Acceleration of scientific computing using graphics hardware

Graham Pullan
Whittle Lab
Engineering Department, University of Cambridge

28 May 2008

I've added some notes that weren't on the original slides to help readers of the online pdf version.

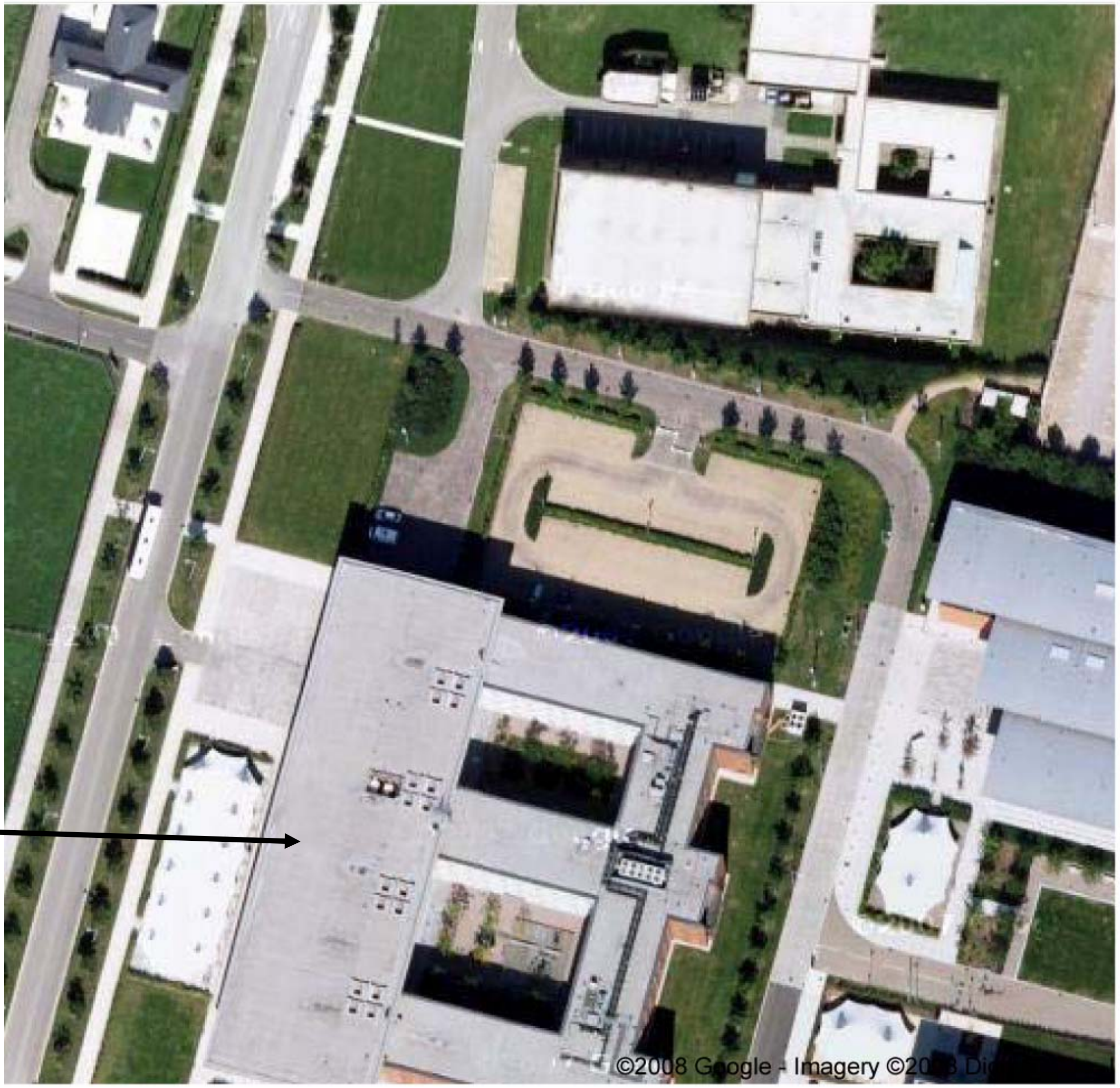
Coming up...

- Background
- CPUs and GPUs
- GPU programming models
- An example – CFD
- Alternative devices
- Conclusions

Part 1: Background

Whittle Lab

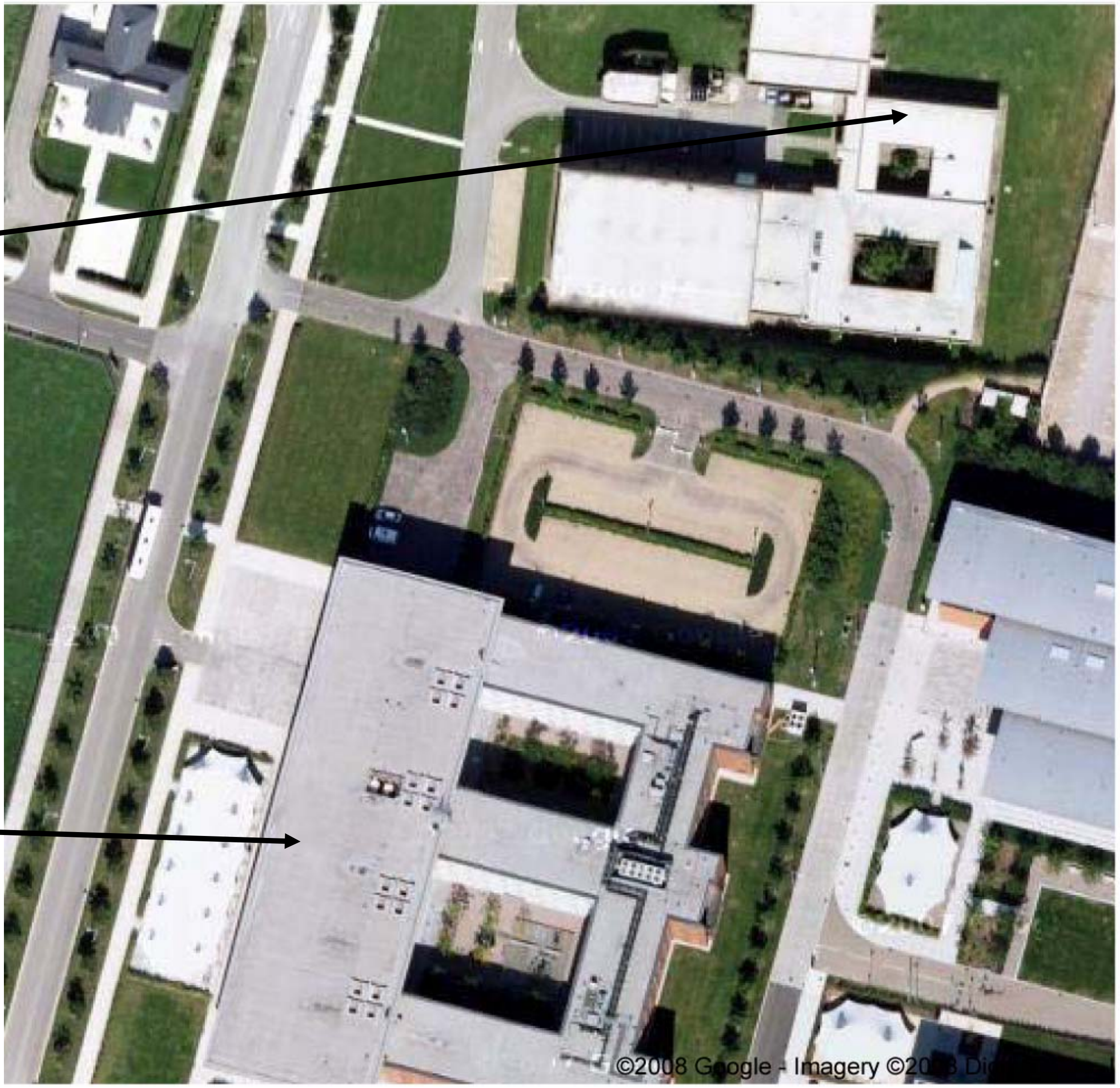
You are here



Whittle Lab

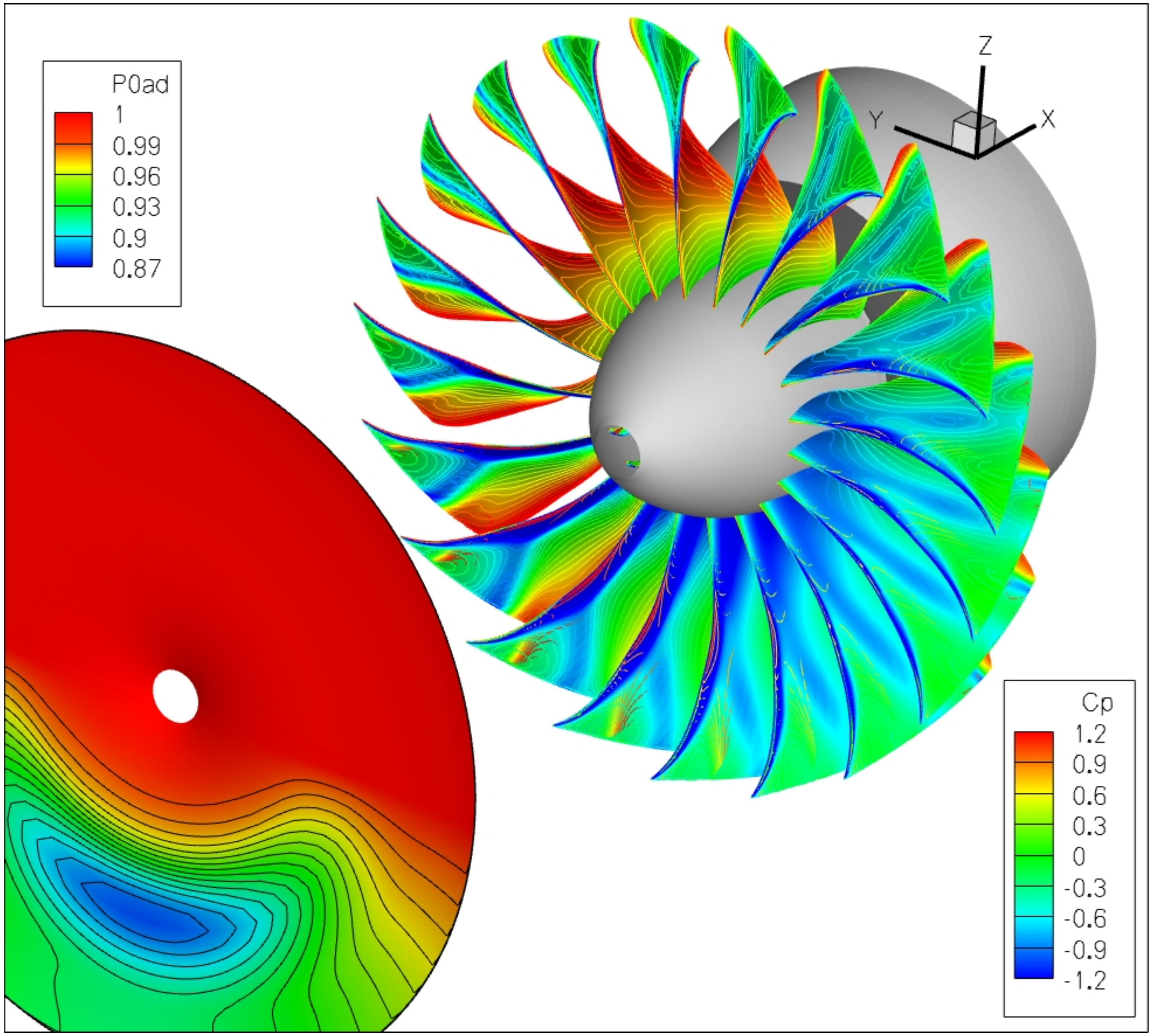
I work here

You are here



Turbomachinery

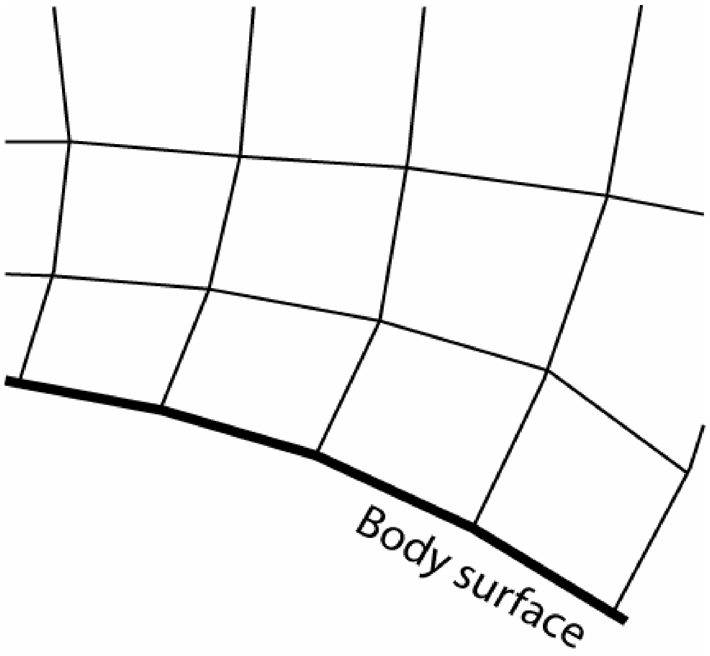
Engine calculation



Courtesy Vicente Jerez Fidalgo, Whittle Lab

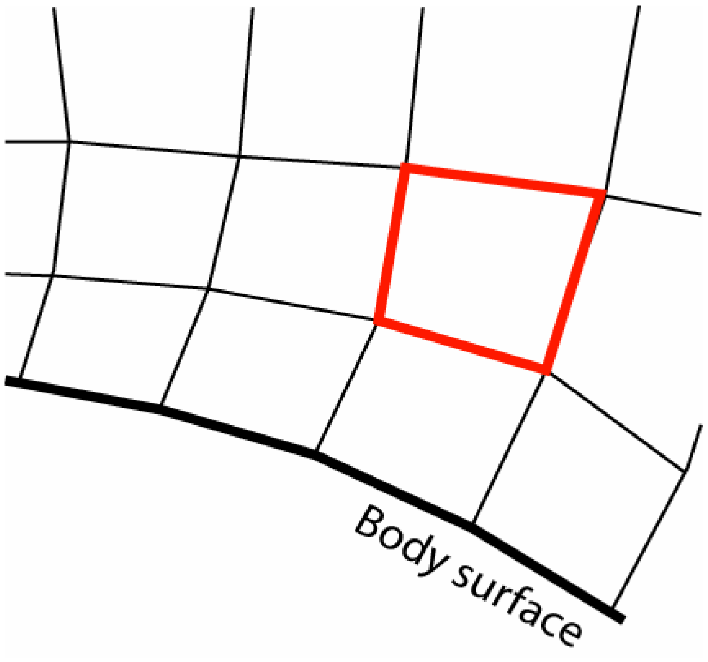
CFD basics

Body-fitted mesh



CFD basics

Body-fitted mesh



For each cell, conserve:

- mass
- momentum
- energy

and update flow properties

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

Graham’s coding experience:

- FORTRAN
- C
- MPI

Graham’s coding experience:

- FORTRAN
- C
- MPI

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.”

