Lattice Boltzmann Method

CUDA Programming

Ravil Dorozhinskii, Hasan Ashraf

Computational Science and Engineering

14 June 2018

Contents

Theory

What is LBM?

Lattice Boltzmann Methods (LBM) are a class of computational fluid dynamics methods for fluid simulation. [1]

Computational fluid dynamics (CFD) is a branch of fluid mechanics that uses numerical analysis and programming to solve and analyze problems that involve fluid flows.

Making different assumption about fluid structure leads us to different mathematical models.

What is LBM?

Lattice Boltzmann Methods (LBM) are a class of computational fluid dynamics methods for fluid simulation. [1]

Computational fluid dynamics (CFD) is a branch of fluid mechanics that uses numerical analysis and programming to solve and analyze problems that involve fluid flows.

Making different assumption about fluid structure leads us to different mathematical models.

What is LBM?

Lattice Boltzmann Methods (LBM) are a class of computational fluid dynamics methods for fluid simulation. [1]

Computational fluid dynamics (CFD) is a branch of fluid mechanics that uses numerical analysis and programming to solve and analyze problems that involve fluid flows.

Making different assumption about fluid structure leads us to different mathematical models.

What is LBM?

Lattice Boltzmann Methods (LBM) are a class of computational fluid dynamics methods for fluid simulation. [1]

Computational fluid dynamics (CFD) is a branch of fluid mechanics that uses numerical analysis and programming to solve and analyze problems that involve fluid flows.

Making different assumption about fluid structure leads us to different mathematical models.

Navier-Stokes - Macroscopic approach

This approach is based on the assumption that the fluid is **a continuum**: made up of a **continuous** substance

Molecular Dynamics - Microscopic approach

The microscopic approach treats the fluid as a set of **separate molecules**. The interaction between molecules is a result of attractive and repulsive forces generated by the molecules themselves.

What is LBM?

Lattice Boltzmann Methods (LBM) are a class of computational fluid dynamics methods for fluid simulation.

Computational fluid dynamics (CFD) is a branch of fluid mechanics that uses numerical analysis and programming to solve and analyze problems that involve fluid flows.

Making different assumption about fluid structure leads us to different mathematical models.

Lattice Boltzmann Methods - Mesoscopic approach

The model represents fluid as a system of particles. **The system is described as a probability distribution over particles**. Each particle has a **probability density f(x,v,t)** which represents the probability that the particle is at position **x** with velocity **v** at time **t**.

Governing equation of LBM

This mesoscopic approach is described by **kinetic theory** which is based on the Boltzmann equation.

The derivative of the probability distribution over time can be described by the so-called **collision operator** $\Omega(t)$

$$\lim_{\delta t \to 0} \frac{1}{\delta t} \left(f \left(\mathbf{x} + \mathbf{v}\delta t, \mathbf{v} + \frac{\mathbf{F}}{m}\delta t, t + \delta t \right) - f(\mathbf{x}, \mathbf{v}, t) \right) = \Omega(t)$$

or

Governing equation of LBM

This mesoscopic approach is described by **kinetic theory** which is based on the Boltzmann equation.

The derivative of the probability distribution over time can be described by the so-called **collision operator** $\Omega(t)$

$$\lim_{\delta t \to 0} \frac{1}{\delta t} \left(f\left(\mathbf{x} + \mathbf{v}\delta t, \mathbf{v} + \frac{\mathbf{F}}{m}\delta t, t + \delta t \right) - f(\mathbf{x}, \mathbf{v}, t) \right) = \Omega(t)$$

or

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla_x + \frac{\mathbf{F}}{m} \cdot \nabla_v \right) f(\mathbf{x}, \mathbf{v}, t) = \Omega(t)$$

where $\mathbf{F}$ is the accumulated external particle force.

Governing equation of LBM

This mesoscopic approach is described by **kinetic theory** which is based on Boltzmann's equation.

The difference of probability distribution over time can be described by so-called **collision operator** $\Omega(t)$

$$\lim_{\delta t \to 0} \frac{1}{\delta t} \left(f\left(\mathbf{x} + \mathbf{v}\delta t, \mathbf{v} + \frac{\mathbf{F}}{m}\delta t, t + \delta t \right) - f(\mathbf{x}, \mathbf{v}, t) \right) = \Omega(t)$$

or

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla_x + \frac{\mathbf{F}}{m} \cdot \nabla_v \right) f(\mathbf{x}, \mathbf{v}, t) = \Omega(t)$$

where $\mathbf{F}$ is the accumulated external particle force.

A crucial assumption of our model is that **the external force is equal to zero**. Hence,

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla_x \right) f(\mathbf{x}, \mathbf{v}, t) = \Omega(t)$$

Discretization: velocity space

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla_x\right) f(\mathbf{x}, \mathbf{v}, t) = \Omega(t)$$

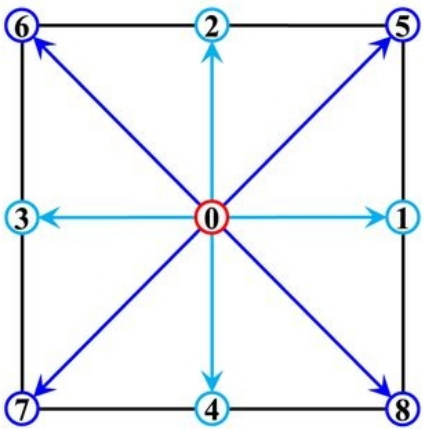
The governing equation is quite **complex** despite its seemingly simple form.

Discretization: velocity space

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla_x\right) f(\mathbf{x}, \mathbf{v}, t) = \Omega(t)$$

The governing equation is pretty **complex** despite its seemingly simple form.

We discretize it in velocity space along **nine directions**:

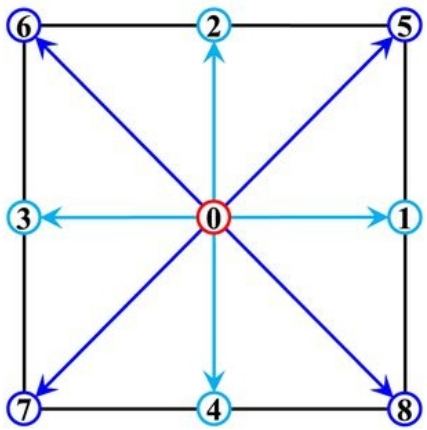


Discretization: velocity space

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla_x\right) f(\mathbf{x}, \mathbf{v}, t) = \Omega(t)$$

The governing equation is pretty **complex** despite its seemingly simple form

We discretize it in velocity space along **nine directions**:



Which leads us to the following equation:

$$\left(\frac{\partial}{\partial t} + e_i \cdot \nabla_x\right) f(\mathbf{x}, e_i, t) = \Omega(t)$$

where e_i is a vector along each direction

Discretization: time

We can then discretize

$$\left(\frac{\partial}{\partial t} + e_i \cdot \nabla_x\right) f(\mathbf{x}, e_i, t) = \Omega(t)$$

over time and write a separate expression for each e_i :

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) - f_i(\mathbf{x}, t) = \Omega_i(t)$$

Discretization: time

We can then discretize

$$\left(\frac{\partial}{\partial t} + e_i \cdot \nabla_x\right) f(\mathbf{x}, e_i, t) = \Omega(t)$$

over time and write a separate expression for each e_i :

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) - f_i(\mathbf{x}, t) = \Omega_i(t)$$

Assumption

The **collision term** Ω turns out to be **very complex** in general. It can be simplified by using a well-known and widely used assumption, the Bhatnagar-Gross-Krook (**BGK**) [3] assumption. It says that the *collision term induces the particle distribution to decay slowly to its equilibrium distribution* f^{eq}.

$$\Omega_i = \frac{1}{\tau}\left(f_i^{\mathbf{eq}} - f_i\right)$$

Discretization: time

- The equilibrium distribution is **a smoothed version of the current distribution**, recomputed from the bulk properties ρ and $\mathbf{u}$ (bulk velocity) [2].
- The equilibrium represents **an expected distribution** which would result in the observed bulk properties [2].
- According to **the BGK assumption**, the equilibrium distribution can be represented by the following:

Discretization: time

- The equilibrium distribution is **a smoothed version of the current distribution**, recomputed from the bulk properties ρ and **u** (bulk velocity) [2].
- The equilibrium represents **an expected distribution** which would result in the observed bulk properties [2].
- According to **the BGK assumption**, the equilibrium distribution can be represented by the following:

$$f_i^{\mathrm{eq}} = \omega_i \rho \left(1 + 3 \frac{e_i \cdot u}{c} + \frac{9}{2} \frac{(e_i \cdot u)^2}{c^2} - \frac{3}{2} \frac{u \cdot u}{c^2} \right)$$

where ω_i - a weighting function dependent on the exact velocity discretization; $c = \frac{1}{\sqrt{3}}$ - the *lattice* speed of sound

Jump from Mesoscopic to Macroscopic World

Let's take a closer look at the equilibrium distribution equation

$$f_i^{\mathrm{eq}} = \omega_i \rho \left(1 + 3 \frac{e_i \cdot u}{c} + \frac{9}{2} \frac{(e_i \cdot u)^2}{c^2} - \frac{3}{2} \frac{u \cdot u}{c^2} \right)$$

To compute this equation, we need to know the bulk **density** and the **velocity**

Jump from Mesoscopic to Macroscopic World

Let's take a closer look at the equilibrium distribution equation

$$f_i^{\mathrm{eq}} = \omega_i \rho \left(1 + 3\frac{e_i \cdot u}{c} + \frac{9}{2}\frac{(e_i \cdot u)^2}{c^2} - \frac{3}{2}\frac{u \cdot u}{c^2}\right)$$

To compute this equation, we need to know the bulk **density** and the **velocity**

These can be computed as follows:

$$\rho(x,t) = \sum_{i=1}^{9} f_i(x,t)$$

$$\rho(x,t)\vec{u}(x,t) = \sum_{i=1}^{9} \vec{e_i} f_i(x,t)$$

Jump from Mesoscopic to Macroscopic World

Let's take a closer look at the equilibrium distribution equation:

$$f_i^{\mathrm{eq}} = \omega_i \rho \left(1 + 3\frac{e_i \cdot u}{c} + \frac{9}{2}\frac{(e_i \cdot u)^2}{c^2} - \frac{3}{2}\frac{u \cdot u}{c^2}\right)$$

To compute this equation, we need to know the bulk **density** and the **velocity**

These can be computed as follows:

$$\rho(x,t) = \sum_{i=1}^{9} f_i(x,t)$$

$$\rho(x,t)\vec{u}(x,t) = \sum_{i=1}^{9} \vec{e_i} f_i(x,t)$$

IMPORTANT!

- All equation above are dimensionless.
- One has to use **scaling** to simulate a real liquid.

All together

The discretized governing equation

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) - f_i(\mathbf{x}, t) = \Omega_i(t)$$

along with the collision operator

$$\Omega_i = \frac{1}{\tau} \left(f_i^{\mathrm{eq}}(\rho(\mathbf{x}, t), u(\mathbf{x}, t)) - f_i(\mathbf{x}, t) \right)$$

where the equilibrium distribution equation has the following form:

$$f_i^{\mathrm{eq}} = \omega_i \rho \left(1 + 3 \frac{e_i \cdot u}{c} + \frac{9}{2} \frac{(e_i \cdot u)^2}{c^2} - \frac{3}{2} \frac{u \cdot u}{c^2} \right)$$

give us **the final equation** for each **direction** i :

All together

The discretized governing equation

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) - f_i(\mathbf{x}, t) = \Omega_i(t)$$

along with the collision operator

$$\Omega_i = \frac{1}{\tau} \left(f_i^{\mathrm{eq}}(\rho(\mathbf{x}, t), u(\mathbf{x}, t)) - f_i(\mathbf{x}, t) \right)$$

where the equilibrium distribution equation has the following form:

$$f_i^{\mathrm{eq}} = \omega_i \rho \left(1 + 3 \frac{e_i \cdot u}{c} + \frac{9}{2} \frac{(e_i \cdot u)^2}{c^2} - \frac{3}{2} \frac{u \cdot u}{c^2} \right)$$

give us **the final equation** for each **direction** i :

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) = f_i(\mathbf{x}, t) - \frac{1}{\tau} \left(f_i(\mathbf{x}, t) - f_i^{\mathrm{eq}}(\rho(\mathbf{x}, t), u(\mathbf{x}, t)) \right)$$

Pipeline

If you look at the final equation

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) = f_i(\mathbf{x}, t) - \frac{1}{\tau}\left(f_i(\mathbf{x}, t) - f_i^{\mathrm{eq}}(\rho(\mathbf{x}, t), u(\mathbf{x}, t))\right)$$

you can easily find two distinct steps, namely:

Pipeline

If you look at the final equation

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) = f_i(\mathbf{x}, t) - \frac{1}{\tau}\left(f_i(\mathbf{x}, t) - f_i^{\mathbf{eq}}(\rho(\mathbf{x}, t), u(\mathbf{x}, t))\right)$$

you can easily find two distinct steps, namely:
Collision:

$$f_i(\mathbf{x}, t)^* = f_i(\mathbf{x}, t) - \frac{1}{\tau}\left(f_i(\mathbf{x}, t) - f_i^{\mathbf{eq}}(\rho(\mathbf{x}, t), u(\mathbf{x}, t))\right)$$

Pipeline

Looking at the final equations

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) = f_i(\mathbf{x}, t) - \frac{1}{\tau}\left(f_i(\mathbf{x}, t) - f_i^{\mathrm{eq}}(\rho(\mathbf{x}, t), u(\mathbf{x}, t))\right)$$

Once can easily find two distinct steps, namely:

Collision:

$$f_i(\mathbf{x}, t)^* = f_i(\mathbf{x}, t) - \frac{1}{\tau}\left(f_i(\mathbf{x}, t) - f_i^{\mathrm{eq}}(\rho(\mathbf{x}, t), u(\mathbf{x}, t))\right)$$

Streaming:

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) = f_i(\mathbf{x}, t)^*$$

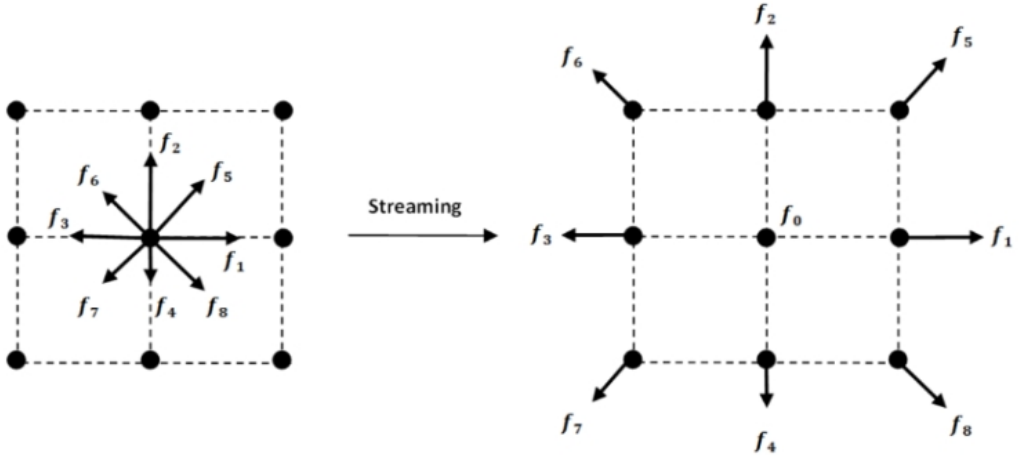
Pipeline

Collision:

$$f_i(\mathbf{x}, t)^* = f_i(\mathbf{x}, t) - \frac{1}{\tau}\left(f_i(\mathbf{x}, t) - f_i^{\mathrm{eq}}(\rho(\mathbf{x}, t), u(\mathbf{x}, t))\right)$$

Streaming:

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) = f_i(\mathbf{x}, t)^*$$



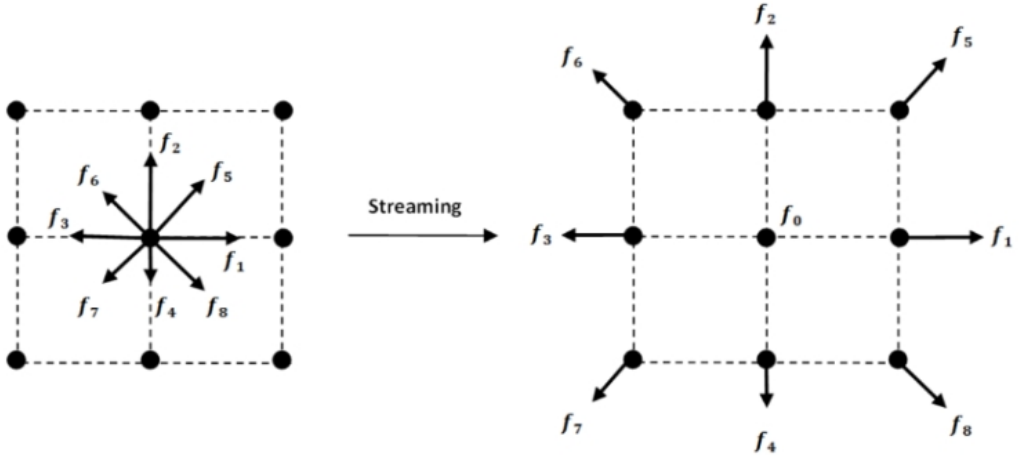
Pipeline

Collision:

$$f_i(\mathbf{x}, t)^* = f_i(\mathbf{x}, t) - \frac{1}{\tau}\left(f_i(\mathbf{x}, t) - f_i^{\mathrm{eq}}(\rho(\mathbf{x}, t), u(\mathbf{x}, t))\right)$$

