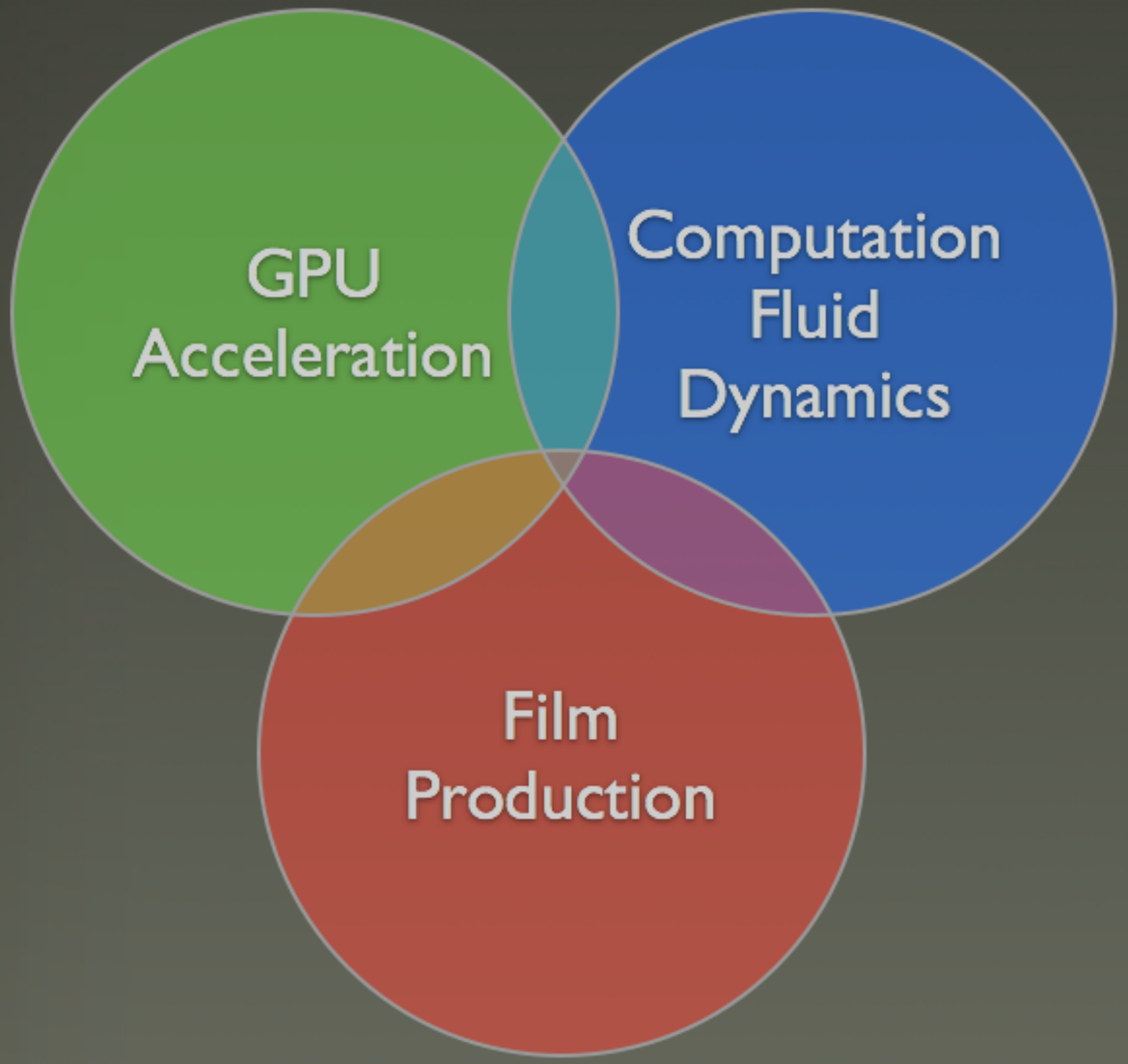
Fast GPU Preconditioning for Fluid Simulations in Film Production

Dan Bailey

double negative visual effects

double negative visual effects

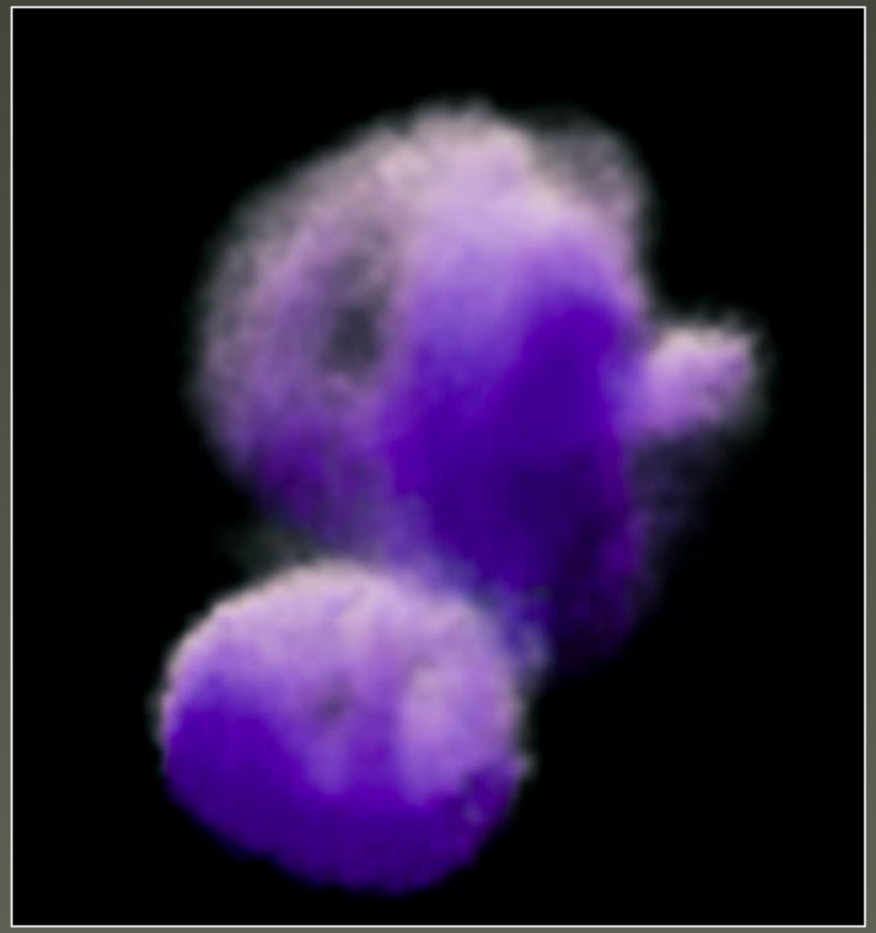


- Complex algorithms to implement
  Hard working with lots of data
- Tricky to debug code running in parallel
  GPU results different to CPU results
- Less willing to trial new technology
  Focus is always on completing shots

double negative visual effects

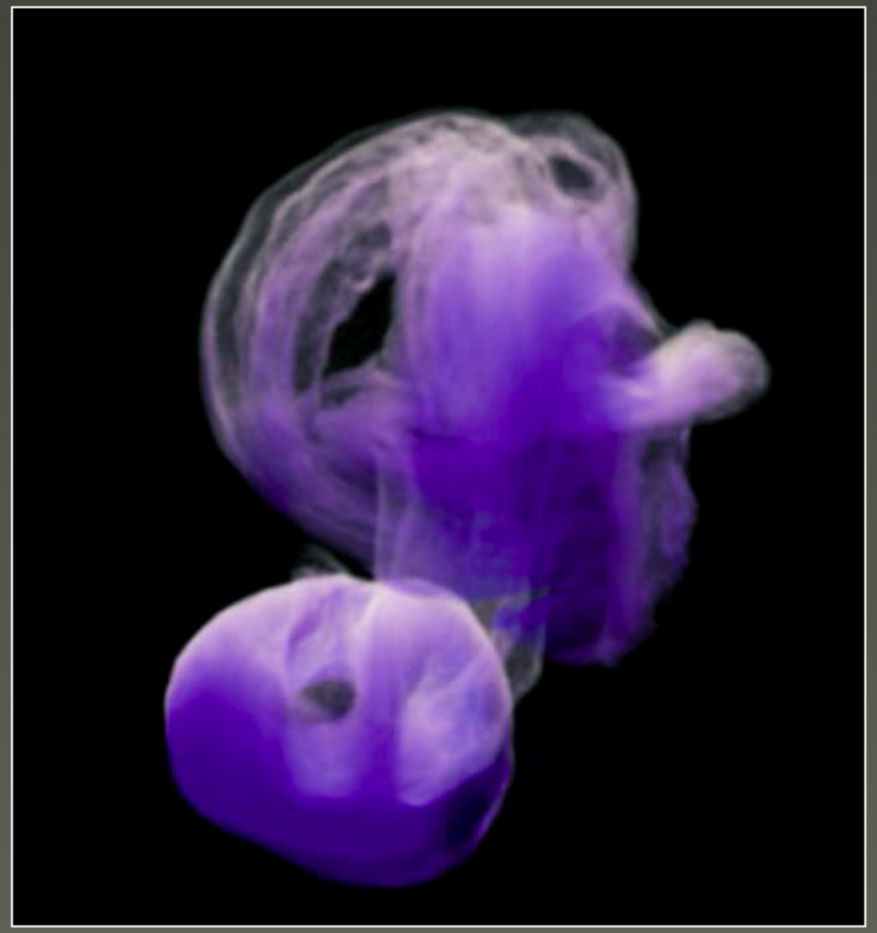
Squirt

- Proprietary Fluid Solver
- Stable Production Tool
- Highly Directable
- Slow to Simulate Large Shots