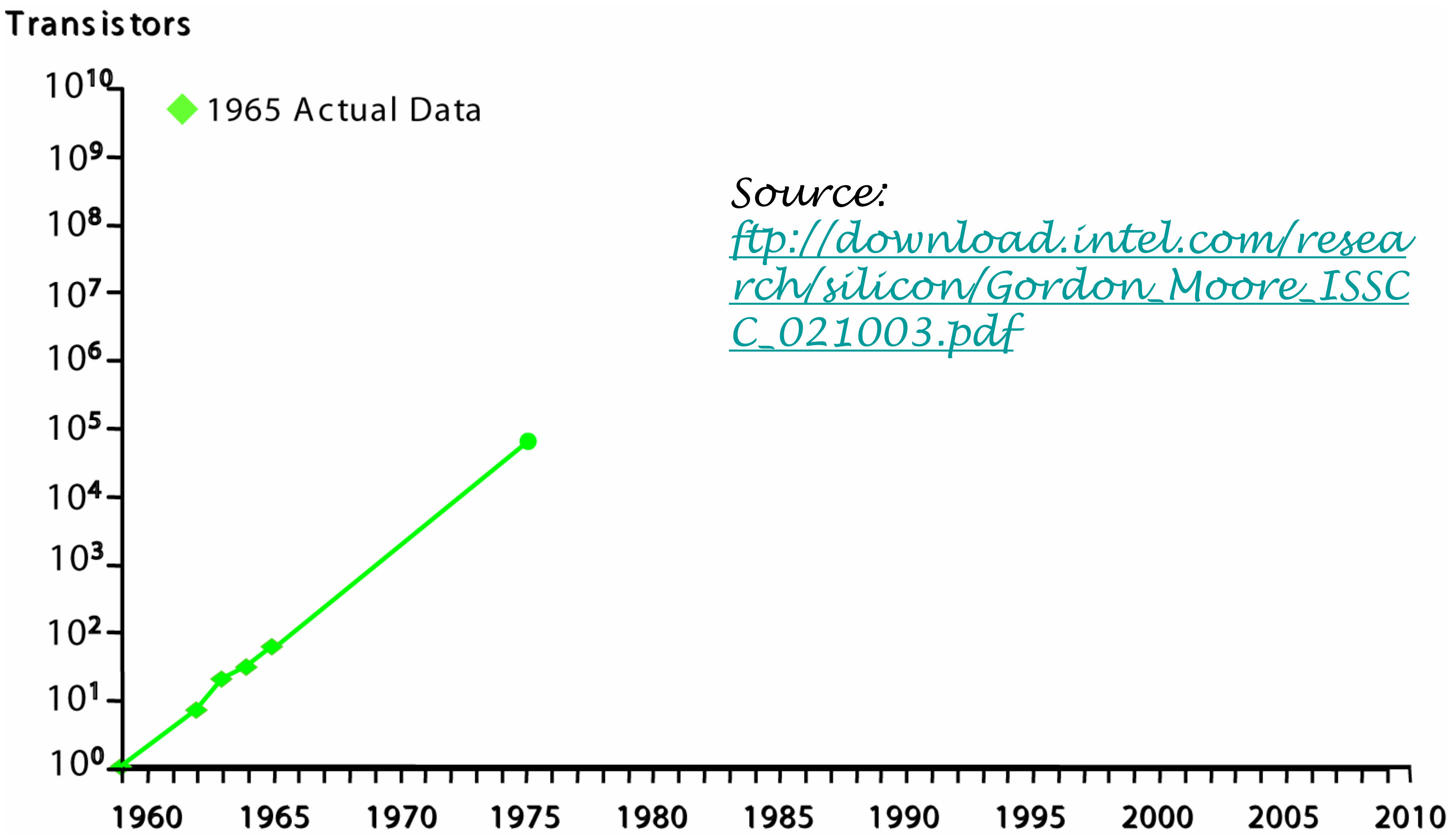
Gordon Moore (Intel), 1965

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]

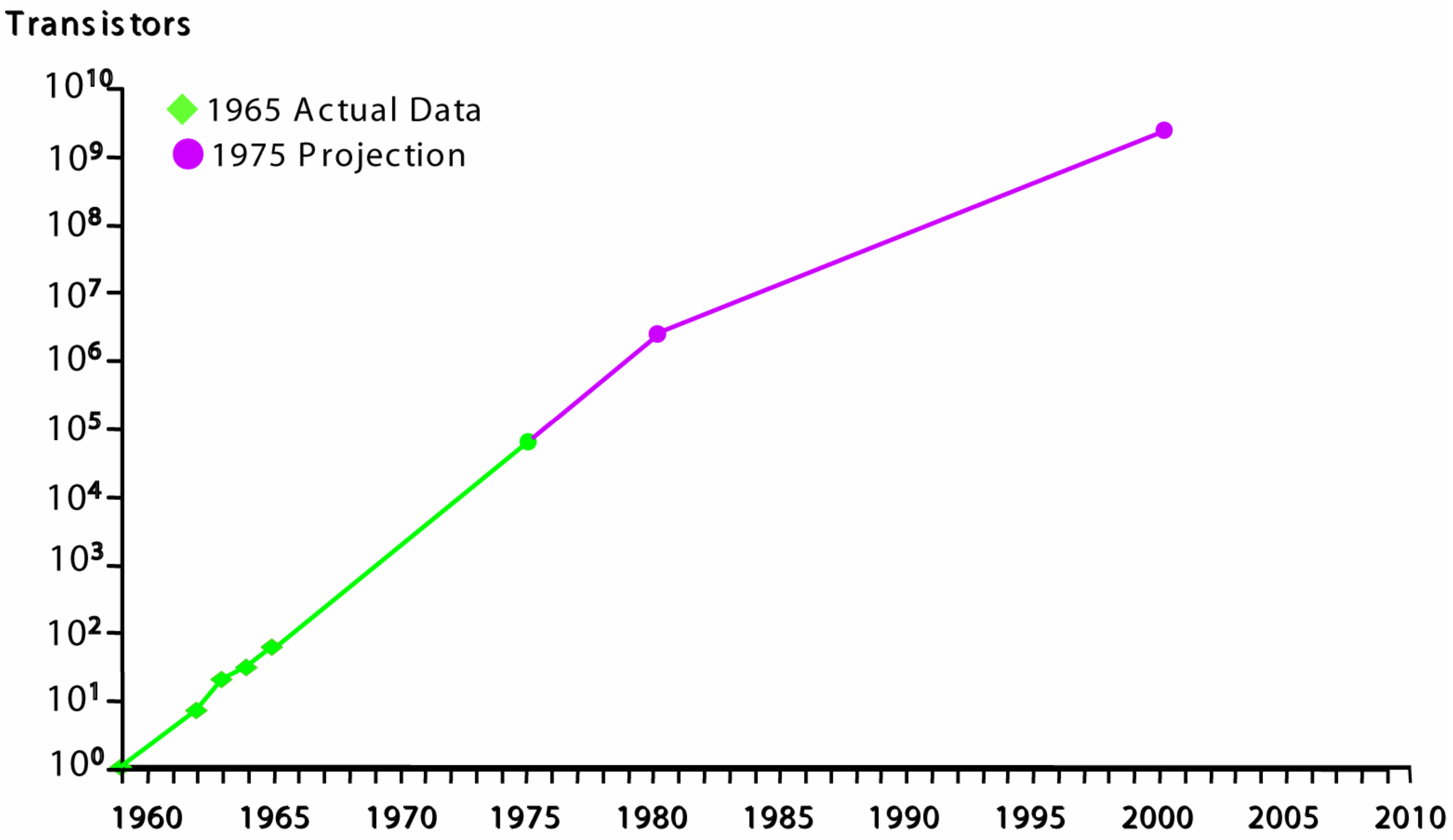
Was Moore right?



Source:
ftp://download.intel.com/research/silicon/Gordon_Moore_ISSCC_021003.pdf

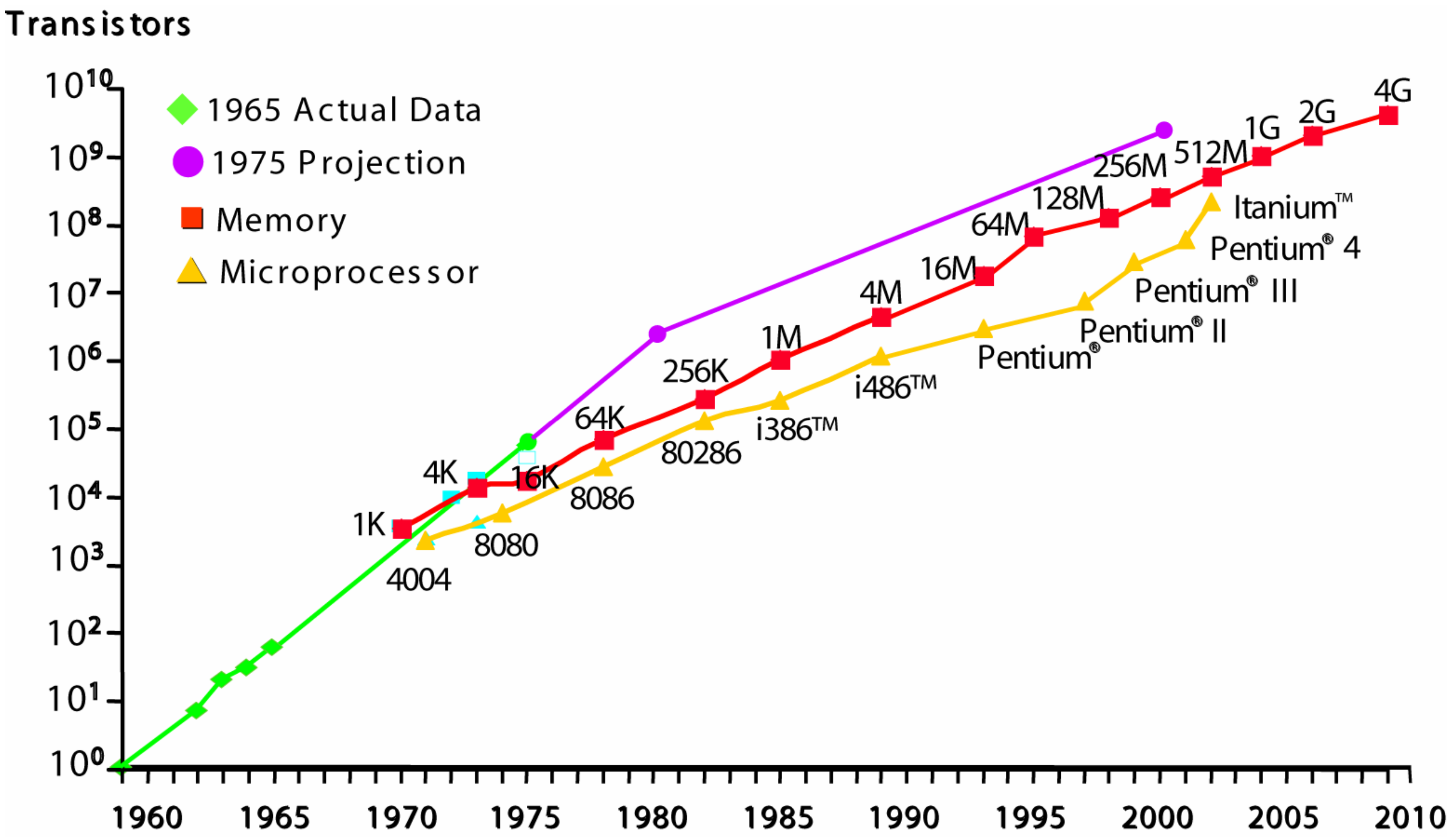
Source: Intel

Was Moore right?



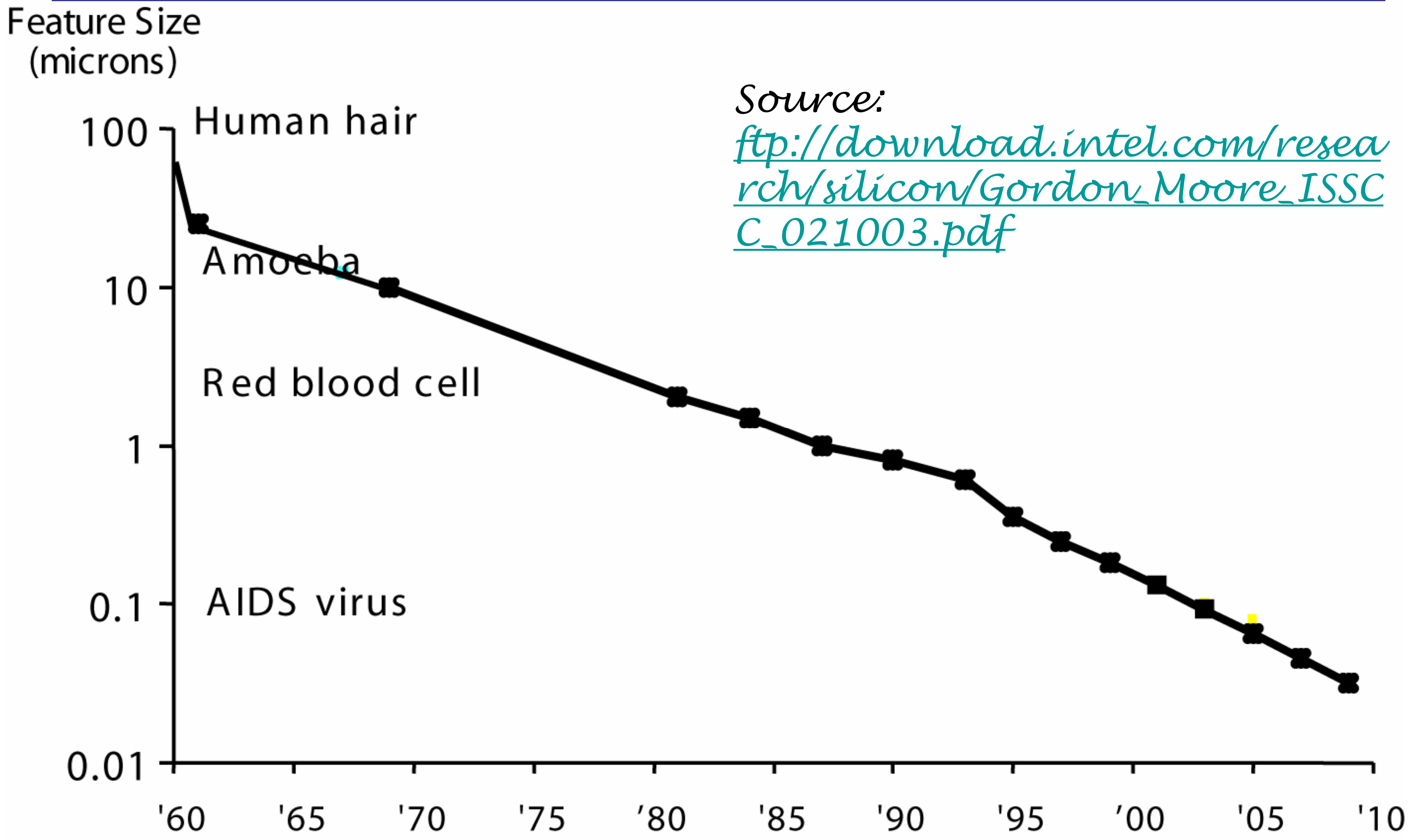
Source: Intel

Was Moore right?



