# LATTICE-BOLTZMANN
## AND COMPUTATIONAL FLUID DYNAMICS

# NAVIER-STOKES EQUATIONS

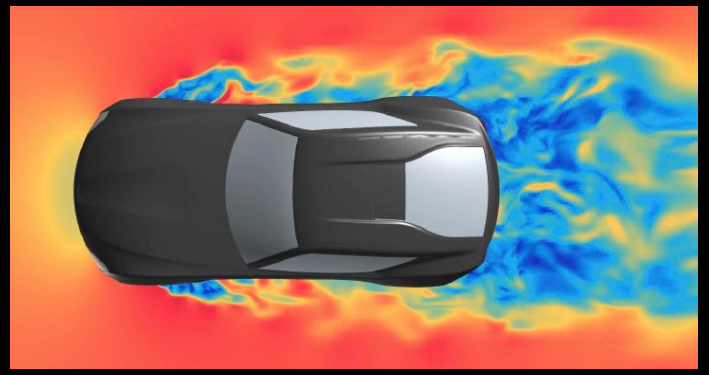
$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u}$$

$$\nabla \cdot \vec{u} = 0$$

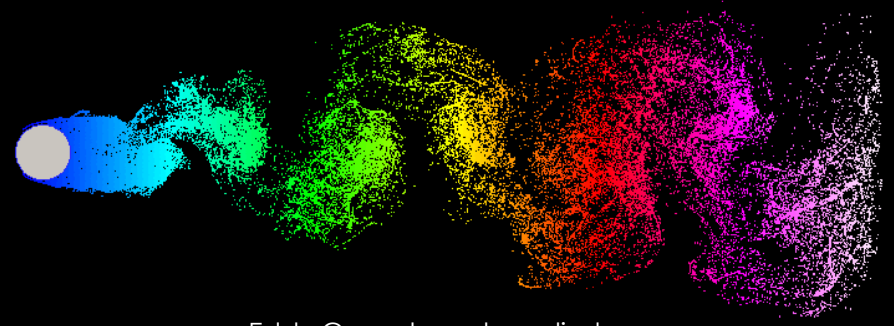
THEY ARE HARD :)

# WHAT IS COMPUTATIONAL FLUID DYNAMICS?

- Branch of Fluid Dynamics which uses computer power to approximate the solutions to equations, because many of the equations describing fluids are very difficult or even impossible to solve exactly.
- E.g. Wind tunnels for the aerodynamic design of aircrafts and automobiles, or simulation of ocean currents



Fluid flowing behind a car

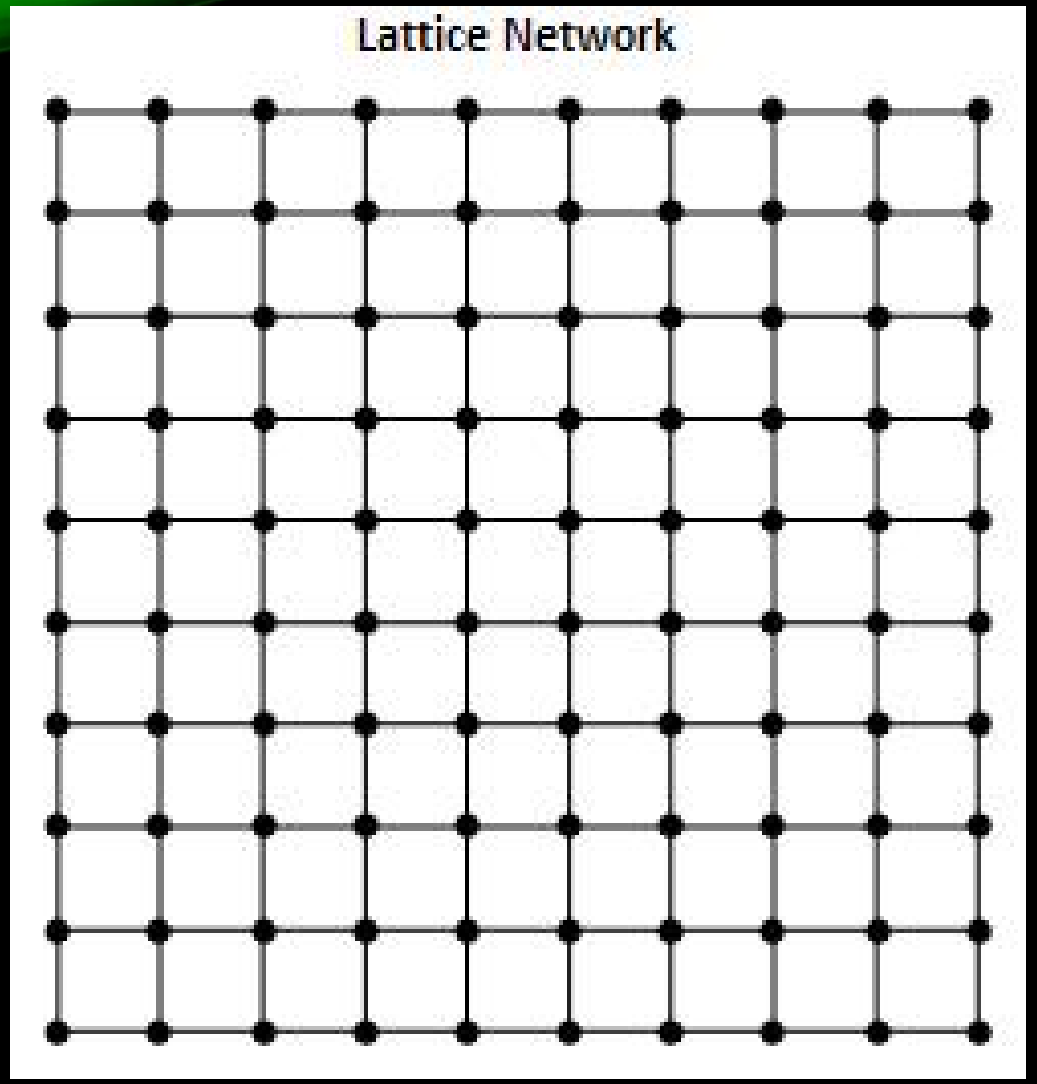


Eddy Currents past a cylinder

# LATTICE-BOLTZMANN ALGORITHM

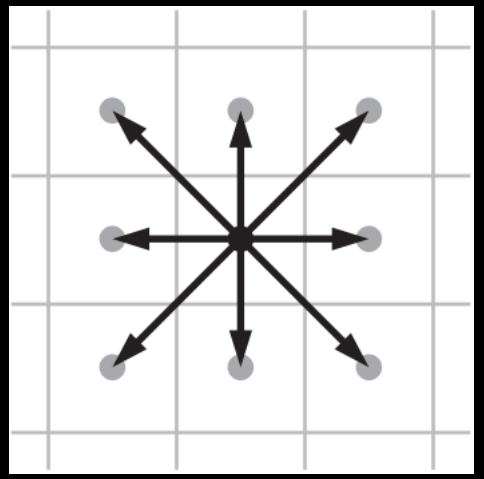
The Discretization of Time and Space of ideal fluids into one neat little algorithm