Fluid Pass

- Low resolution velocity pass
- Traditional fluid solve

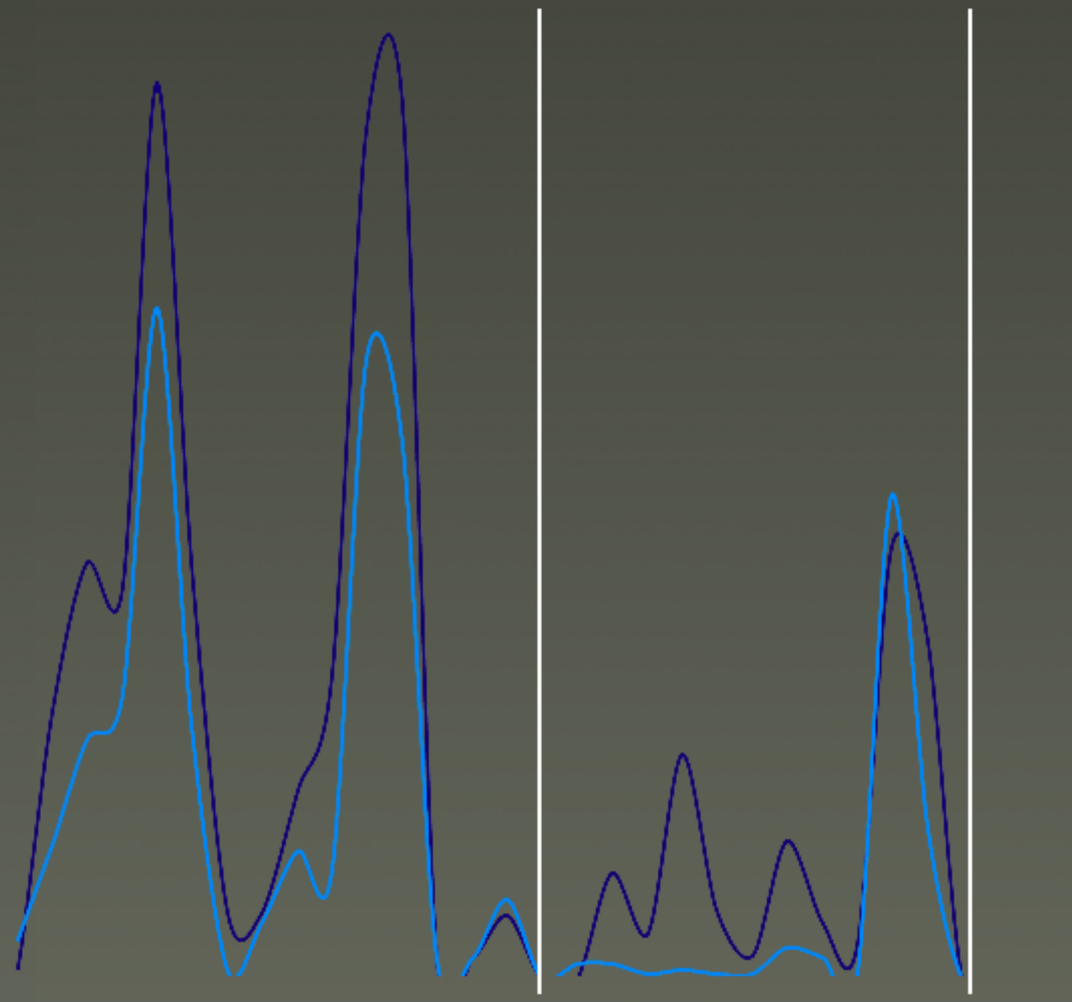


Marker Pass

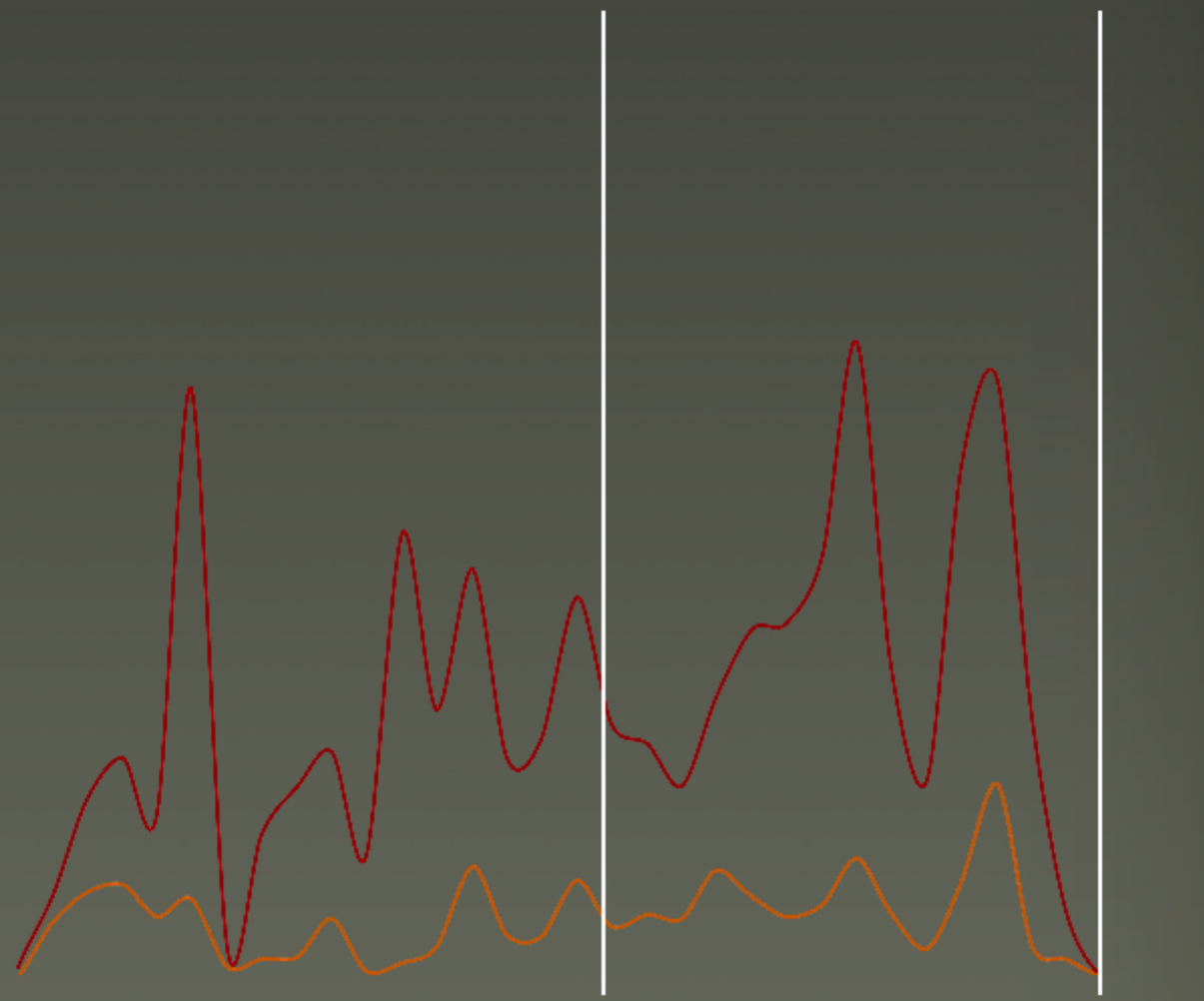
- High resolution detail pass
- Density emission replaced by particle emission

double negative visual effects

Inception



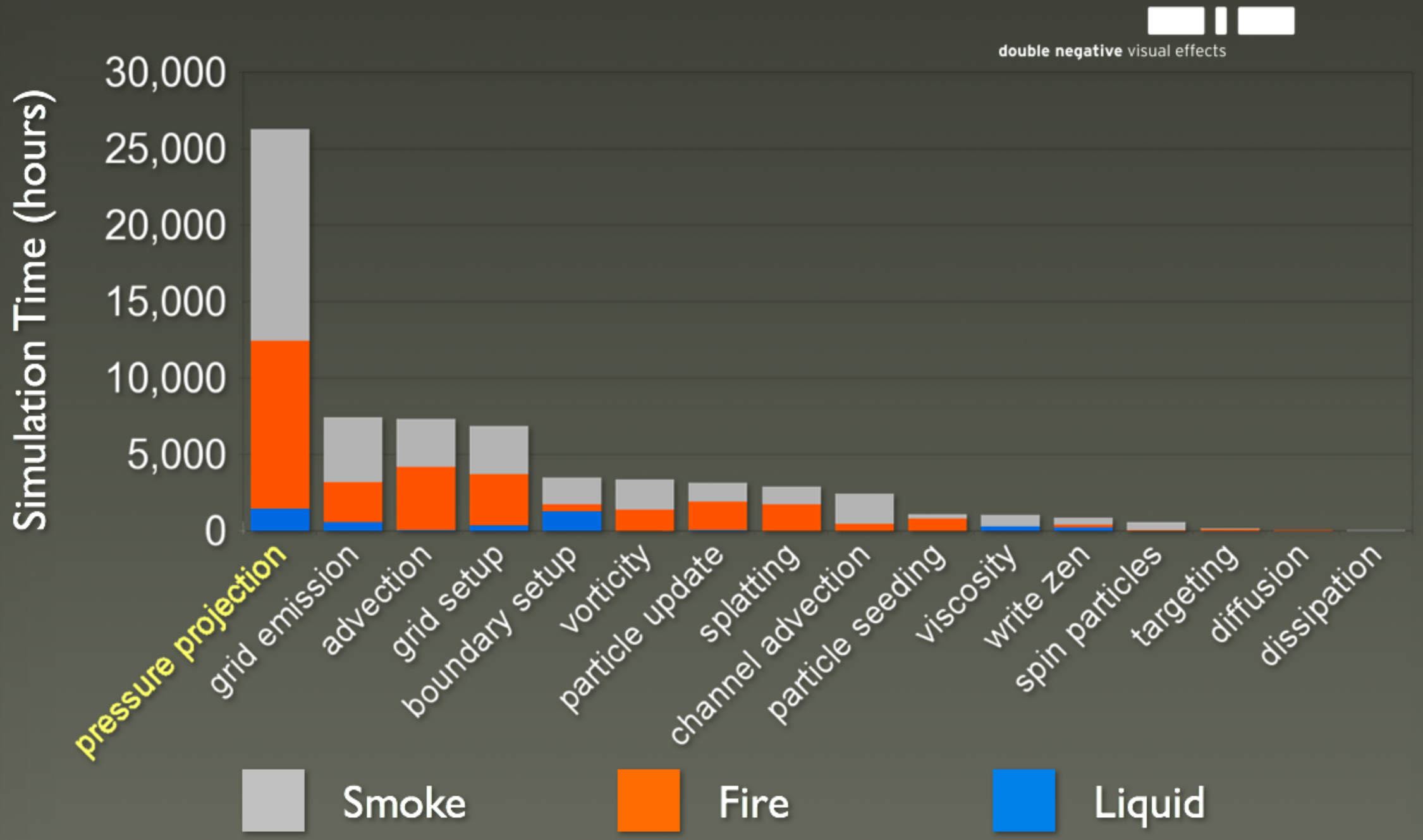
Sorcerer's Apprentice



double negative visual effects

Siggraph 2009 - New Orleans

- Directable, high-resolution simulation of fire on the GPU (Horvath, Geiger - ILM)
- Sprite Rendering and Particle Simulation (Disney)
- Splat - Sprite Rendering (Sony Picture Imageworks)
- V-Ray, OptiX, OpenCL, etc...


double negative visual effects
Simulation Time (hours)
30,000
25,000
20,000
15,000
10,000
5,000
0
pressure projection
grid emission
advection
grid setup
boundary setup
vorticity
particle update
splatting
channel advection
particle seeding
viscosity
write zen
spin particles
targeting
diffusion
dissipation
Smoke
Fire
Liquid

Navier Stokes Equations:

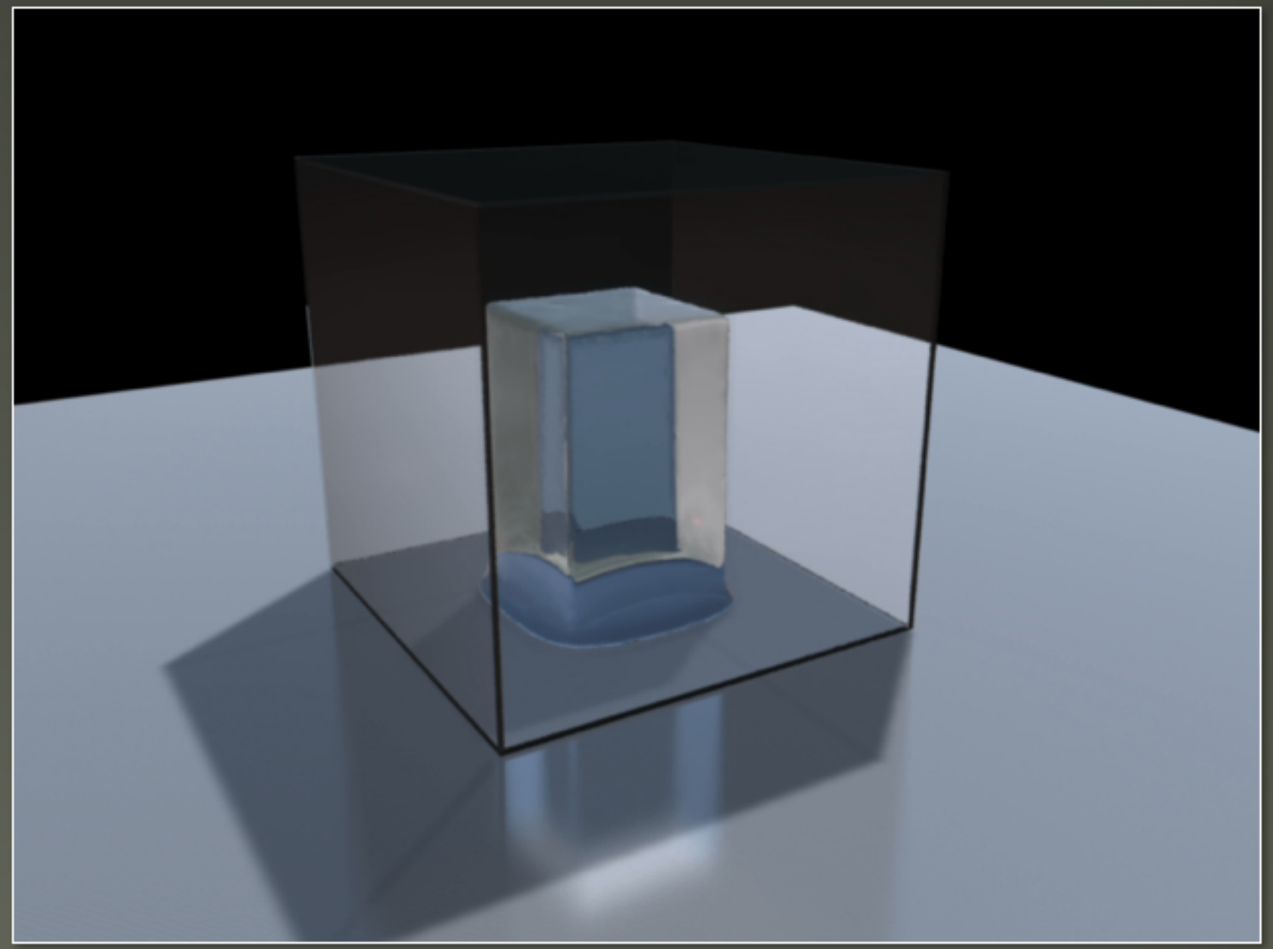
momentum equation

(1) $$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu \nabla^2 u + f$$

(2) $$\nabla \cdot u = 0$$

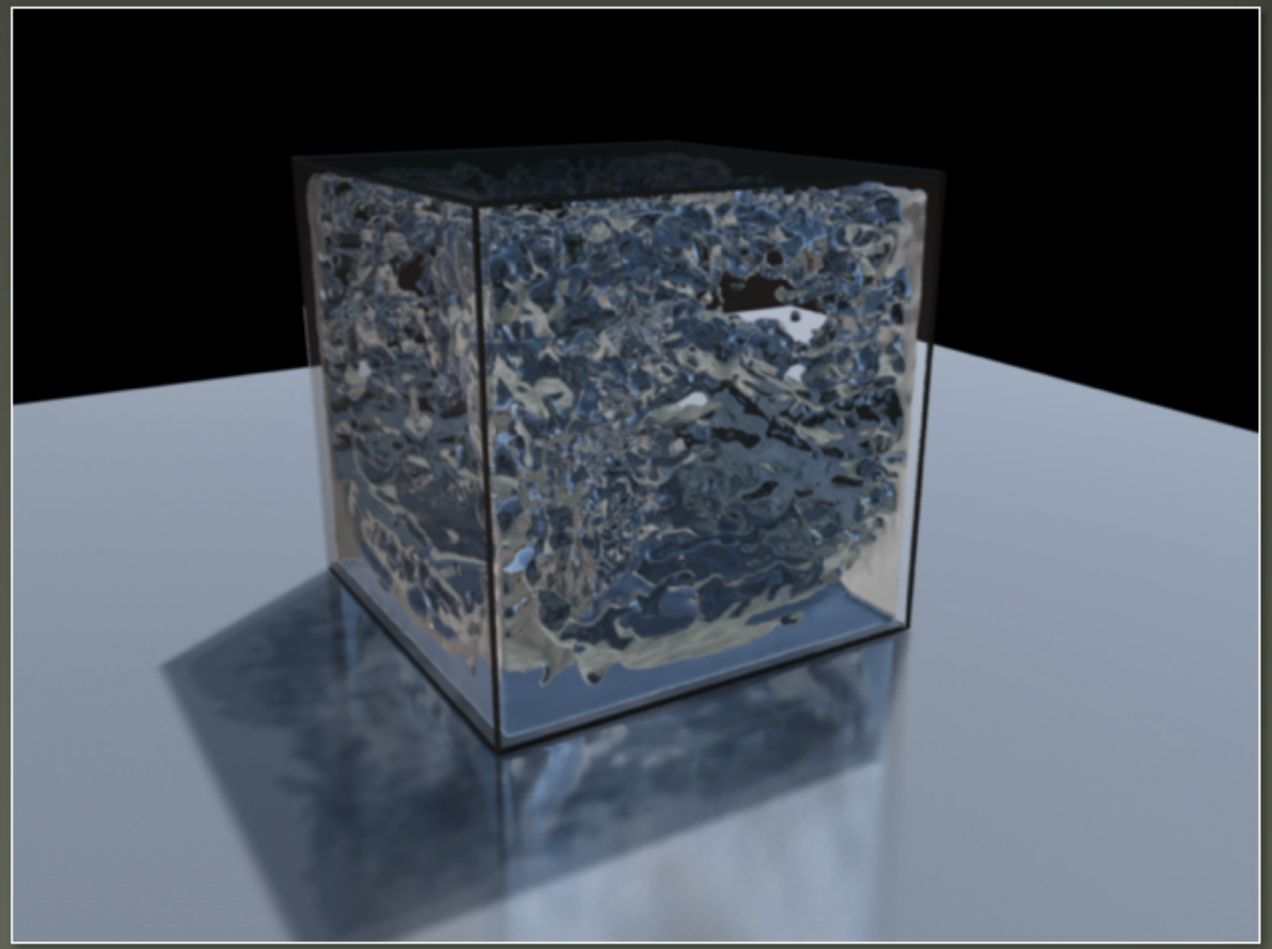
incompressibility condition

Aim for a divergence-free velocity field

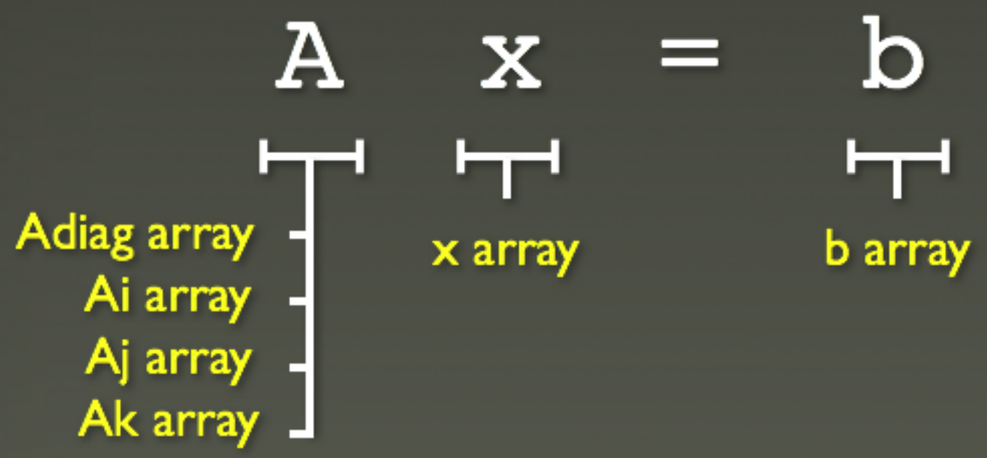


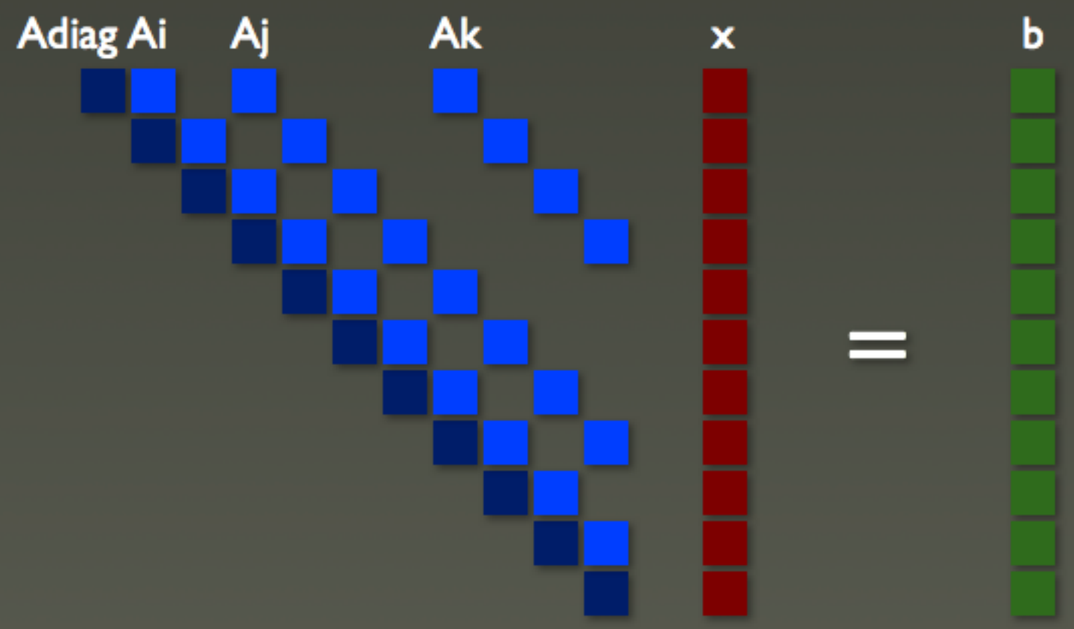
Values are put into a pressure Poisson equation

Can be solved iteratively using a Poisson solver



double negative visual effects





Matrices and vectors stored in 3D arrays

Iteratively solve using conjugate gradients

Minimise Ax - b = 0

Stop at specified residual tolerance

The max() and dot() operations are trivial:

```
dot(s, z) = s[0] * z[0] +
            s[1] * z[1] +
            ...
```

multiplication performed into temp array

Then use reductions for sum and max

(See NVidia SDK for reduction examples)

```
r = b
ρ = dot(r, r)
if ρ == 0: end
s = z
for i = 0:10000
    z = As
    α = ρ / dot(s, z)
    r = r - αz
    x = x + αz
    if max(r) < tolerance: end
    ρ* = dot(r, r)
    β = ρ* / ρ
    s = βs + r
    ρ = ρ*
```