Source: Intel

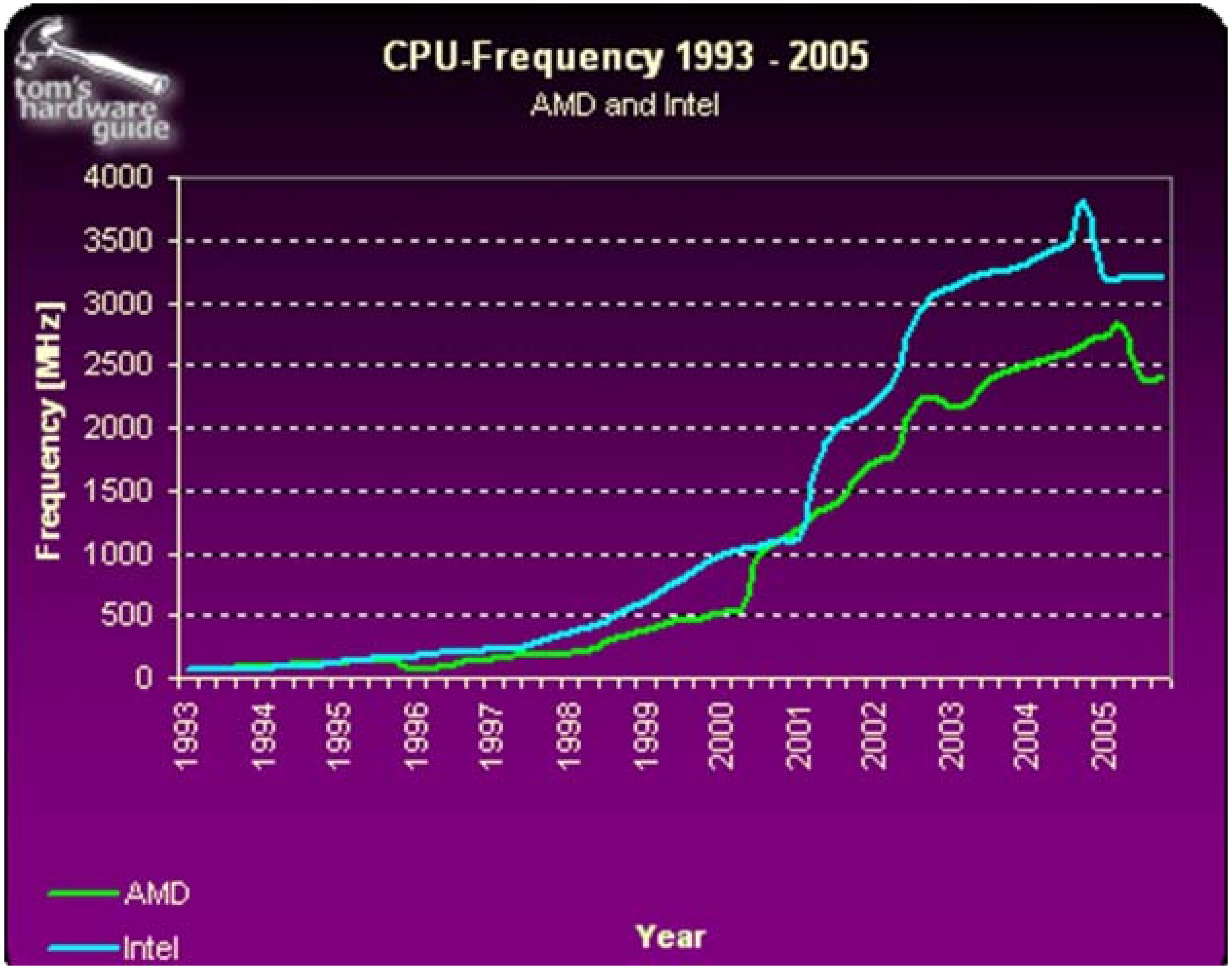
Feature size



Source:
ftp://download.intel.com/research/silicon/Gordon_Moore_ISSCC_021003.pdf

Source: Intel

Clock speed



Source: *http://www.tomshardware.com/2005/11/21/the_mother_of_all_cpu_charts_2005/index.html*

Power – the Clock speed limiter?

- 1 GHz CPU requires ≈ 25 W
- 3 GHz CPU requires ≈ 100 W

Power – the Clock speed limiter?

- 1 GHz CPU requires ≈ 25 W
- 3 GHz CPU requires ≈ 100 W

“*The total of electricity consumed by major search engines in 2006 approaches 5 GW.*” – Wired / AMD

Source: http://www.hotchips.org/hc19/docs/keynote2.pdf

What to do with all these transistors?

Parallel computing

Multi-core chips are either:

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

or

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

Today's commodity MIMD chips: CPUs

Intel Core 2 Quad

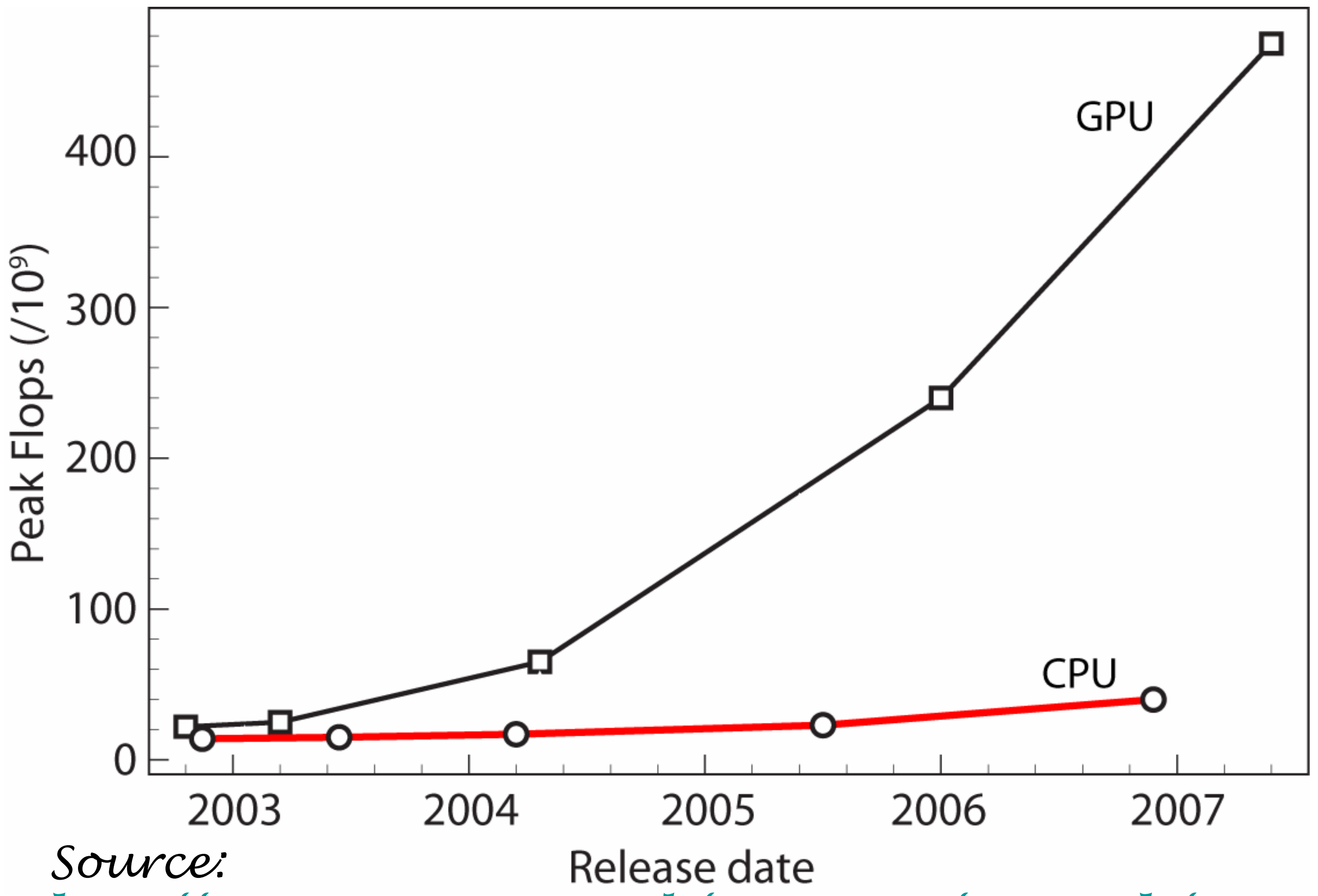
- 4 cores
- 2.4 GHz
- 65nm features
- 582 million transistors
- 8MB on chip memory

Today’s commodity SIMD chips: GPUs

NVIDIA 8800 GTX

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

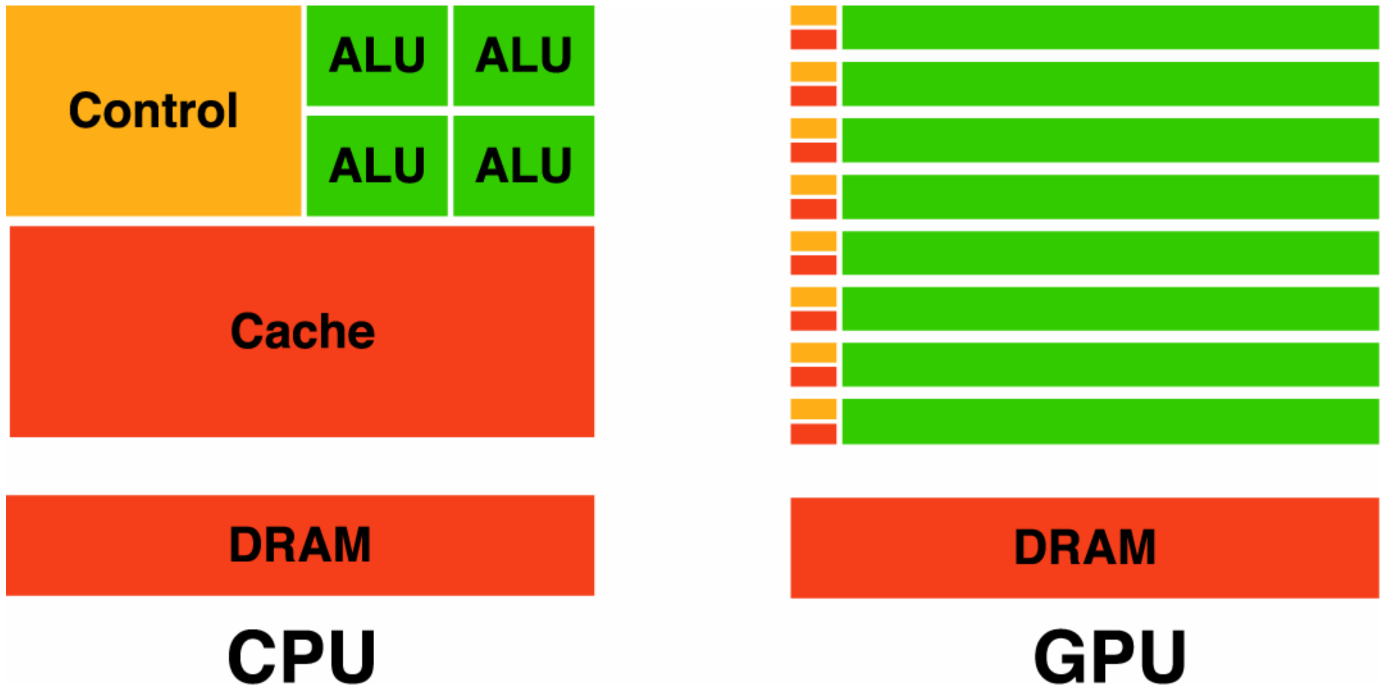
CPUs vs GPUs



Source: *http://www.eng.cam.ac.uk/~gp10006/research/Brandvik_Pullan_2008a_DRAFT.pdf*

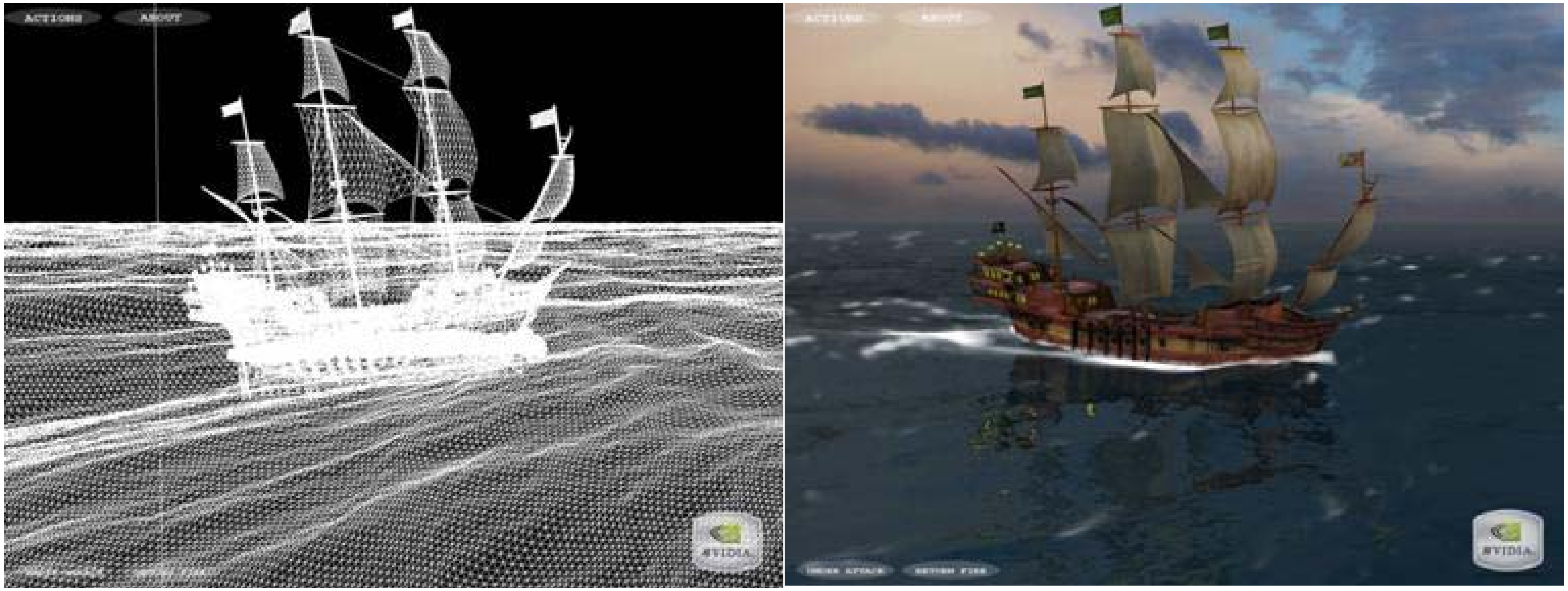
CPUs vs GPUs

Transistor usage:



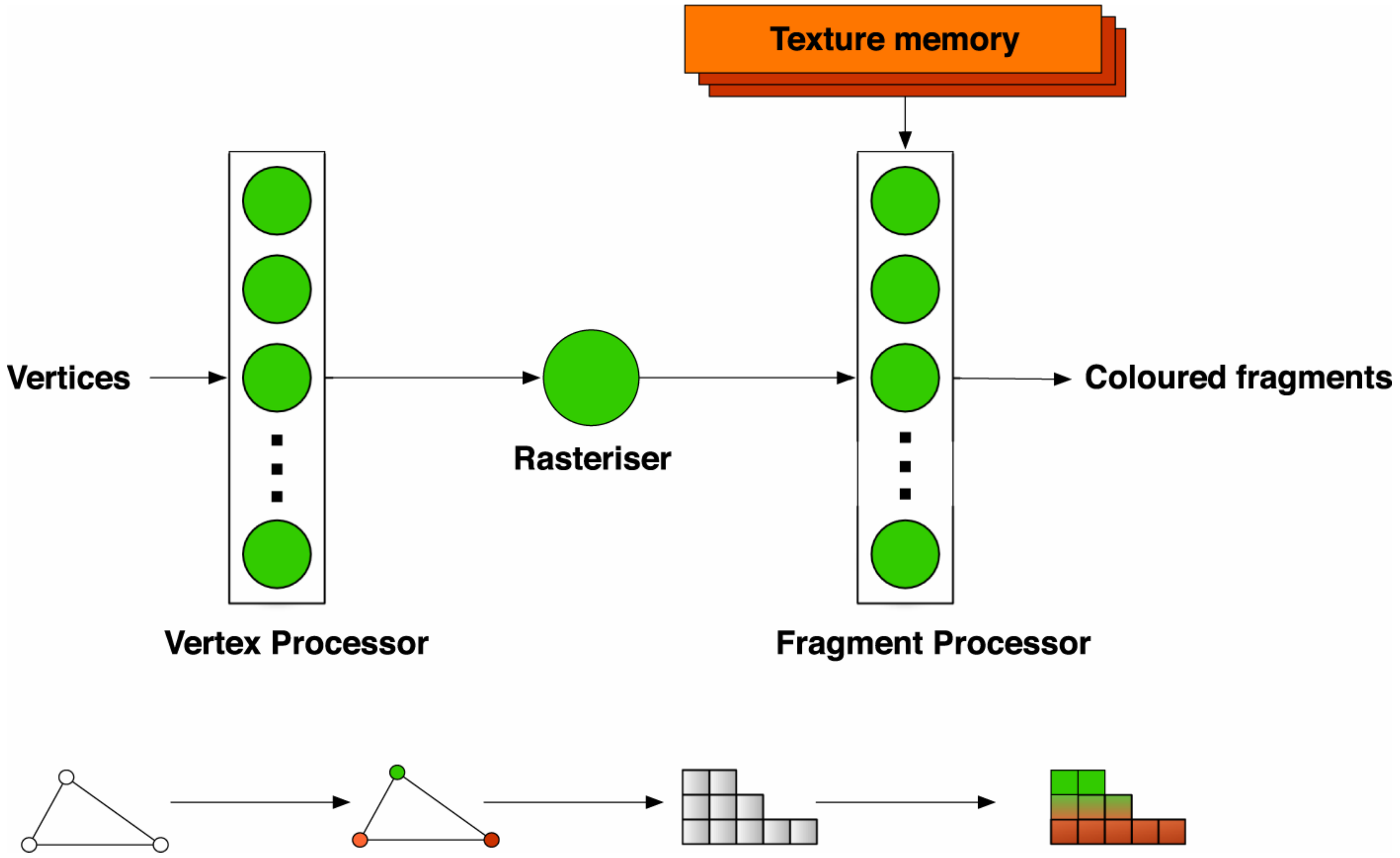
Source: NVIDIA CUDA SDK documentation

Graphics pipeline



Source:
ftp://download.nvidia.com/developer/presentations/2004/Perfect_Kitchen_Art/English_Evolution_of_GPUs.pdf

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!

Part 3: Programming methods

3 Generations of GPGPU (Owens, 2008)

3 Generations of GPGPU (Owens, 2008)

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

3 Generations of GPGPU (Owens, 2008)

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

3 Generations of GPGPU (Owens, 2008)

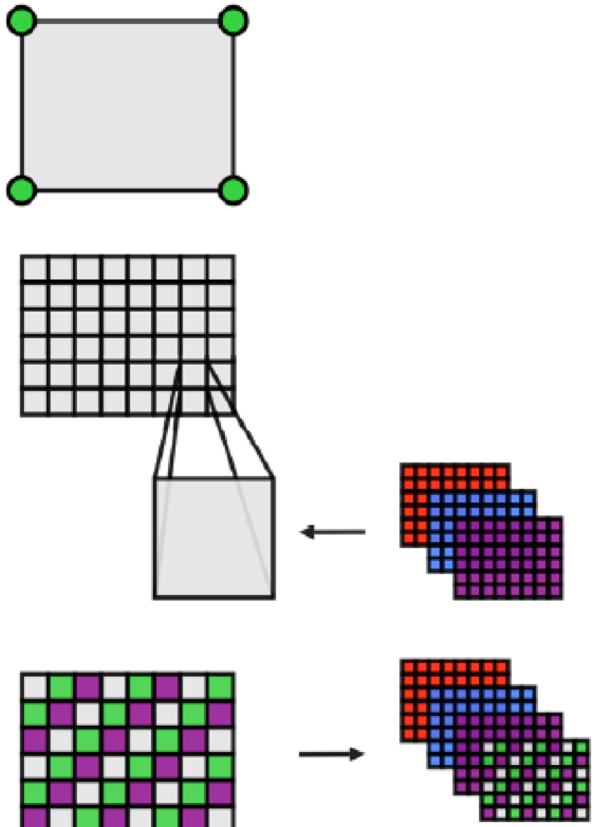
- Making it work at all:
  - Primitive functionality and tools (graphics APIs)
  - Comparisons with CPU not rigorous
- Making it work better:
  - Easier to use (higher level APIs)
  - Understanding of how best to do it
- Doing it right:
  - Stable, portable, modular building blocks

Source:

http://www.ece.ucdavis.edu/~jowens/talks/intel-santaclara-070420.pdf

GPU – Programming for graphics

Courtesy, John Owens, UC Davis



Application specifies geometry – GPU rasterizes

Each fragment is shaded (SIMD)

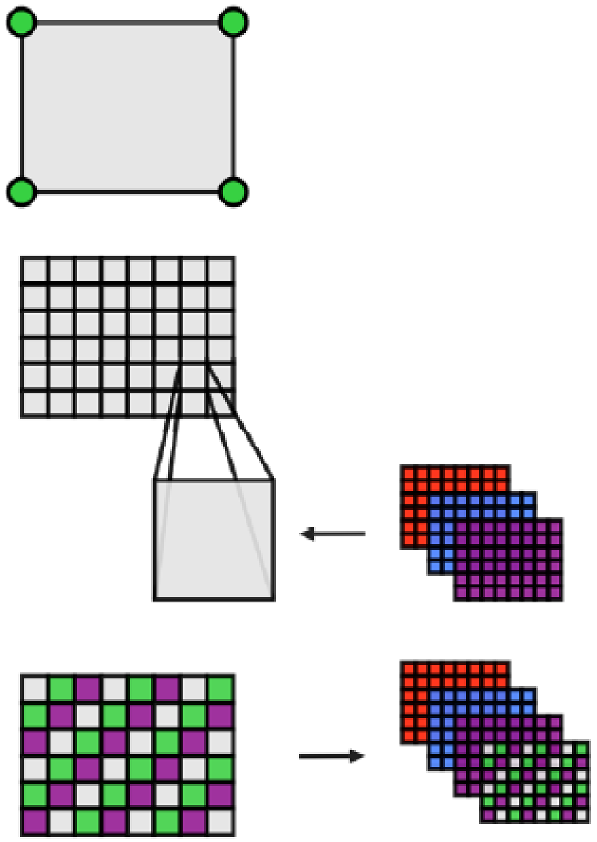
Shading can use values from memory (textures)

Image can be stored for re-use

Source:

http://www.ece.ucdavis.edu/~jowens/talks/intel-santaclara-070420.pdf

GPGPU programming ("old-school")



Draw a quad

Run a SIMD program over each fragment

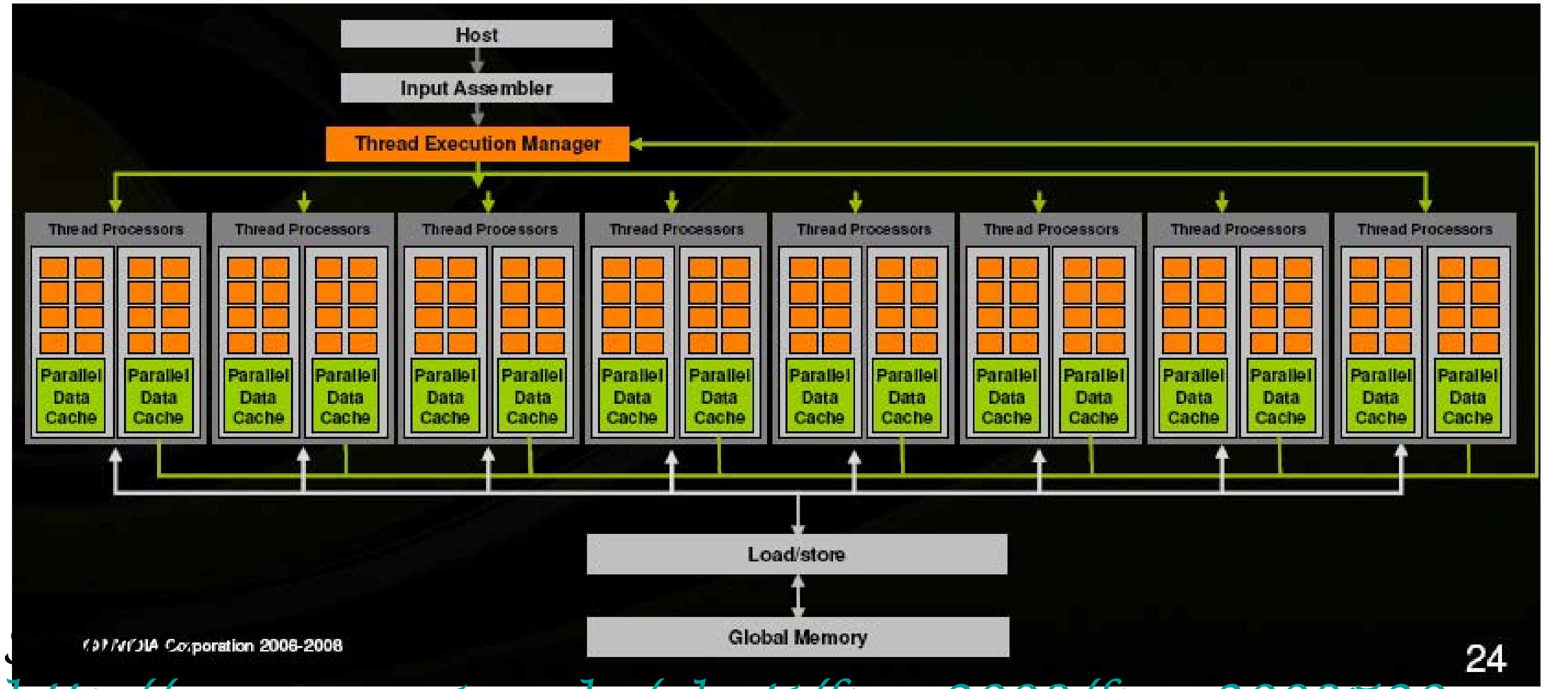
Gather is permitted from texture memory

Resulting buffer can be stored for re-use

Courtesy, John Owens, UC Davis

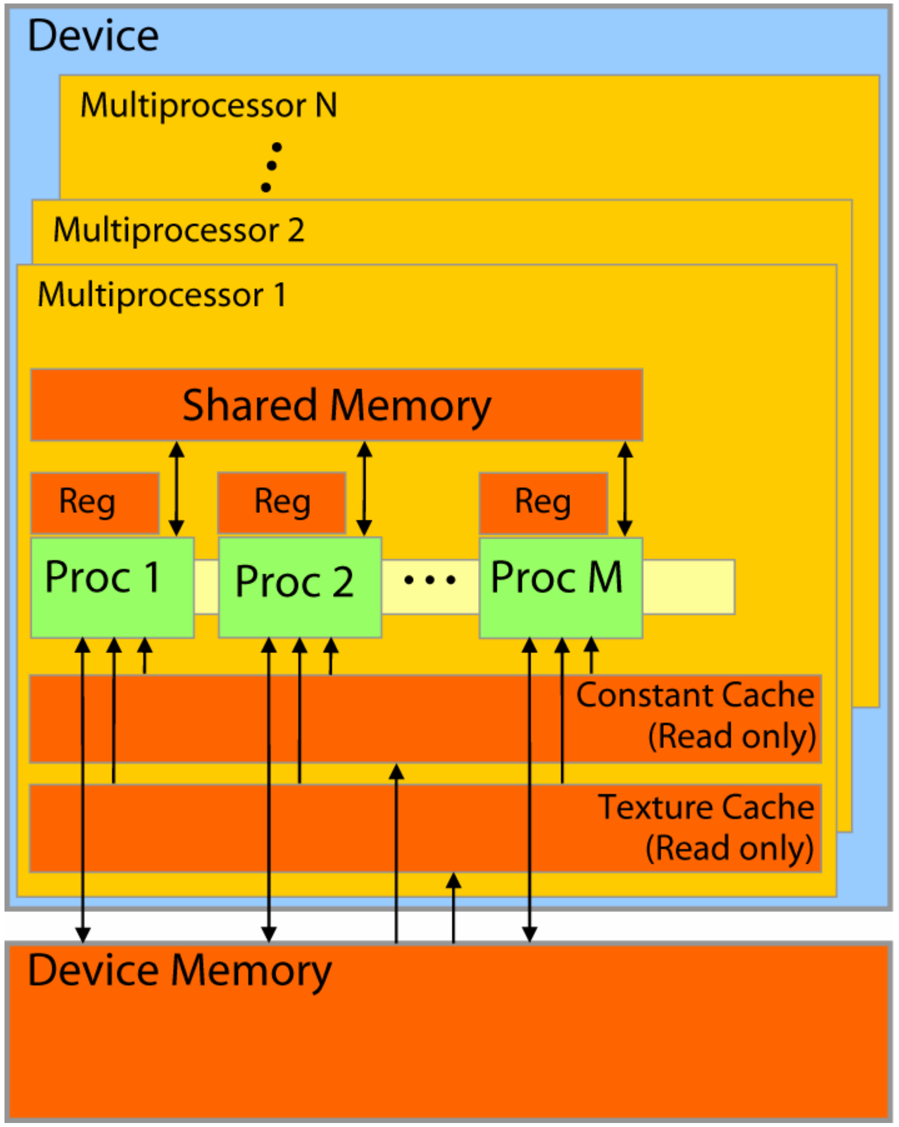
NVIDIA G80 hardware implementation

- Vertex/fragment processors replaced by Unified Shaders
- Now view GPU as massively parallel co-processor
- Set of (16) SIMD MultiProcessors (8 cores)