# THE LATTICE



# LATTICE-BOLTZMANN IN 2D

- The Lattice-Boltzmann algorithm in 2D is the called D2Q9
- All of the flow is directed into 9 vectors
  - 0 vector
  - Horizontal and Vertical Vectors
  - 45° Vectors
- Weights are used to determine the likelihood of the fluid to go in the direction of the vector



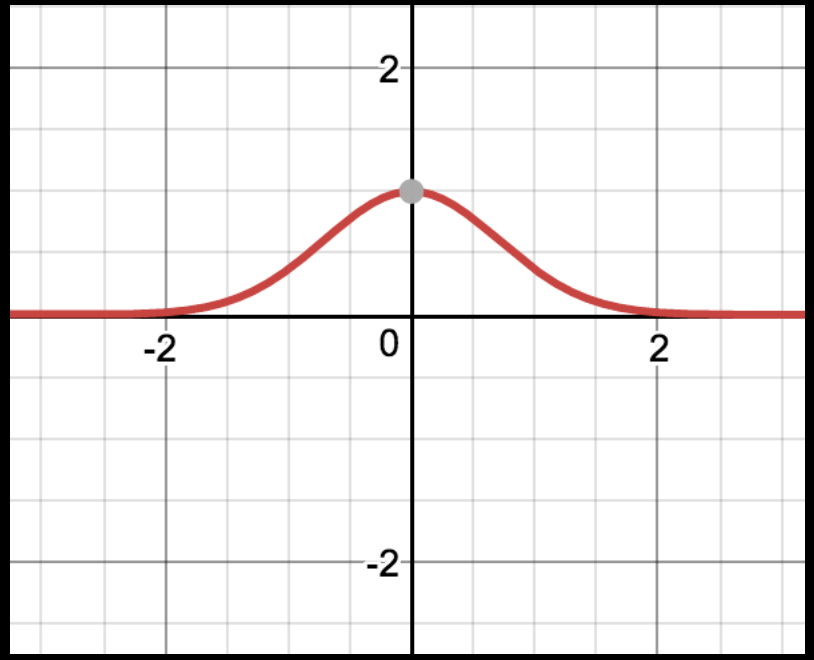
$$e_0 = 0$$
$$e_1 = (1,0) \quad e_5 = (1,1)$$
$$e_2 = (0,1) \quad e_6 = (-1,1)$$
$$e_3 = (-1,0) \quad e_7 = (-1,-1)$$
$$e_4 = (0,-1) \quad e_8 = (1,-1)$$

$$w_0 = \frac{4}{9},\ w_1 = w_2 = w_3 = w_4 = \frac{1}{9},\ w_5 = w_6 = w_7 = w_8 = \frac{1}{9}$$

# MAXWELL-BOLTZMANN DISTRIBUTION

- The Maxwell-Boltzmann distribution describes the average distribution of non-interacting material particles over various energy states in equilibrium.

$$D(\vec{v}) = \frac{m}{2\pi kT} e^{-\frac{m|\vec{v}|^2}{2kT}}$$



# DERIVING THE WEIGHTS

- When this equation is integrated over a velocity range and multiplied by a directional velocity, it gives the system of equation to solve for the weights (the 9 vectors in the star)
- For example:

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} v_x^2 D(\vec{v}) dv_x dv_y \sum_{i=0}^{8} (e_{i,x} \cdot c)^2 w_i$$

- This equality only works under the assumption that $c^2 = \frac{3kT}{m}$

# INITIAL CONDITIONS

At positions where there is fluid

- $n_0 = \frac{4}{9}(1 - 1.5v^2)$
- $n_E = \frac{1}{9}(1 - 3v + 3v^2)$
- $n_W = \frac{1}{9}(1 - 3v + 3v^2)$
- $n_N = \frac{1}{9}(1 - 1.5v^2)$
- $n_S = \frac{1}{9}(1 - 1.5v^2)$
- $n_N = \frac{1}{9}(1 - 1.5v^2)$
- $n_{NE} = \frac{1}{36}(1 + 3v + 3v^2)$
- $n_{NE} = \frac{1}{36}(1 + 3v + 3v^2)$
- $n_{NE} = \frac{1}{36}(1 - 3v + 3v^2)$
- $n_{NE} = \frac{1}{36}(1 - 3v + 3v^2)$
- $density = 1$

At the position where there is a wall

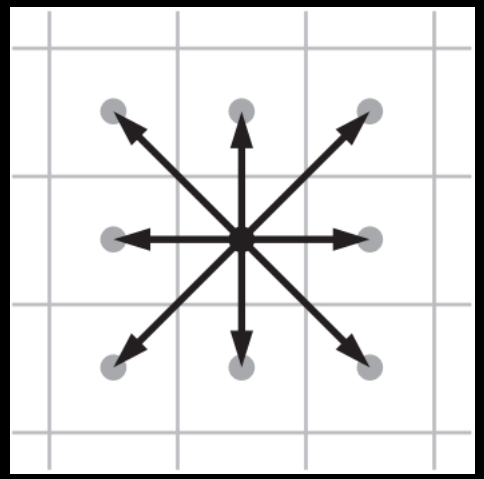
Everything is zero!