Streaming:

$$f_i(\mathbf{x} + e_i \delta t, t + \delta t) = f_i(\mathbf{x}, t)^*$$

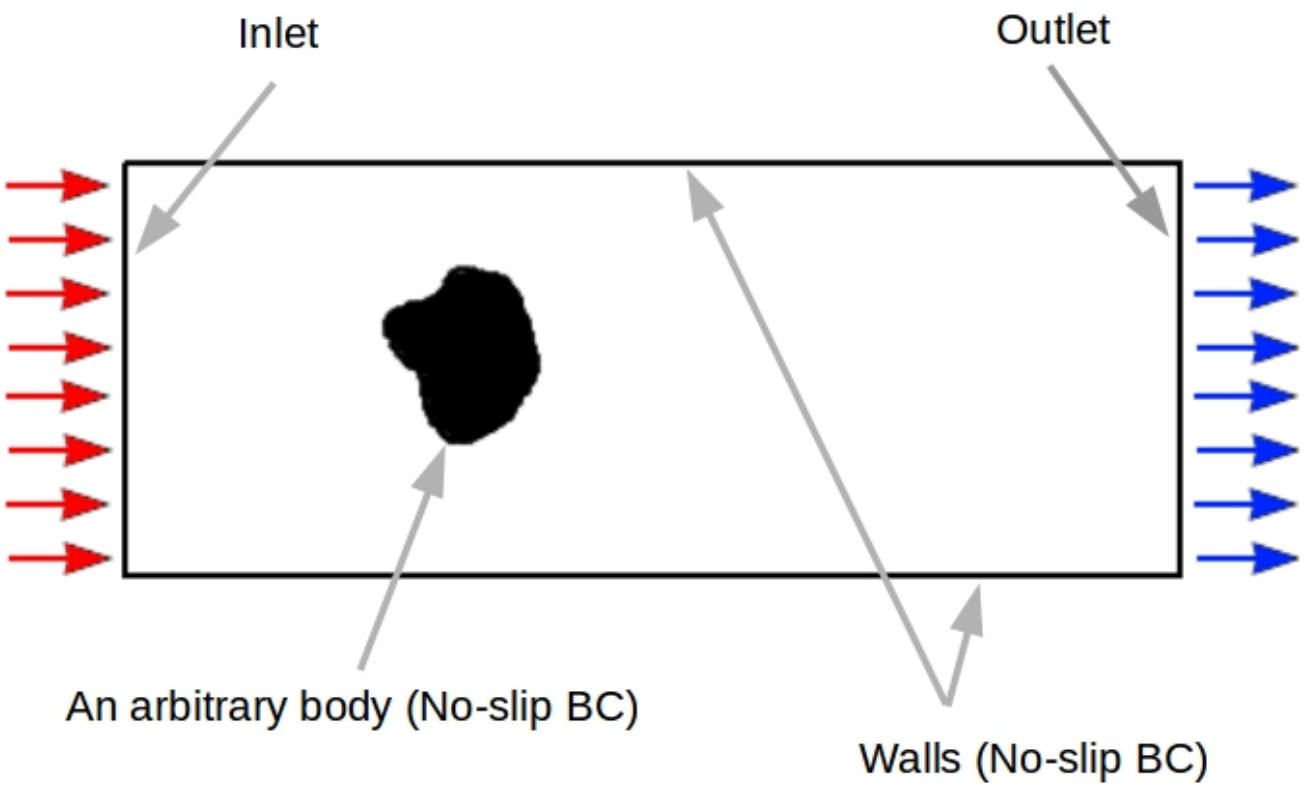


NOTE:

One can easily rewrite **collision-streaming** steps as **streaming-collision** ones

Boundary conditions

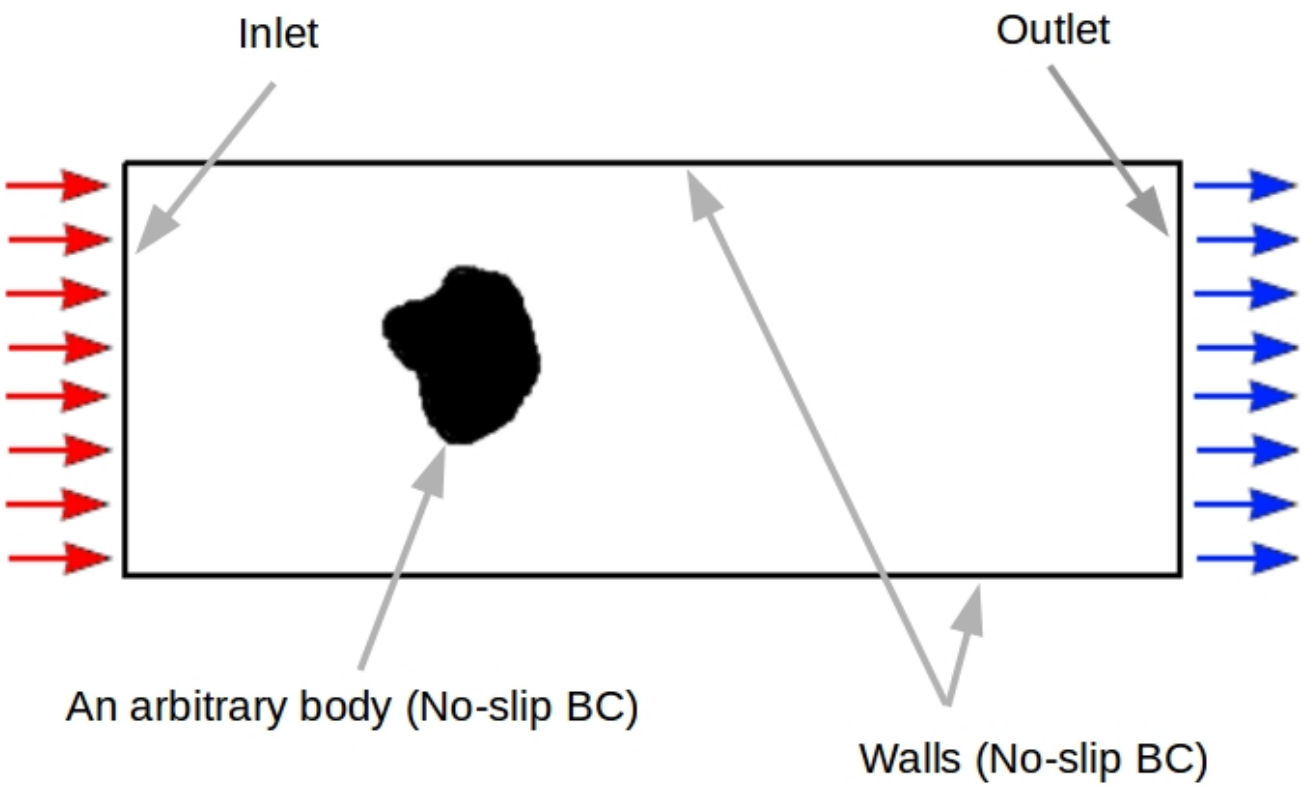
We want to simulate a wind tunnel with an arbitrary object inside



But ...

Boundary conditions

We want to simulate a wind tunnel with an arbitrary object inside



The last piece of LBM theory that we have not discussed yet is the **Boundary Conditions** and their implementation

Boundary conditions

- There is a whole "zoo" of boundary conditions for Lattice Boltzmann.

Boundary conditions

- There is a whole "zoo" of boundary conditions for Lattice Boltzmann.
- Each implementation has its advantages and disadvantages (***e.g.***, *accuracy vs programming complexity*)

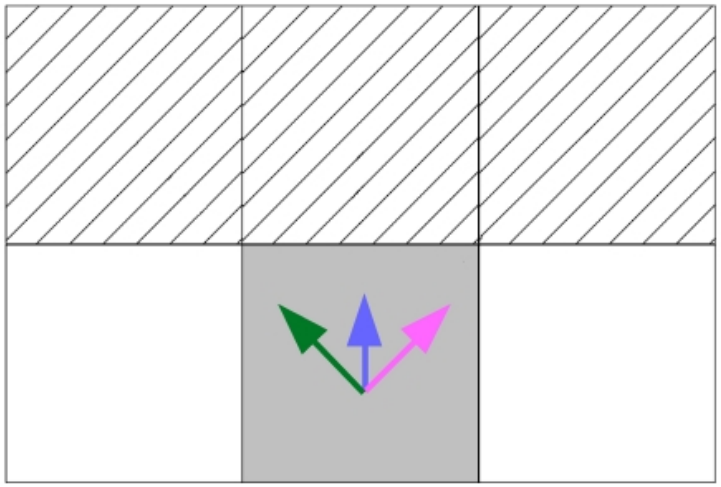
Boundary conditions

- There is a whole "zoo" of boundary conditions for Lattice Boltzmann.
- Each implementation has its advantages and disadvantages (***e.g.***, *accuracy vs programming complexity*.)
- We are going to demonstrate an implementation of the **"non-slip"** (so-called **bounce back**) boundary conditions to show you how they work.

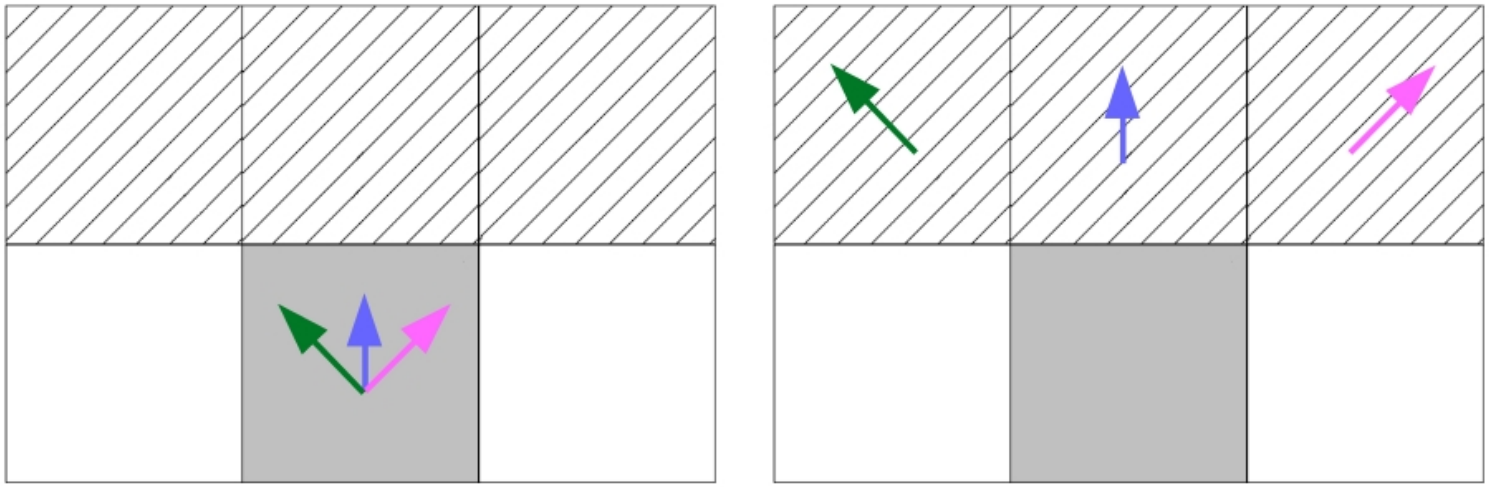
Boundary conditions

- There is a whole "zoo" of boundary conditions for Lattice Boltzmann.
- Each implementation has its advantages and disadvantages (***e.g.***, *accuracy vs programming complexity*.)
- We are going to demonstrate an implementation of the **"non-slip"** (so-called **bounce back**) boundary conditions to show you how they work.
- Please, read corresponding literature to get an idea of how different boundary conditions can be implemented.

