragment



Gather is permitted from texture memory

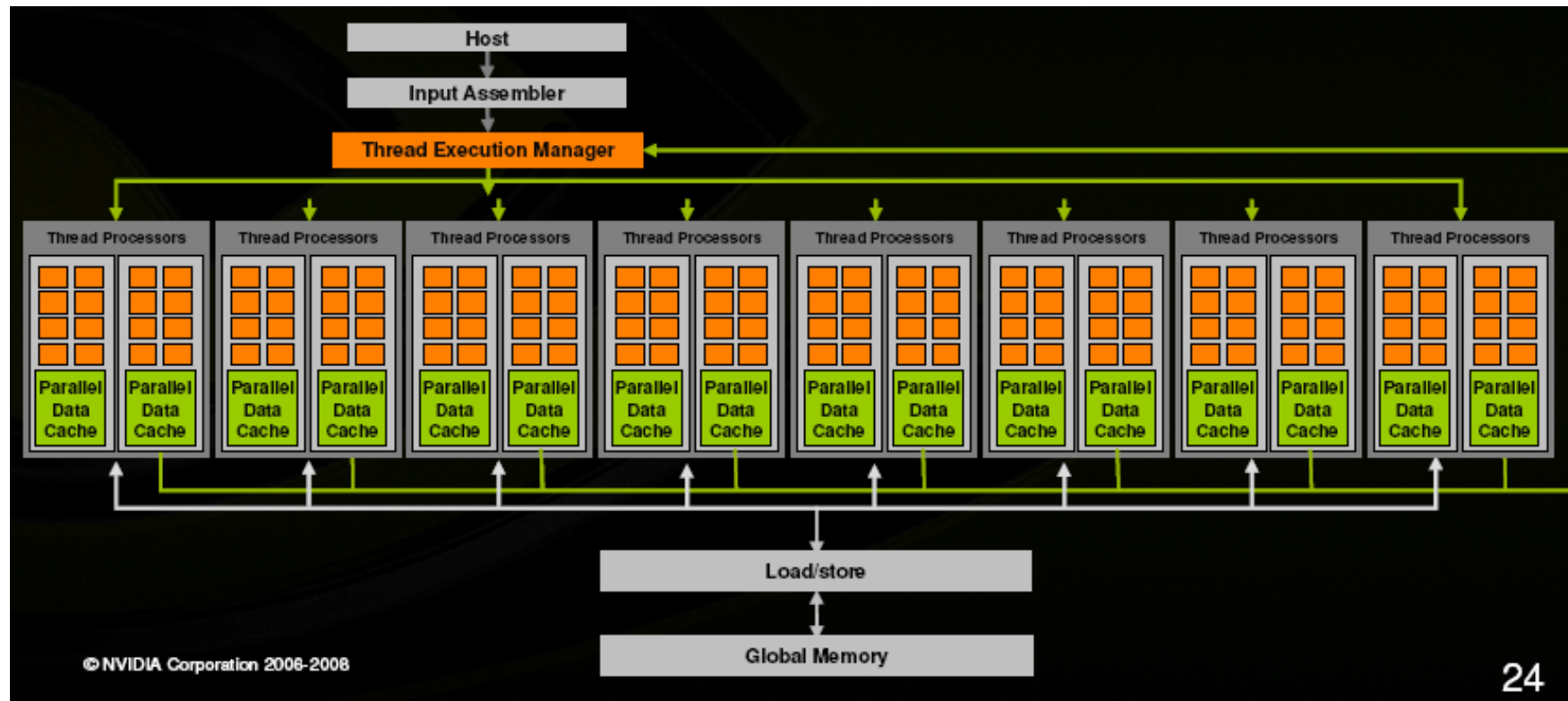


Resulting buffer can be stored for re-use

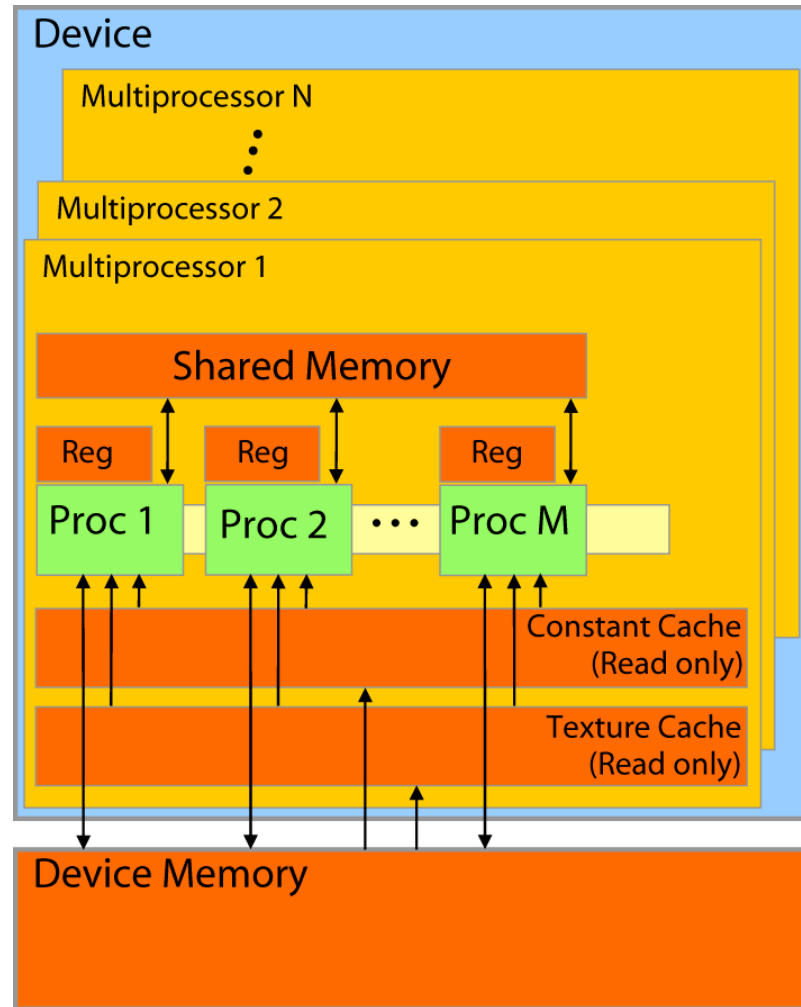


# NVIDIA G80 hardware implementation

- Vertex/fragment processors replaced by Unified Shaders
- Now view GPU as massively parallel co-processor