# LATTICE-BOLTZMANN ALGORITHM (3 STEPS)

- Collisions
- Streaming
- Boundary Conditions

# COLLISIONS

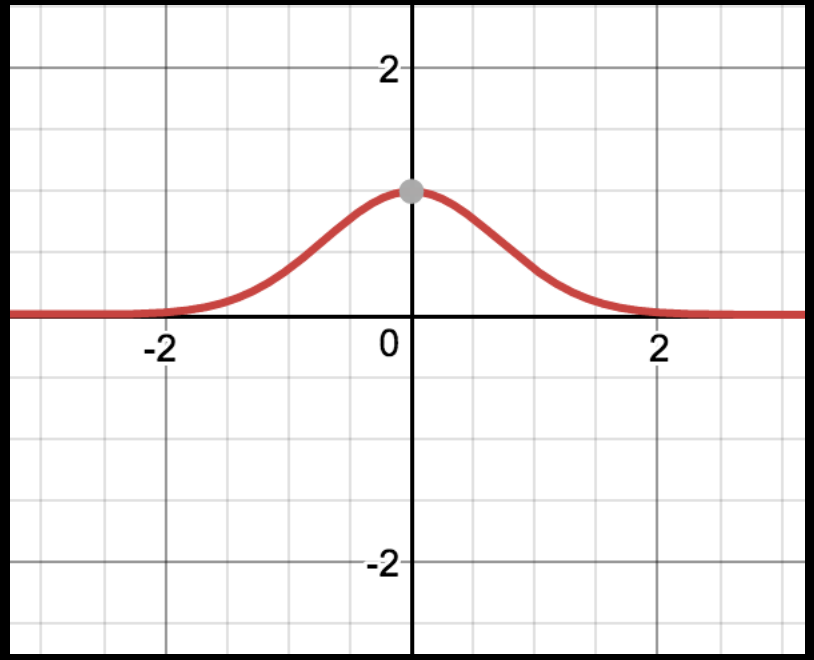
- Inside the collision step, the simulation uses the redistributes all the velocity vectors back into the 9 discrete number density vectors.
- The collision step is the most computationally intensive step in the algorithm.
- To think about the collision step conceptually, think about this step as the redistribution of the incoming vectors back into the four directions



# MAXWELL-BOLTZMANN DISTRIBUTION

- The Maxwell-Boltzmann distribution describes the average distribution of non-interacting material particles over various energy states in equilibrium.

$$D(\vec{v}) = \frac{m}{2\pi kT} e^{-\frac{m|\vec{v}|^2}{2kT}}$$



What about when the particles are not in thermal equilibrium?

# GENERALIZING THE FORMULA

- The velocity for each particle is a combination of the the macroscopic velocity and the thermal velocity:

$$\vec{e}_i \cdot c = \vec{u} + \vec{v} \qquad \text{or} \qquad \vec{v} = \vec{e}_i \cdot c - \vec{u}$$

- Then plug updated description into the distribution

$$D(\vec{v}) = \frac{m}{2\pi kT} e^{-\frac{m|\vec{e}_i \cdot c - u\ |^2}{2kT}}$$

# GENERALIZING THE FORMULA PART II

- Then to approximate the distribution function, use a the Taylor Series for $e^x$:

$$D(\vec{v}) \approx w_i(1 + \frac{3\vec{e}_i \cdot \vec{u}}{c} + \frac{9}{2} \cdot \left(\frac{\vec{e}_i \cdot \vec{u}}{c}\right)^2 - \frac{3}{2} \cdot \frac{\vec{u}^2}{c^2})$$

- Use the definition of macroscopic velocity densities to get the macroscopic densities for each particle:

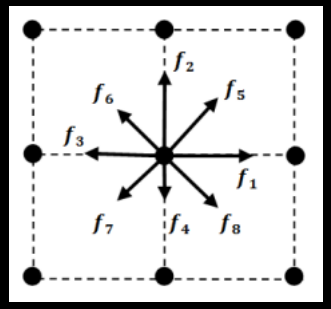
$$n_i^{eq} = \rho D(\vec{v}) \qquad n_i^{eq} = \rho[w_i\left(1 + \frac{3\vec{e}_i \cdot \vec{u}}{c} + \frac{9}{2} \cdot \left(\frac{\vec{e}_i \cdot \vec{u}}{c}\right)^2 - \frac{3}{2} \cdot \frac{\vec{u}^2}{c^2}\right)]$$

- Finally, the formula can be used:

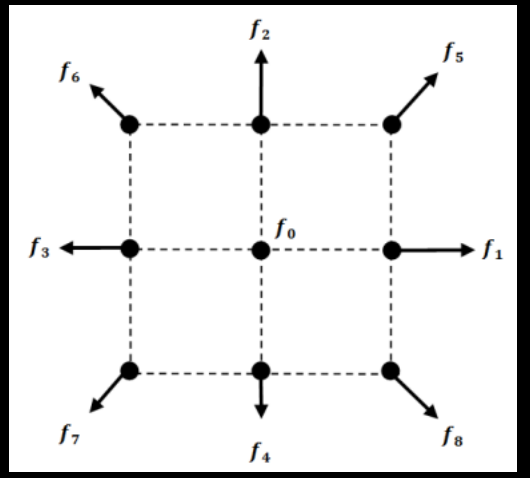
$$n_i^{new} = n_i^{old} + \omega(n_i^{eq} - n_i^{old})$$

# STREAMING

- After the vectors collide and yield the correct number densities, This step is very simple – the program "streams" the vectors to adjacent steps
- This step only works under the assumption that $|u| \ll c$ so that particles can only stream one lattice point away.