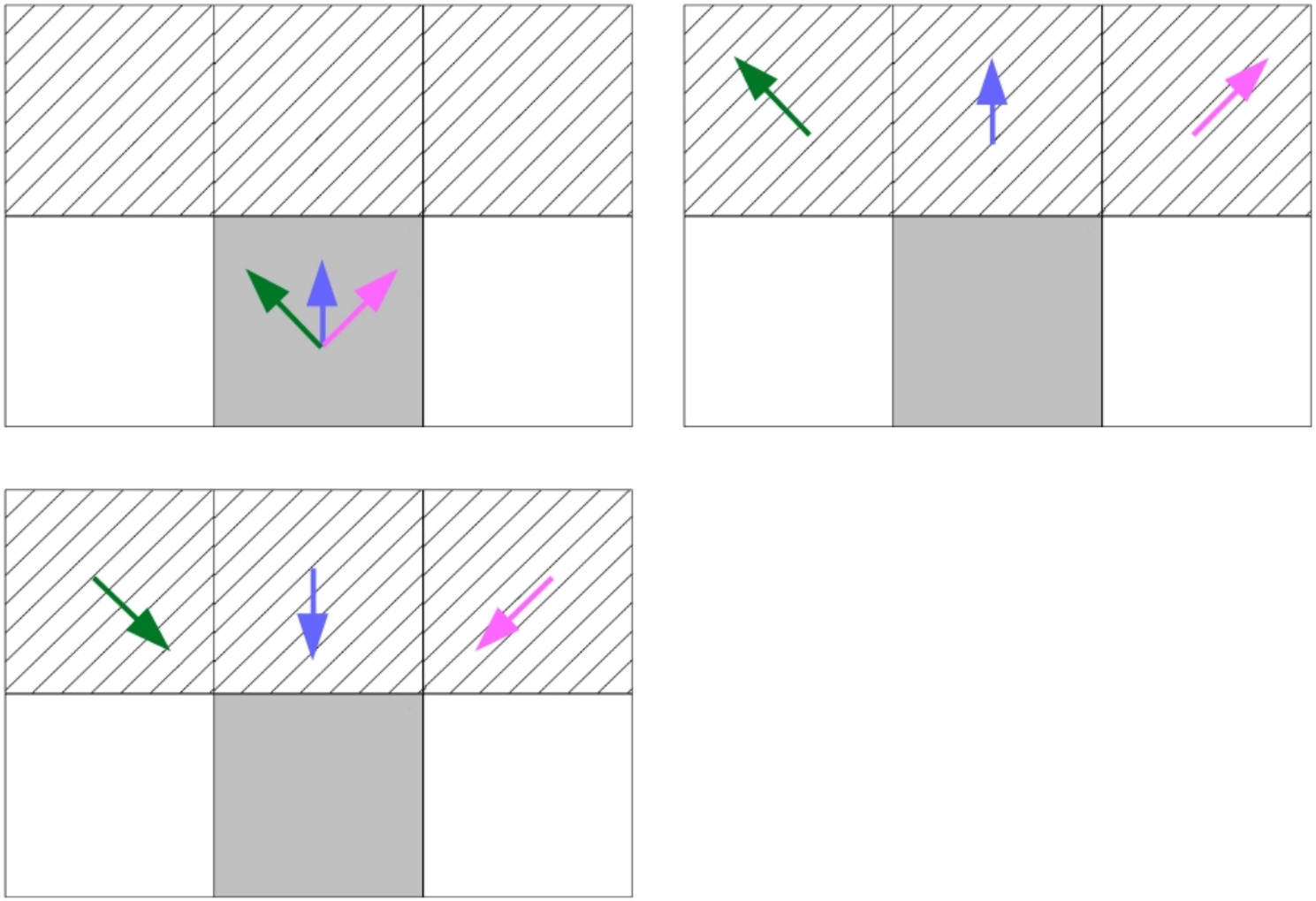
Bounce Back BC



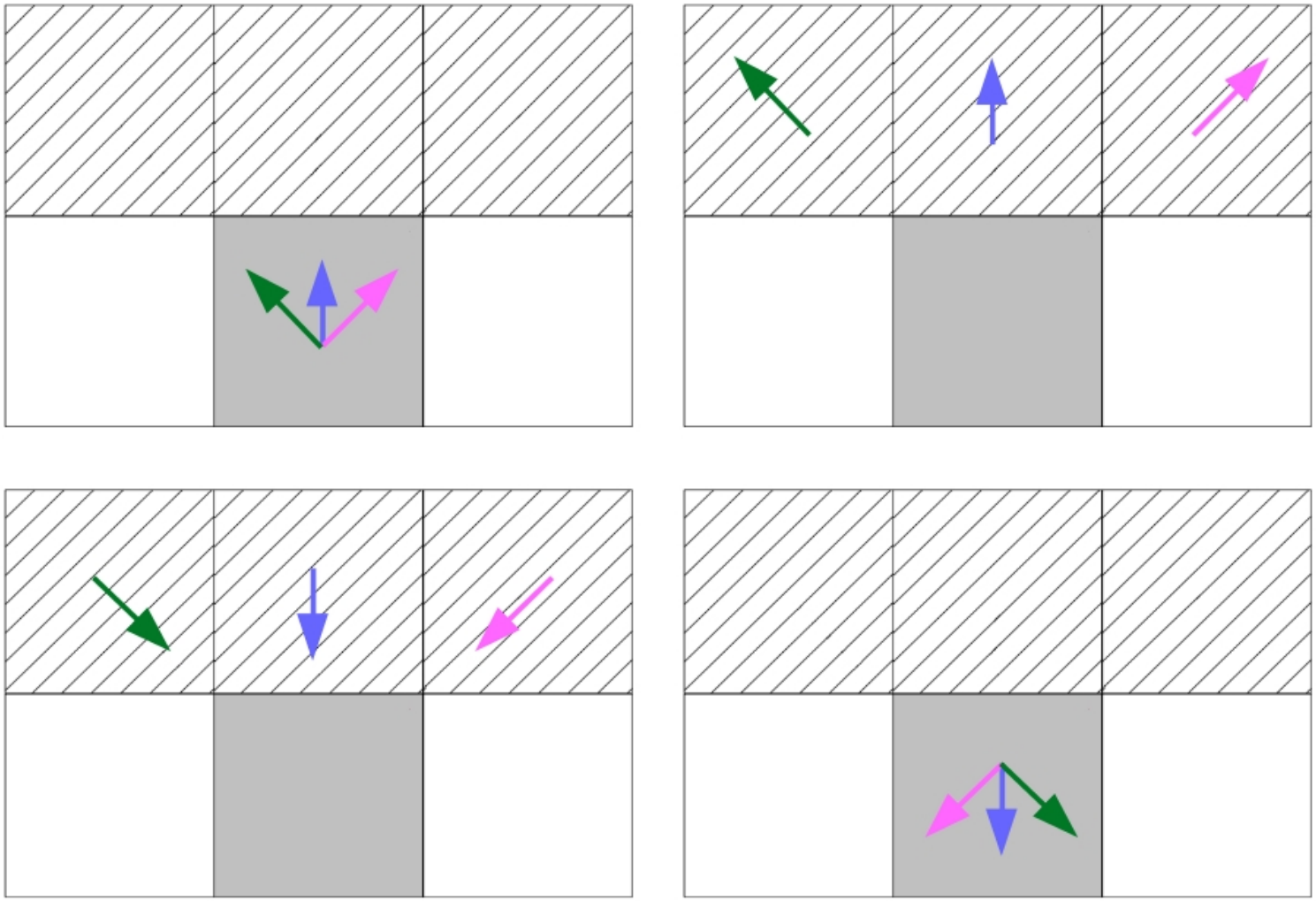
Bounce Back BC



Bounce Back BC



Bounce Back BC



LBM Algorithmic Properties

Advantages:

- LBM with BKG assumptions is easy to implement
- Streaming and Collision are **embarrassingly parallel** steps
- It is a **matrix free** method (no need to store the entire matrix)
  - equations are built on the fly

LBM Algorithmic Properties

Advantages:

- LBM with BKG assumptions is easy to implement
- Streaming and Collision are **embarrassingly parallel** steps
- It is a **matrix free** method (no need to store the entire matrix)
  - equations are built on the fly

Disadvantages:

- The algorithm is **memory bound**

LBM Algorithmic Properties

Advantages:

- LBM with BKG assumptions is easy to implement
- Streaming and Collision are **embarrassingly parallel** steps
- It is a **matrix free** method (no need to store the entire matrix)
  - equations are built on the fly

Disadvantages:

- The algorithm is **memory bound**
- The streaming step does not have any computational load. It **just moves data** from one place to another. Hence, it creates huge memory traffic.

LBM Algorithmic Properties

Advantages:

- LBM with BKG assumptions is easy to implement.
- Streaming and Collision are **embarrassingly parallel** steps.
- It is a **matrix free** method (no need to store the entire matrix).
  - equations are built on the fly

Disadvantages:

- The algorithm is **memory bound**
- The streaming step does not have any computational load. It **just moves data** from one place to another. Hence, it creates huge memory traffic.
- To build an equation on the fly, we have to differentiate between elements according to their types. Thus, we need to assign **a flag to each element**. Hence, we have to hold and process an additional data structure.

LBM Algorithmic Properties

Advantages:

- LBM with BKG assumptions is easy to implement
- Streaming and Collision are **embarrassingly parallel** steps
- It is a **matrix free** method (no need to store the entire matrix)
  - equations are built on the fly

Disadvantages:

- The algorithm is **memory bound**
- The streaming step does not have any computational load. It **just moves data** from one place to another. Hence, it creates huge memory traffic.
- To build an equation on the fly, we have to differentiate between elements according to their types. Thus, we need to assign **a flag to each element**. Hence, we have to hold and process additional data structure.

Note

We have to **check flags** for each neighboring element to build the right equation **in each iteration**. This leads to a **huge ladder of if-else statements** which lead to **branch divergence** on the GPU.

Disadvantages

We have to **check flags** for each neighboring element to build the right equation **in each iteration**. This leads to a **huge ladder of if-else statements** which lead to **branch divergence** on the GPU.

- Perform **functional decomposition** before the main loop.

Disadvantages

We have to **check flags** for each neighboring element to build the right equation **in each iteration**. This leads to a **huge ladder of if-else statements** which lead to **branch divergence** on the GPU.

- Perform **functional decomposition** before the main loop.
- Update elements of the **same type** separately.

Disadvantages

We have to **check flags** for each neighboring element to build the right equation **in each iteration**. This leads to a **huge ladder of if-else statements** which lead to **branch divergence** on the GPU.

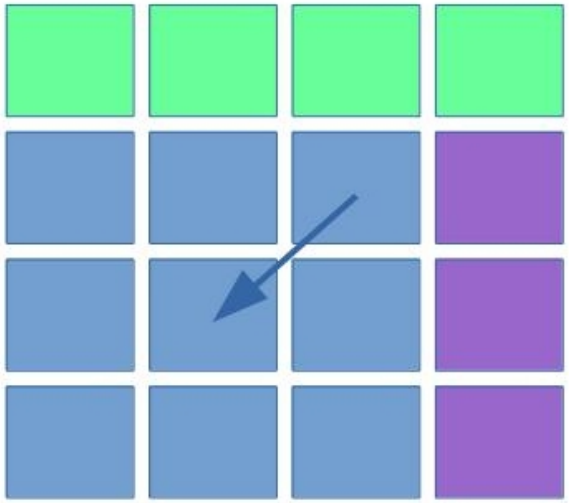
- Perform **functional decomposition** before the main loop.
- Update elements of the **same type** separately.
- In other words, remember **where, what and how** to update.

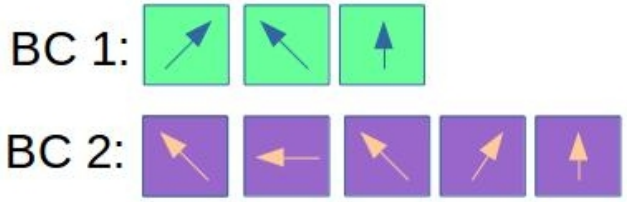
Disadvantages

We have to **check flags** for each neighboring element to build the right equation **in each iteration**. This leads to a **huge ladder of if-else statements** which lead to **branch divergence** on the GPU.

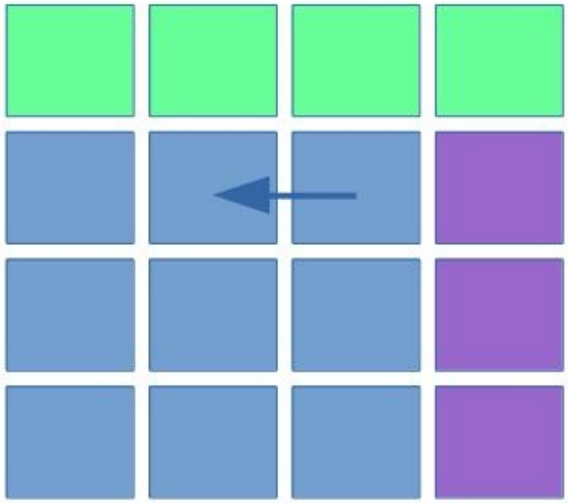
- Perform **functional decomposition** before the main loop.
- Update elements of the **same type** separately.
- In other words, remember **where, what and how** to update.
- Put elements of the same type in a **separate data structure**.