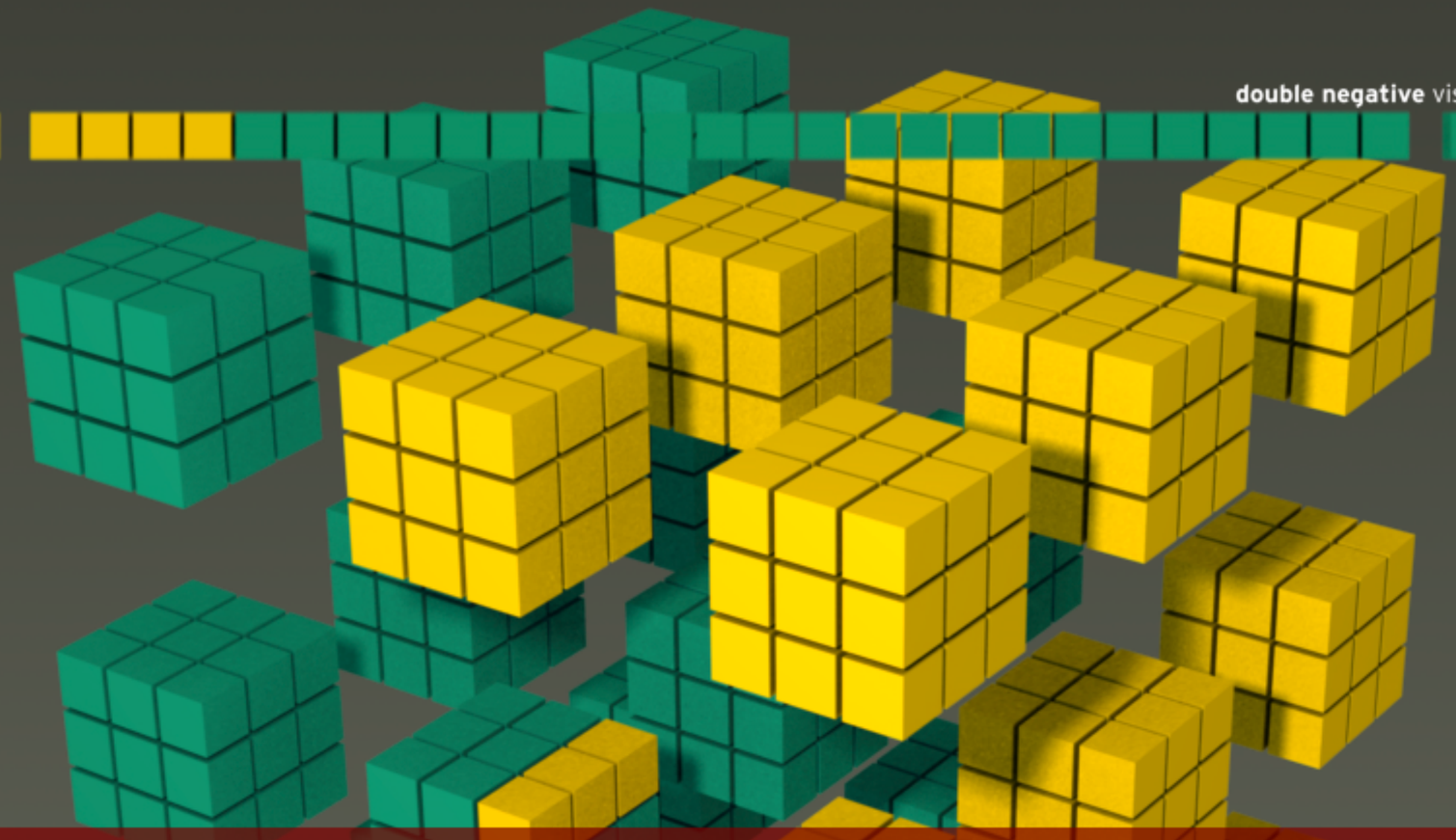
Hardest part to optimise

```
r = b
ρ = dot(r, r)
if ρ == 0: end
s = z
for i = 0:10000
    z = As
    α = ρ / dot(s, z)
    r = r - αz
    x = x + αz
    if max(r) < tolerance: end
    ρ* = dot(r, r)
    β = ρ* / ρ
    s = βs + r
    ρ = ρ*
```

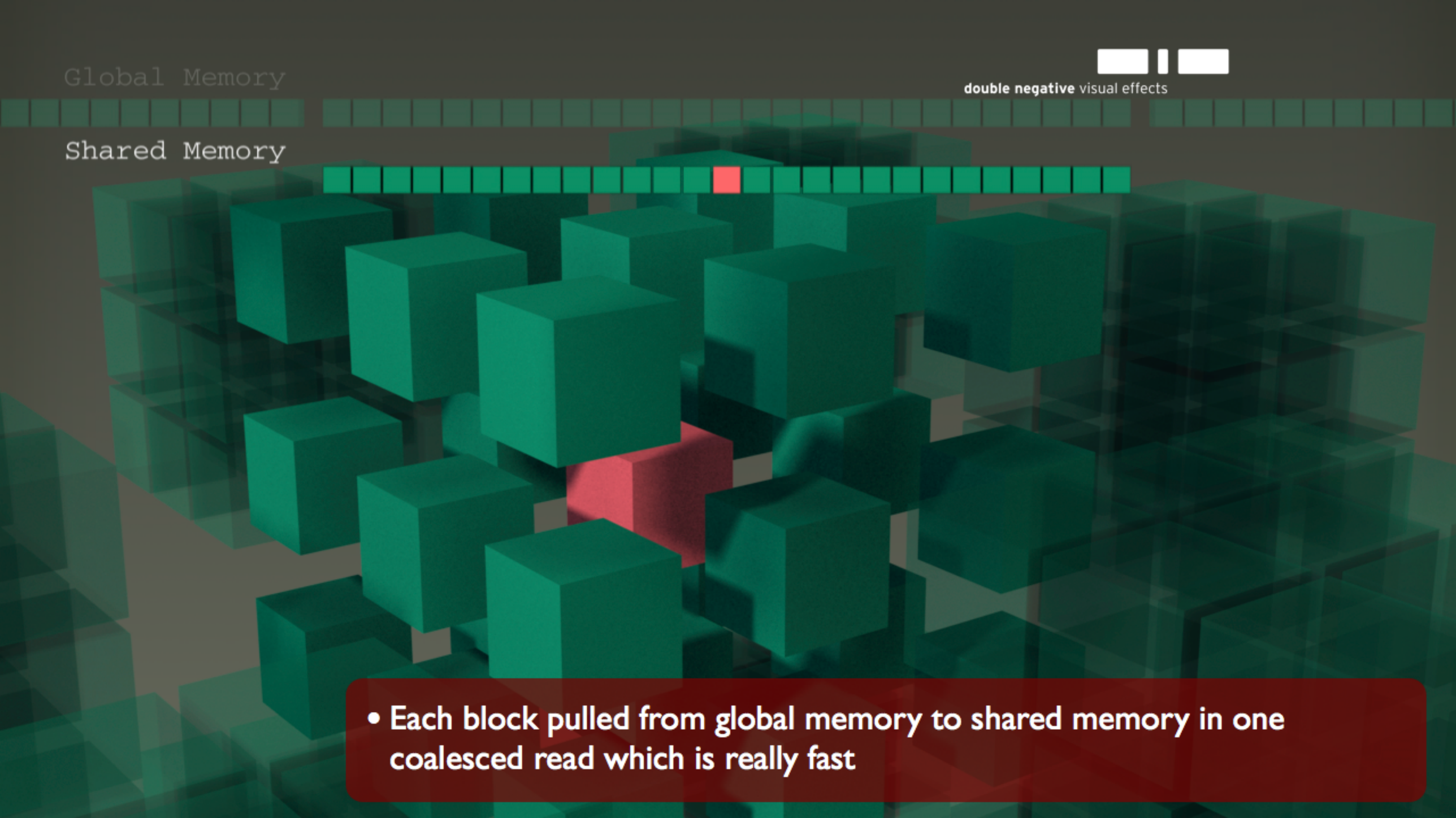
```
z(i, j, k) = Adiag(i, j, k) * s(i, j, k) +
             Ai(i, j, k) * s(i + 1, j, k) +
             Aj(i, j, k) * s(i, j + 1, k) +
             Ak(i, j, k) * s(i, j, k + 1) +
             Ai(i - 1, j, k) * s(i - 1, j, k) +
             Aj(i, j - 1, k) * s(i, j - 1, k) +
             Ak(i, j, k - 1) * s(i, j, k - 1);
```

repeated, uncoalesced
global memory access

- Algorithms based around grids use neighbouring values heavily
- Computation is very simple, so kernels heavily limited by memory bandwidth
- Essential to make use of coalescing to get data out of global memory faster



- Grid stored in a 3D array arranged contiguously in memory per block
- Block size is actually 8 x 8 x 4 (and 8 x 8 x 8 on Fermi)
- Domains padded with empty memory regions to ensure domain widths are a multiple of the block size



- Each block pulled from global memory to shared memory in one coalesced read which is really fast

- Values read from shared memory using consistent relative offsets
- Offsets can be calculated at compile time as our block size is fixed
- Slices of neighbouring blocks also need to be pulled into shared memory

The process involved in using "blocked" kernels:

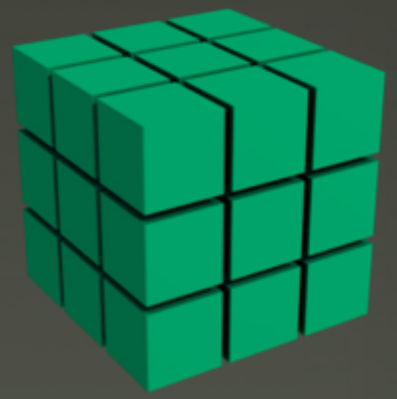
Get tile (i, j, k) values

Get tile (i, j, k) values

double negative visual effects

threadIdx.x

```
tileX = 8
tileY = 8
tileZ = 8
```

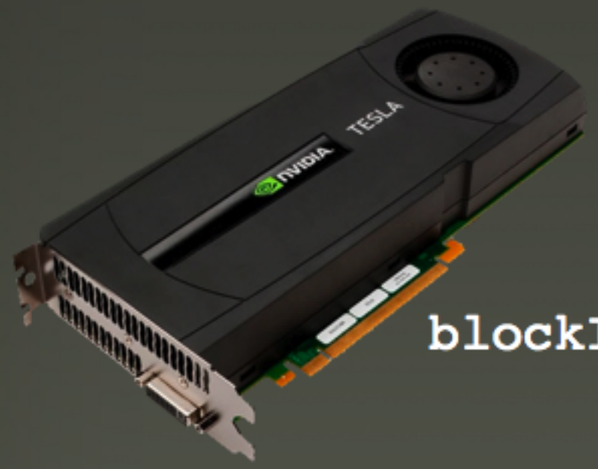


```
int k = threadIdx.x / tileX.tileY;
int j = (threadIdx.x - k.tileX.tileY) / tileX;
int i = threadIdx.x - k.tileX.tileY - j.tileX;
```

Get block (i, j, k) values

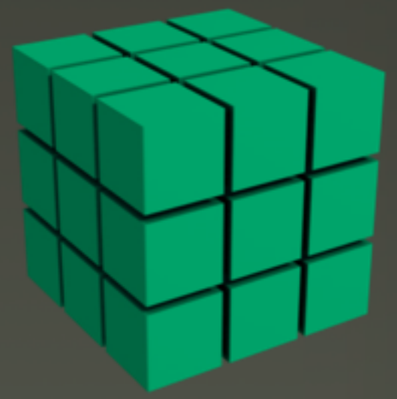
Get block (i, j, k) values

double negative visual effects



blockIdx.x

```
bx = (# blocks in x)
by = (# blocks in y)
bz = (# blocks in z)
```



```
int k = blockIdx.x / bx.by;
int j = (blockIdx.x - k.bx.by) / bx;
int i = blockIdx.x - k.bx.by - j.bx;
```

Calculate relative indices (ijk, iMjk, iPjk, ...)