# NVIDIA G80 hardware implementation



Divide 128 cores into  
16 Multiprocessors (MPs)

- Each MP has:
  - Registers
  - Shared memory
  - Read only constant cache
  - Read only texture cache

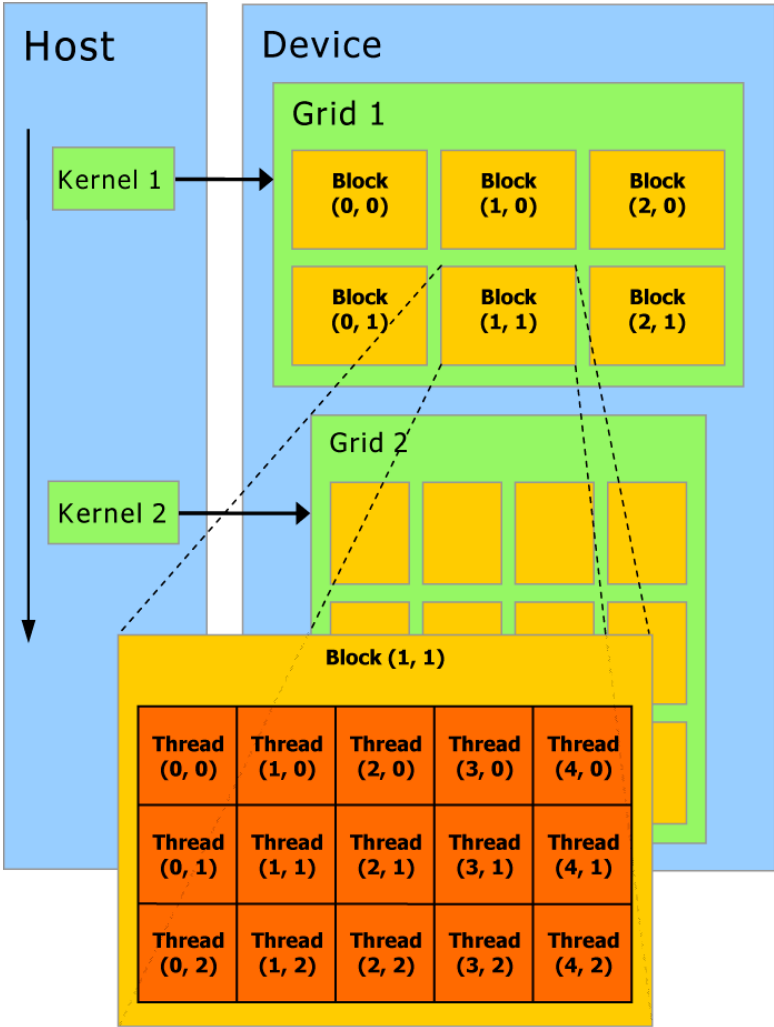


# NVIDIA's CUDA programming model

- G80 chip supports MANY active *threads*: 12,288
- Threads are lightweight:
  - Little creation overhead
  - “instant” switching
  - Efficiency achieved through 1000's of threads
- Threads are organised into *blocks* (1D, 2D, 3D)
- Blocks are further organised into a *grid*



# Kernels, grids, blocks and threads



# Kernels, grids, blocks and threads

- Organisation of threads and blocks is key abstraction

# Kernels, grids, blocks and threads

- Organisation of threads and blocks is key abstraction
- Software:
  - Threads from one block may cooperate:
    - Using data in shared memory
    - Through synchronising

# Kernels, grids, blocks and threads

- Organisation of threads and blocks is key abstraction
- Software:
  - Threads from one block may cooperate:
    - Using data in shared memory
    - Through synchronising
- Hardware:
  - A block runs on one MP
  - Hardware free to schedule any block on any MP
  - More than one block can reside on one MP

# CUDA implementation

- CUDA implemented as extensions to C
- CUDA programs:
  - explicitly manage host and device memory:
    - allocation
    - transfers
  - set thread blocks and grid
  - launch kernels
  - are compiled with the CUDA `nvcc` compiler