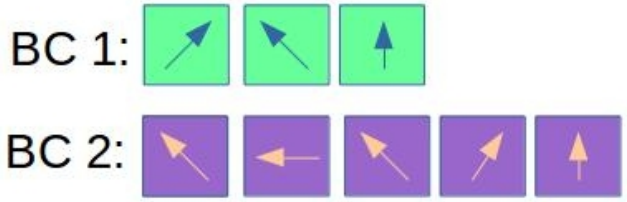
Scanning



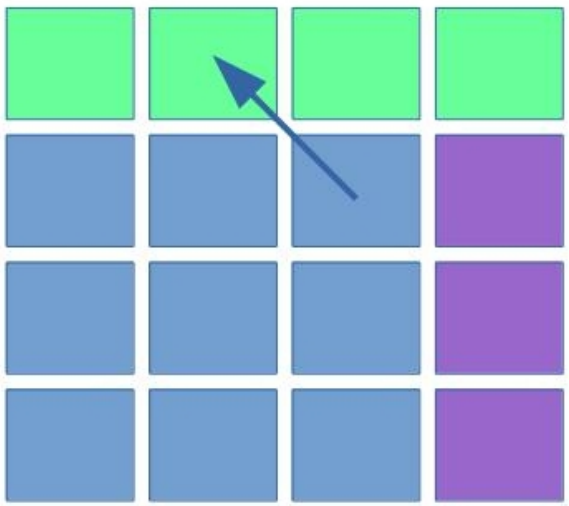


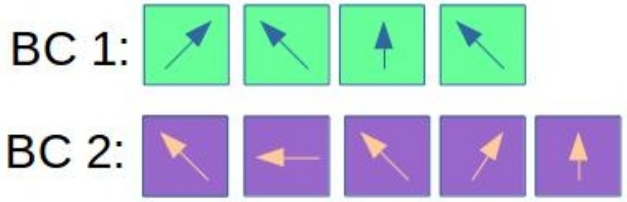
Scanning



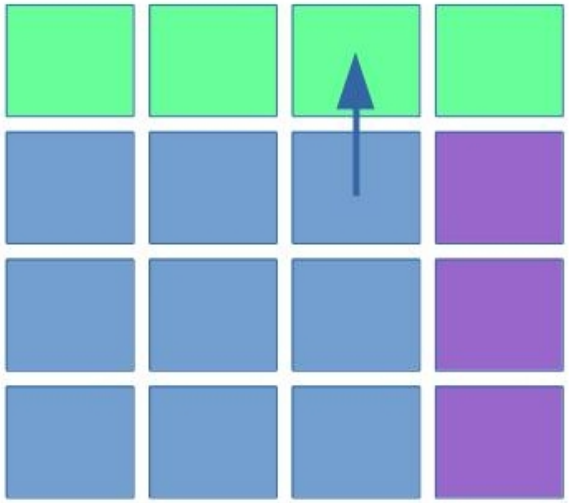


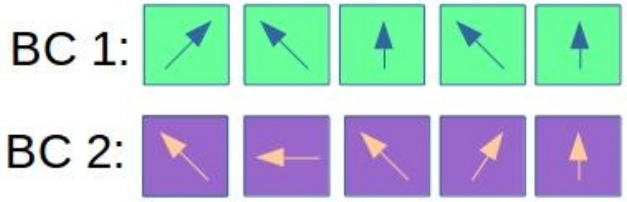
Scanning



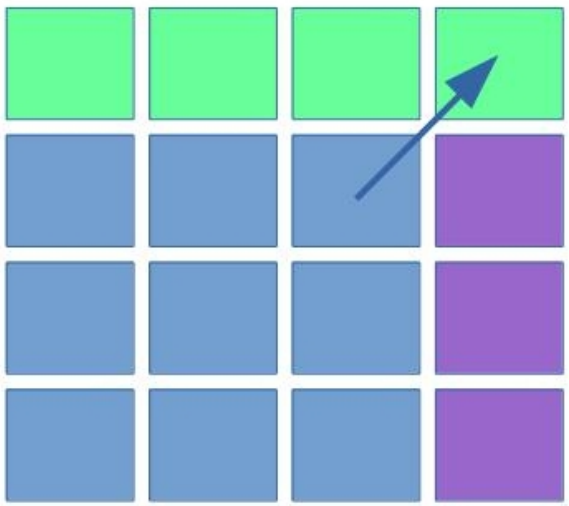


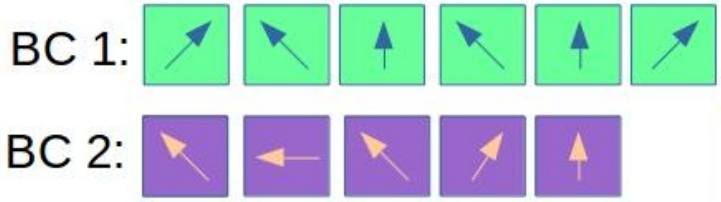
Scanning



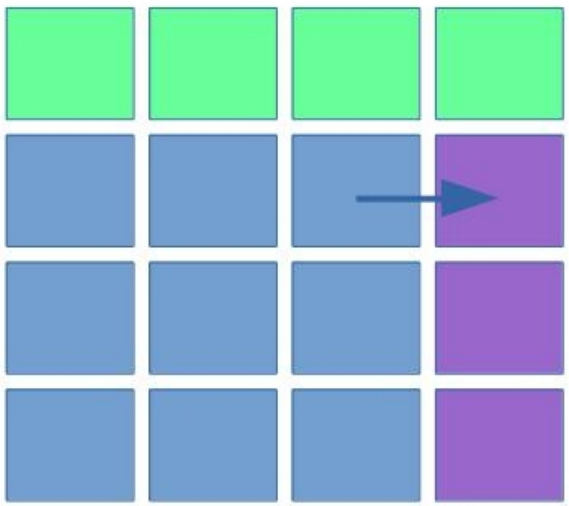


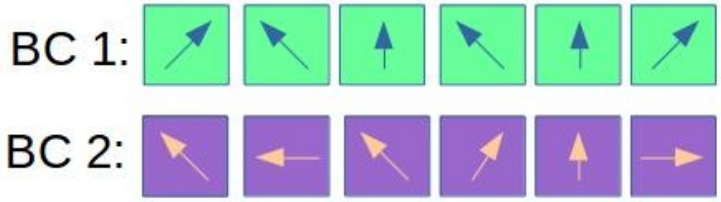
Scanning



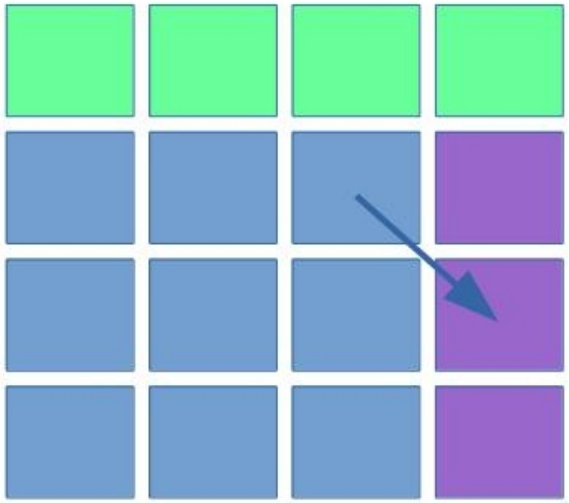


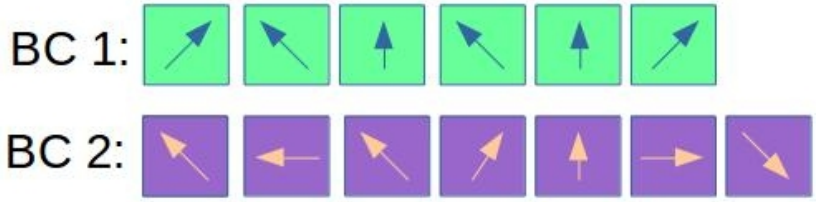
Scanning



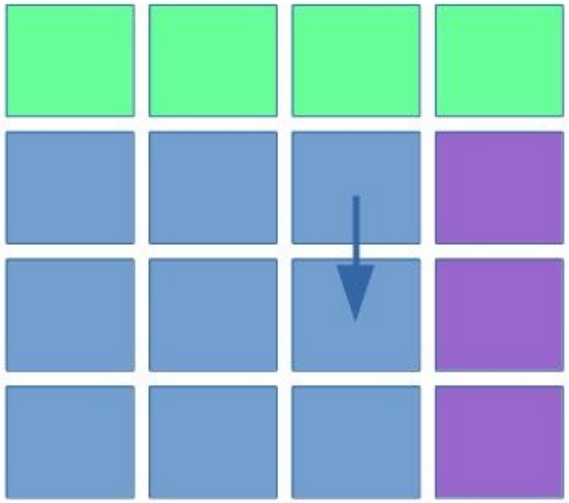


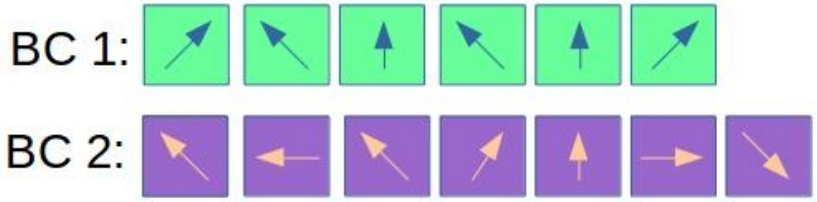
Scanning





Scanning





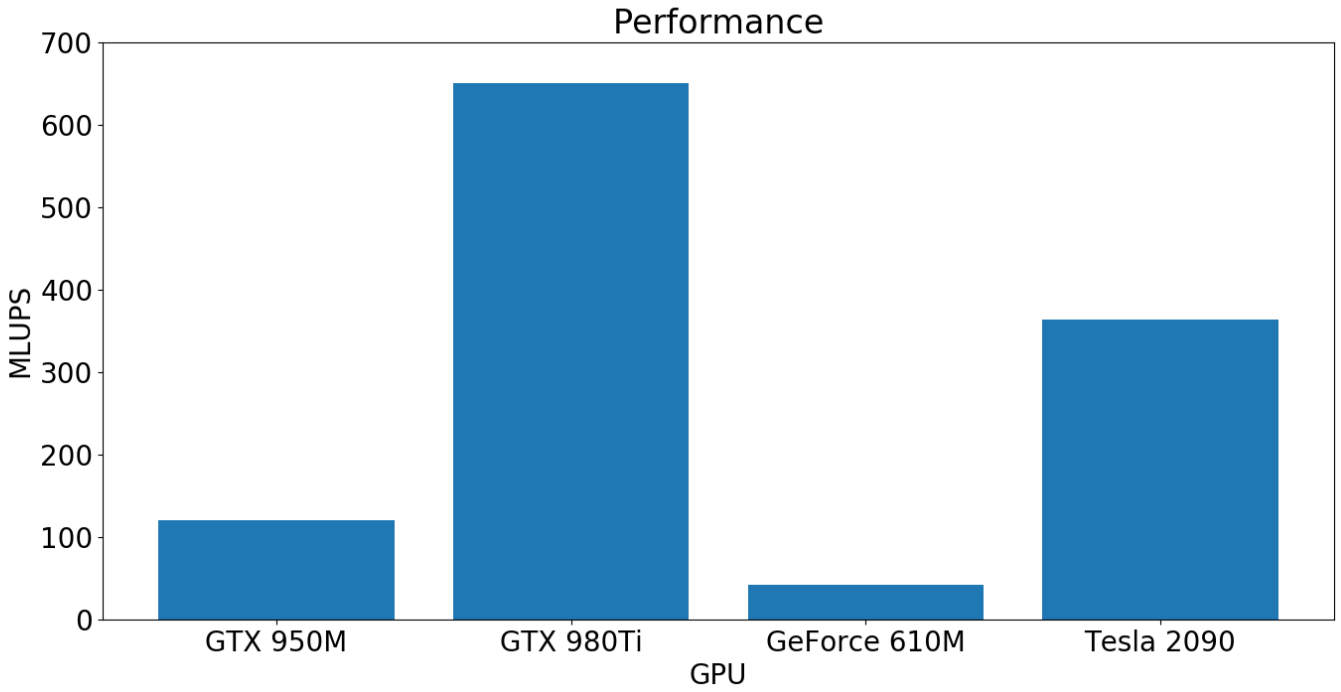
Results

Time for some videos.

Performance

Performance on GPUs

We ran the application on a number of different GPUs.

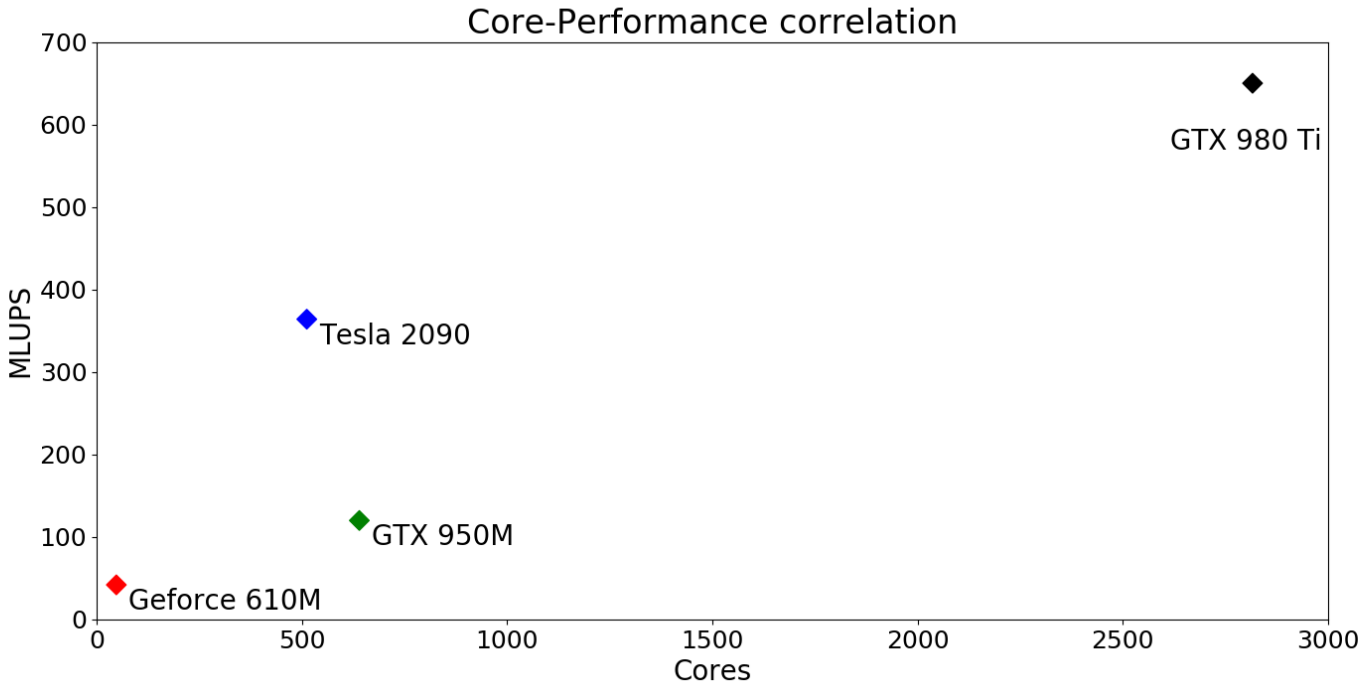


Core-Performance correlation

How do you think the core count is correlated with the performance?

Core-Performance correlation

How do you think the core count is correlated with the performance?

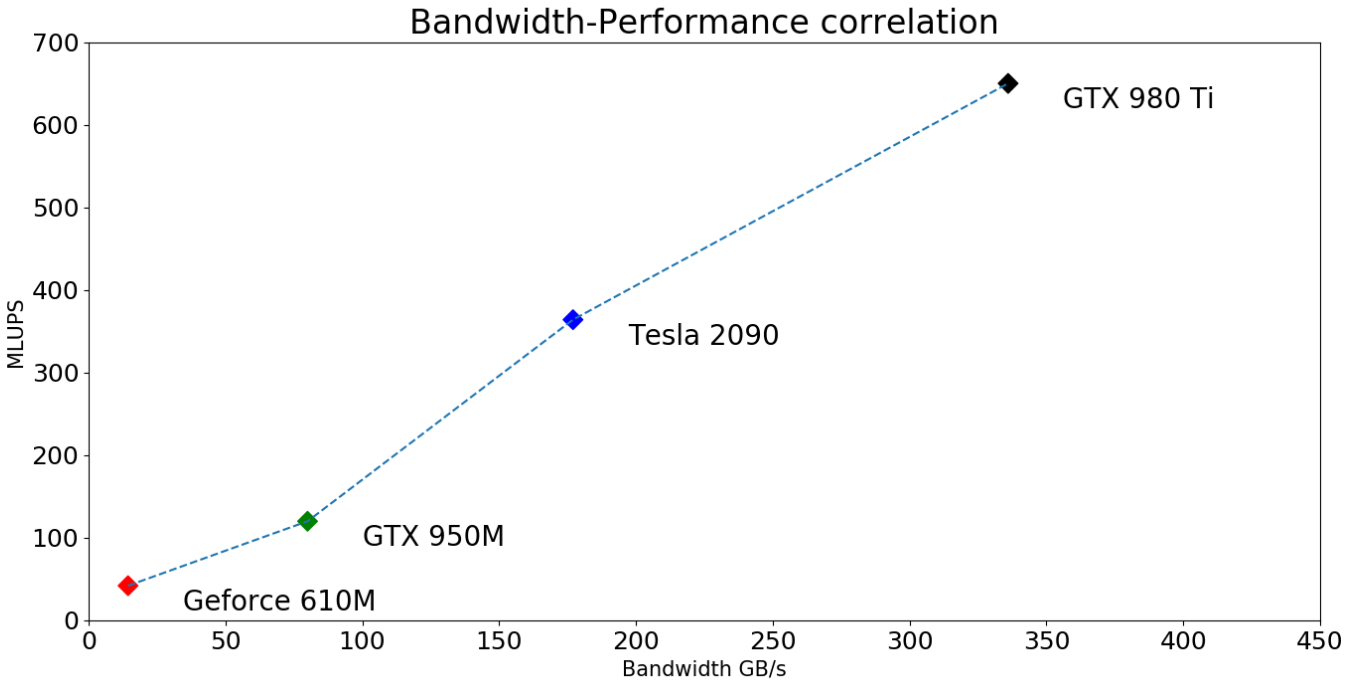


Bandwidth-Performance correlation

What about the **bandwidth?**

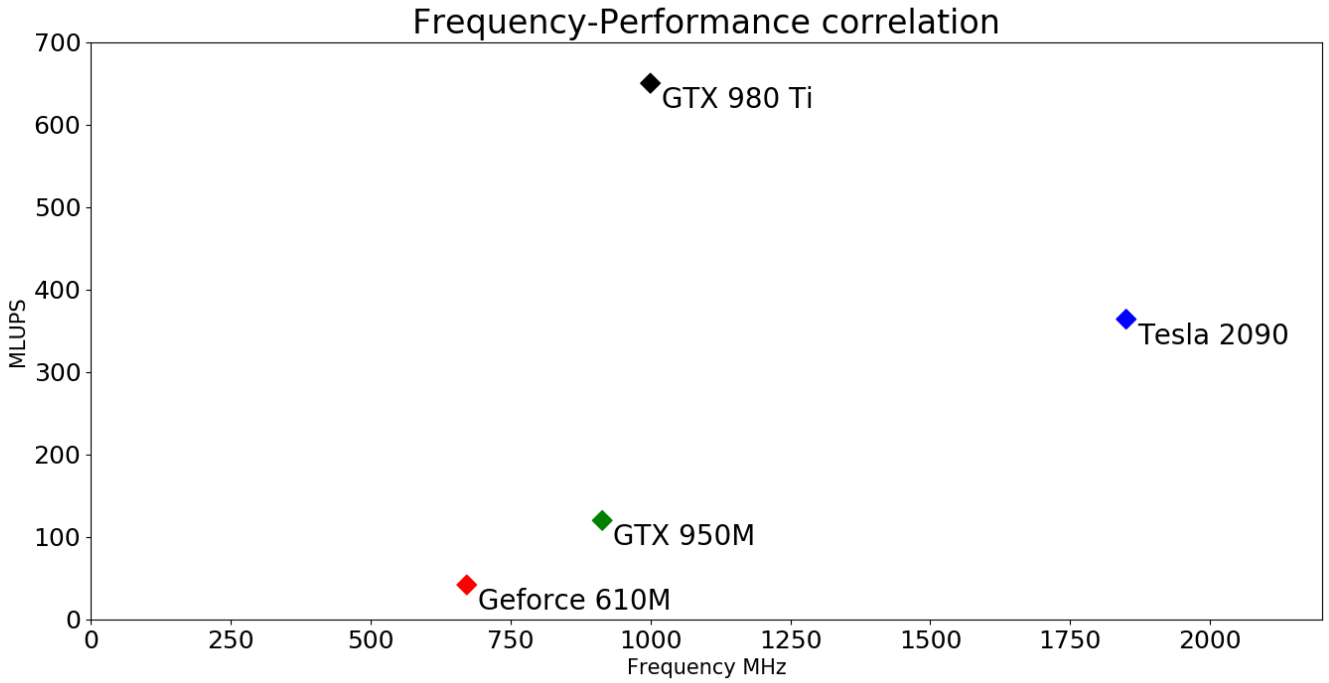
Bandwidth-Performance correlation

What about the **bandwidth?**



Frequency-Performance correlation

Let us take a look at frequency too.



CUDA Optimization and Memory

General Steps Towards Optimization

Some general principles for optimization:

- Minimize host-device memory interactions.
- Use coalesced memory access.
- Set kernel execution parameters to achieve higher occupancy.

General Steps Towards Optimization

Some general principles for optimization:

- Minimize host-device memory interactions.
- Use coalesced memory access.
- Set kernel execution parameters to achieve higher occupancy.
- Pay attention to register and shared memory usage.
- Divergent code can significantly reduce performance. In the worst case, by a factor of 32.

Memory Hierarchy

- CPUs have low bandwidth and hide latency through caching
- GPUs have higher bandwidth but also higher latency: 400 - 800 cycles

Memory Hierarchy

- CPUs have low bandwidth and hide latency through caching
- GPUs have higher bandwidth but also higher latency: 400 - 800 cycles

	Size	Latency, cycles	Bandwidth, GB/s
Register	300B	1	-
L1 cache	32KB	3	150
L3 cache	4MB	20	30
CPU-local memory	4GB	200	15
CPU-remote memory	4GB	300	10
GPU-device memory	**4GB**	**400-800**	**256-320**
PCI Express 2.0 (x8-x16)	-	-	**4-8**

Memory and Computational Limitations

An application can be either memory or compute bounded.

- Memory bounded - Not enough bandwidth to utilize all FPUs.
- Compute bounded - All FPUs are occupied but we have more data available.

Memory and Computational Limitations

An application can be either memory or compute bounded.

- Memory bounded - Not enough bandwidth to utilize all FPUs.
- Compute bounded - All FPUs are occupied but we have more data available.

A simple heuristic for determining memory boundedness:

- Switch from double to float.
- Memory bounded if speedup is small - for example 2.

Memory and Computational Limitations

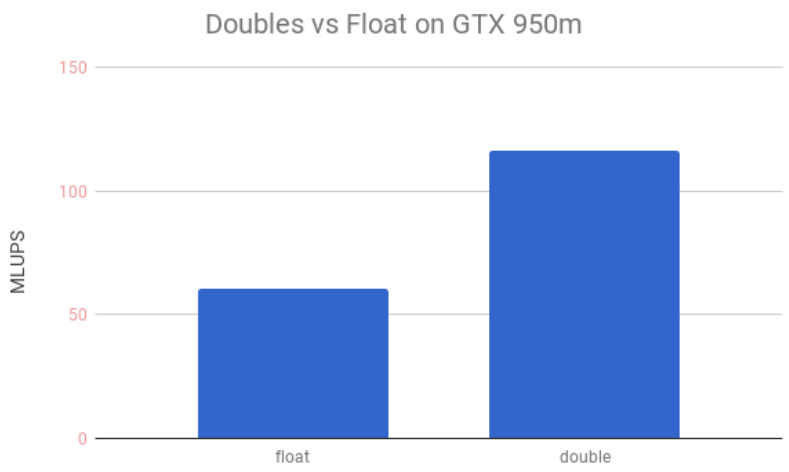
An application can be either memory or compute bounded.

- Memory bounded - Not enough bandwidth to utilize all FPUs.
- Compute bounded - All FPUs are occupied but we have more data available.