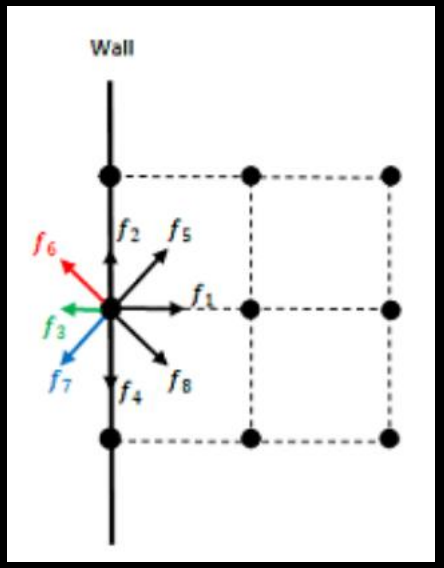


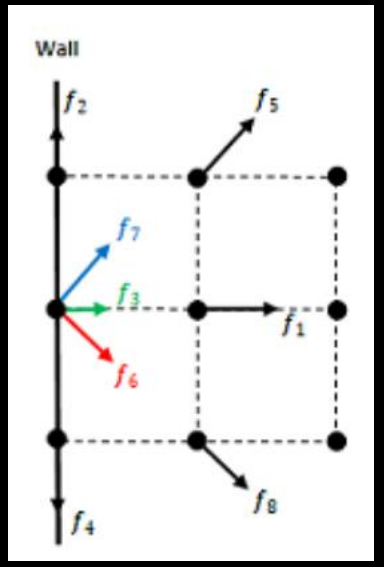

(It's a stream)



# BOUNDARY CONDITIONS

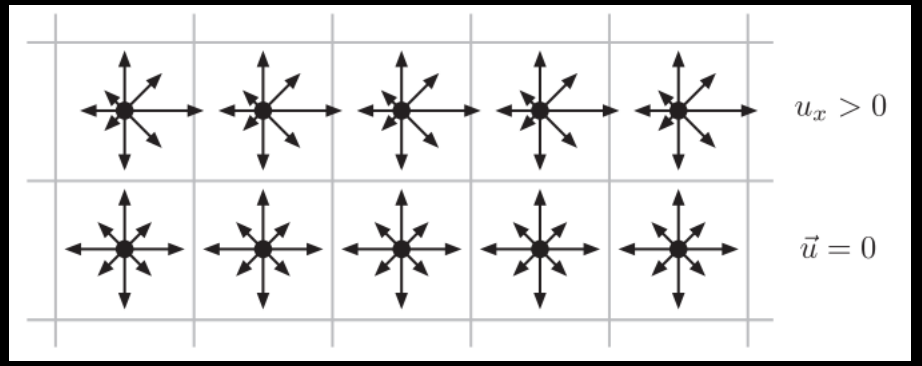
- In this simulation, we use the "bounce-back" algorithm to handle cases where particles collide with boundaries.
- When a vector hits a boundary, the vector is then sent in the opposite direction



# VISCOSITY

- Viscosity can be seen as the loss of the factor which energy is lost.
- As seen in the scenario below, the fluid in the top layer is flowing to the right, but the fluid in the bottom layer has no net motion.
- So if the fluid is viscous, the upper level will better drag the lower layers along with it.
- Viscous solutions accelerate the process of the reaching equilibrium as seen in the step function for collisions




$$n_i^{new} = n_i^{old} + \omega(n_i^{eq} - n_i^{old})$$

# CUDA AND OPENGL
## WITH LATTICE-BOLTZMANN

# WHY DO GRAPHICS ACCELERATION?


Enhances performance for algorithms that are:

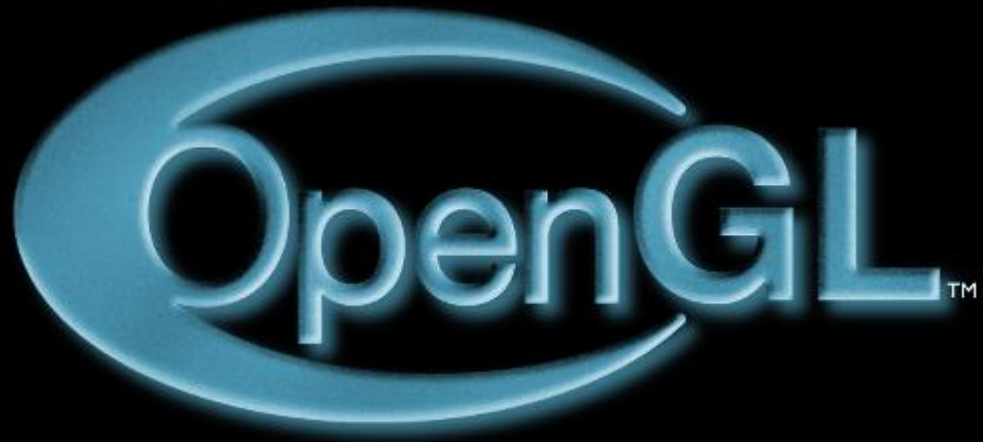
-massively parallel
-order independent
-computation intensive

LBM is all three of these things!

# SOME GPU TERMS:

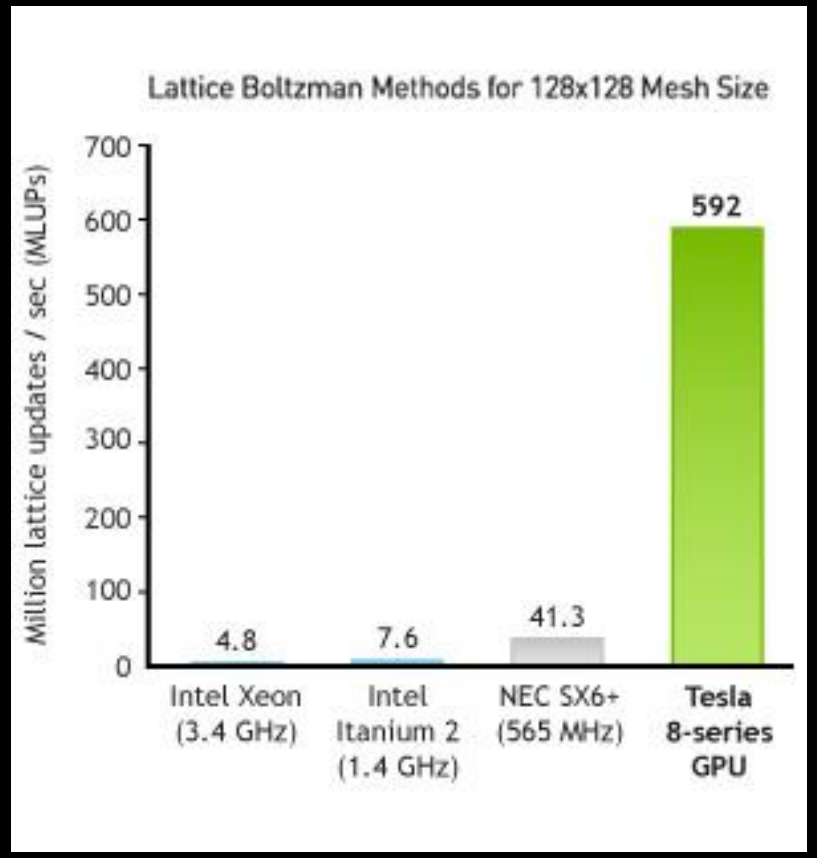
- CUDA - Compute Unified Device Architecture
  - Proprietary (Nvidia)
  - **Computation** focused