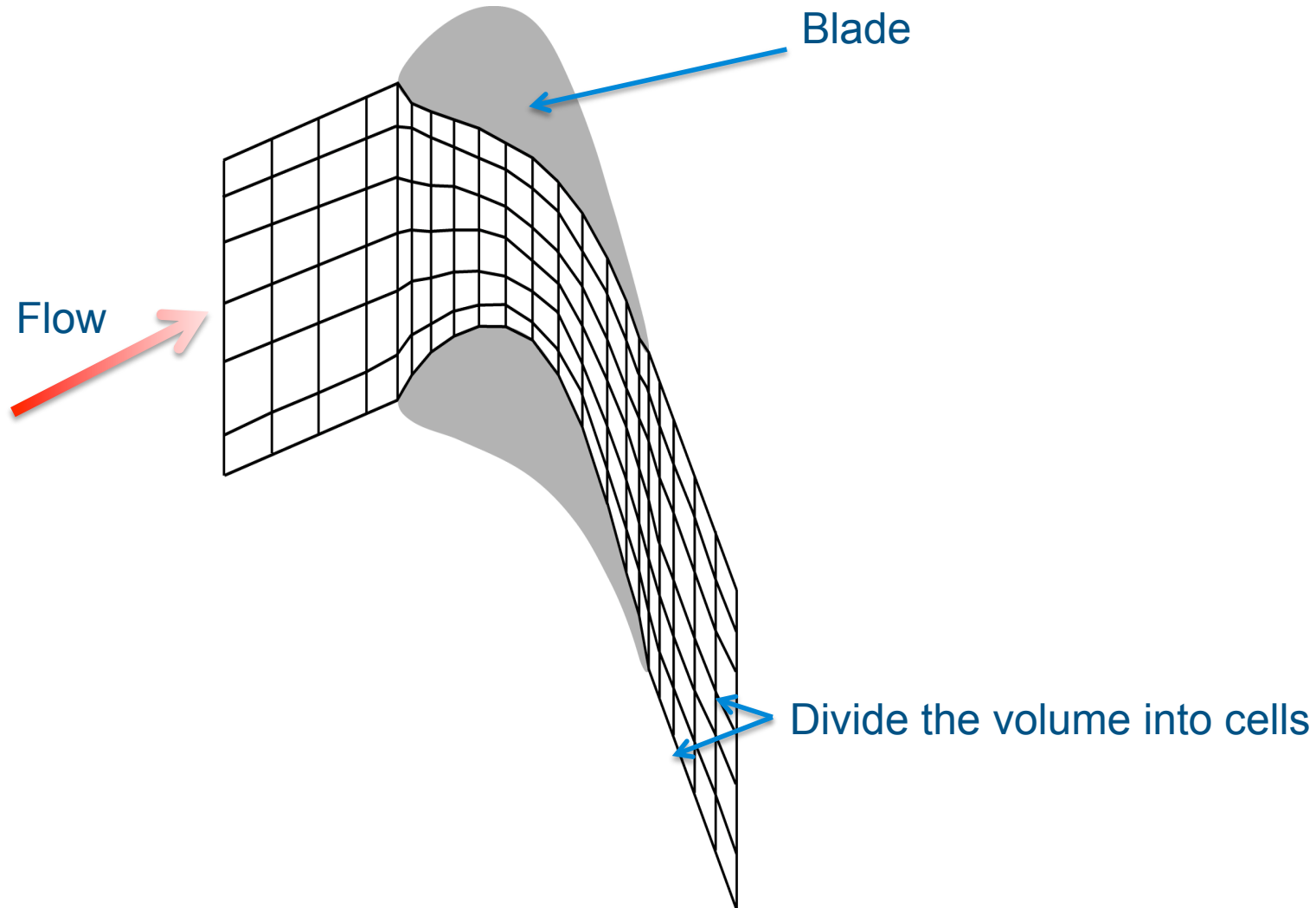


UNIVERSITY OF  
CAMBRIDGE

800 YEARS  
1209 ~ 2009

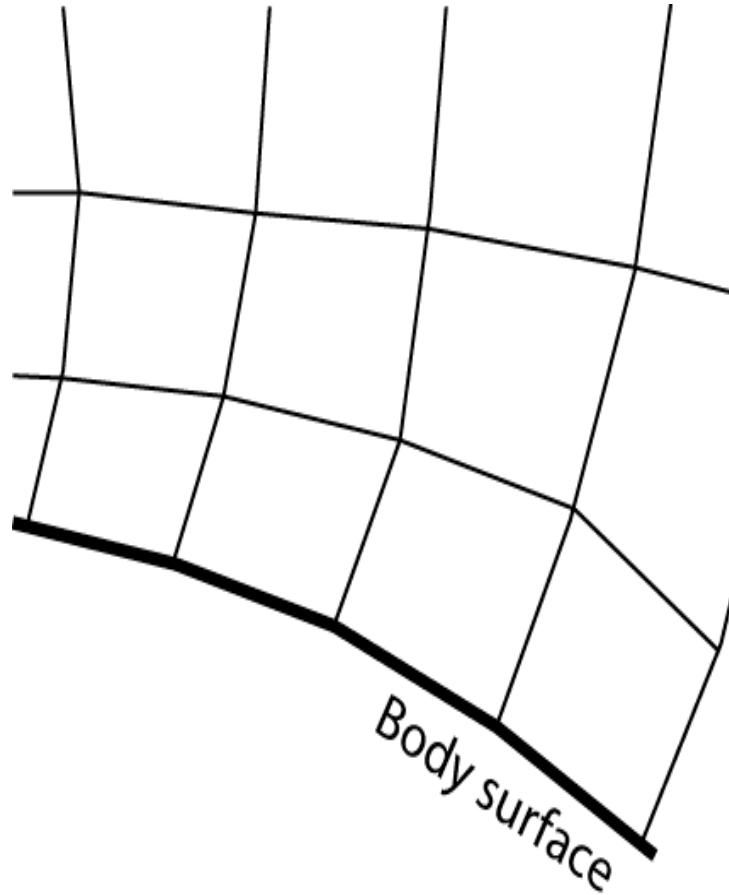
# Part 4: Application to CFD

# Introduction to CFD

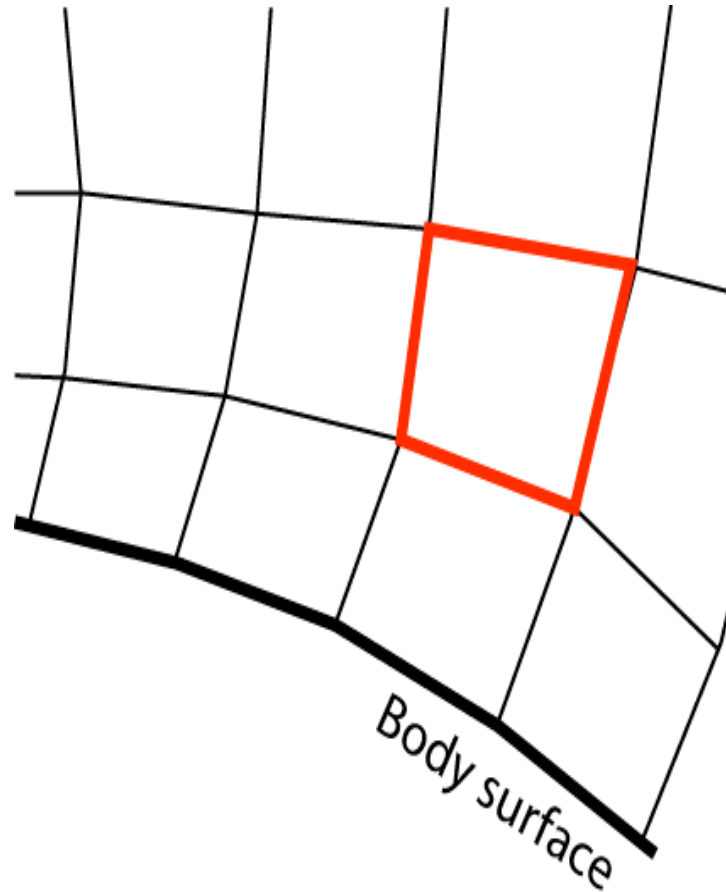




# Governing equations for each cell



# Governing equations for each cell



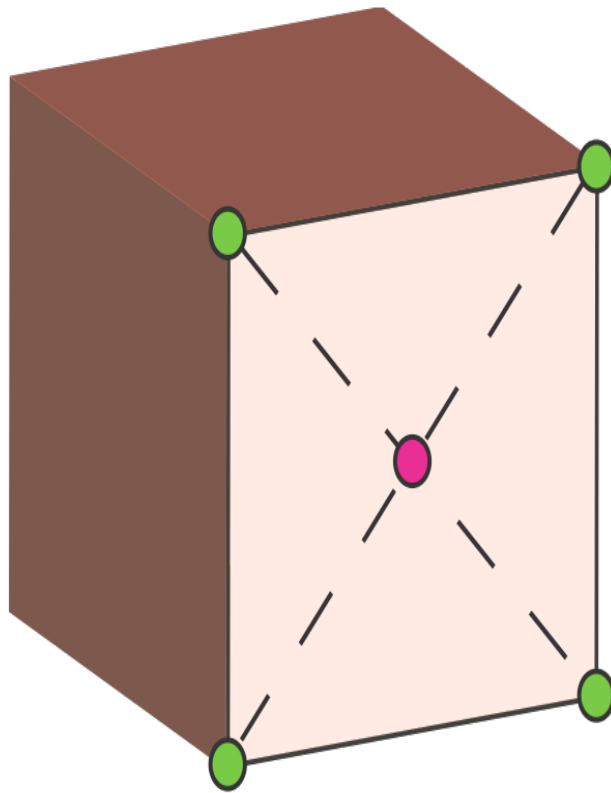