Compute Capability	3.0	5.0, 5.2	6.0
Float Throughput	192	128	64
Double Throughput	8	4	32
Ratio	24	32	2

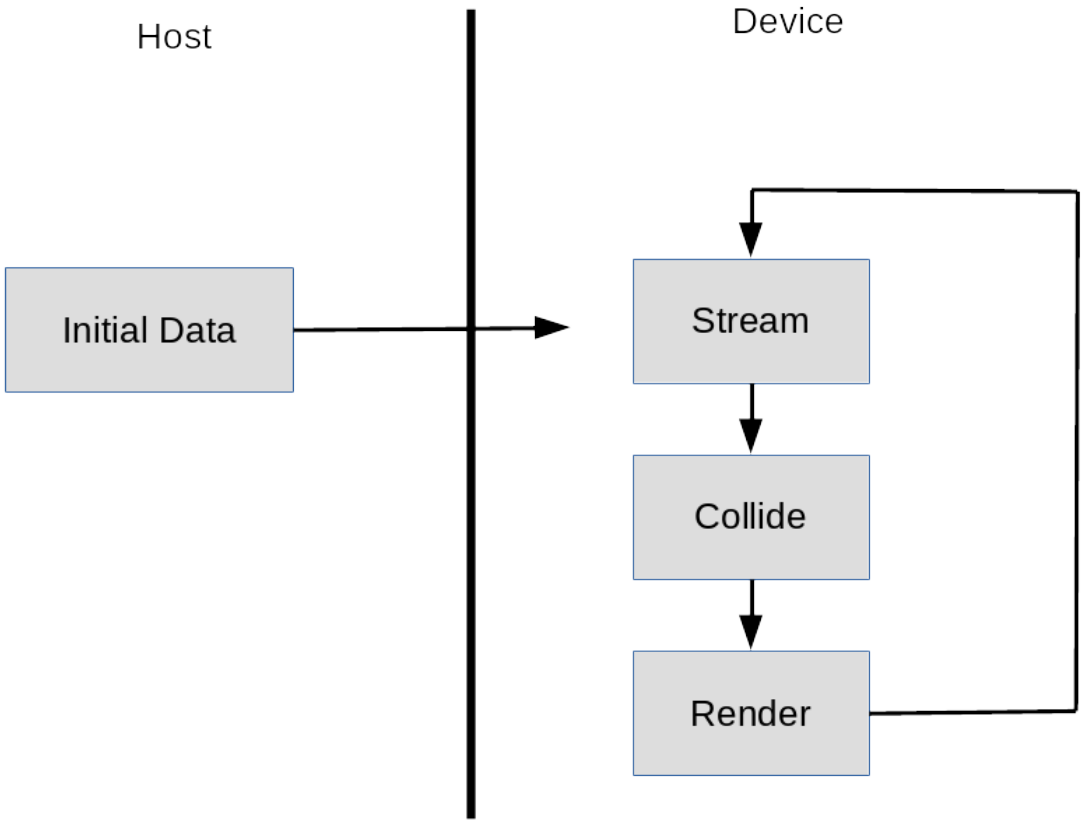
A simple heuristic for determining memory boundedness:

- Switch from double to float.
- Memory bounded if speedup is small - for example 2.



Host - Device Memory Interactions

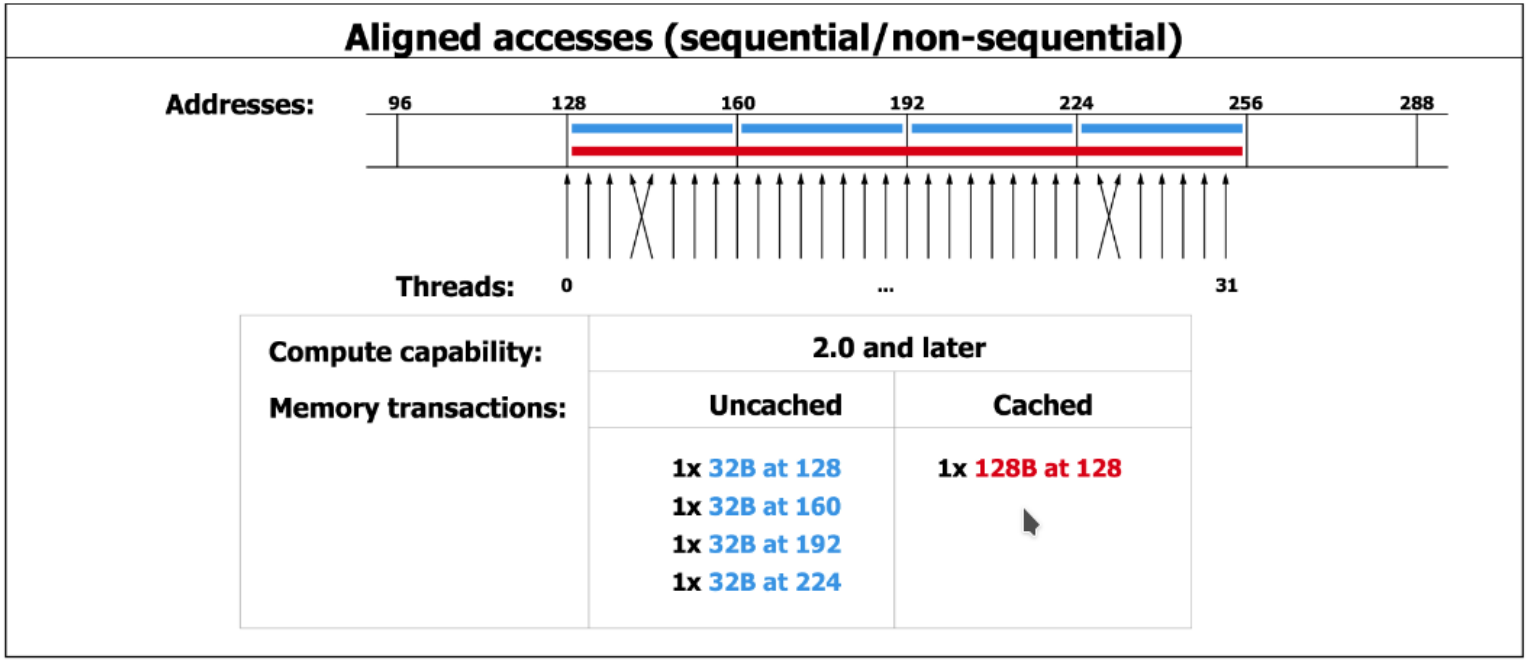
Since data transfers between the CPU and GPU are limited by the PCIe bus, our code only requires host-device memory interactions in the beginning. Thereafter, GPU computations do not require memory interactions with the CPU.



Coalesced Memory Access

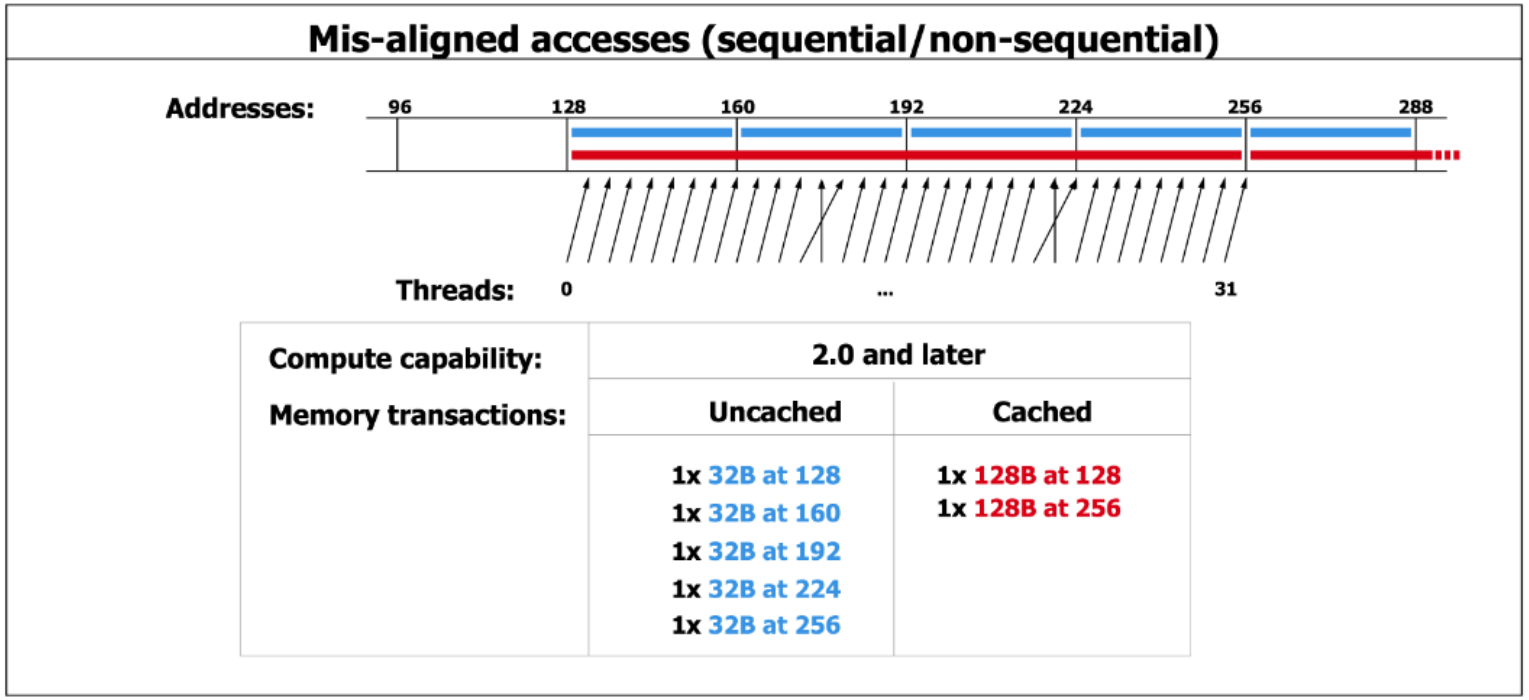
What is coalesced memory access?

If threads in a warp access nearby memory locations then these memory requests can be **coalesced** into fewer transactions [4].



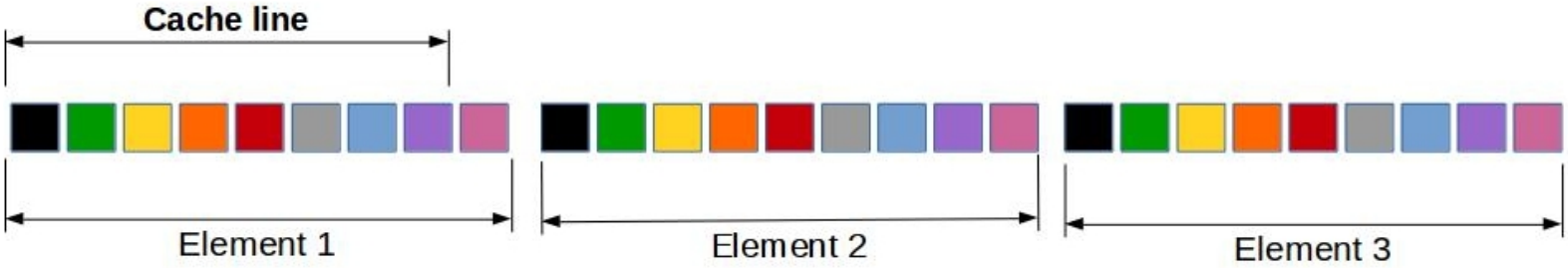
What is coalesced memory access?

If threads in a warp access nearby memory locations then these memory requests can be **coalesced** into fewer transactions[4].



Coalesced Memory: CPUs vs GPUs

CPU approach: **Array of Structures**



GPU approach: **Structure of Arrays**

CUDA Execution Parameters

Hardware limits on the number of threads and blocks:

Compute Capability	2.0	3.0	5.0
Max. blocks	65535	$2^{31} - 1$	$2^{31} - 1$
Max. threads per block	1024	1024	1024
Warp size	32	32	32
Resident blocks per MP	8	16	32
Resident threads per MP	1536	2048	2048
Resident warps per MP	48	64	64

CUDA Execution Parameters

There is a similar **limit** on the number of registers:

Compute Capability	2.0	3.0	5.0
Number of registers per MP	32 K	64 K	64 K
Max. registers per block	32 K	64 K	64 K
Max. registers per thread	63	255	255

Why are these important?

CUDA Execution Parameters

There is a similar **limit** on the number of registers:

Compute Capability	2.0	3.0	5.0
Number of registers per MP	32 K	64 K	64 K
Max. registers per block	32 K	64 K	64 K
Max. registers per thread	63	255	255

Why are these important?

They all affect **occupancy**.

Occupancy

$$Occupancy = \frac{\text{Active warps per multiprocessor}}{\text{Max. resident warps per multiprocessor}}$$

Occupancy

$$Occupancy = \frac{\text{Active warps per multiprocessor}}{\text{Max. resident warps per multiprocessor}}$$

- Low occupancy means that you cannot hide memory latency by switching between different warps.

Occupancy

$$Occupancy = \frac{\text{Active warps per multiprocessor}}{\text{Max. resident warps per multiprocessor}}$$

- Low occupancy means that you cannot hide memory latency by switching between different warps.
- On the other hand, high occupancy can lead to register spillage and thereby decrease performance.

Occupancy

$$Occupancy = \frac{\text{Active warps per multiprocessor}}{\text{Max. resident warps per multiprocessor}}$$

- Low occupancy means that you cannot hide memory latency by switching between different warps.
- On the other hand, high occupancy can lead to register spillage and thereby decrease performance.
- Therefore, we need to balance at least two things:
  1. Choose grid and block size such that we can maximize multiprocessor usage.

Occupancy

$$Occupancy = \frac{\text{Active warps per multiprocessor}}{\text{Max. resident warps per multiprocessor}}$$

- Low occupancy means that you cannot hide memory latency by switching between different warps.
- On the other hand, high occupancy can lead to register spillage and thereby decrease performance.
- Therefore, we need to balance at least two things:
  1. Choose grid and block size such that we can maximize multiprocessor usage.
  2. At the same, ensure that registers used per thread are low enough to fit everything in the register file.

Occupancy Example: Threads and Blocks

Compute capability 2.0: 1536 threads and 8 blocks per multiprocessor.

$$\text{Total threads per MP} = \text{Blocks per MP} \times \text{Threads per block}$$

- For a kernel with 32 threads per block, we need 1536 / 32 = 48 blocks to fully occupy the MP. However, devices with compute capability 2.0 **only support 8 blocks/MP.**

Occupancy Example: Threads and Blocks

Compute capability 2.0: 1536 threads and 8 blocks per multiprocessor.

$$\text{Total threads per MP} = \text{Blocks per MP} \times \text{Threads per block}$$

- For a kernel with 32 threads per block, we need 1536 / 32 = 48 blocks to fully occupy the MP. However, devices with compute capability 2.0 **only support 8 blocks/MP.**
- To fully occupy such a GPU, we need to launch kernels with 1536 / 8 = 192 threads per block.

Occupancy Example: Threads and Blocks

Compute capability 2.0: 1536 threads and 8 blocks per multiprocessor.

$$\text{Total threads per MP} = \text{Blocks per MP} \times \text{Threads per block}$$

- For a kernel with 32 threads per block, we need 1536 / 32 = 48 blocks to fully occupy the MP. However, devices with compute capability 2.0 **only support 8 blocks/MP.**
- To fully occupy such a GPU, we need to launch kernels with 1536 / 8 = 192 threads per block.
- This gives 6 warps per block and a total of $6 \times 8 = 48$ warps / MP. This is the maximum number of active warps per MP for such a device.

Occupancy Example: Registers

Devices with compute capability 2.0 have **32,768** registers per multiprocessor.
Continuing with the example, suppose we have a kernel that uses 21 registers per thread.

Occupancy Example: Registers

Devices with compute capability 2.0 have **32,768** registers per multiprocessor.
Continuing with the example, suppose we have a kernel that uses 21 registers per thread.

- With 1536 threads per MP, our register usage is 1536 * 21 = **32,256** . In this case we are fine.

Occupancy Example: Registers

Devices with compute capability 2.0 have **32,768** registers per multiprocessor.
Continuing with the example, suppose we have a kernel that uses 21 registers per thread.

- With 1536 threads per MP, our register usage is 1536 * 21 = **32,256** . In this case we are fine.
- Now assume register usage per kernel is 22. Then to run 1536 threads, we would need 1536 * 22 = **33,972** registers. This will lead to **register spilling** .

Occupancy Example: Registers

Devices with compute capability 2.0 have **32,768** registers per multiprocessor.
Continuing with the example, suppose we have a kernel that uses 21 registers per thread.

- With 1536 threads per MP, our register usage is 1536 * 21 = **32,256** . In this case we are fine.
- Now assume register usage per kernel is 22. Then to run 1536 threads, we would need 1536 * 22 = **33,972** registers. This will lead to **register spilling** .
- Here reducing the occupancy by reducing the number of threads might speed up the application: 32768 / 22 = **1489 max. threads** without register spilling. We can only have 7 active blocks with 192 threads per block and a **total of 1344 threads** .