- OpenGL – Open Graphics Language
  - Open Source!
  - **Graphics** focused

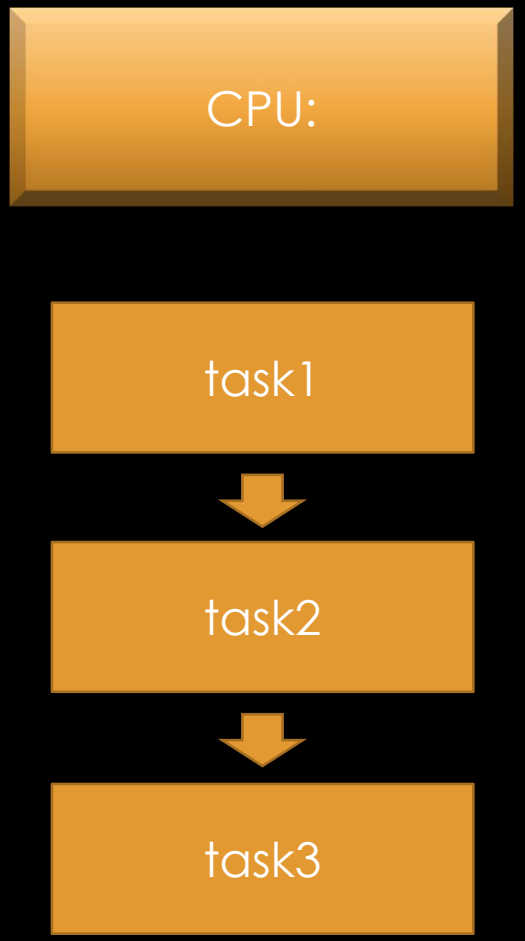


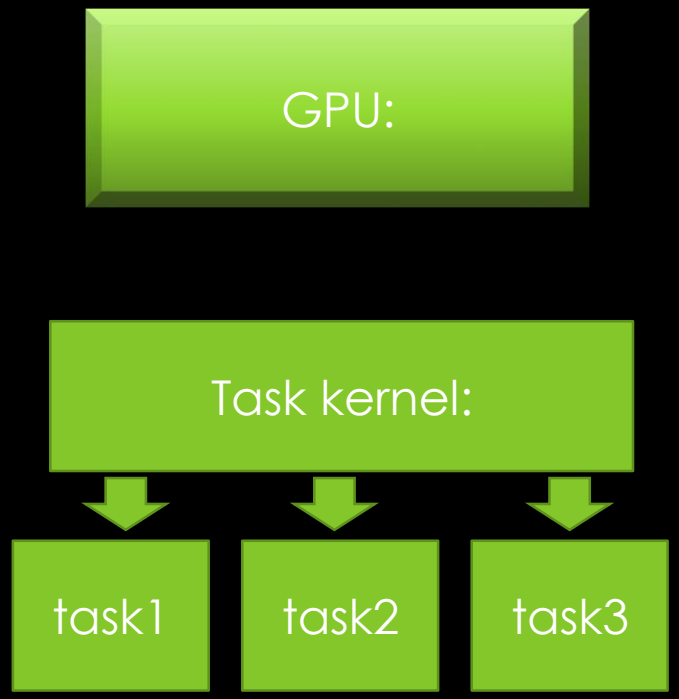

- (OpenCL – Open Compute Language)
  - Open Source!
  - **Computation** focused
  - We did **not** use this!

# PARALLEL VS SERIAL COMPUTING:




Courtesy Nvidia



# PARALLEL PROGRAMMING LEVELS

Course

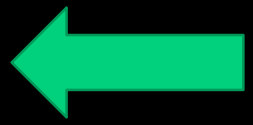
Fine

Grid/Cluster computing

MPI (Message Passing Interface)

Multi-threading

SIMD (single instruction multiple data)