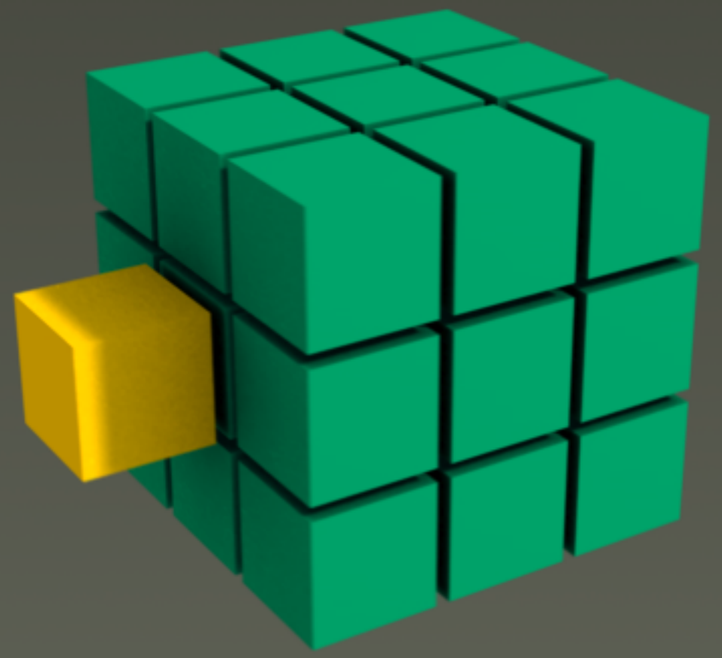
Calculate relative indices (ijk, iMjk, iPjk, ...)