Getting register usage data

Use the compiler flag `-Xptxas -v` to get register usage per kernel.

```
ptxas info    : Compiling entry function '_Z7PrintBCP18BoundaryConditions' for 'sm_5
ptxas info    : Function properties for _Z7PrintBCP18BoundaryConditions
    8 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 20 registers, 328 bytes cmem[0]
ptxas info    : Compiling entry function '_Z27UpdatePopulationFieldDevicePfS_S_' for
ptxas info    : Function properties for _Z27UpdatePopulationFieldDevicePfS_S_
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 32 registers, 344 bytes cmem[0]
```

NVIDIA Occupancy Calculator

In order to figure out the right block and thread distribution for a given number of registers per thread, you can use NVIDIA's occupancy calculator.

CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	5.0
1.b) Select Shared Memory Size Config (bytes)	65536

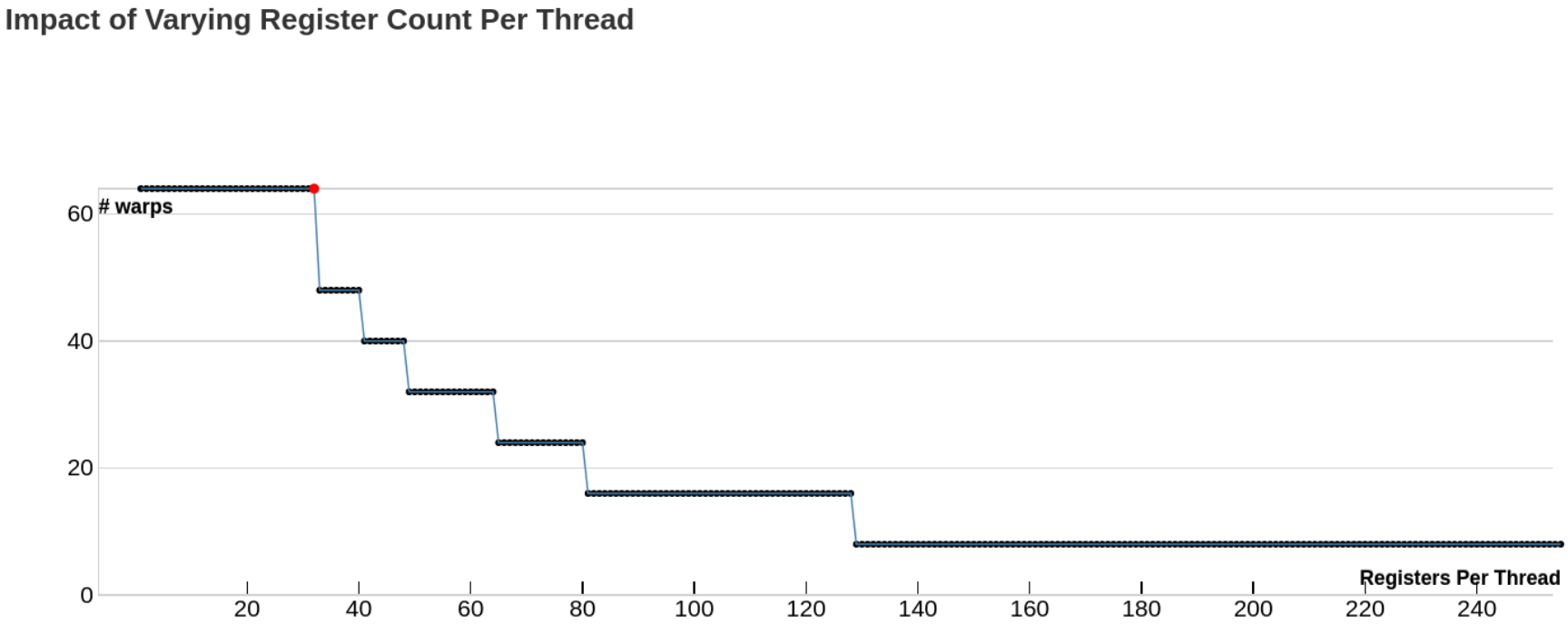
2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	32
Shared Memory Per Block (bytes)	4096

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	8
Occupancy of each Multiprocessor	100%

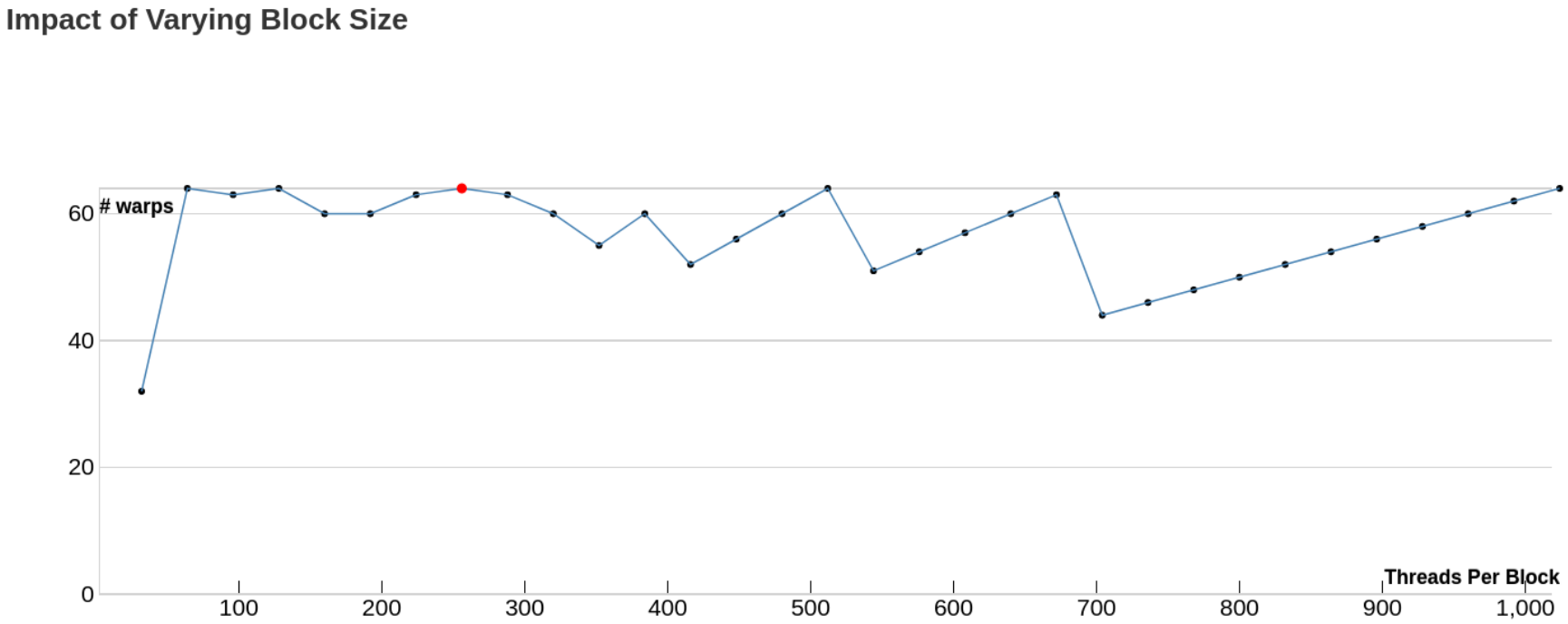
NVIDIA Occupancy Calculator

In order to figure out the right block and thread distribution for a given number of registers per thread, you can use NVIDIA's occupancy calculator.



NVIDIA Occupancy Calculator

In order to figure out the right block and thread distribution for a given number of registers per thread, you can use NVIDIA's occupancy calculator.



Nvidia Visual Profiler

You can use Nvidia's Visual Profiler to find the achieved occupancy.

Analysis | GPU Details (Summary) | CPU Details | OpenACC Details | Console | Settings

Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem	Achieved Occupancy
UpdatePopulationFieldDevice(float*, float*, flo	400	1.66246 ms	32	0	0	0.967
StreamDevice(float*, float*, int*)	400	1.47414 ms	25	0	0	0.958
UpdateVelocityFieldDevice(float*, float*, float*,	400	985.775 µs	30	0	0	0.942
UpdateDensityFieldDevice(float*, float*, int*)	400	828.649 µs	20	0	0	0.933
InitArrayDevice(float*, float, int)	21	77.919 µs	10	0	0	0.795
TreatNonSlipBC(int*, float*, int)	400	44.753 µs	10	0	0	0.611
TreatInflowBC(int*, float*, float*, float*, int)	400	23.457 µs	11	0	0	0.164
TreatOutflowBC(int*, float*, float*, float*, int)	400	32.734 µs	19	0	0	0.146

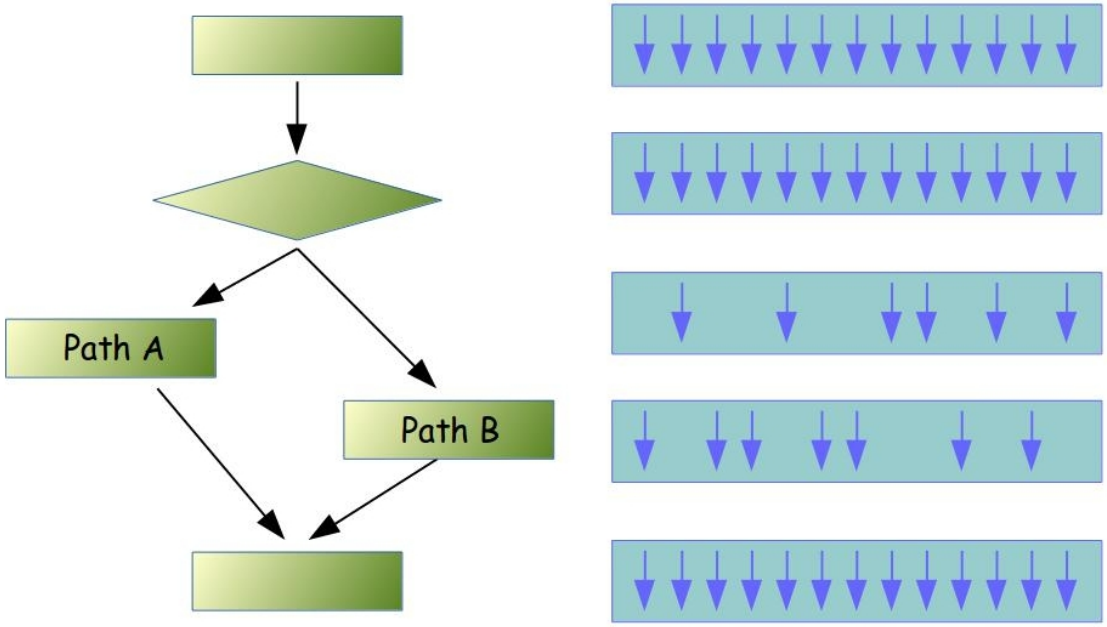
Branch divergence within a warp

Branch Divergence

- You have probably already heard about **branch divergence within a warp**.
- You might also have seen this figure:

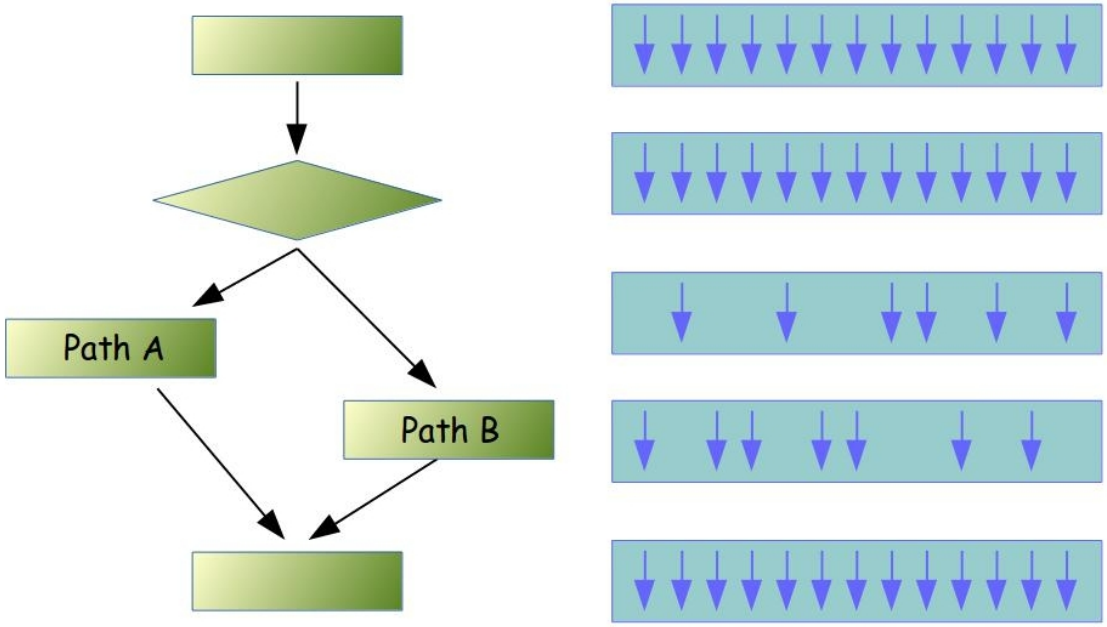
Branch Divergence

- You have probably already heard about **branch divergence within a warp**.
- You might also have seen this figure:



Branch Divergence

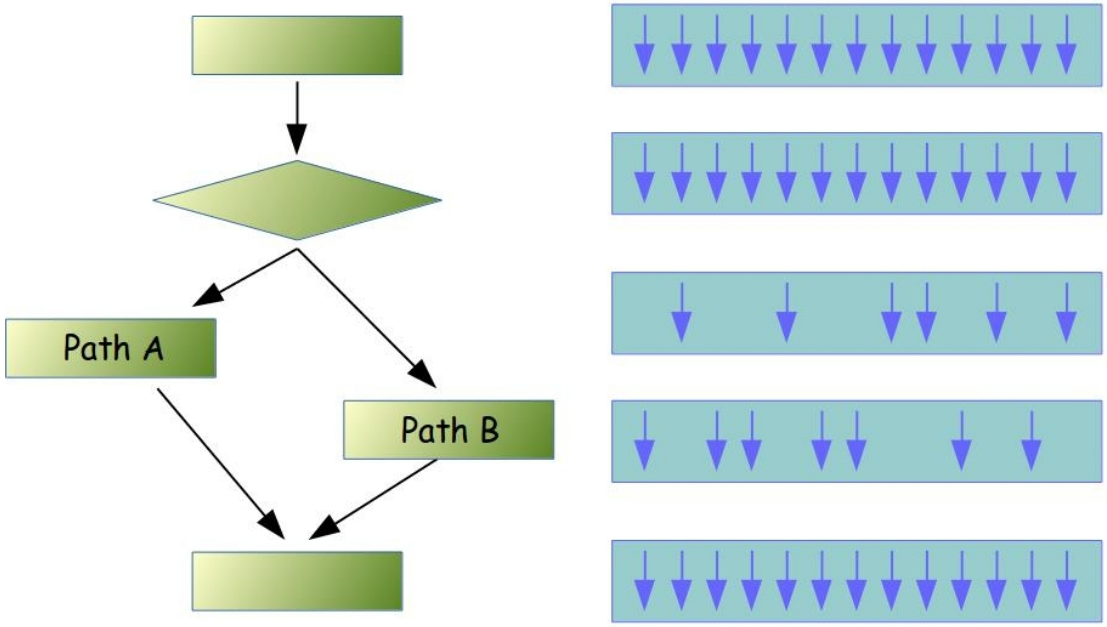
- You have probably already heard about **branch divergence within a warp**.
- You might also have seen this figure:



- You know that branches are executed in «lockstep»

Branch Divergence

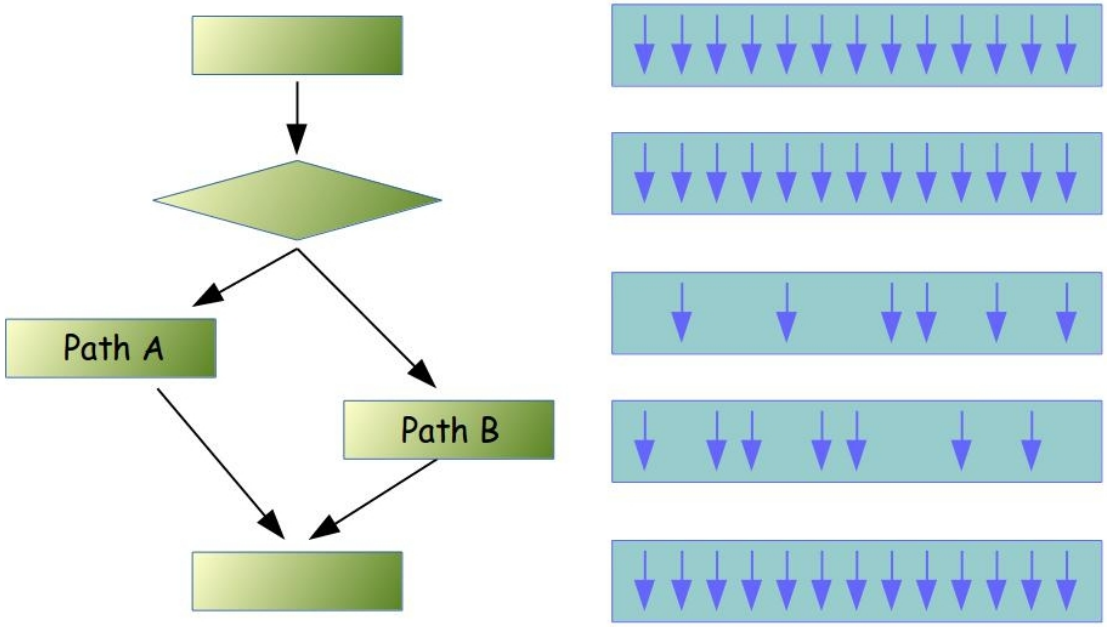
- You have probably already heard about **branch divergence within a warp**.
- You might also have seen this figure:



- You know that branches are executed in «lockstep»
- You might already know that this leads to almost **50% performance loss**

Branch Divergence

- You have probably already heard about **branch divergence within a warp**.
- You might also have seen this figure:



- You know that branches are executed in «lockstep»
- You might already know that this leads to almost **50% performance loss**
- Question: is this **always** true?