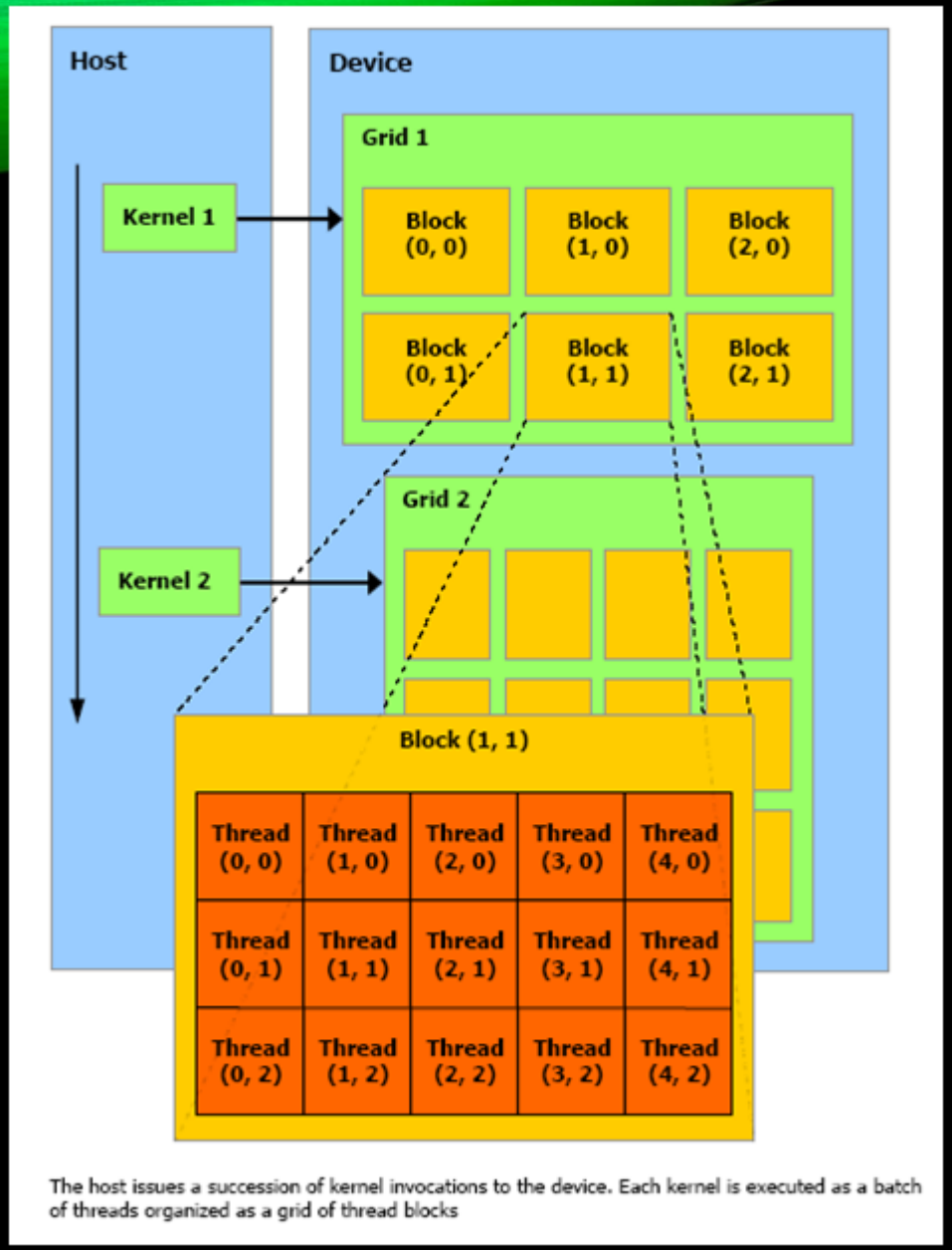


**We are here**

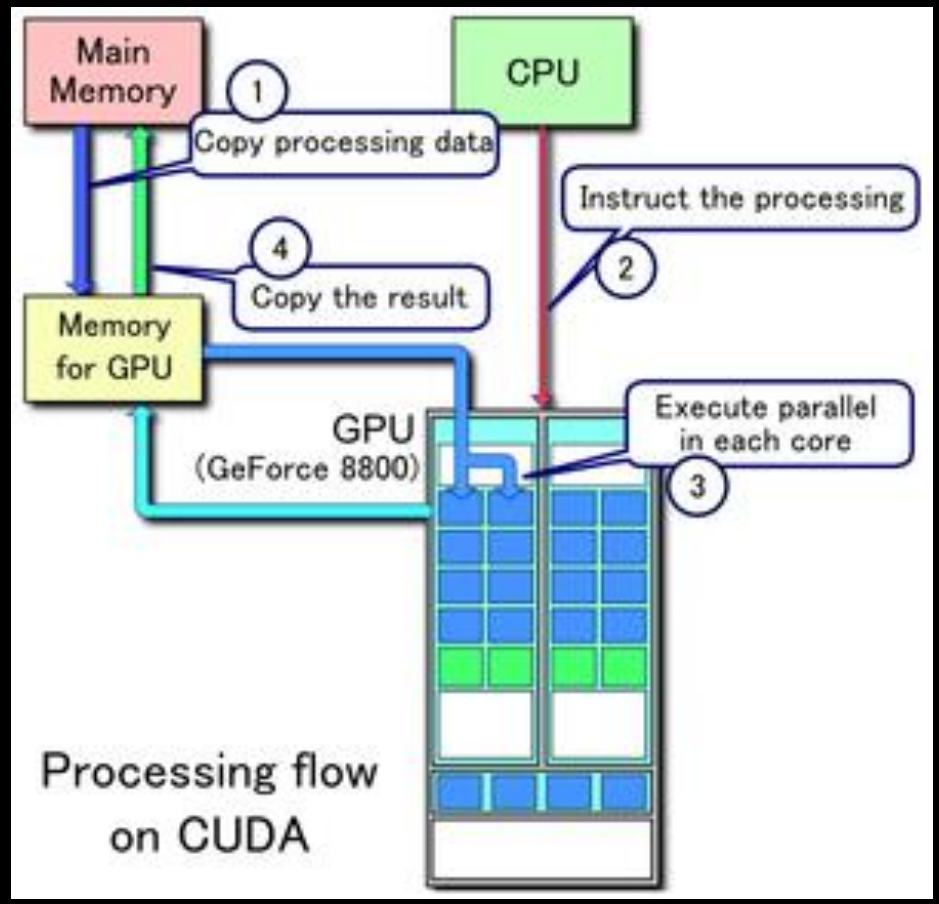
# VIDEO:

Courtesy Nvidia

# HOW CUDA WORKS:

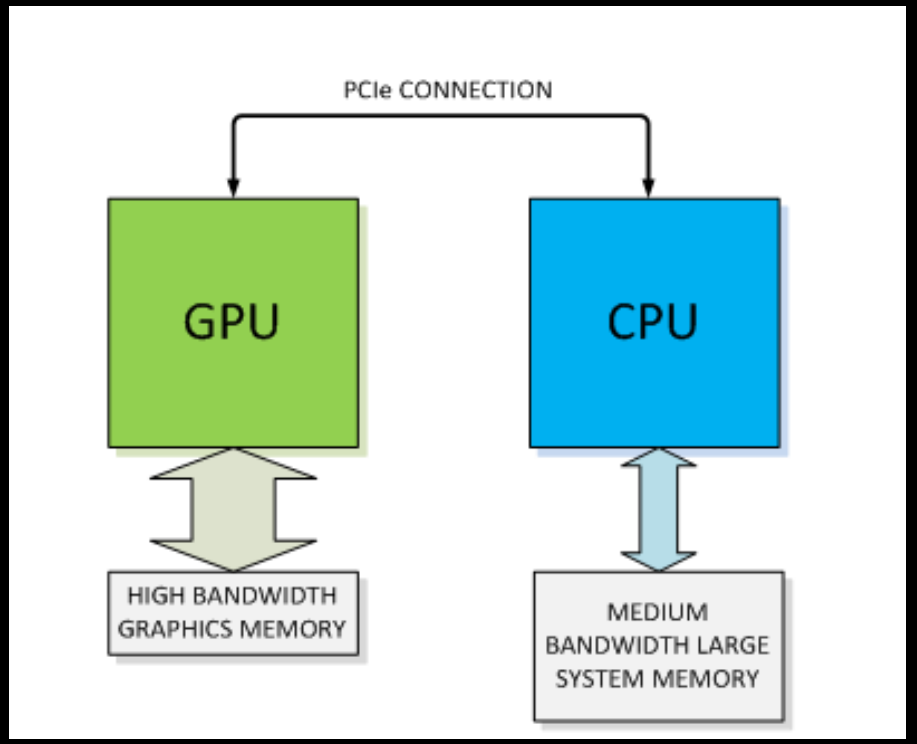



Courtesy Colorado School of Mines




Courtesy Wikimedia

# THE IMPORTANCE OF MEMORY



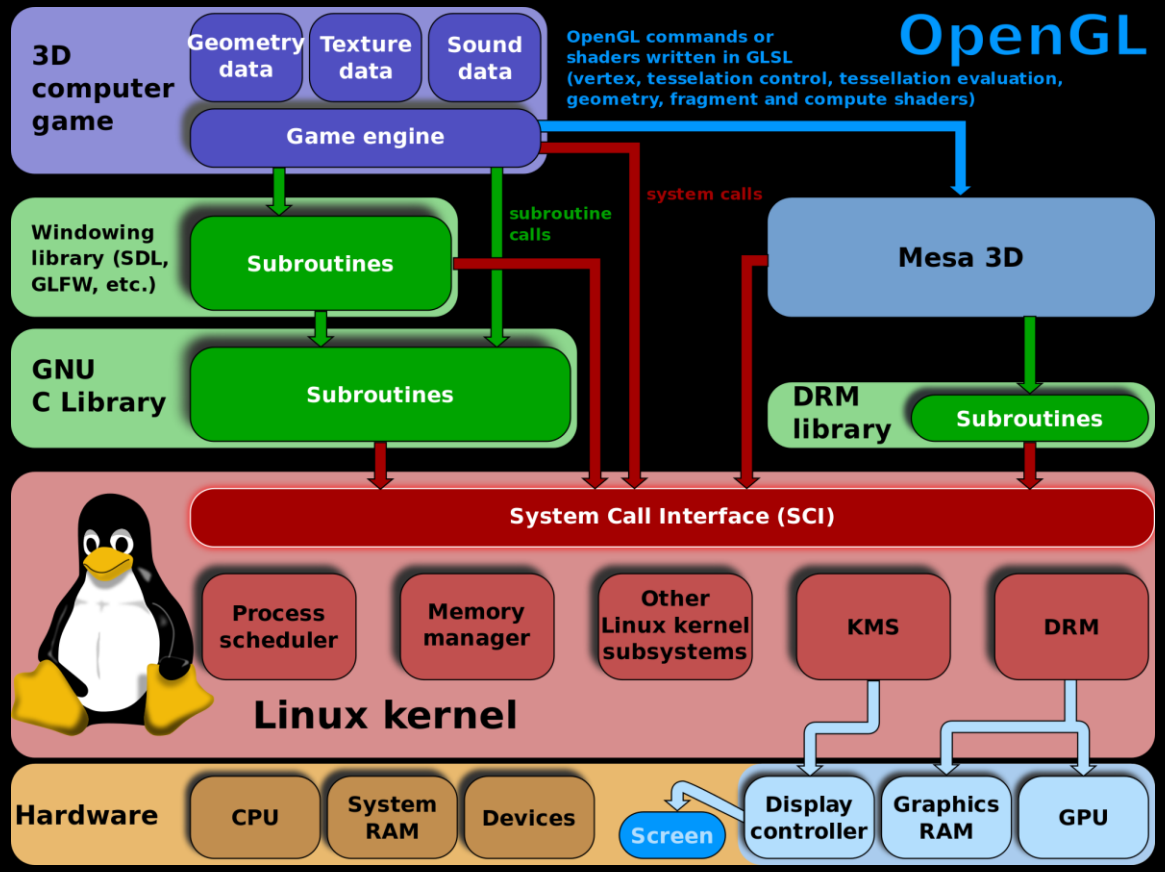

Courtesy Nvidia

PCIe is VERY slow:
5-10GB/s

GPU and CPU memory access is VERY fast:
up to 150GB/s (GPU)

So…
-Avoid repeated GPU/CPU Memcpy calls
-Prestore all vital data on GPU
-Keep frame data on GPU

# WHY OPENGL?




Courtesy Wikimedia

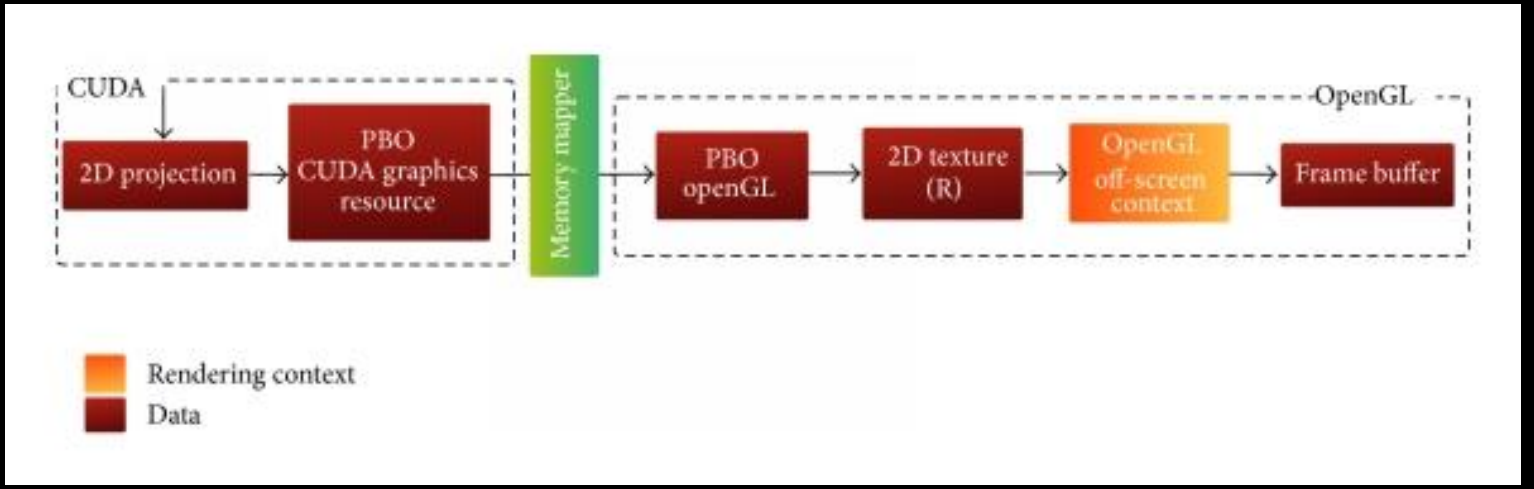
-Platform independent!
  -OSX, Linux, and Windows!