Conserve:

- Mass
- Momentum
- Energy



# Example: mass conservation

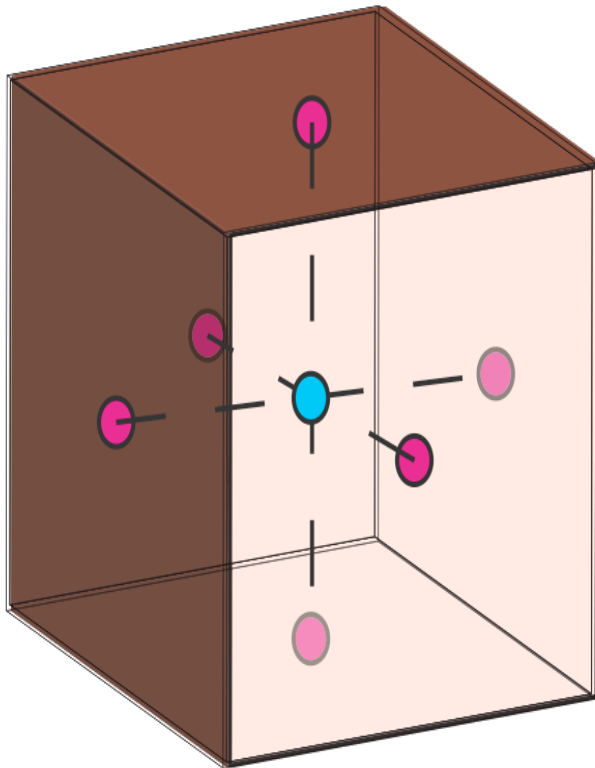
- Evaluate mass fluxes on each face



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

# Example: mass conservation

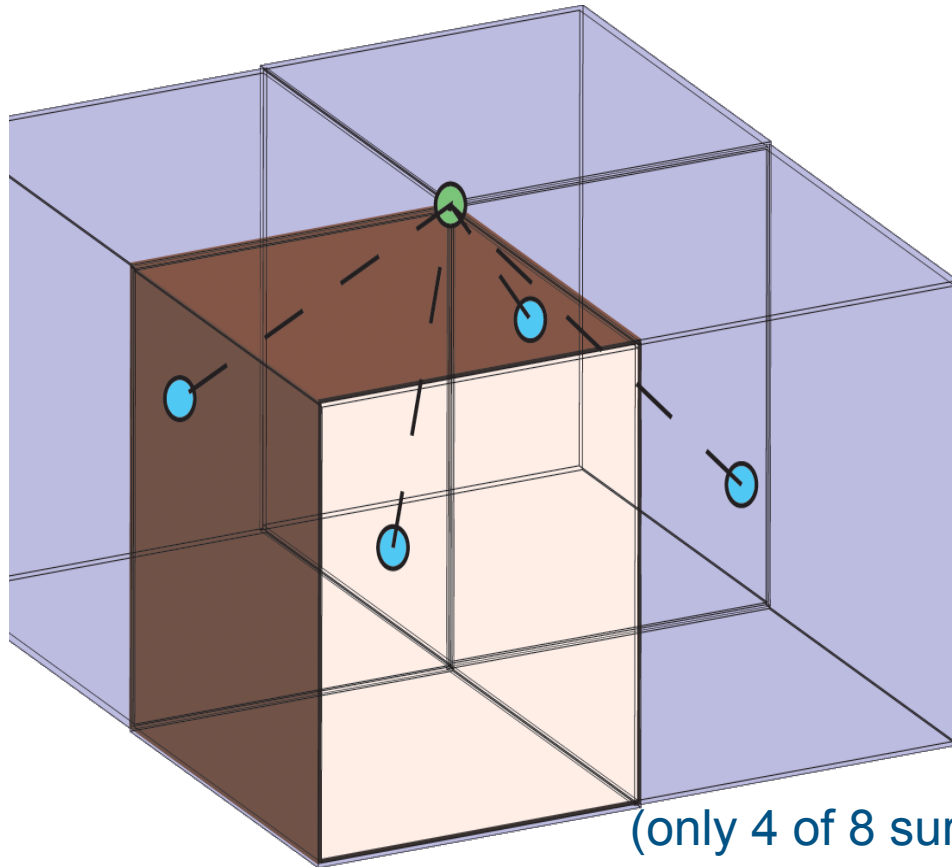
- Sum fluxes on faces to find density change in cell