S
http://www.ece.wisc.edu/~kati/fpga2008/fpga2008%20workshop%20-%2006%20NVIDIA%20-%20Kirk.pdf

NVIDIA G80 hardware implementation



Divide 128 cores into

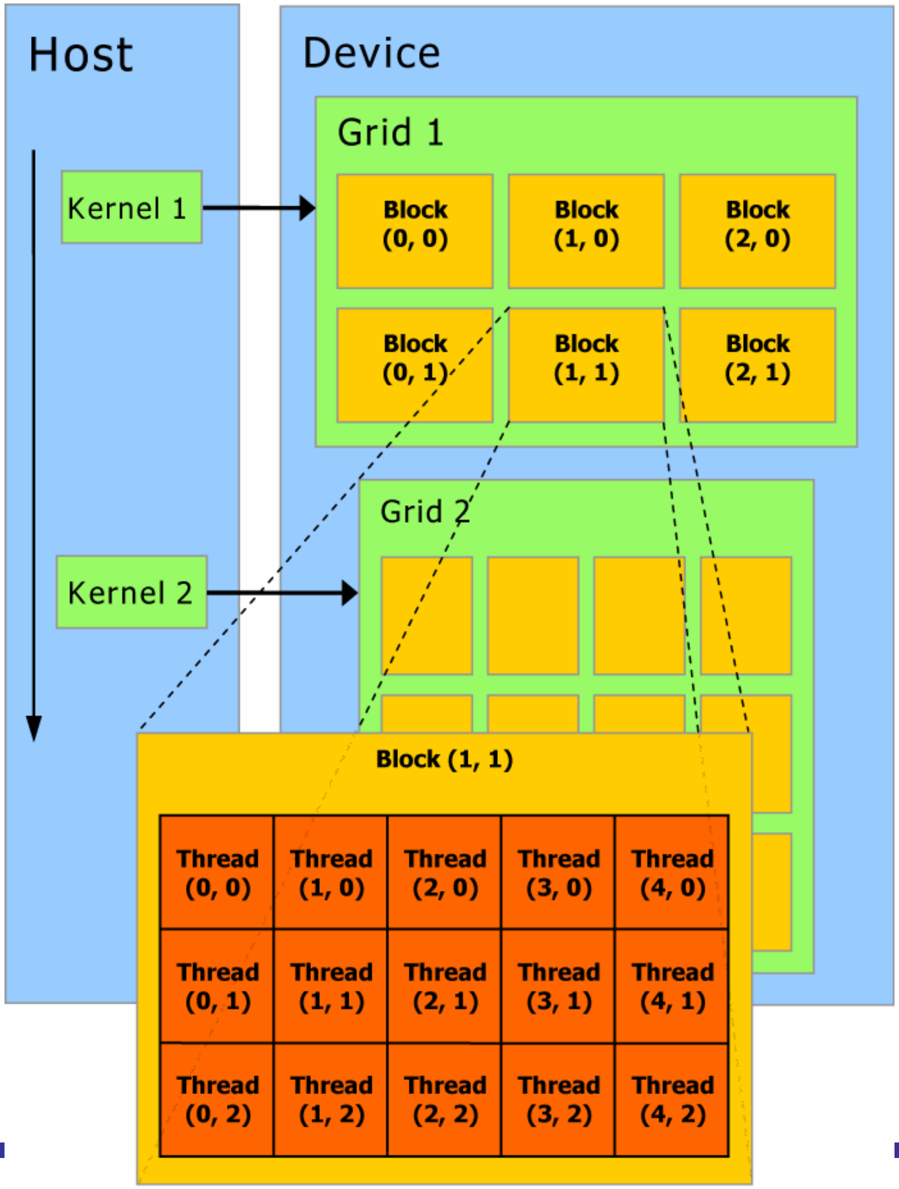
16 Multiprocessors (MPs)

- 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

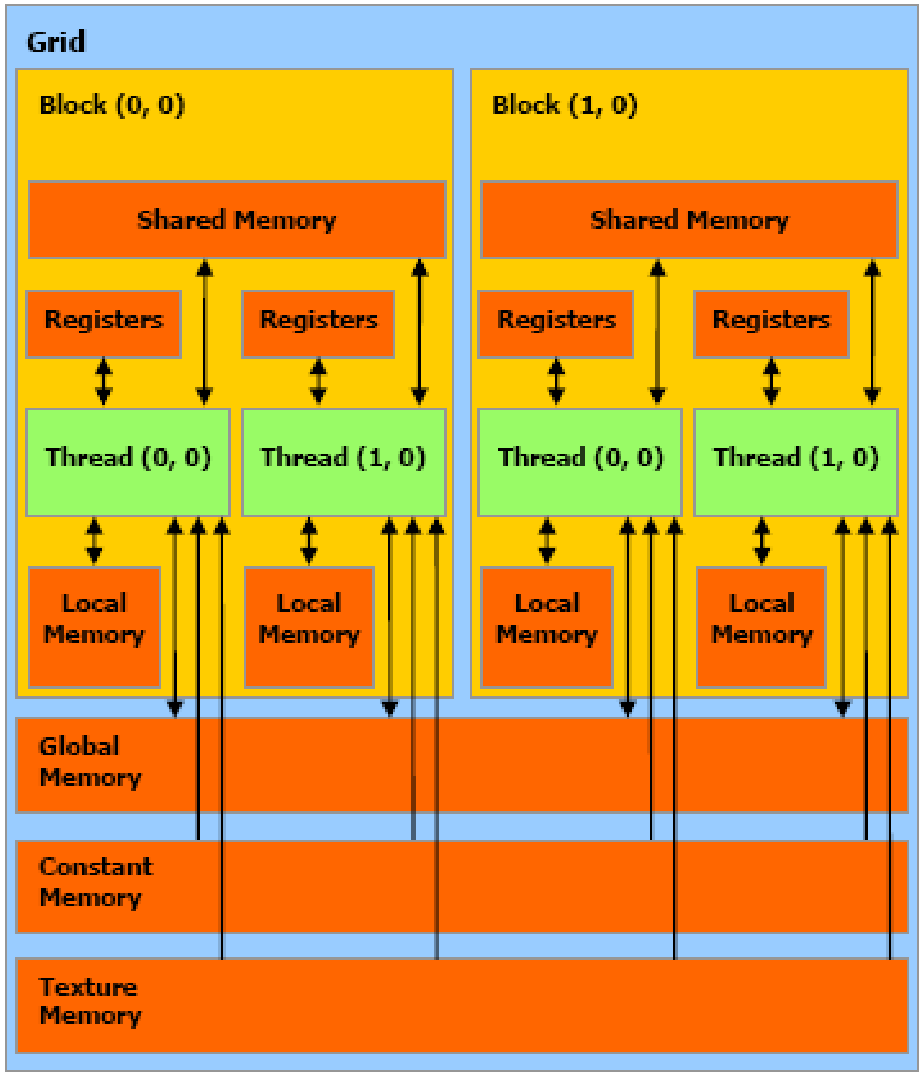
Kernels, grids, blocks and threads

- 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

Kernels, grids, blocks and threads



CUDA implementation

- CUDA implemented as extensions to C

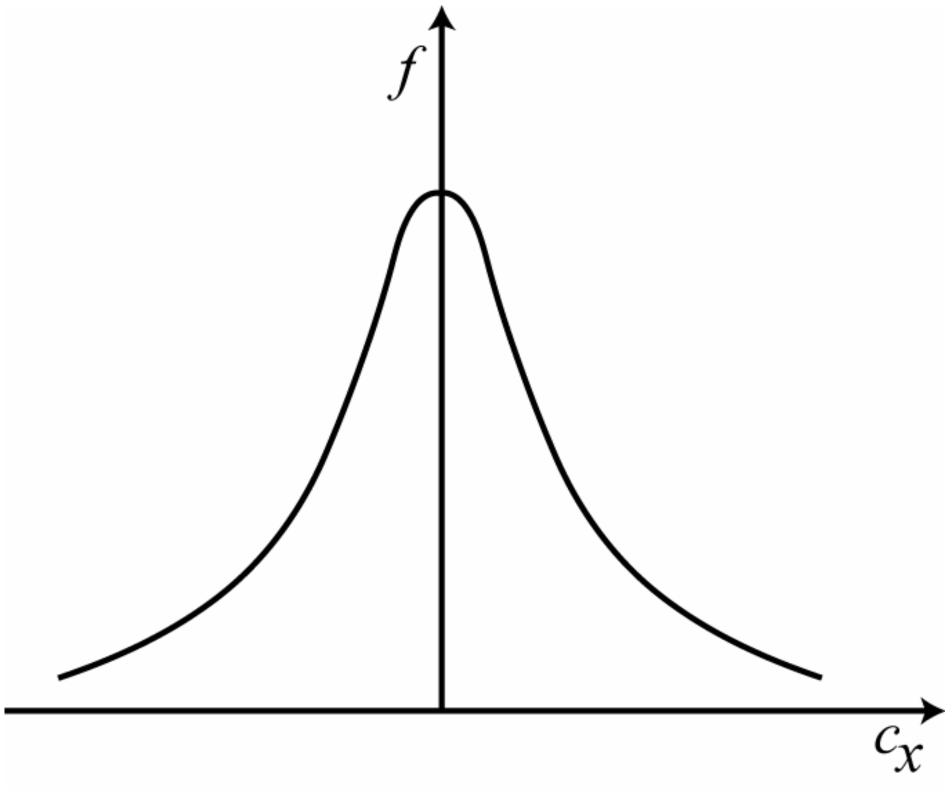
- CUDA programs:
  - explicitly manage host and device memory:
    - allocation
    - transfers
  - set thread blocks and grid
  - launch kernels
  - are compiled with the CUDA **nvcc** compiler

Part 4: An example – CFD

Distribution function

$$f = f(\mathbf{c}, \mathbf{x}, t)$$

c is **microscopic** velocity



$$\rho = \int f \; d\mathbf{c}$$

$$\rho\mathbf{u} = \int \mathbf{c}f \; d\mathbf{c}$$

u is **macroscopic** velocity

Boltzmann equation

The evolution of f:

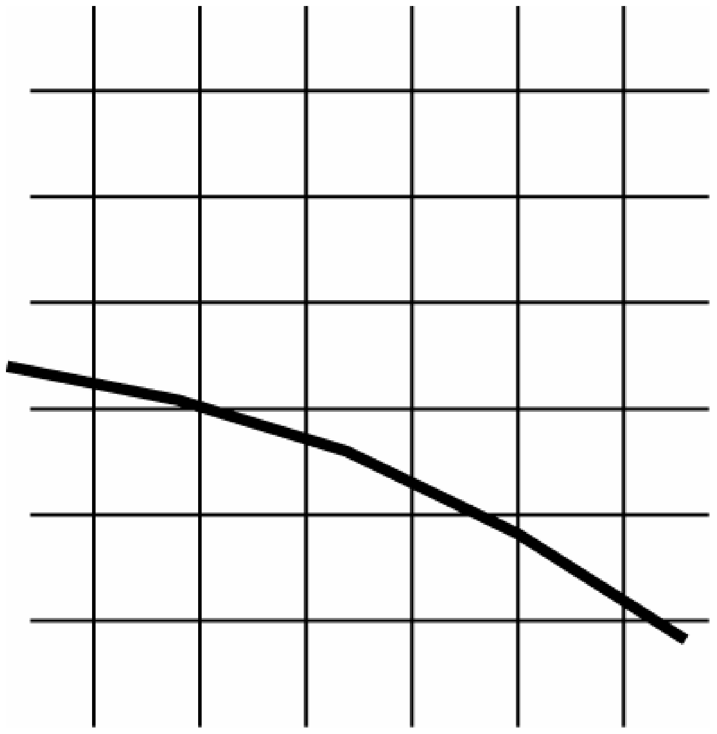
$$\frac{\partial f}{\partial t} + \mathbf{u} \cdot \nabla f = \left.\frac{\partial f}{\partial t}\right|_{collisions}$$

Major simplification:

$$\frac{\partial f}{\partial t} + \mathbf{u} \cdot \nabla f = -\frac{1}{\tau}(f - f^{eq})$$