double negative visual effects

Modify offsets for block boundaries

Modify offsets for block boundaries

double negative visual effects



Load global memory into shared memory

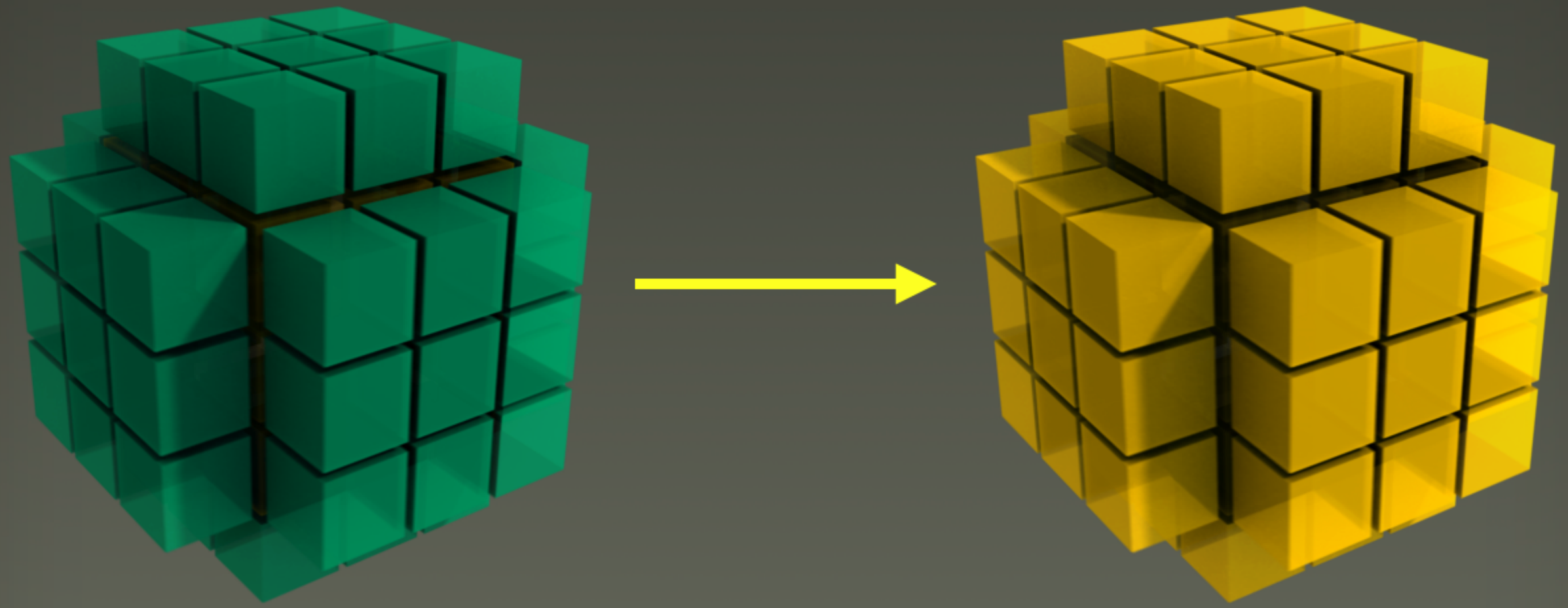
Load global memory into shared memory

double negative visual effects

Load block boundaries into shared memory

Load block boundaries into shared memory

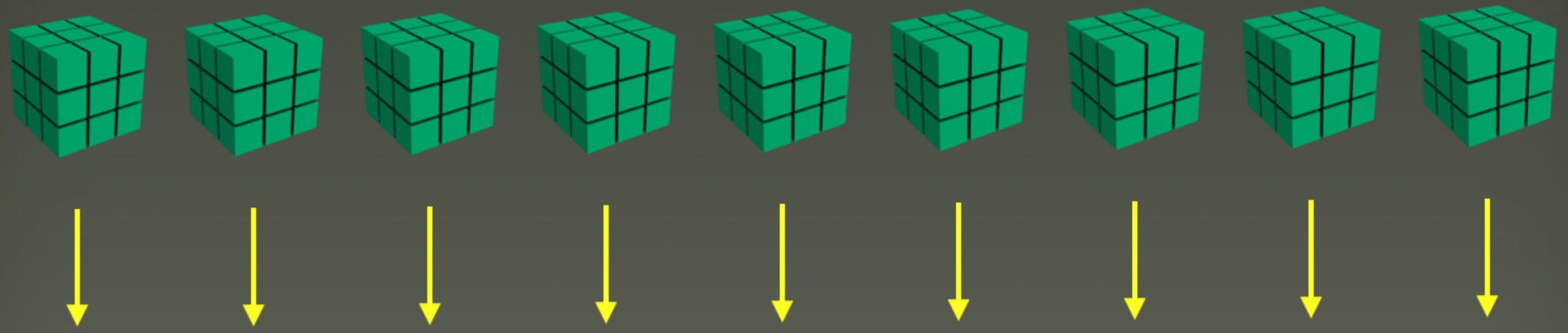
double negative visual effects



__syncthreads()

__syncthreads()

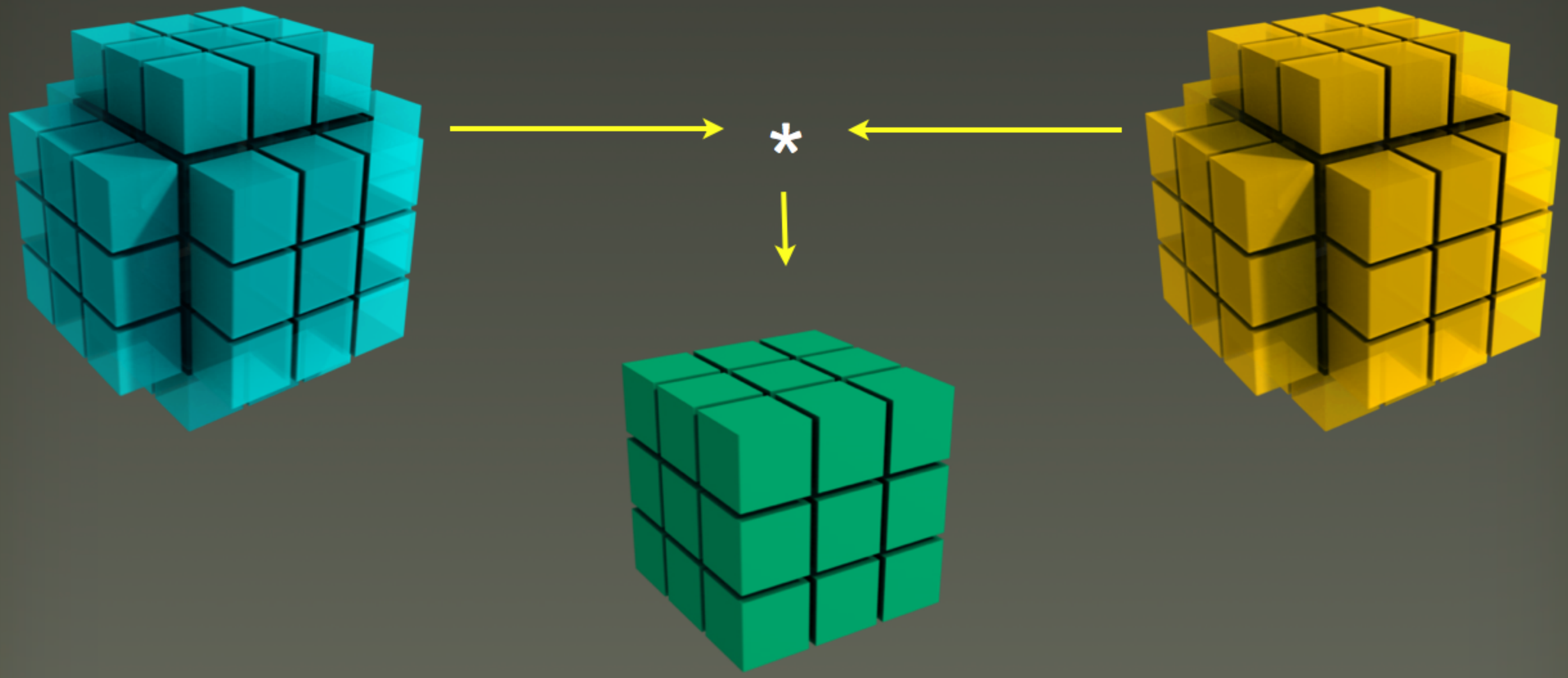
double negative visual effects



Multiply texture memory with shared memory

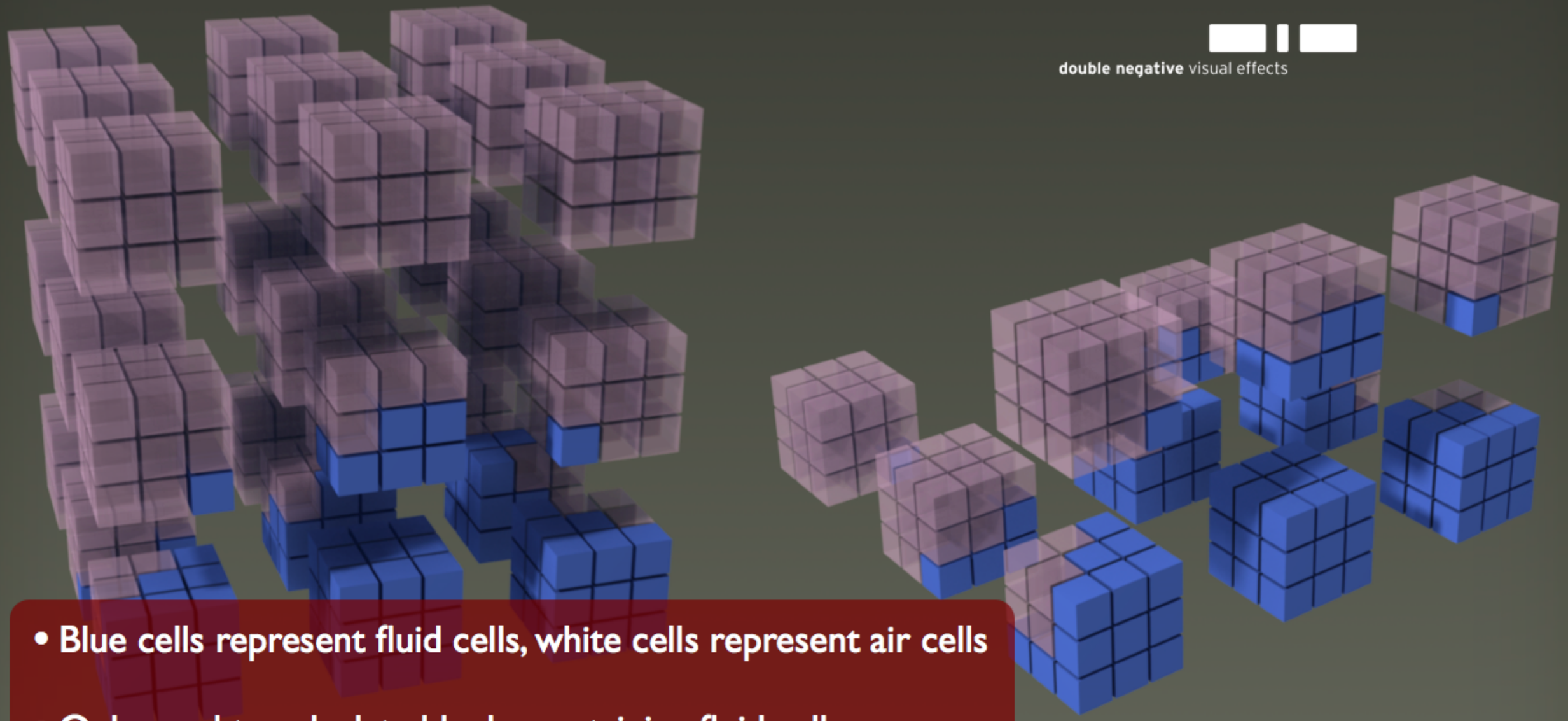
Multiply texture memory with shared memory

double negative visual effects



Save result back to global memory

double negative visual effects



- Blue cells represent fluid cells, white cells represent air cells
- Only need to calculate blocks containing fluid cells
- Highly effective optimisation

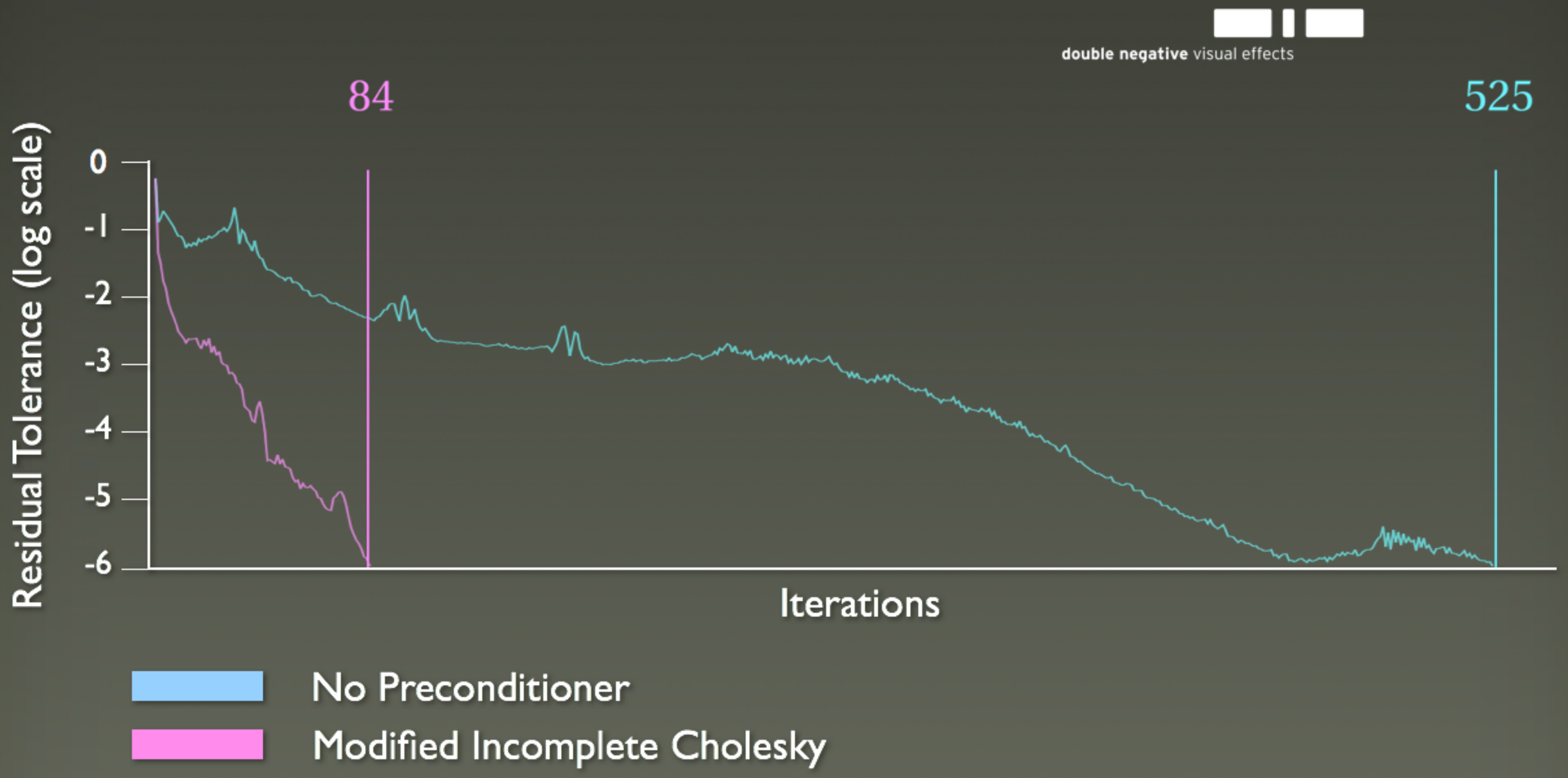
double negative visual effects

GPU Memory Regions

Temporary Arrays → Global Memory —coalescing→ Shared Memory

Input Arrays → Texture Memory

Block Cache → Constant Memory


double negative visual effects
84
525
Residual Tolerance (log scale)
0
-1
-2
-3
-4
-5
-6
Iterations
No Preconditioner
Modified Incomplete Cholesky

```
r = b
C = form_preconditioner(A)
z = apply_preconditioner(r, C)
ρ = dot(r, r)
if ρ == 0: end
s = z
for i = 0:10000
    z = As
    α = ρ / dot(s, z)
    r = r - αz
    x = x + αz
    if max(r) < tolerance: end
    z* = apply_preconditioner(r, C)
    ρ* = dot(z*, r)
    β = ρ* / ρ
    s = βs + z*
    ρ = ρ*
```

double negative visual effects

Blocked kernels are crucial when applying a preconditioner

Preconditioners

Jacobi - trivial to implement, but very ineffective

Factorisation:

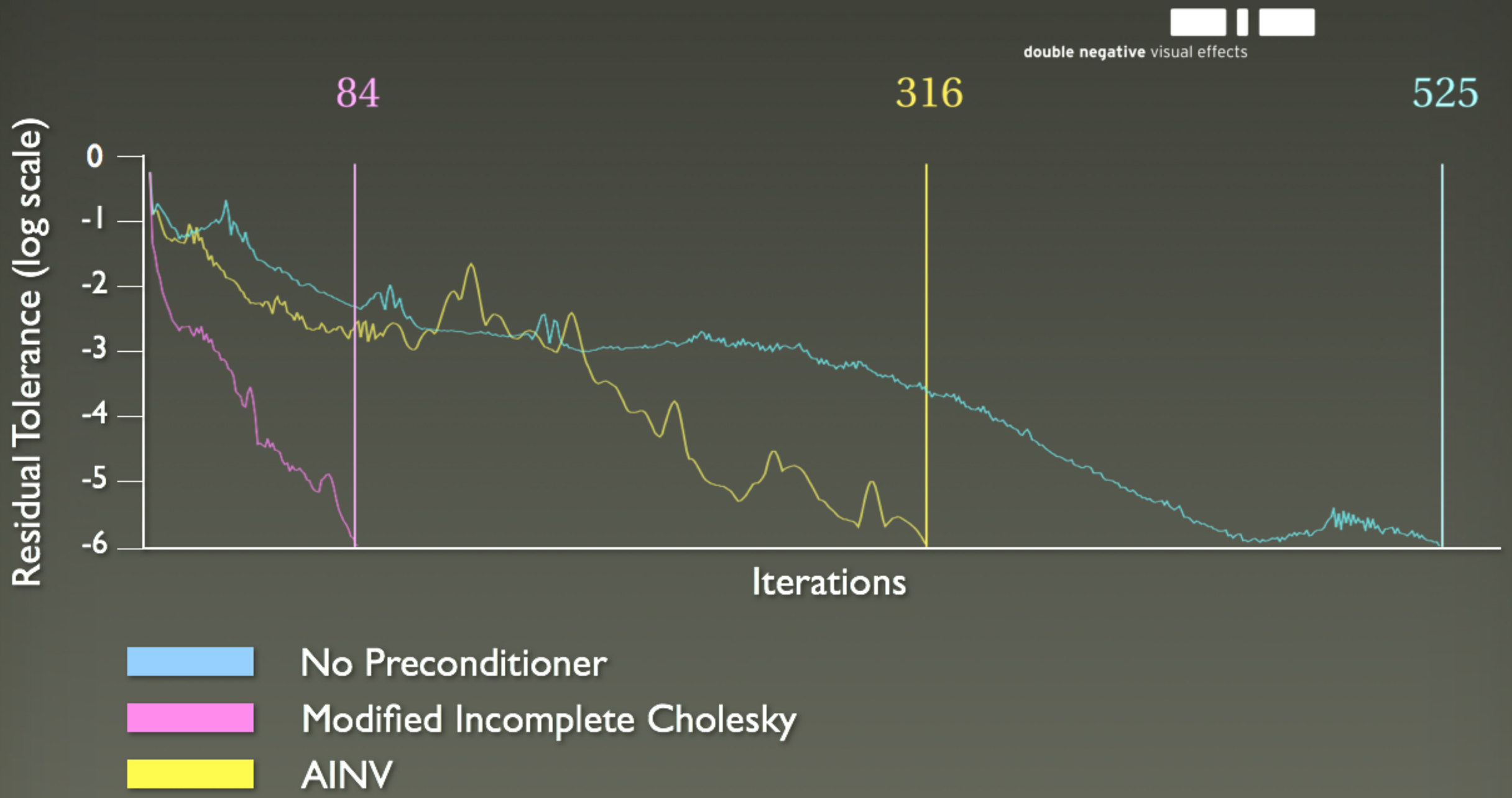
- Incomplete Cholesky (IC)
- Incomplete LU (ILU)