$$\Delta\rho_{cell} = \sum F_{mass}$$

# Example: mass conservation

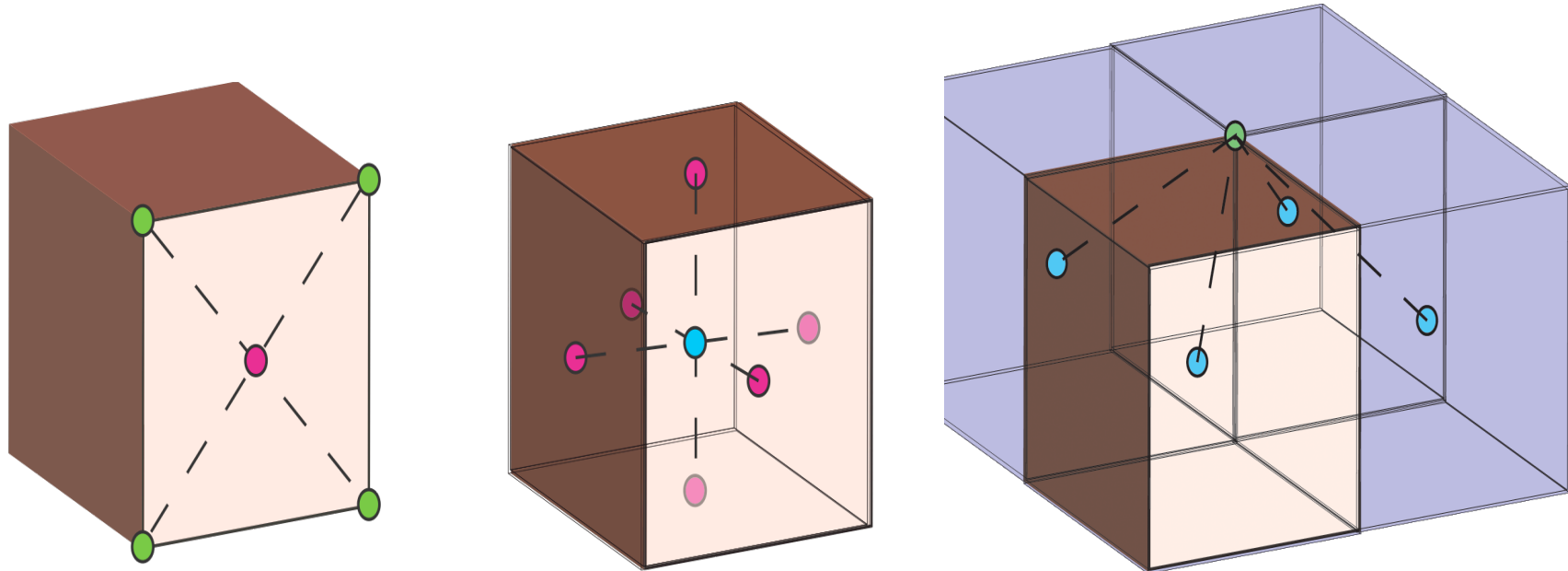
- Update density



(only 4 of 8 surrounding cells shown)

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

# Similarity of steps



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

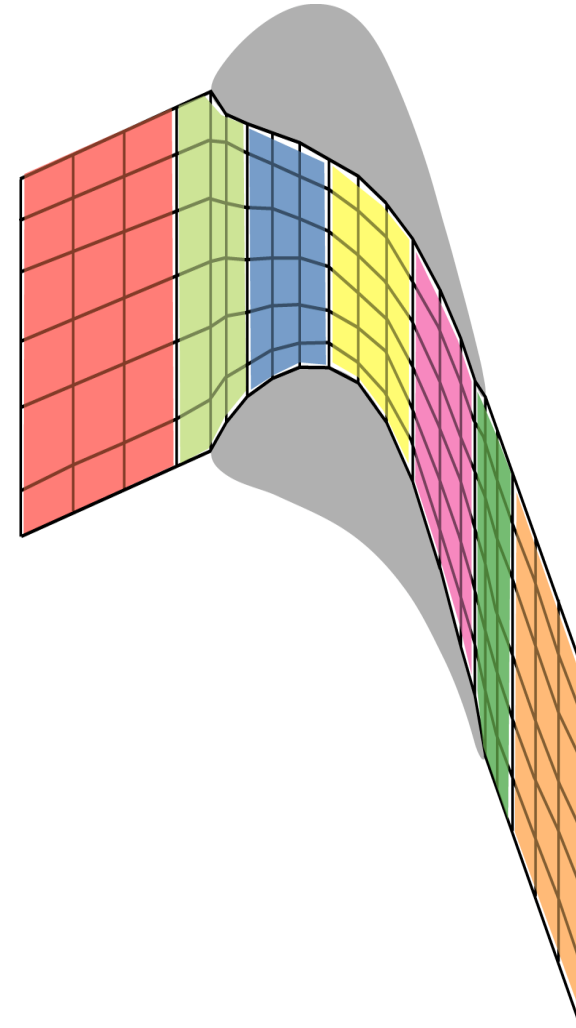
# Similarity of equations

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



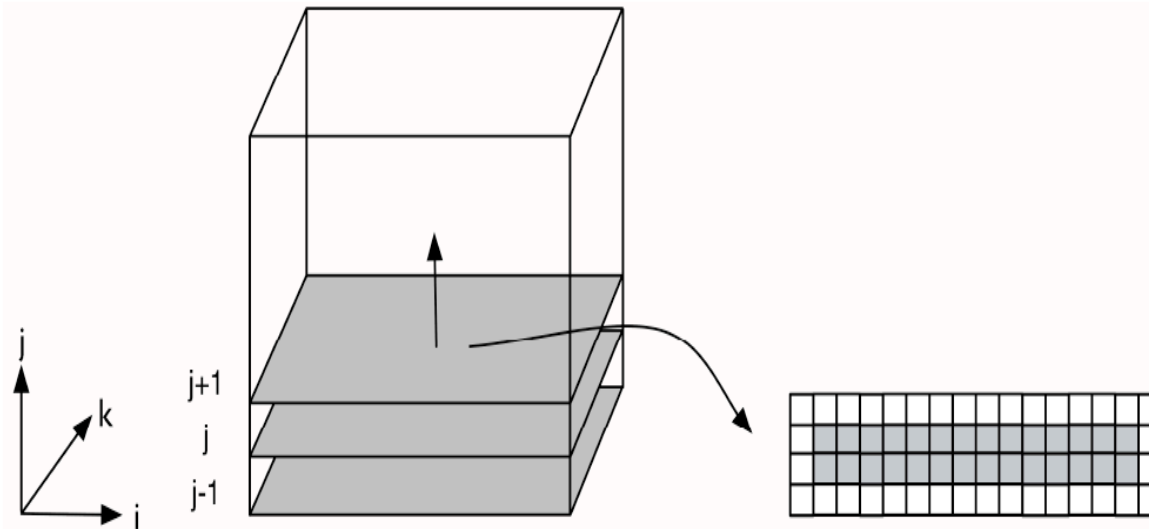
# Overall strategy

- 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)