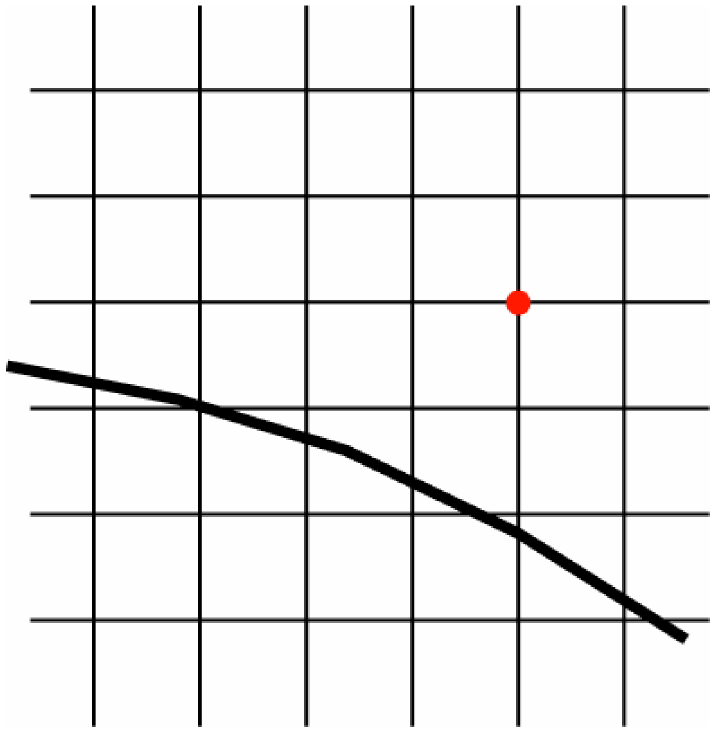
Lattice Boltzmann Method

Uniform mesh (lattice)

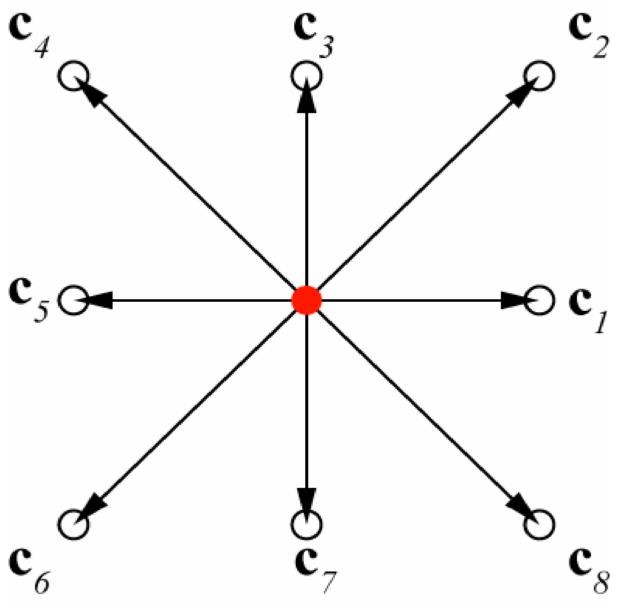


Lattice Boltzmann Method

Uniform mesh (lattice)



Restrict microscopic velocities to a finite set:



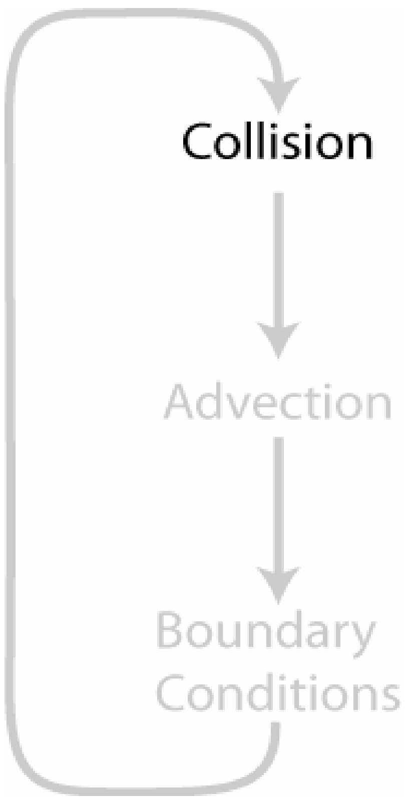
$$\rho = \sum_{\alpha} f_{\alpha} \qquad \rho\mathbf{u} = \sum_{\alpha} f_{\alpha}\mathbf{c}_{\alpha}$$

Macroscopic flow

For 2D, 9 velocities recover

- Isothermal, incompressible Navier-Stokes eqns
- With viscosity: $\nu = \left(\tau - \frac{1}{2} \right) \frac{\Delta x^2}{\Delta t}$

Solution procedure



1. Evaluate macroscopic properties:

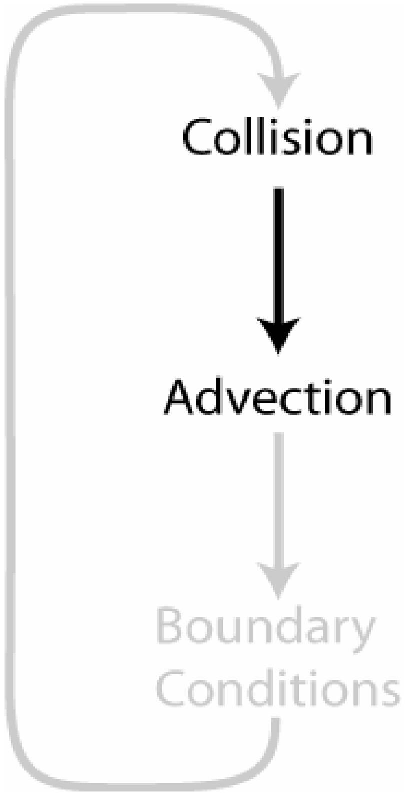
$$\rho = \sum_\alpha f_\alpha \qquad \rho\mathbf{u} = \sum_\alpha f_\alpha \mathbf{c}_\alpha$$

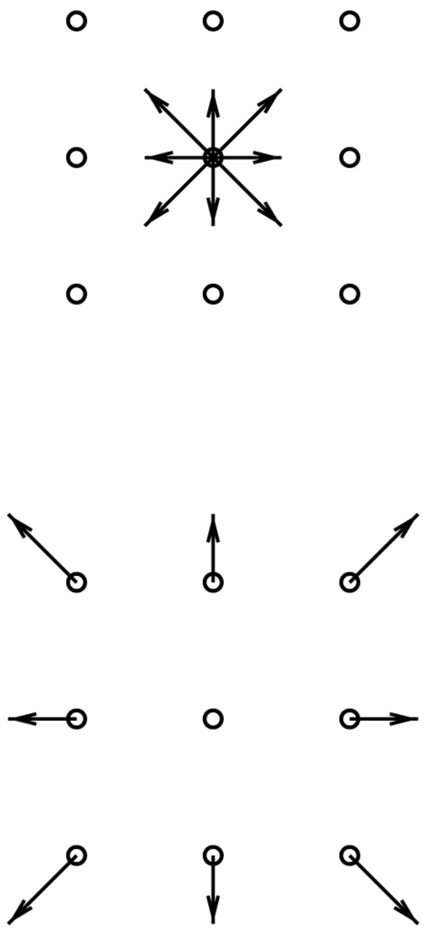
2. Evaluate $f_\alpha^{eq}(\rho, \mathbf{u})$

3. Find

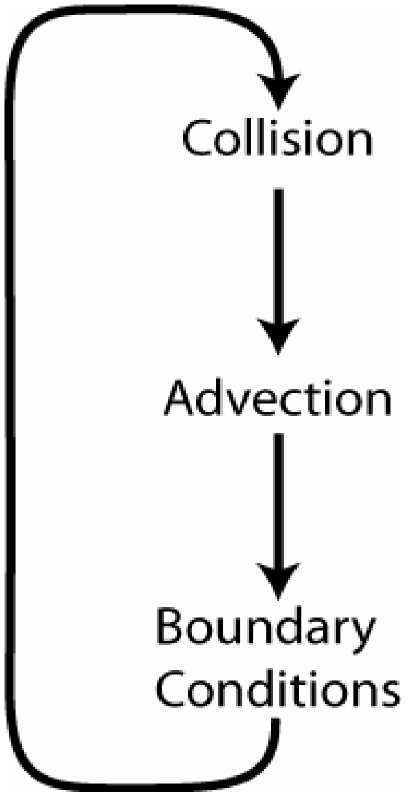
$$f_\alpha^* = f_\alpha - \frac{1}{\tau}\left(f_\alpha - f_\alpha^{eq}\right)$$

Solution procedure





Solution procedure



Simple prescriptions at boundary nodes

CPU code: main.c

```
/* Memory allocation */
f0 = (float *)malloc(ni*nj*sizeof(float));
 ...

/* Main loop */
Stream (...args...);
Apply_BCs (...args...);
Collide (...args...);
```

GPU code: main.cu

```
/* allocate memory on host */
f0 = (float *)malloc(ni*nj*sizeof(float));

/* allocate memory on device */
cudaMallocPitch((void **)&f0_data, &pitch,
                sizeof(float)*ni, nj);

cudaMallocArray(&f0_array, &desc, ni, nj);

/* Main loop */
Stream (...args...);
Apply_BCs (...args...);
Collide (...args...);
```

CPU code – collide.c

```
for (j=0; j<nj; j++) {
  for (i=0; i<ni; i++) {
      i2d = I2D(ni,i,j);
/* Flow properties */
      density = ...function of f’s ...
      vel_x = ...   “
      vel_y = ...   “
/* Equilibrium f’s */
      f0eq = ... function of density, vel_x, vel_y ...
      f1eq = ...     “
/* Collisions */
      f0[i2d] = rtau1 * f0[i2d] + rtau * f0eq;
      f1[i2d] = rtau1 * f1[i2d] + rtau * f1eq;
      ...
  }
}
```

GPU code – collide.cu – kernel wrapper

```
void collide( ... args ...)
{
/* Set thread blocks and grid */
    dim3 grid = dim3(ni/TILE_I, nj/TILE_J);
    dim3 block = dim3(TILE_I, TILE_J);

/* Launch kernel */
    collide_kernel<<<grid, block>>>(... args ...);

}
```

GPU code – collide.cu - kernel

```
/* Evaluate indices */
i = blockIdx.x*TILE_I + threadIdx.x;
j = blockIdx.y*TILE_J + threadIdx.y;
i2d = i + j*pitch/sizeof(float);
/* Read from device global memory */
f0now = f0_data[i2d];
f1now = f1_data[i2d];

/* Calc flow, feq, collide, as CPU code */

/* Write to device global memory */
f0_data[i2d] = rtau1 * f0now + rtau * f0eq;
f1_data[i2d] = rtau1 * f1now + rtau * f1eq;
```

GPU code – stream.cu – kernel wrapper

```
void stream( ... args ...)
{
/* Copy linear memory to CUDA array */
  cudaMemcpy2DToArray(f1_array, 0, 0,
      (void *)f1_data, pitch,sizeof(float)*ni, nj,
       cudaMemcpyDeviceToDevice);
/* Make CUDA array a texture */
  f1_tex.filterMode = cudaFilterModePoint;
  cudaBindTextureToArray(f1_tex, f1_array));
/* Set threads and launch kernel */
  dim3 grid = dim3(ni/TILE_I, nj/TILE_J);
  dim3 block = dim3(TILE_I, TILE_J);
  stream_kernel<<<grid, block>>>(... args ...);
}
```

GPU code – stream.cu – kernel

```
/* indices */

i = blockIdx.x*TILE_I + threadIdx.x;
j = blockIdx.y*TILE_J + threadIdx.y;
i2d = i + j*pitch/sizeof(float);

/* stream using texture fetches */
f1_data[i2d] = tex2D(f1_tex, (i-1), j);
f2_data[i2d] = tex2D(f2_tex, i, (j-1));
...
```

CPU / GPU demo

Results

- 2D Lattice Boltzmann code: 15x speedup GPU vs CPU
- Real CFD is more complex:
  - more kernels
  - 3D
- To improve performance, make use of shared memory

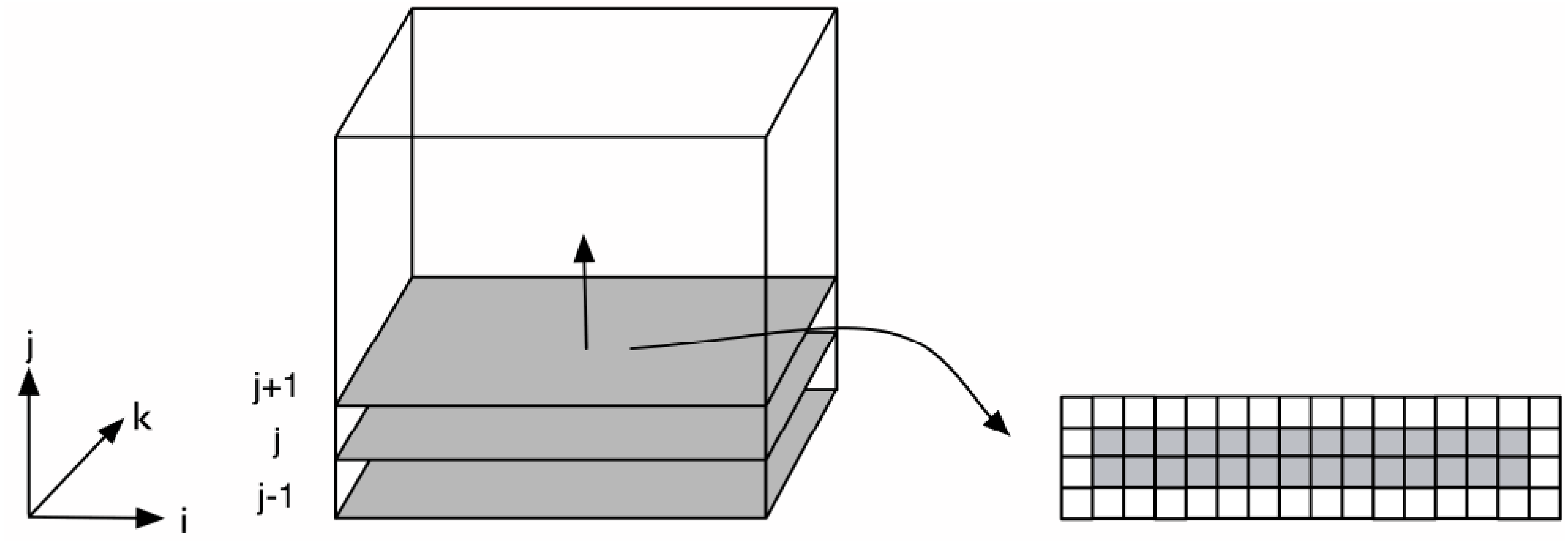
3D stencil operations

- Most CFD operations use nearest neighbour lookups (***stencil*** operations)

- e.g. 7 point stencil: centre point + 6 nearest neighbours

- Load data into shared memory
- Perform stencil ops
- Export results to device global memory
- Read in more data into shared memory

Stencil operations



3D sub-domain

Threads in one plane

Source:

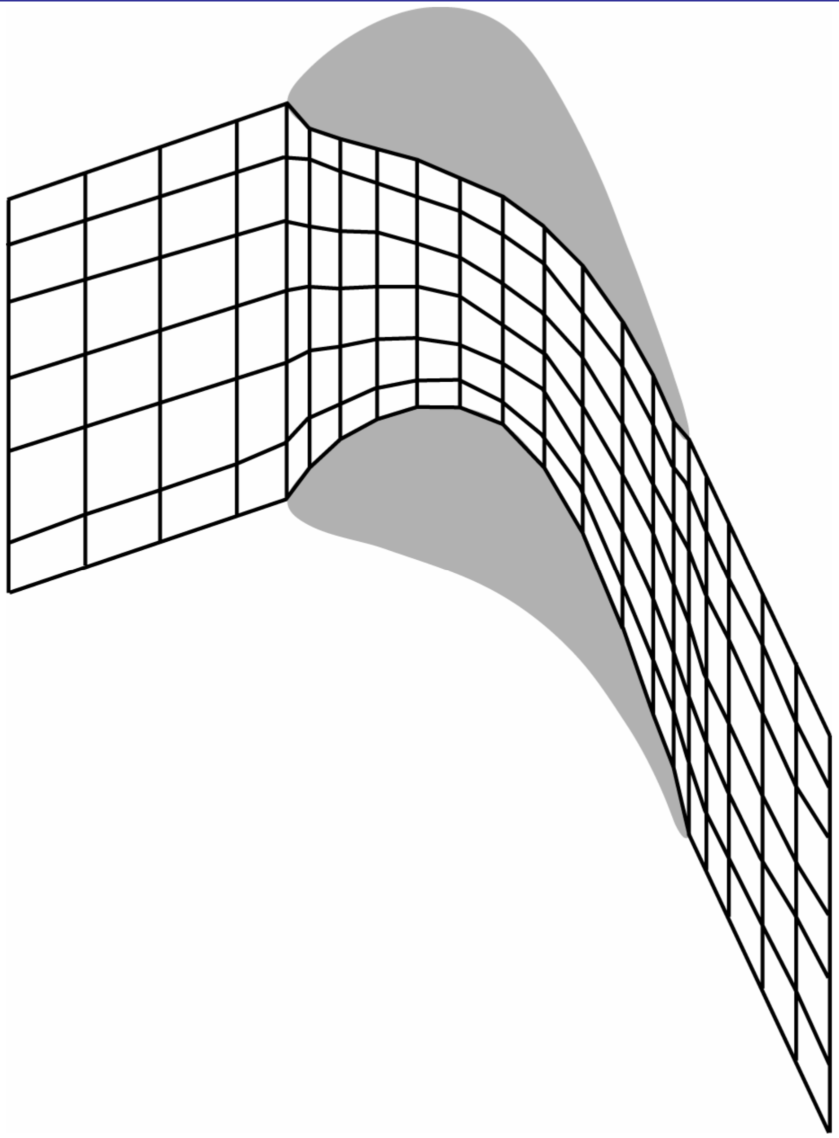
http://www.eng.cam.ac.uk/~gp10006/research/Brandvik_Pullan_2008a_DRAFT.pdf

CUDA stencil kernel

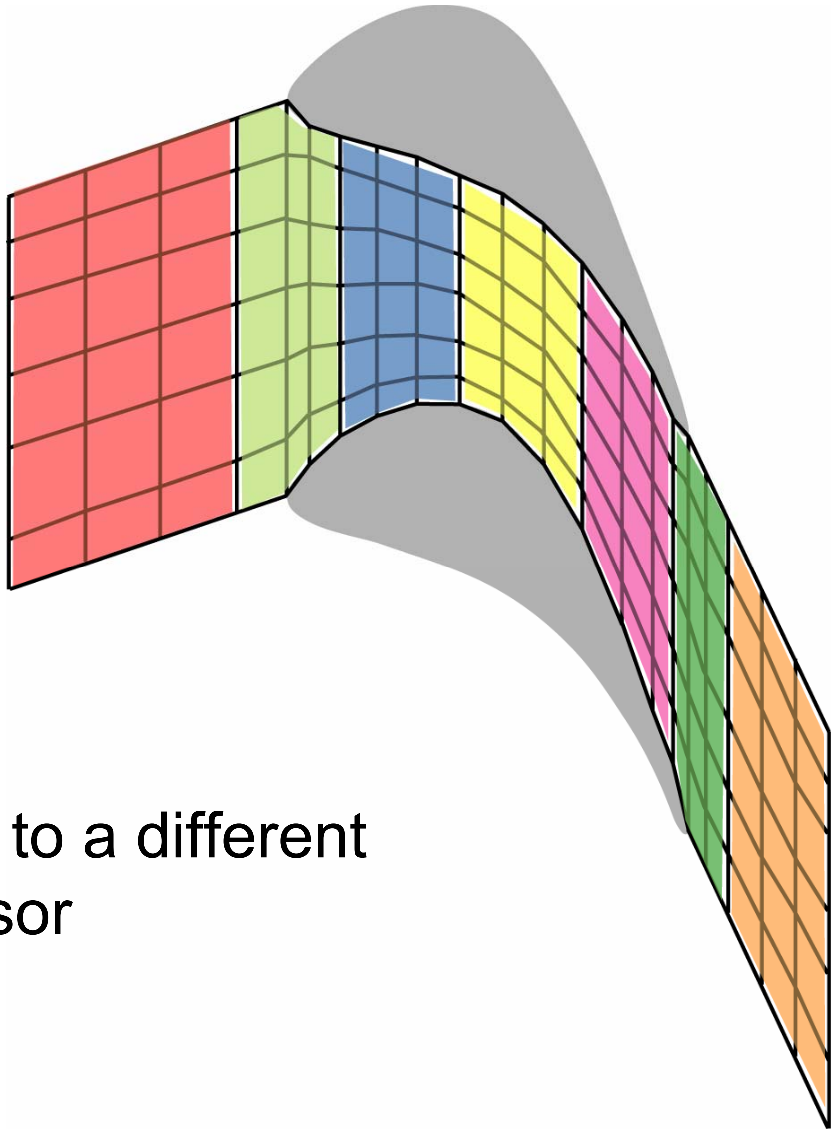
```
__global__ void smooth_kernel(float sf, float
   *a_data, float *b_data){

/* 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 ...*/
```

Typical grid – CUDA partitioning

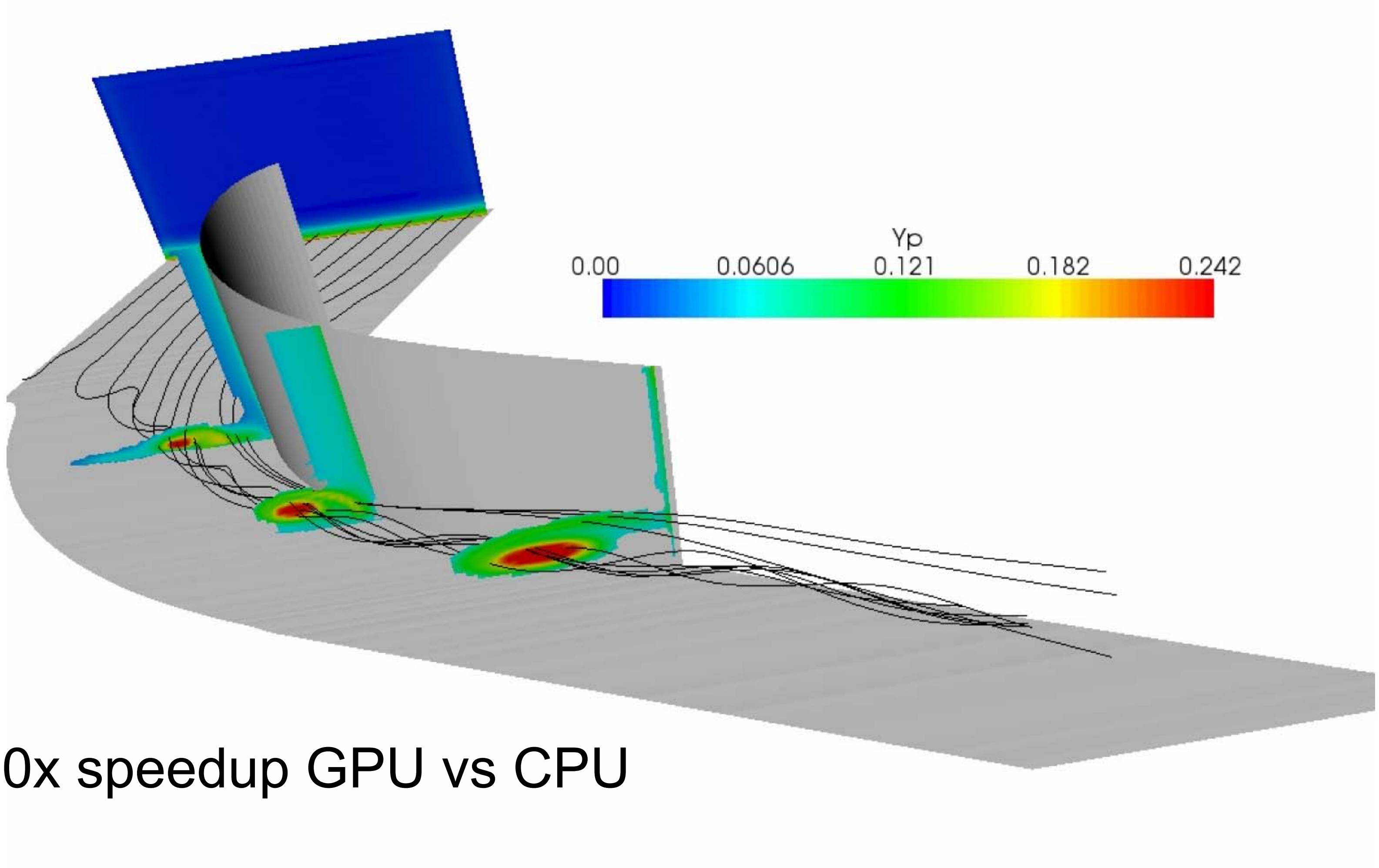


Typical grid – CUDA partitioning



Each colour to a different multiprocessor

3D results



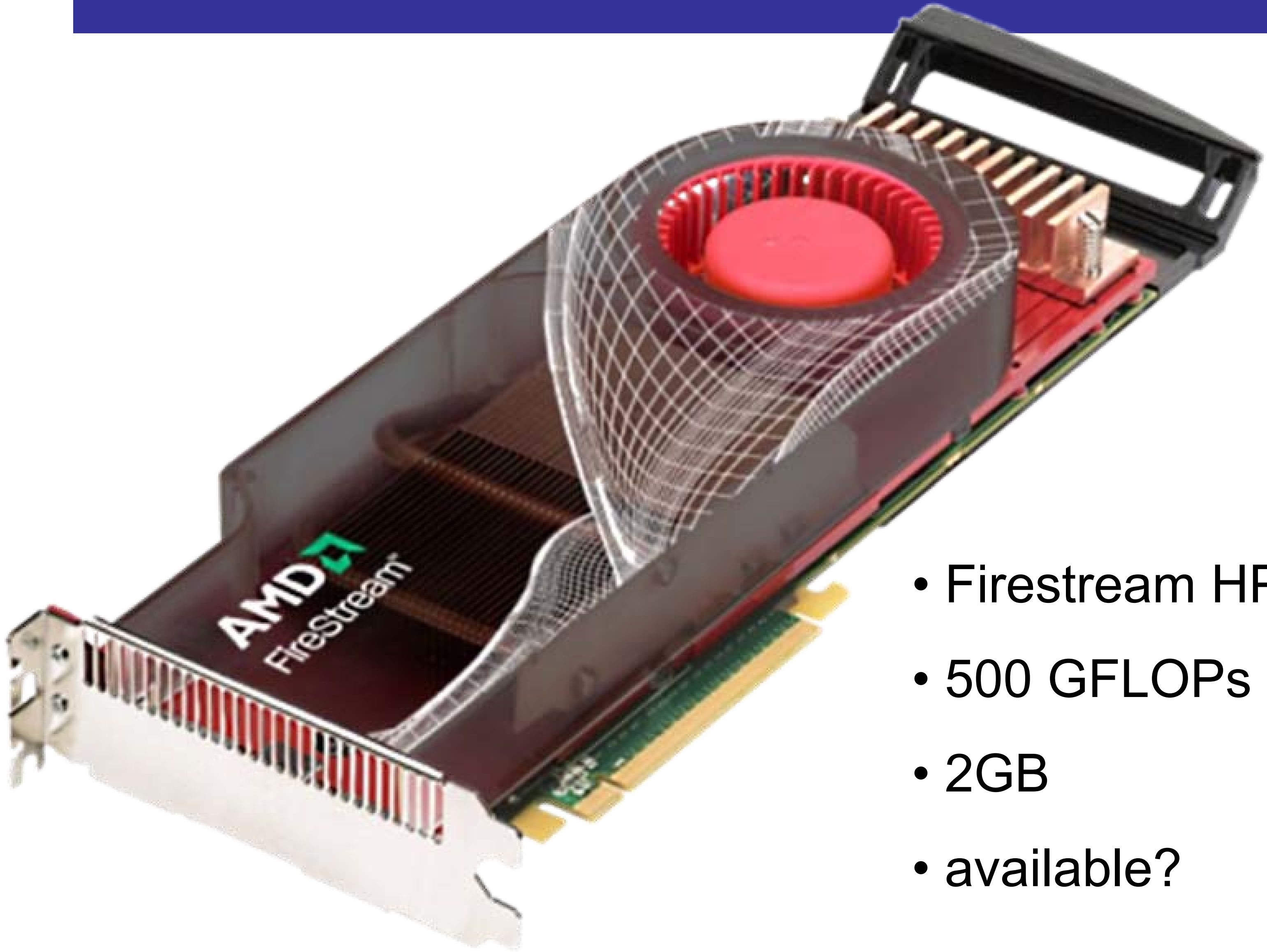
30x speedup GPU vs CPU

Part 5: NVIDIA – the only show in town?