] very robust, but notoriously hard to parallelise

Approximate Inverse:

- SParse Approximate Inverse (SPAI)
- Factorised Sparse Approximate Inverse (FSAI)
- Approximate INVerse (AINV)

] less effective, but more natural parallelism


double negative visual effects
84
316
525
Residual Tolerance (log scale)
0
-1
-2
-3
-4
-5
-6
Iterations
No Preconditioner
Modified Incomplete Cholesky
AINV

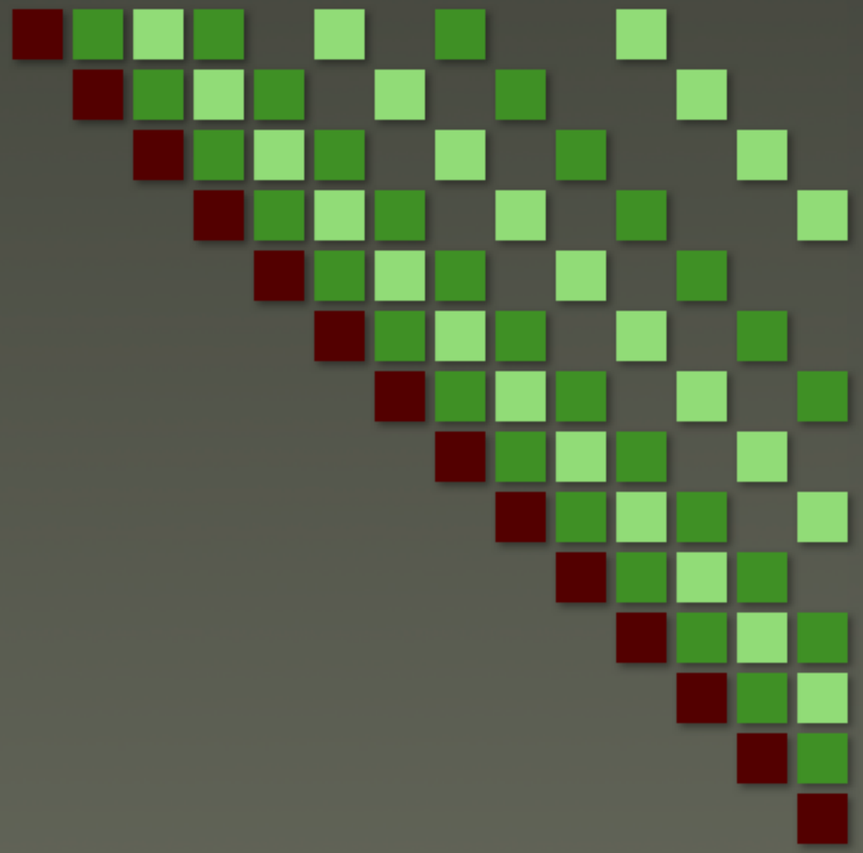
AINV Algorithm

- Outer-product form constructs an A-conjugate set of vectors from the standard basis
- FSAI potentially better as formulation done in parallel, but more complicated to implement
- Traditional approach is to use a drop tolerance or "postfiltration" process

double negative visual effects

```
z(i, j, k) =

    r(i + 1, j, k) * ci(i, j, k) +
    r(i, j + 1, k) * cj(i, j, k) +
    r(i, j, k + 1) * ck(i, j, k) +
    r(i, j, k) * cdiag(i, j, k) +
    r(i - 1, j, k) * ci(i - 1, j, k) +
    r(i, j - 1, k) * cj(i, j - 1, k) +
    r(i, j, k - 1) * ck(i, j, k - 1) +
    r(i + 1, j - 1, k) * cij(i + 1, j, k) +
    r(i - 1, j + 1, k) * cij(i, j + 1, k) +
    r(i + 1, j, k - 1) * cik(i + 1, j, k) +
    r(i - 1, j, k + 1) * cik(i, j, k + 1) +
    r(i, j + 1, k - 1) * cjk(i, j + 1, k) +
    r(i, j - 1, k + 1) * cjk(i, j, k + 1);
```



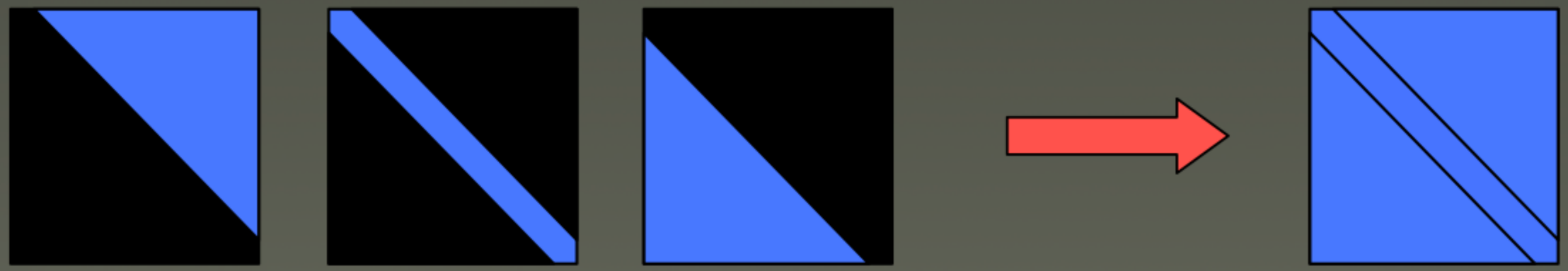
Represents the sparsity of the upper triangular matrix

double negative visual effects

AINV Algorithm

$$M = ZD^{-1}Z^T \approx A^{-1}$$

Upper Triangular Matrix Z
(Approximate Inverse Factors)

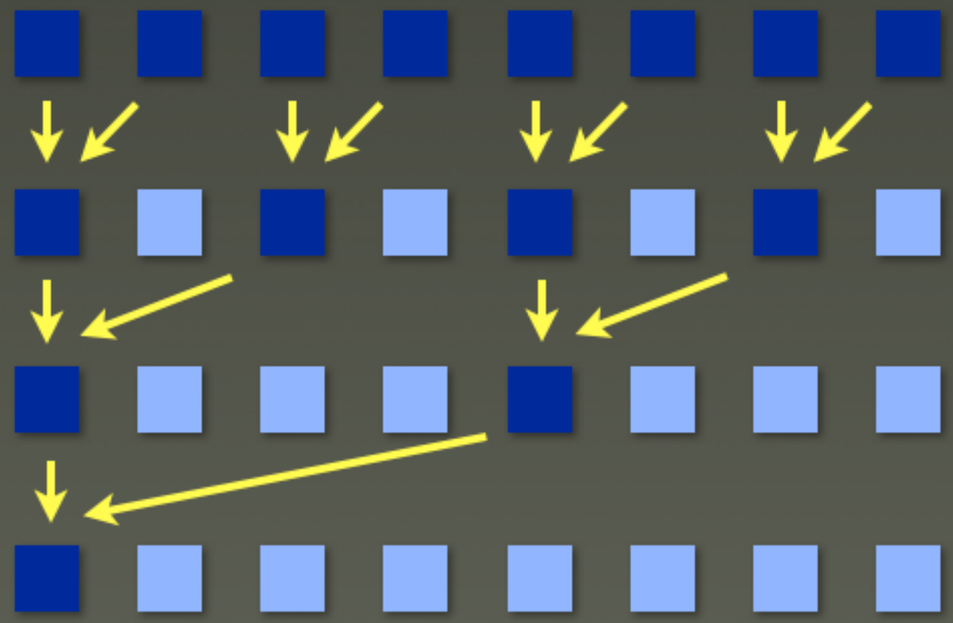


Diagonal Matrix D
(Pivots)

Obtaining Identical Results on CPU and GPU

Reduction:

- Floating point addition is non-associative
- Order of operations is different
- Floating point error will be different



During development process, copy array onto CPU and perform sequential operation

Obtaining Identical Results on CPU and GPU

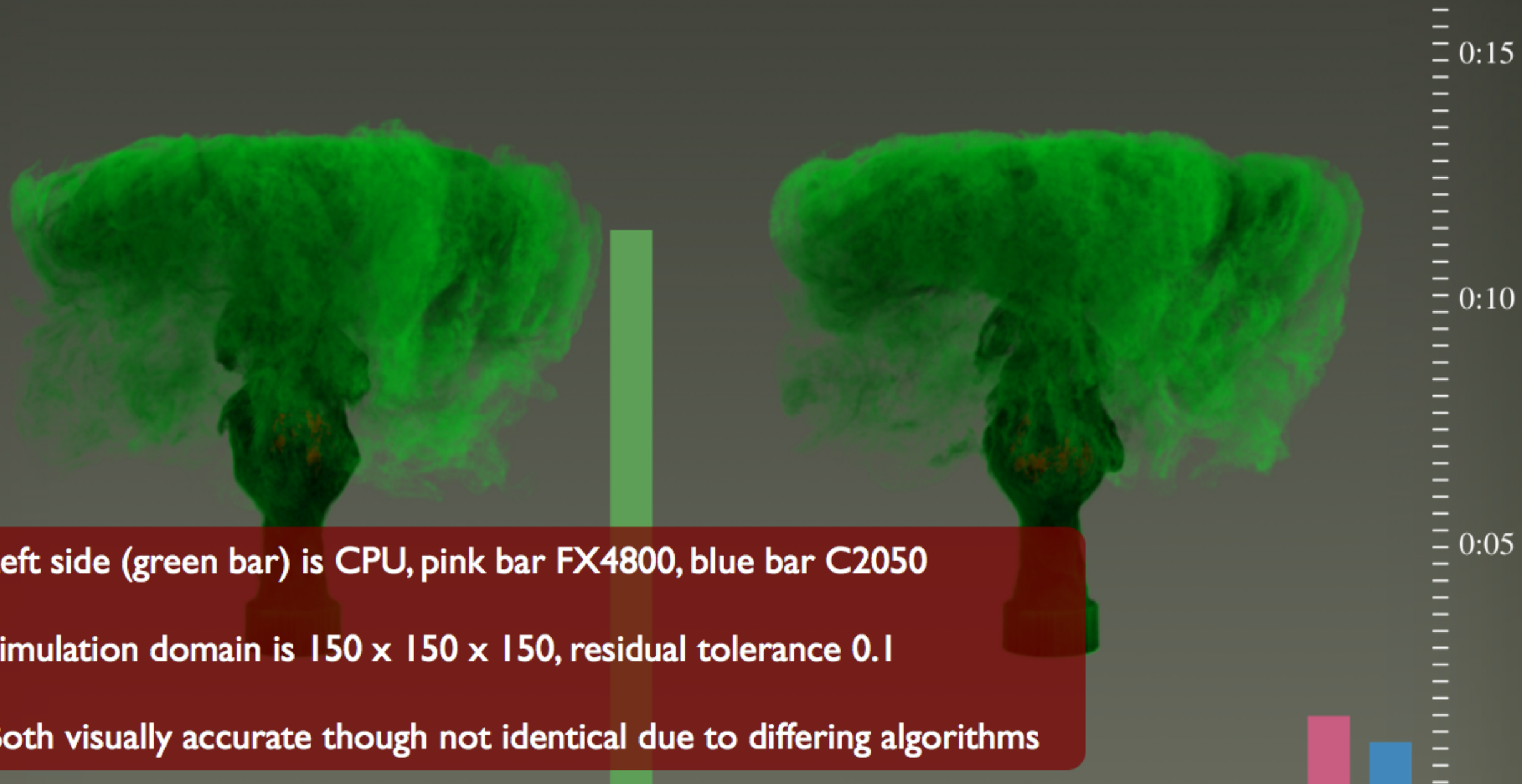
Fused Multiply-Add (FMA):