Branch Divergence

What **assumptions** did we make to get a **50% performance loss** ?

Branch Divergence

What **assumptions** did we make to get a **50% performance loss** ?

- One **half** of the warp executes **Branch A** and the second half executes **Branch B**

Branch Divergence

What **assumptions** did we make to get a **50% performance loss** ?

- One **half** of the warp executes **Branch A** and the second half executes **Branch B**.
- The **workloads** are **equal**.

Branch Divergence

What **assumptions** did we make to get a **50% performance loss** ?

- One **half** of the warp executes **Branch A** and the second half executes **Branch B**.
- The **workloads** are **equal**.

Extreme case

- Consider **32** distinct **branches**.
- Each **thread** executes **one branch**.
- All branches have **the same workload**.
- The **slow-down** is **x32** in that case.

Branch Divergence

What **assumptions** did we make to get a **50% performance loss** ?

- One **half** of the warp executes **Branch A** and the second half executes **Branch B**.
- The **workloads** are **equal**.

Extreme case

- Consider **32** distinct **branches**.
- Each **thread** executes **one branch**.
- All branches have **the same workload**.
- The **slow-down** is **x32** in that case.

But let us consider our case one more time

Branch Divergence

Let's compare two implementation of the same algorithm: **with** branching

```
__global__ void UpdateVelocityFieldDevice(real *velocity,
                                          real *population,
                                          real *density,
                                          int *flag_field) {
    int num_lattices = parameters_device.num_lattices;
    short int num_directions = parameters_device.discretization;

    int thread_id = threadIdx.x + blockIdx.x * blockDim.x;

    while (thread_id < num_lattices) {
        if (flag_field[thread_id] == FLUID) { // <= branching
            real lattice_velocity_x = 0.0;
            real lattice_velocity_y = 0.0;

            for (short int component = 0; component < num_directions; ++component) {
                real distribution = population[component * num_lattices + thread_id]
                lattice_velocity_x += coords_device[component] * distribution;
                lattice_velocity_y += coords_device[num_directions + component] * di
            }

            real inverse_density = 1.0 / density[thread_id];
            velocity[thread_id] = inverse_density * lattice_velocity_x;
            velocity[num_lattices + thread_id] = inverse_density * lattice_velocity_
        }
        thread_id += blockDim.x * gridDim.x;
    }
}

\\ MLUPS: 42.79 | 19.98  [float | double] (GeForce 610M)
```

Branch Divergence

Let's compare two implementation of the same algorithm: **without** branching

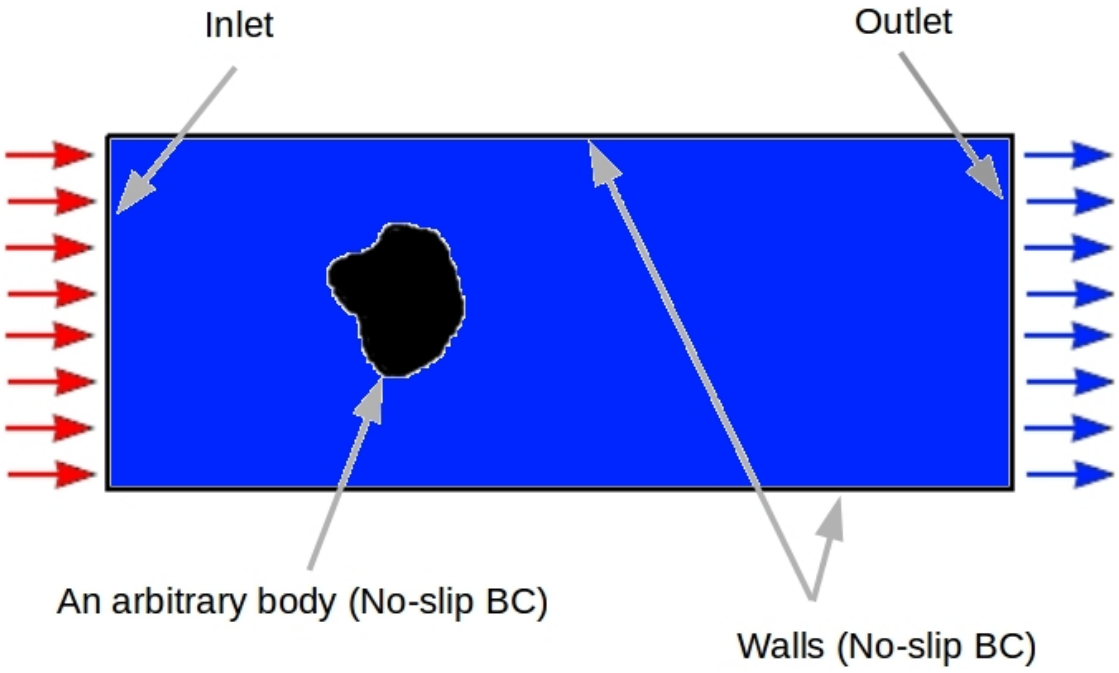
```
__global__ void UpdateVelocityFieldDevice_A(real *velocity,
                                            real *population,
                                            real *density,
                                            int *flag_field,
                                            int *fluid_indices, // additional parame
                                            int num_fluid_lattices) {
    int num_lattices = parameters_device.num_lattices;
    short int num_directions = parameters_device.discretization;

    int thread_id = threadIdx.x + blockIdx.x * blockDim.x;
    while (thread_id < num_fluid_lattices) {
        int index = fluid_indices[thread_id]; // <-----

        real lattice_velocity_x = 0.0;
        real lattice_velocity_y = 0.0;

        for (short int component = 0; component < num_directions; ++component) {
            real distribution = population[component * num_lattices + index];
            lattice_velocity_x += coords_device[component] * distribution;
            lattice_velocity_y += coords_device[num_directions + component] * distrib
        }

        real inverse_density = 1.0 / density[index];
        velocity[index] = inverse_density * lattice_velocity_x;
        velocity[num_lattices + index] = inverse_density * lattice_velocity_y;

        thread_id += blockDim.x * gridDim.x;
    }
}

\\ MLUPS: ??.?? | ??.??  [float | double] (GeForce 610M)
```

Branch Divergence

Let's compare two implementations of the same algorithm: **without branching**

```
__global__ void UpdateVelocityFieldDevice_A(real *velocity,
                                            real *population,
                                            real *density,
                                            int *flag_field,
                                            int *fluid_indices, // additional parame
                                            int num_fluid_lattices) {
    int num_lattices = parameters_device.num_lattices;
    short int num_directions = parameters_device.discretization;

    int thread_id = threadIdx.x + blockIdx.x * blockDim.x;
    while (thread_id < num_fluid_lattices) {
        int index = fluid_indices[thread_id]; // it leads to indirect addressing

        real lattice_velocity_x = 0.0;
        real lattice_velocity_y = 0.0;

        for (short int component = 0; component < num_directions; ++component) {
            real distribution = population[component * num_lattices + index];
            lattice_velocity_x += coords_device[component] * distribution;
            lattice_velocity_y += coords_device[num_directions + component] * distrib
        }

        real inverse_density = 1.0 / density[index];
        velocity[index] = inverse_density * lattice_velocity_x;
        velocity[num_lattices + index] = inverse_density * lattice_velocity_y;

        thread_id += blockDim.x * gridDim.x;
    }
}

\\ MLUPS: 42.22 | 18.44  [float | double] (GeForce 610M)
```

Branch Divergence



- Almost **95%** of cells are fluid cells whereas there are only approximately **5%** boundary cells.
- The work load is **significantly** different during the **Collision** step.
  - For a **«FLUID»** cell, update **density, velocity, distribution**.
  - **Do nothing** for a **BC**.

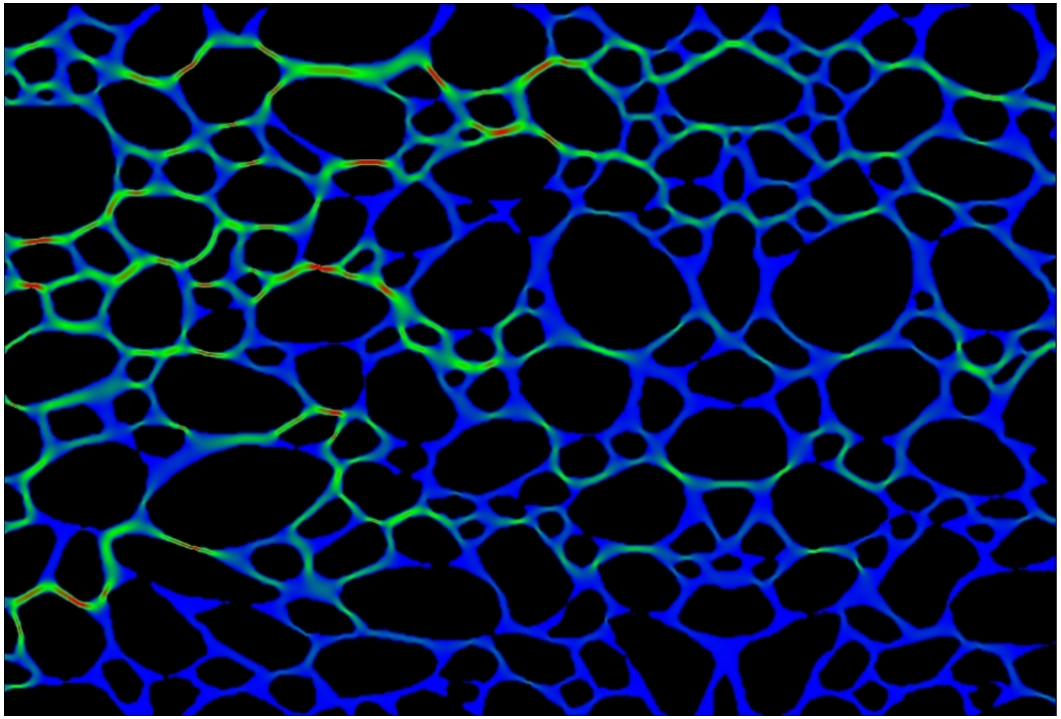
Branch Divergence

- We have just seen an example where a branching code **outperforms** a non-branching code.
- In this case, branch divergence helps preserve the key property of the algorithm: **structured mesh** (direct memory addressing).
- The implementation without branching is, in its turn, a **linked list**: the address of the next element must be fetched from the memory.
- Moreover, it needs **additional memory transfers** (expensive)

Question: What about the **Porous Media case** ? Is that also true for that scenario?

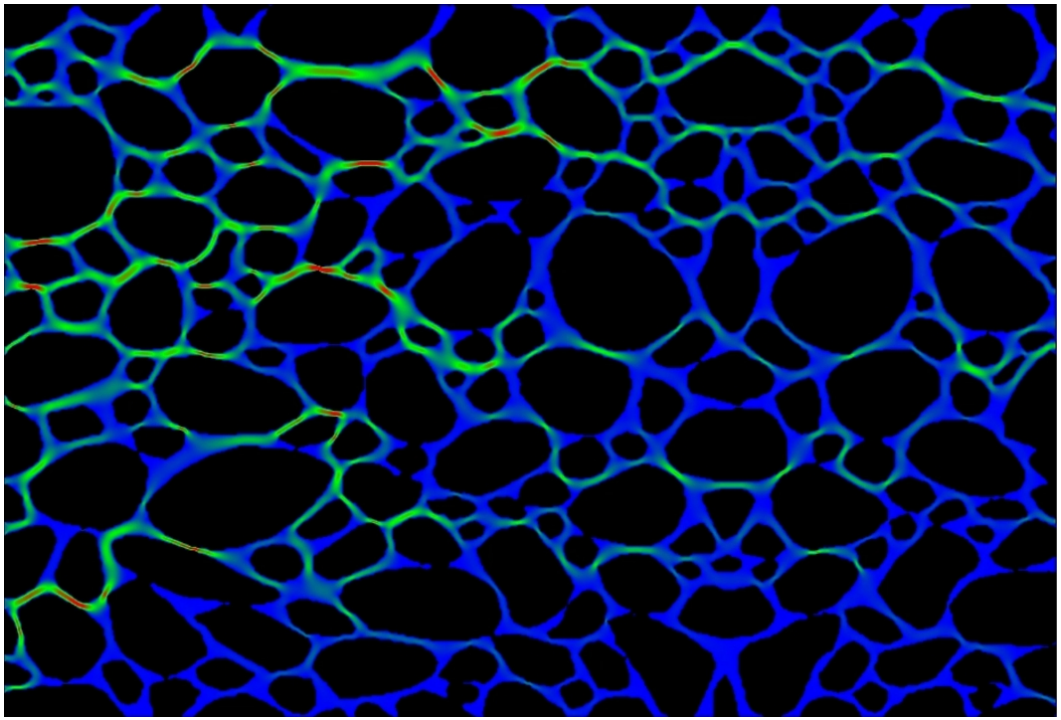
Let's take a look...