# Efficient stencil operations



- Launch one thread per element in an  $i$ - $k$  plane
- Load enough planes into shared memory as needed by stencil
- Update elements in plane (store in global device memory)
- Load new ( $i$ - $k$ ) plane – repeat, iterate in  $j$  direction

# CUDA example

```
__global__ void smooth_kernel(float sf, float *a_data, float *b_data){
    /* shared memory array */
    __shared__ float a[16][3][5];
    /* fetch first planes */
    a[i][0][k] = a_data[i0m10];
    a[i][1][k] = a_data[i000];
    a[i][2][k] = a_data[i0p10];
    __syncthreads();
    /* compute */
    b_data[i000] =
        sf1*a[i][1][k] + sfd6*(a[im1][1][k] +
        a[ip1][1][k] + a[i][0][k] +
        a[i][2][k] + a[i][1][km1] + a[i][1][kp1])
    /* load next "j" plane and repeat ...*/
}
```

# 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!)

# 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 */
                b[i000] = sf1*a[i000] +
                    sfd6*(a[im100] + a[ip100] +
                        a[i0m10] + a[i0p10]
                        + a[i00m1] + a[i00p1]);
            }
        }
    }
}
```



# GPU implementation

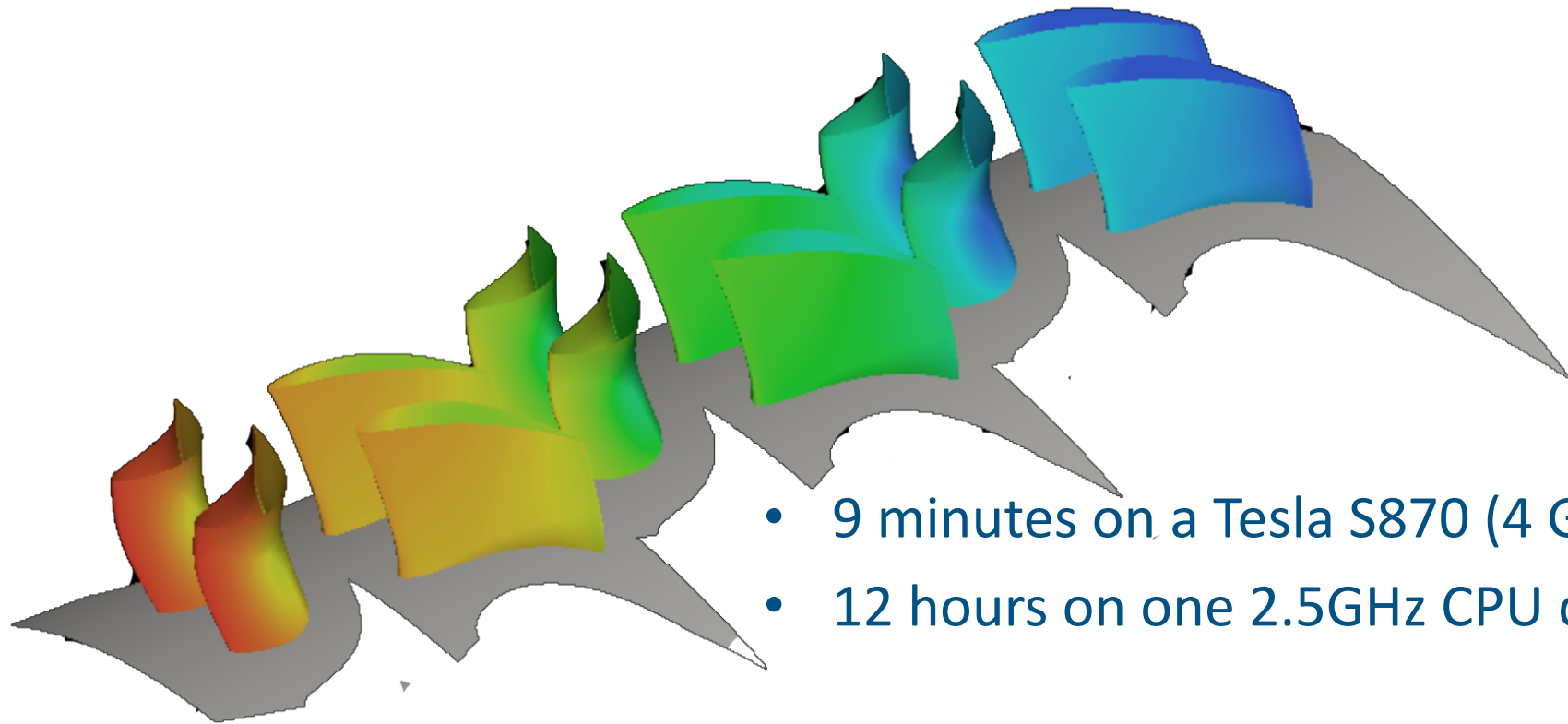
```
__global__ void smooth_kernel(float sf, float *a_data, float *b_data){
    /* shared memory array */
    __shared__ float a[16][3][5];
    /* fetch first planes */
    a[i][0][k] = a_data[i0m10];
    a[i][1][k] = a_data[i000];
    a[i][2][k] = a_data[i0p10];
    __syncthreads();
    /* compute */
    b_data[i000] =
        sf1*a[i][1][k] + sfd6*(a[im1][1][k] +
        a[ip1][1][k] + a[i][0][k] +
        a[i][2][k] + a[i][1][km1] + a[i][1][kp1])
    /* load next "j" 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