NVIDIA

- 4 Tesla HPC GPUs
- 500 GFLOPs peak per GPU
- 1.5GB per GPU

AMD



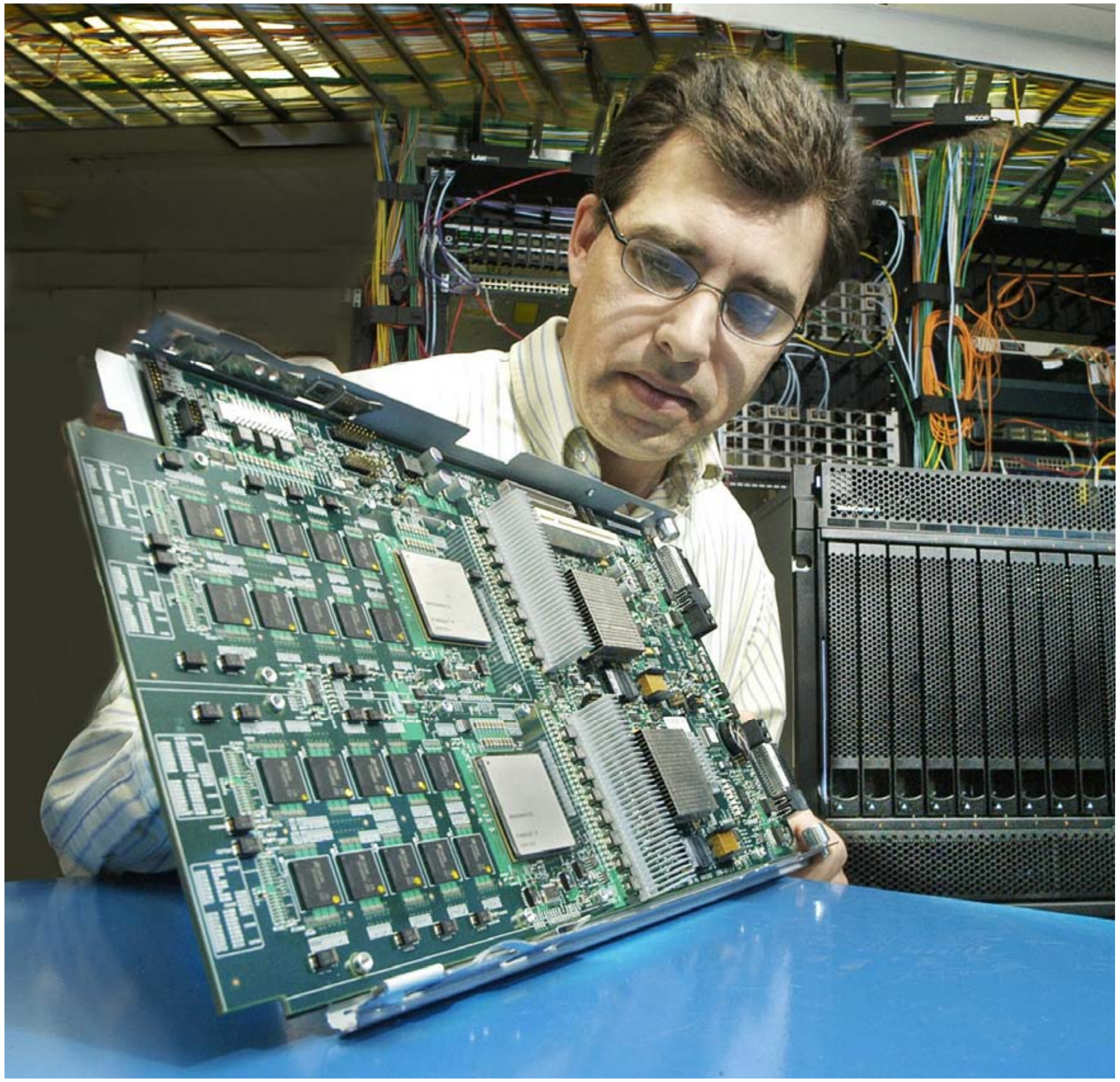
- Firestream HPC GPU
- 500 GFLOPs
- 2GB
- available?

ClearSpeed

80 GFLOPs

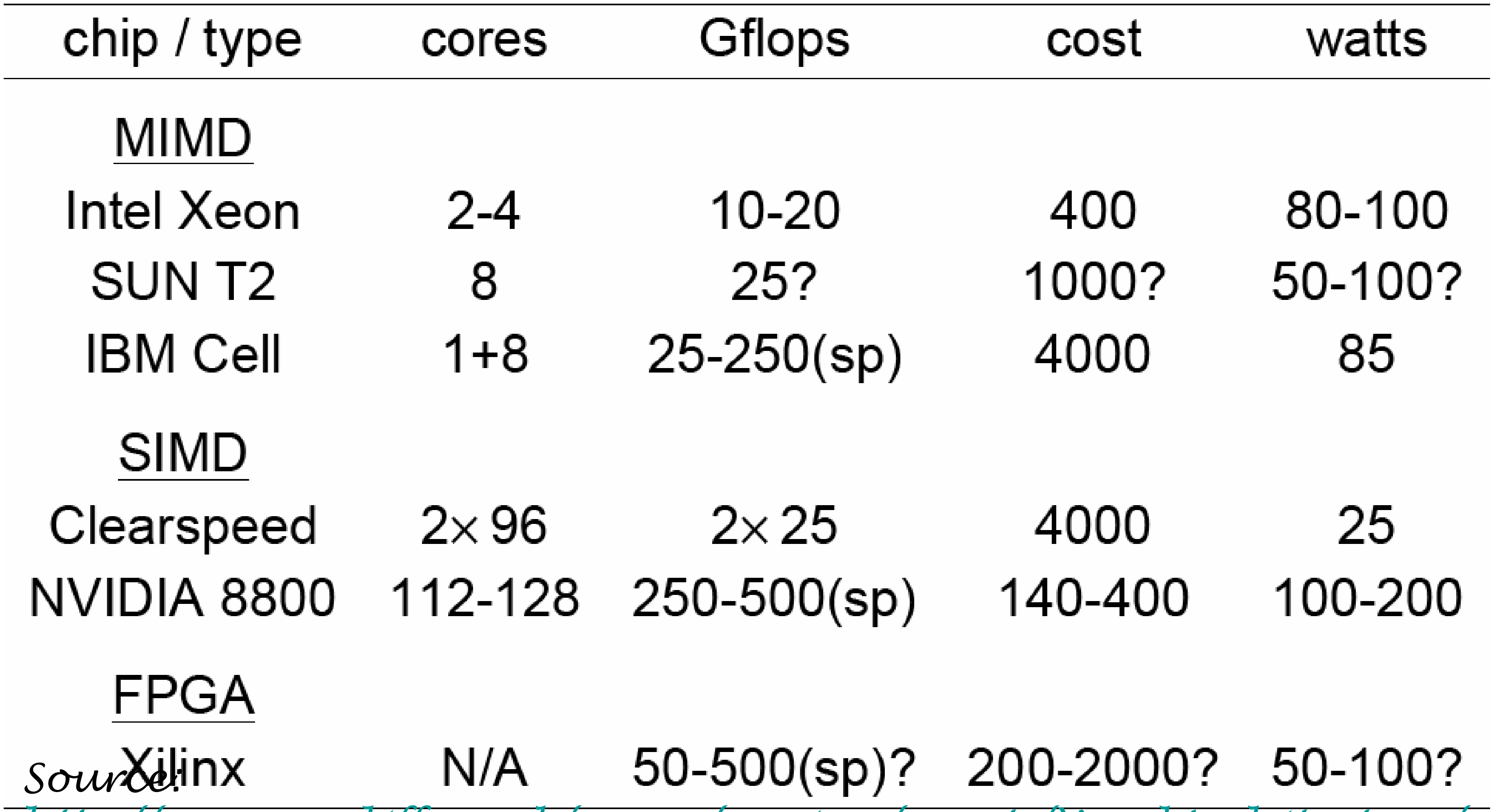
35 W !

IBM Cell BE



25 x 8 GFLOPs

Chip comparison (Giles 2008)



chip / type	cores	Gflops	cost	watts
<u>MIMD</u>				
Intel Xeon	2-4	10-20	400	80-100
SUN T2	8	25?	1000?	50-100?
IBM Cell	1+8	25-250(sp)	4000	85
<u>SIMD</u>				
Clearspeed	2×96	2×25	4000	25
NVIDIA 8800	112-128	250-500(sp)	140-400	100-200
<u>FPGA</u>				
Xilinx	N/A	50-500(sp)?	200-2000?	50-100?

Source:
http://www.cardiff.ac.uk/arcca/services/events/NovelArchitecture/Mike-Giles.pdf

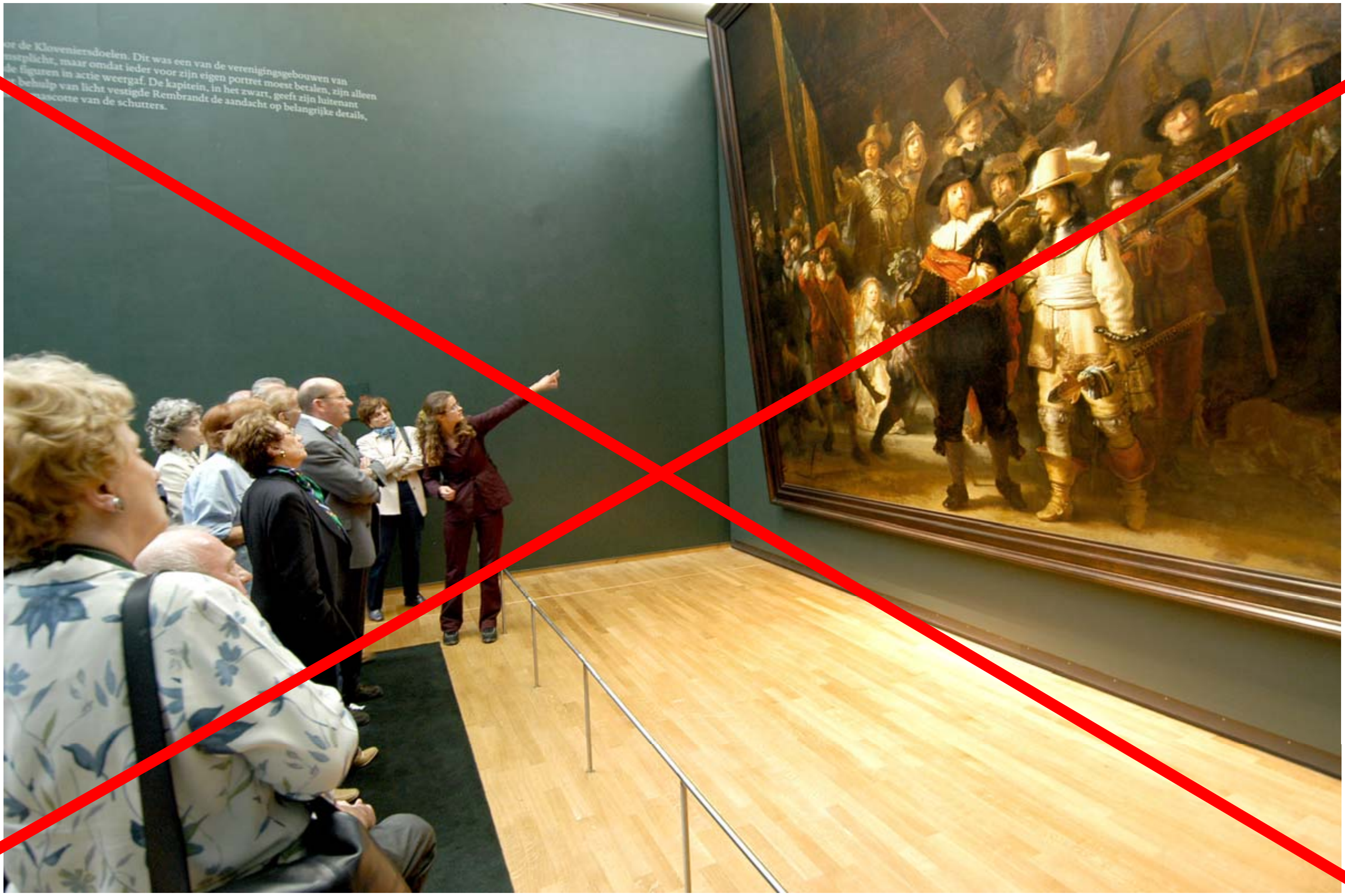
Too much choice!

- Each device has
  - different hardware characteristics
  - different software (C extensions)
  - different developer tools
- How can we write code for all SIMD devices for all applications?

Big picture – all devices, all problems?

or de Kloveniersdoelen. Dit was een van de verenigingsgebouwen van
nstplicht, maar omdat ieder voor zijn eigen portret moest betalen, zijn alleen
de figuren in actie weergaf. De kapitein, in het zwart, geeft zijn luitenant
et behulp van licht vestigde Rembrandt de aandacht op belangrijke details,
s de mascotte van de schutters.

Forget the big picture


or de Kloveniersdoelen. Dit was een van de verenigingsgebouwen van
nstplicht, maar omdat ieder voor zijn eigen portret moest betalen, zijn alleen
de figuren in actie weergaf. De kapitein, in het zwart, geeft zijn luitenant
behulp van licht vestigde Rembrandt de aandacht op belangrijke details,
mascotte van de schutters.

Tackle the dwarves!

The View from Berkeley (7 “dwarves”)

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce

Source:
http://view.eecs.berkeley.edu/wiki/Main_Page

The View from Berkeley (13 dwarves?)

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

The View from Berkeley (13 dwarves?)

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. ***Structured Grids***
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

SBLOCK (Brandvik)

- Tackle structured grid, stencil operations dwarf
- Define kernel using high level Python abstraction
- Generate kernel for a range of devices from same definition: CPU, GPU, Cell
- Use MPI to handle multiple devices

SBLOCK kernel 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])"""}
```

SBLOCK – CPU implementation (C)

```
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]);
      }
    }
  }
}
```

SBLOCK – GPU implementation (CUDA)

```
__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 ...*/
```

Benefits of SBLOCK

So long as the task fits the dwarf:

- Programmer need not learn every device library
- Optimal device code is produced
- Code is future proofed (so long as back-ends are available)

Part 6: Conclusions

Conclusions

- Many science applications fit the SIMD model
- GPUs are commodity SIMD chips
- Good speedups (10x – 100x) can be achieved

Conclusions

- Many science applications fit the SIMD model
- GPUs are commodity SIMD chips
- Good speedups (10x – 100x) can be achieved

- GPGPU is evolving (Owens, UC Davis):

1. Making it work at all (graphics APIs)
2. Doing it better (high level APIs)
3. Doing it right (portable, modular building blocks)

Conclusions

- Many science applications fit the SIMD model
- GPUs are commodity SIMD chips
- Good speedups (10x – 100x) can be achieved

- GPGPU is evolving (Owens, UC Davis):
  1. Making it work at all (graphics APIs)
  2. Doing it better (high level APIs)
  3. Doing it right (portable, modular building blocks)

Acknowledgements and info

- Research student: Tobias Brandvik (CUED)
- Donation of GPU hardware: NVIDIA

http://dx.doi.org/10.1109/JPROC.2008.917757

http://www.gpgpu.org

http://www.oerc.ox.ac.uk/research/many-core-and-reconfigurable-supercomputing