- FMA performs rounding once
- Requires two operations on CPU
- Floating point error will be different

```
__device__ double mul(double a, double b)
{
#ifdef USE_FMA
    return a * b;
#else
    return __dmul_rn(a, b);
#endif
}
```

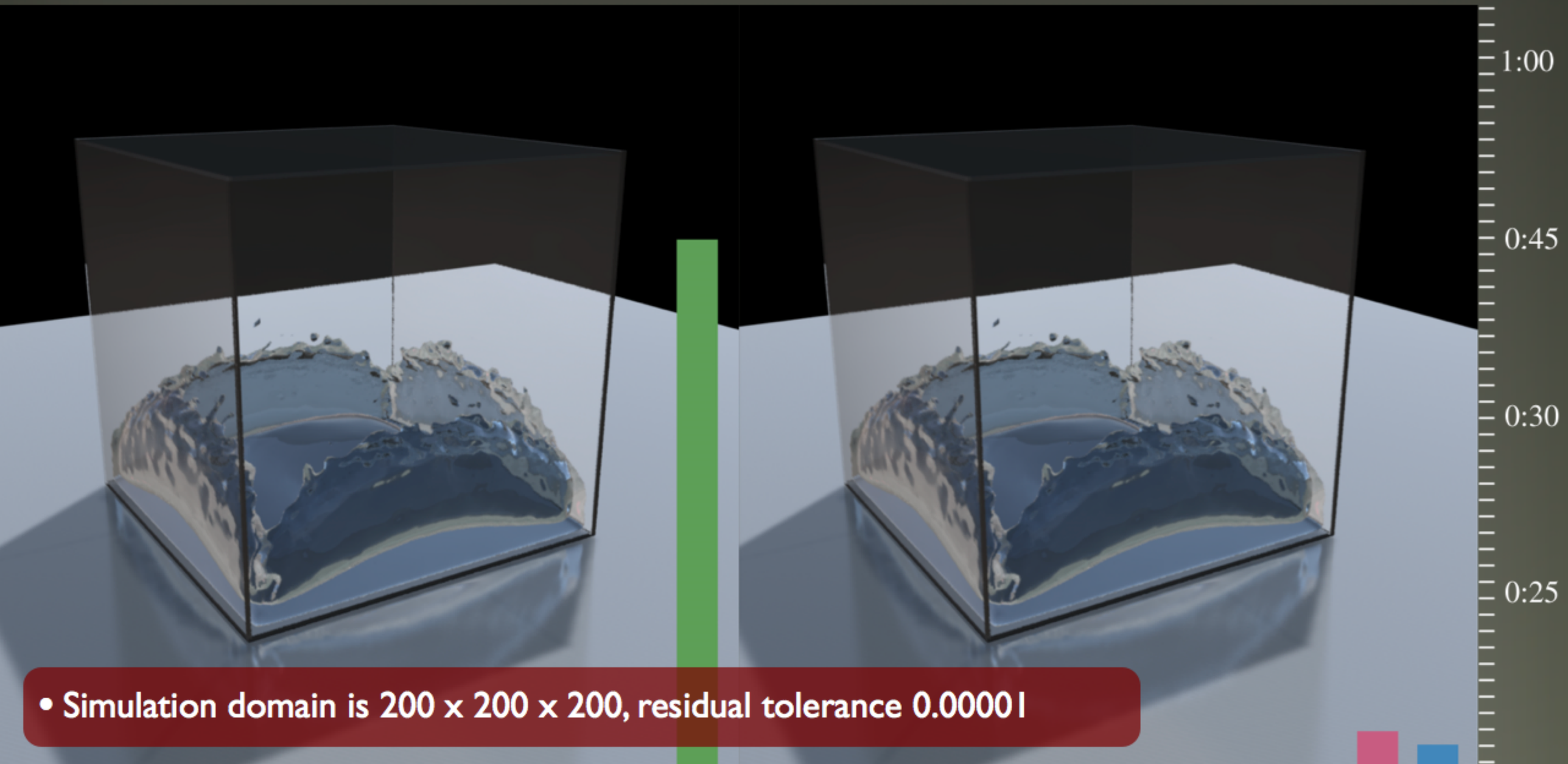
Use special instruction to force GPU to do multiply and addition as two operations

double negative visual effects



- Left side (green bar) is CPU, pink bar FX4800, blue bar C2050
- Simulation domain is 150 x 150 x 150, residual tolerance 0.1
- Both visually accurate though not identical due to differing algorithms

double negative visual effects



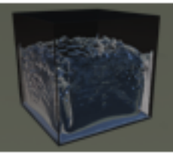
- Simulation domain is 200 x 200 x 200, residual tolerance 0.00001

double negative visual effects

1.95 MV (125 x 125 x 125) Residual Tolerance = 0.1 CPU Time = 43m 14s

	Transfer	Projection	Speed Up	Total Speed Up
FX4800	9m 53s	3m 34s	12.1 x	3.2 x
C2050	8m 12s	2m 11s	19.8 x	4.2 x

8.0 MV (200 x 200 x 200) Residual Tolerance = 0.00001 CPU Time = 10h 57m

	Transfer	Projection	Speed Up	Total Speed Up
FX4800	18m 51s	55m 5s	11.9 x	8.9 x
C2050	19m 9s	32m 18s	20.3 x	12.8 x

Double Negative now has a GPU renderfarm!

Currently 15 machines in a air conditioned room on site:

- 14 machines with Quadro FX4800 GPUs
- 1 machine with a Tesla C1060 GPU

Artists are gaining confidence in using this new technology

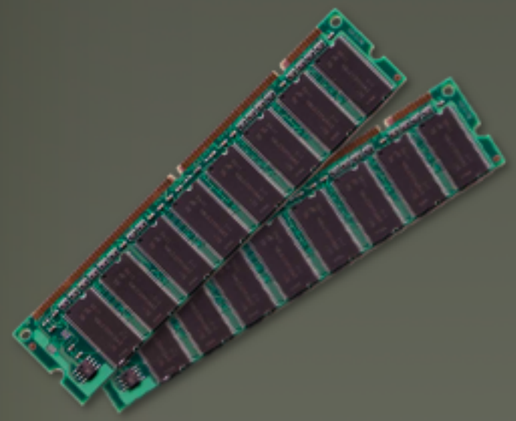
Issues with retaining the same look during development

Scalability

Maximum simulation domain sizes:

Important to be scalable for the future

Quadro FX4800	Tesla C1060
16.5 MV	38 MV
$\approx 255^3$	$\approx 335^3$



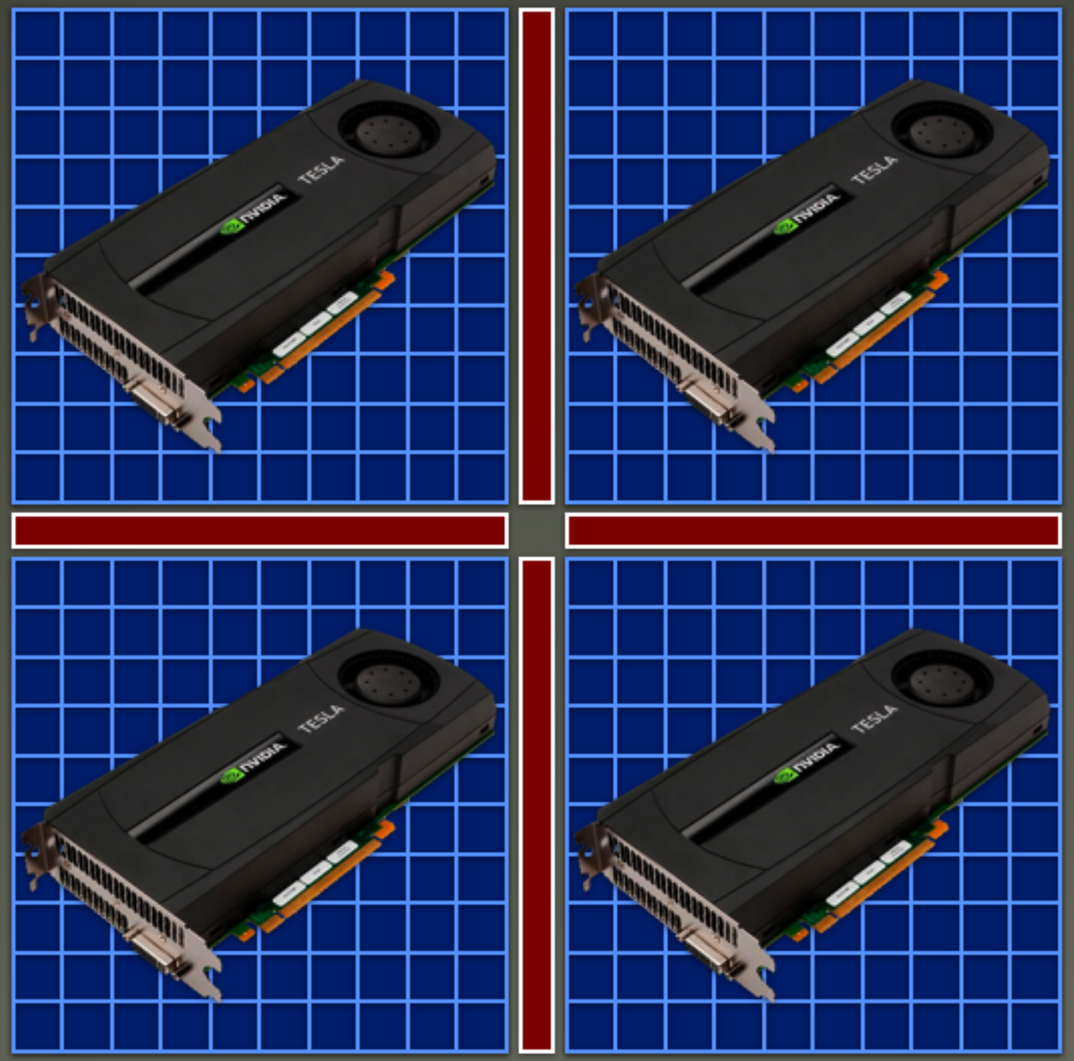
5-6 GB/s

Global Memory

144 GB/s
coalescing

Shared Memory

Scalability



- Multi-GPU machines allow for much larger domain sizes
- Memory transfer now only required for edge cells
- Provides flexibility for handling any project or simulation

double negative visual effects

GPU Strategy

- Always looking for better ways of parallel preconditioning our Poisson solver
- Pressure projection is now no longer the bottleneck
- Blocking layout an important step in moving more fluids computation to GPU
- Scalability needs further investigation
- Confidence of artists in using the GPU has increased