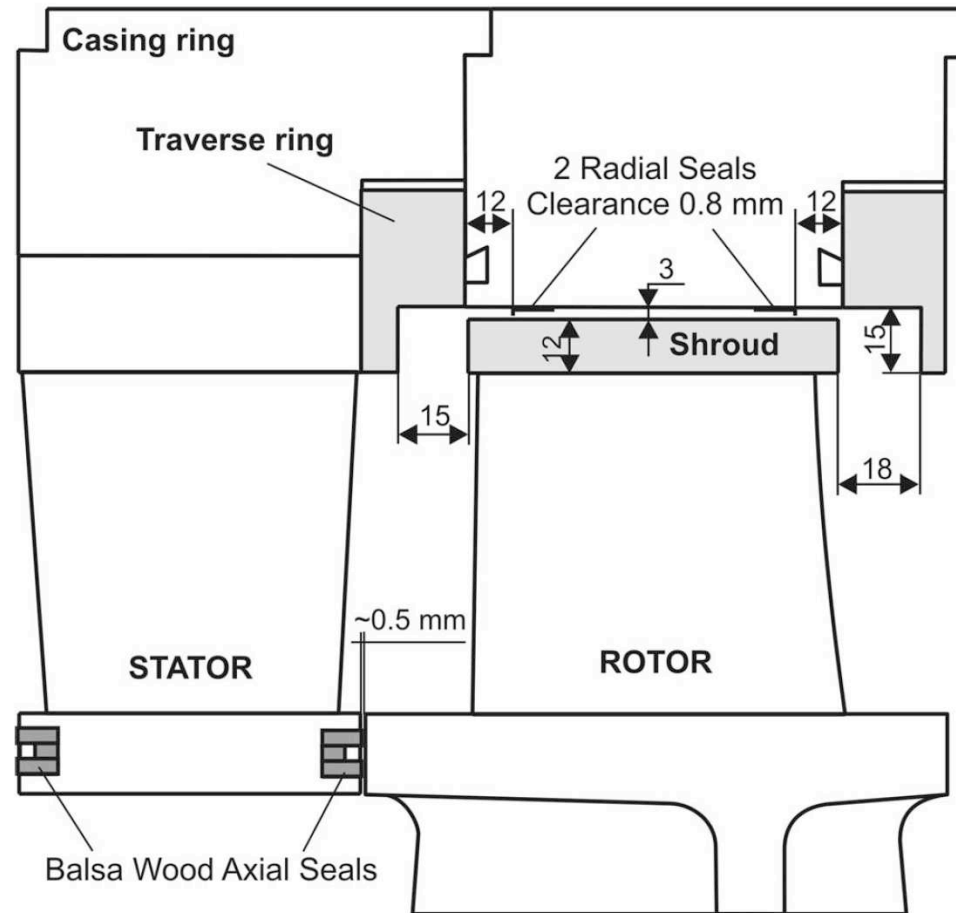


- 9 minutes on a Tesla S870 (4 GPUs)
- 12 hours on one 2.5GHz CPU core



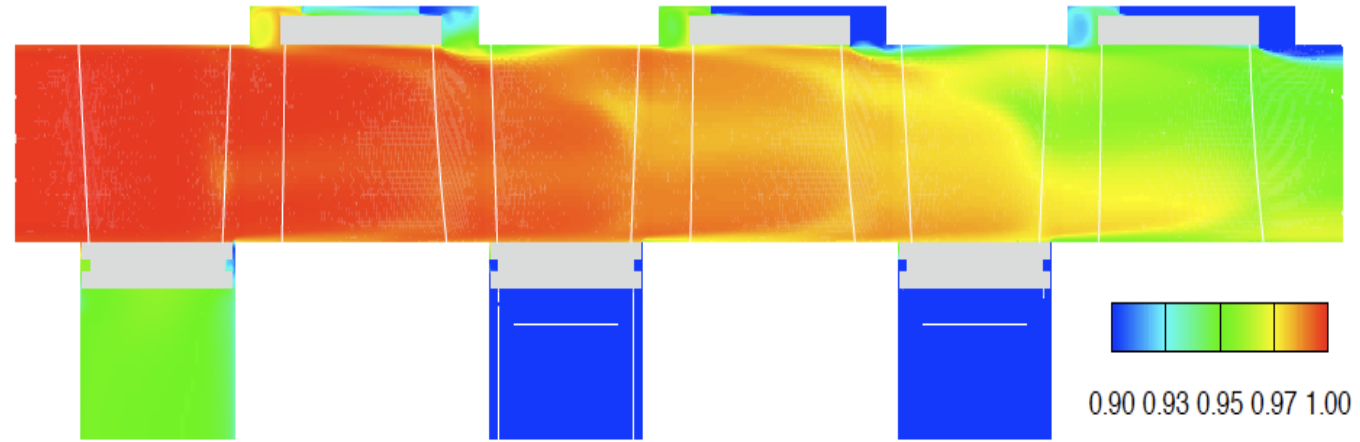


# Application to 3 stage turbine

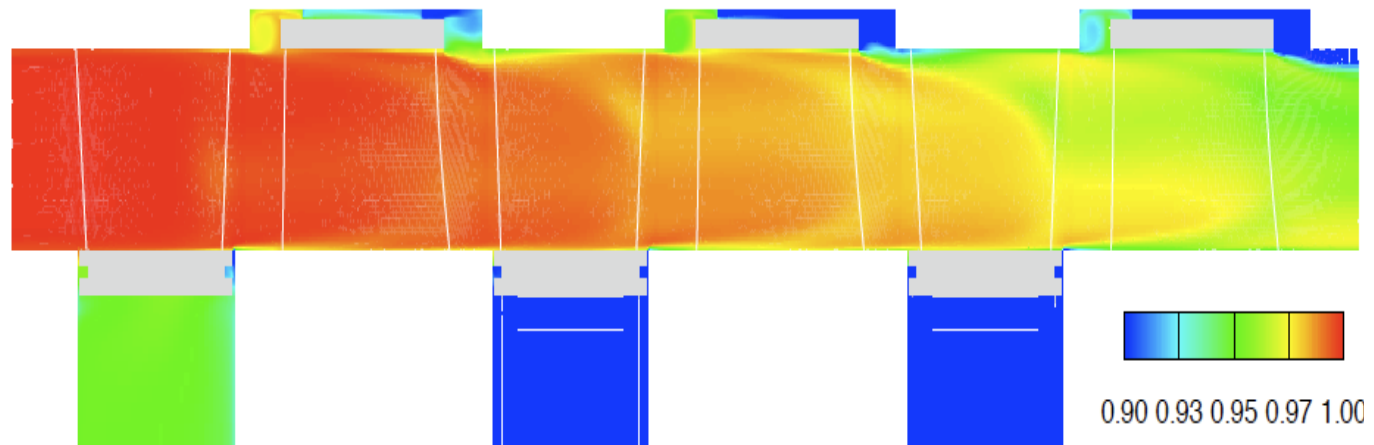


# FORTRAN & CUDA comparison

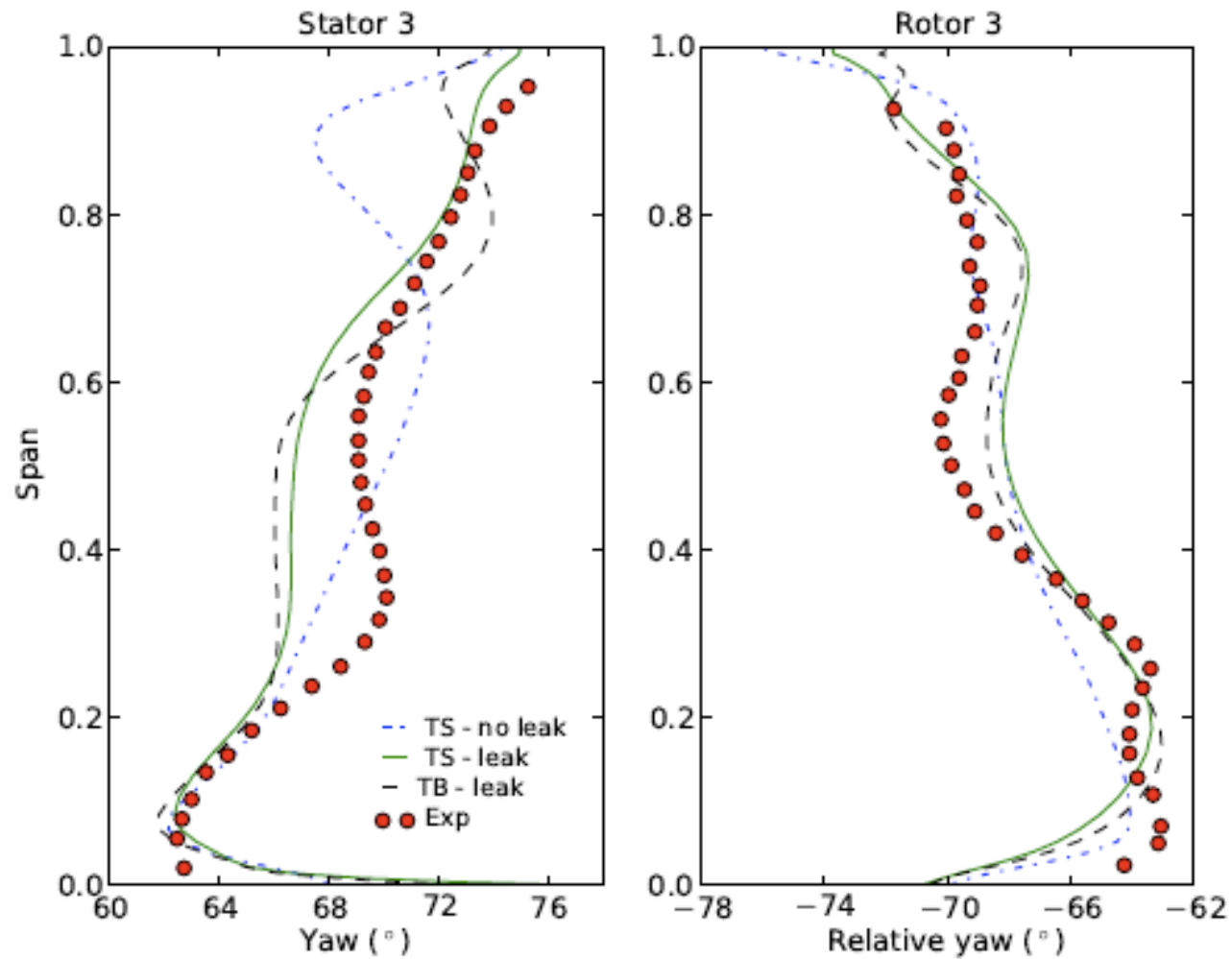
Fortran



CUDA



# Comparison to experimental data



# 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

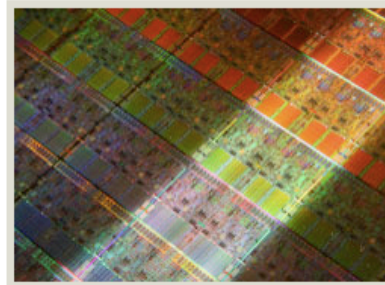
# More information

[www.many-core.group.cam.ac.uk](http://www.many-core.group.cam.ac.uk)

## many-core.group

 [University of Cambridge](#) > [many-core.group](#)

Many-core computing devices have large numbers of processors (cores) on a single chip. Such configurations are attractive because they can achieve a greater performance (calculations per second) for a given amount of electrical power than their single-core cousins. CPUs are heading down this route with dual-core and quad-core processors now commonplace. However, accelerator add-on cards or chips are also available today which have over 100 cores; of these, the graphics processing unit (GPU) is the most widespread.




**many-core.group** is a site where researchers at Cambridge University who are using many-core devices to accelerate their scientific applications can show their results and describe their experiences.


### Events


27 Feb 2009


**NVIDIA Personal SuperComputer Seminar, Cambridge**


### On this site:

 [GPGPU](#)

 [People](#)

 [Projects](#)

 [Events archive](#)

 [Contact](#)

# Lattice Boltzmann demo