-Low-Level 3d graphics engine
  **-FAST!!!**

-Supports CUDA/gl interop!
  -**data never leaves GPU**

**Games** that use opengl:
-Left 4 Dead
-Doom 3
-Portal
-Counter Strike
-Far Cry
-Hitman
-and many more!

# CUDA/GL INTEROP EXPLAINED:




Courtesy ResearchGate

-All memory resides inside GPU! (so it's fast!)
-Double buffering prevents flickering

-PBO's use uchar4's, so 1 byte each for R,G,B and Alpha channels

-2d textures are mapped onto screen-sized rectangle object
    -rendered with an ortholinear matrix (because 3d…)

# OUR SOLUTION:

- The program:

- 1 node = 1 thread
  - 1080p sim = ~2 million threads!
  - $O(n^2)$
- 3 kernels:
  - Collide
  - Stream
  - Bounceback
- And lots of CPU-side functions!

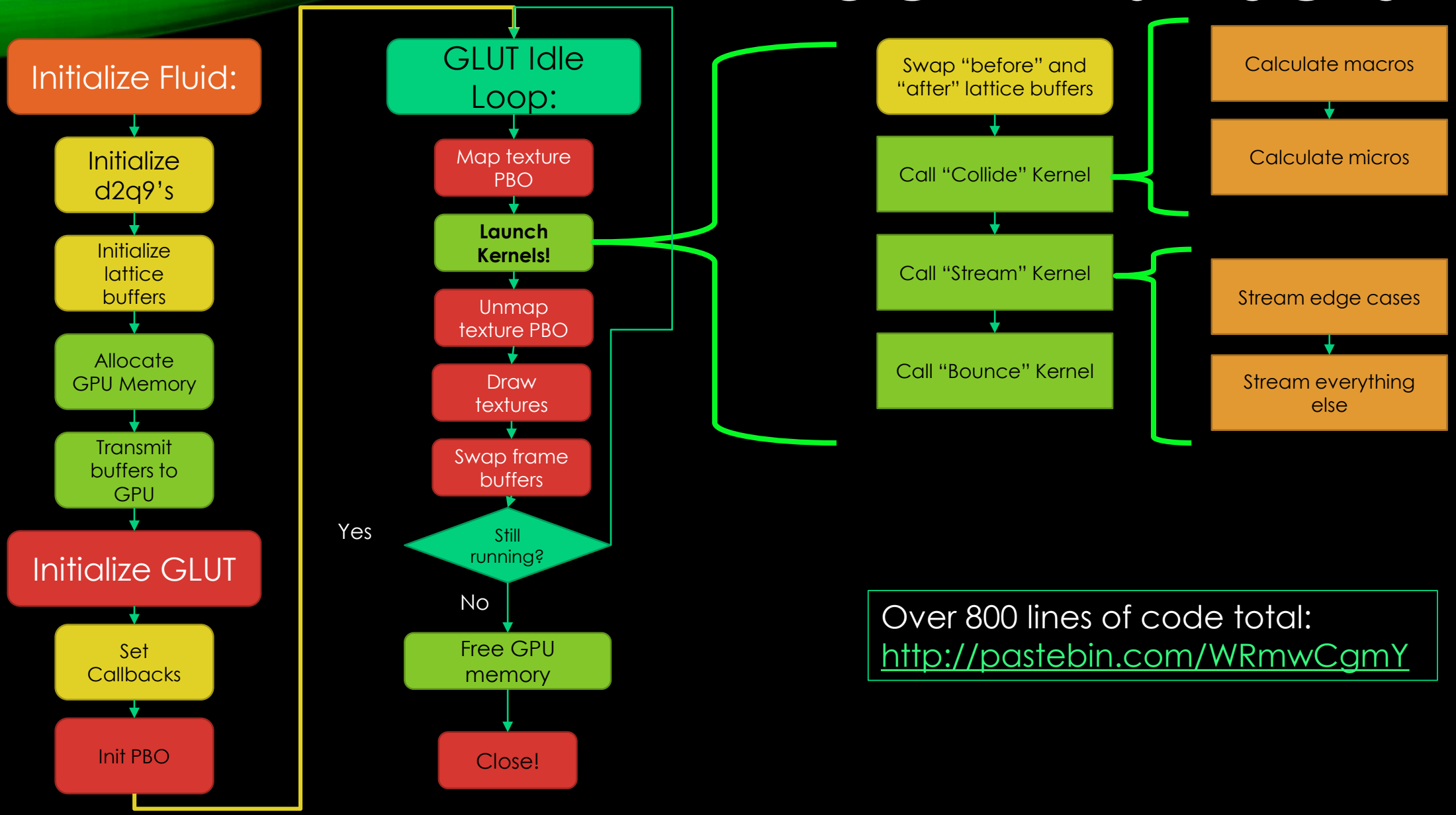
- Our Setup:

- Programmed with C++ and Nvidia Nsight for Visual Studio

- Hardware:
  - GTX 1060 and i5 CPU

- Can't run it on a laptop…

# PROGRAM STRUCTURE:




Over 800 lines of code total:
http://pastebin.com/WRmwCgmY

# DIFFICULTIES OF CUDA:

- Crappy debugging
  - Custom error handling code is necessary
  - No step through or anything AFAIK
  - Print statements are a pain
- Everything is more complicated
  - Memory allocation, kernel setup, and cpu/gpu interop
  - No access to non-gpu variables
- Interop with opengl is incredibly complex to get running
  - PBO's, texture memory, and memory mapping ☹

# THANKS!