Branch Divergence: Porous Media case



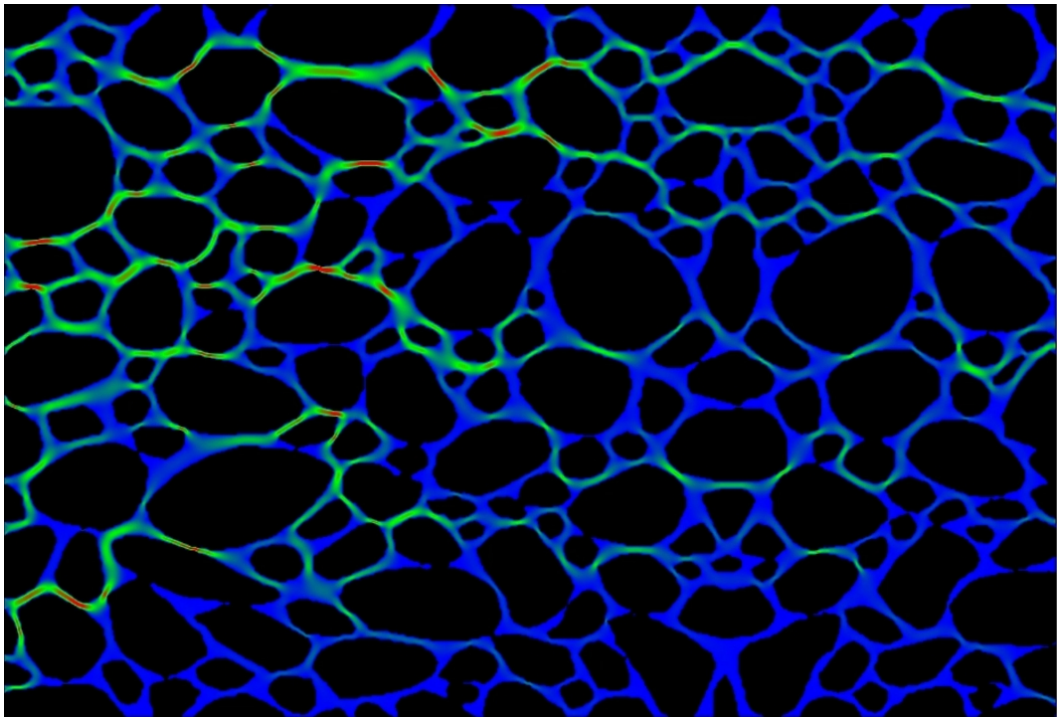
MLUPS (with branching): 46.21 | 26.55 [float | double] (Geforce 610M)
MLUPS (without branching): ??.?? | ??.?? [float | double] (Geforce 610M)

Branch Divergence: Porous Media case



MLUPS (with branching): 46.21 | 26.55 [float | double] (Geforce 610M)
MLUPS (without branching): **49.85 | 28.02** [float | double] (Geforce 610M)

Branch Divergence: Porous Media case



MLUPS (with branching): 46.21 | 26.55 [float | double] (Geforce 610M)
MLUPS (without branching): **49.85 | 28.02** [float | double] (Geforce 610M)

Note

The right decision heavily depends on your specific scenario

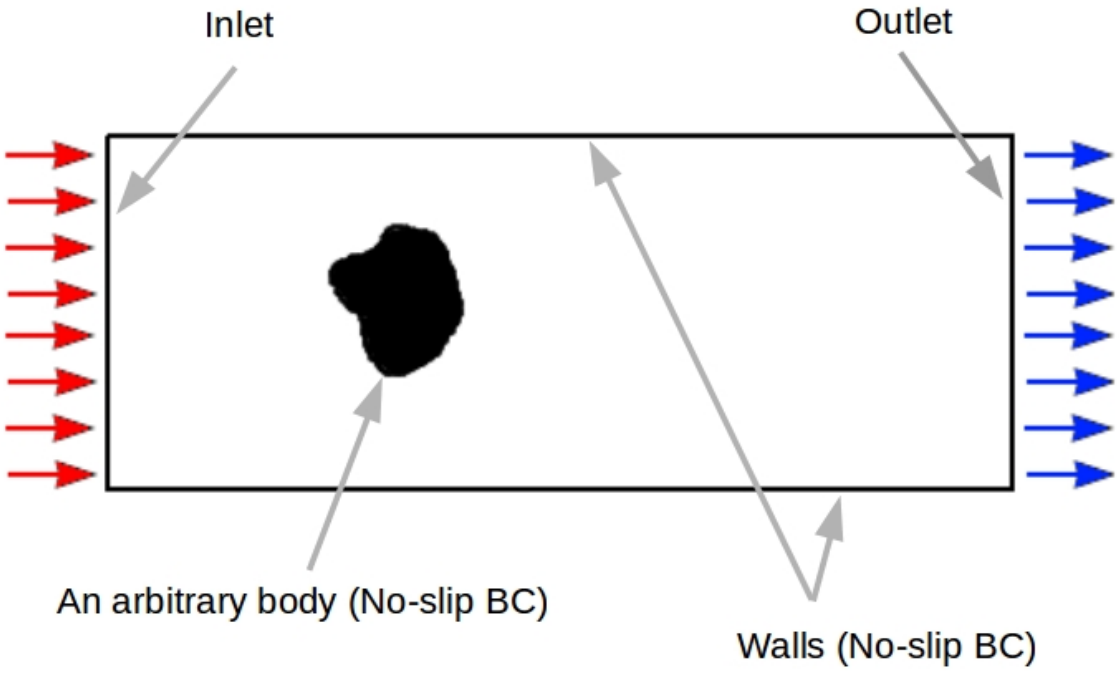
Branch Divergence

- We have just seen an example where a branching code **outperforms** a non-branching code.
- In this case, branch divergence helps preserve the key property of the algorithm: **structured mesh** (direct memory addressing).
- The implementation without branching is, in its turn, a **linked list**: the address of the next element must be fetched from the memory.
- Moreover, it needs **additional memory transfers** (expensive)
- **But** sometimes it is **better to get rid of branching**.

Question: What about Boundary Elements? Can we still use branching?

Let's look at the computational scheme one more time...

Branch Divergence: Boundary Conditions



- Almost **5%** of cells are boundary cells. The rest are fluid cells.
- The workload is **significantly** different during **Boundary Update** step.
  - For **«FLUID»**, **do nothing**
  - For a **boundary cell**, **update** corresponding boundary.

Branch Divergence: Boundary Conditions

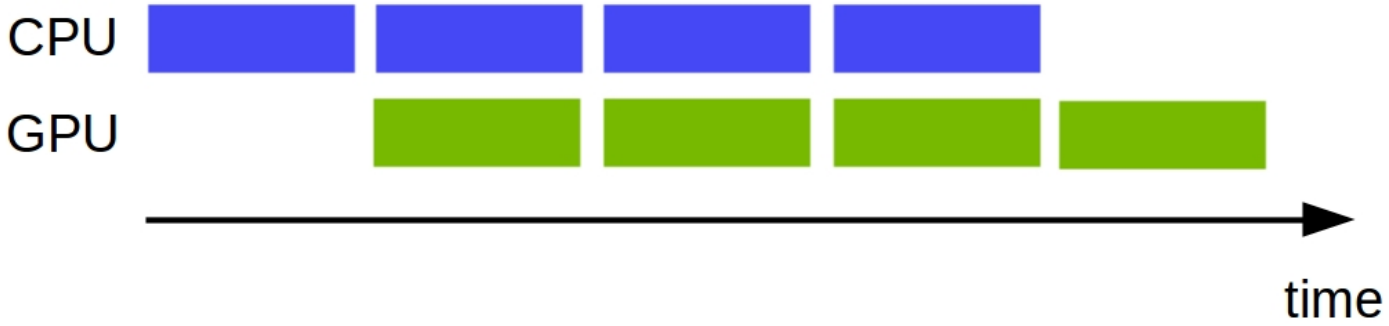
- For boundary cells, **branching** can lead to **wastage of resources**.
- Most **cores are usually idle** and are probably **waiting for a thread** in their warp.
- It is much better to **update boundary elements separately** from «FLUID» elements.

- Additionally it is better to update each boundary condition separately.
- This approach brings us **functional parallelism**
- This is a great opportunity to try out **asynchronous kernel launch** and execution.

Asynchronous kernel launch

Synchronicity in Cuda

- CUDA calls can be either **synchronous** or **asynchronous** with respect to the host
  - **Synchronous:** submit work and wait for completion
  - **Asynchronous:** submit work and return immediately
- Kernel launches are non-blocking and automatically overlap with the host



Cuda Streams

- A stream is a **queue** of device work
  - The host places work in the queue and continues on immediately
  - Device schedules work from streams when resources are free
- All CUDA operations are placed within a stream
- Operations within the **same stream** are **ordered** **(FIFO)** and cannot overlap
- Operations in **different streams** are **unordered** and can overlap

Managing Streams

- Declares a stream handle: *cudaStream_t stream;*
- Allocates a stream: *cudaStreamCreate(&stream);*
- Deallocates a stream: *cudaStreamDestroy(stream);*

Managing Streams

- Declares a stream handle: *cudaStream_t stream;*
- Allocates a stream: *cudaStreamCreate(&stream);*
- Deallocates a stream: *cudaStreamDestroy(stream);*

Question: How to assign work to a stream?

Managing Streams

- Declares a stream handle: *cudaStream_t stream;*
- Allocates a stream: *cudaStreamCreate(&stream);*
- Deallocates a stream: *cudaStreamDestroy(stream);*

Question: How to assign work to a stream?

- Stream is the 4th launch parameter

$$kernel \lll blocks, threads, smem, stream \ggg ();$$

- Streams are passed to some API calls

$$cudaMemcpyAsync(dst, src, size, dir, stream);$$

Default Stream

- If no stream is specified explicitly, a call will be assigned to **the default stream**: **Stream 0**

Default Stream

- If no stream is specified explicitly, a call will be assigned to **the default stream**: **Stream 0**
- Stream 0 has its own special synchronization rule:

Rule

Operations in stream 0 cannot overlap other streams
In other words, the hardware will wait untill all other streams finish their work before scheduling **stream 0**

Default Stream

- If no stream is specified explicitly, a call will be assigned to **the default stream**: **Stream 0**
- Stream 0 has its own special synchronization rule:

Rule

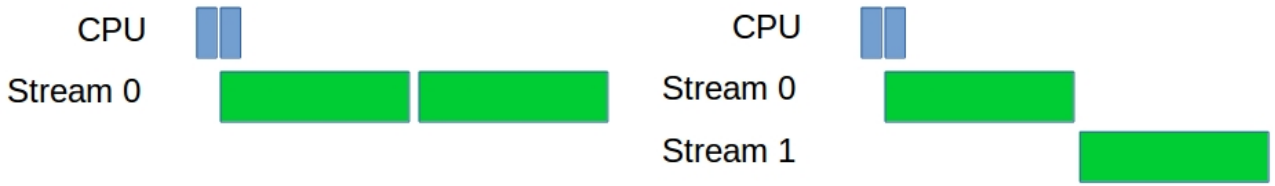
Operations in stream 0 cannot overlap with other streams.
In other words, the hardware will wait untill all other streams finish their work before scheduling **stream 0**.

Default Stream

- If no stream is specified explicitly, a call will be assigned to **the default stream**: **Stream 0**
- Stream 0 has its special synchronization rule:

Rule

Operations in stream 0 cannot overlap with other streams.
In other words, the hardware will wait untill all other streams finish their work before scheduling **stream 0**.



Default Stream

- If no stream is specified explicitly, a call will be assigned to **the default stream**: **Stream 0**
- Stream 0 has its own special synchronization rule:

Rule

Operations in stream 0 cannot overlap with other streams.
In other words, the hardware will wait untill all other streams finish their work before scheduling **stream 0**.

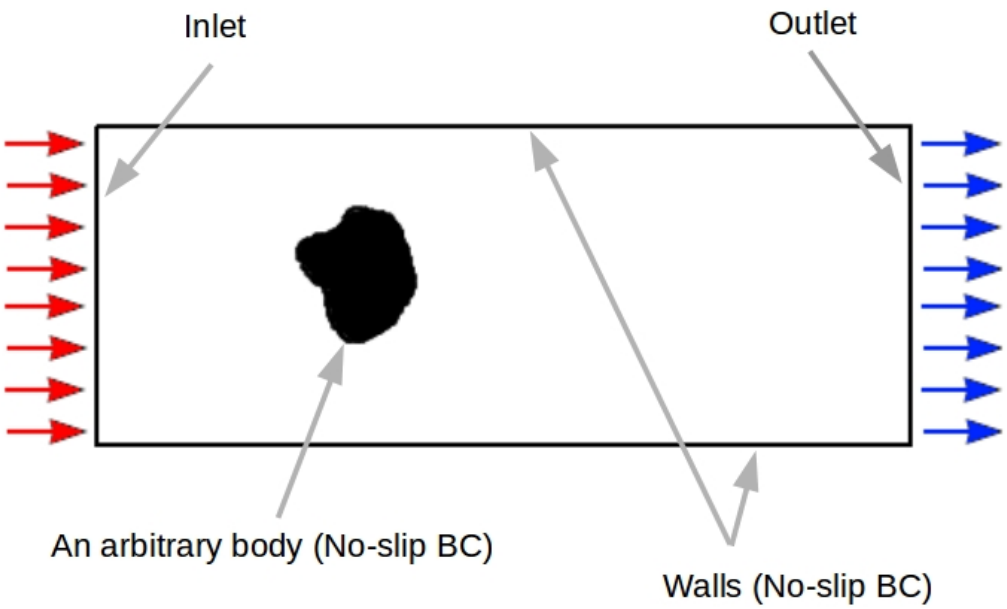
Strategies

Question: When to use asynchronous CUDA calls?

- When your kernels **cannot** occupy the entire device (all Streaming Multiprocessors)
  Attention: you need **function parallelism** for that

- When you can overlay kernel execution with data transfer between the host and device memory

Example: update of **Boundary Conditions**



Performance gain: data transfer overlap

Performance can be improved **significantly** if you can **overlay data transfer** with kernel execution.

Advice: Try to completely hide communication between host and device

Use:

$$cudaMemcpyAsync(dst, src, size, dir, stream);$$

Performance gain: function parallelism

In general it is **quite difficult** to get some decent performance gain using **functional parallelism**: usually your independent kernels are **too small**

Performance gain: function parallelism

In general it is **quite difficult** to get decent performance gain using **functional parallelism**: usually your independent kernels are **too small**

Before

Performance gain: function parallelism

In general it is **quite difficult** to get decent performance gain using **functional parallelism**: usually your independent kernels are **too small**

Before

After

Question: What performance gain did we get?

Performance gain: function parallelism

In general it is **quite difficult** to get decent performance gain using **functional parallelism**: usually your independent kernels are **too small**

Before

After

Question: What performance gain did we get?
Answer: **1.449%**

Why?

Performance gain: function parallelism

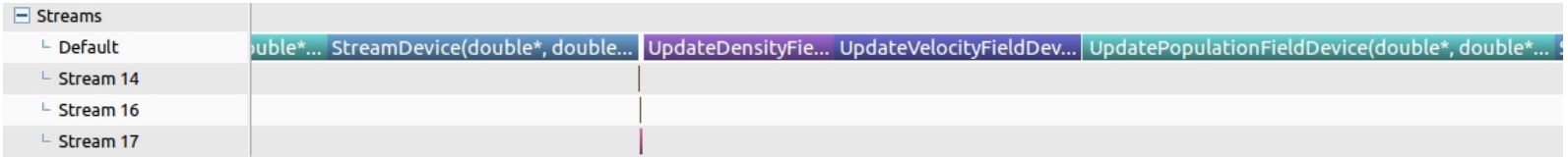
In general it is **quite difficult** to get decent performance gain using **functional parallelism**: usually your independent kernels are **too small**

Before

After

Question: What performance gain did we get?
Answer: **1.449%**

Iteration view



Performance gain: function parallelism

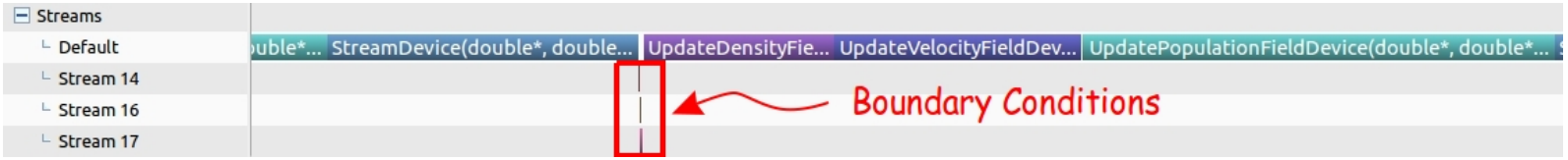
In general it is **quite difficult** to get some decent performance gain using **functional parallelism**: usually your independent kernels are **too small**

Before

After

Question: What performance gain did we get?
Answer: **1.449%**

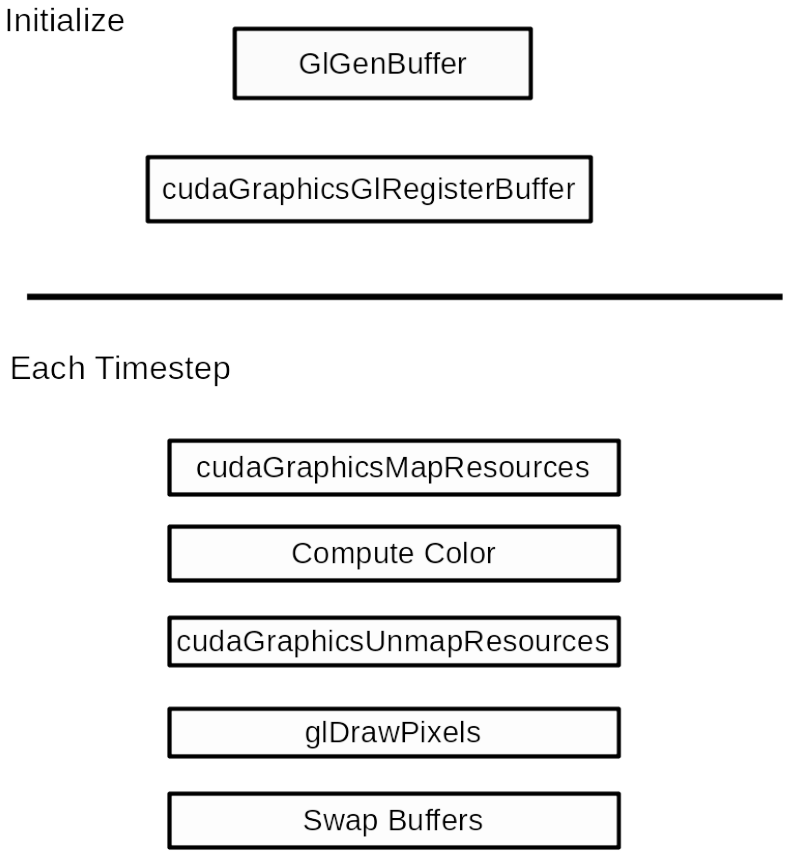
Iteration view



OpenGL

CUDA OpenGL interoperation

To visualize the results, we use a **shared buffer** between CUDA and OpenGL. This shared buffer maps the velocity magnitude for each lattice to an RGB value.



Adding Interactivity

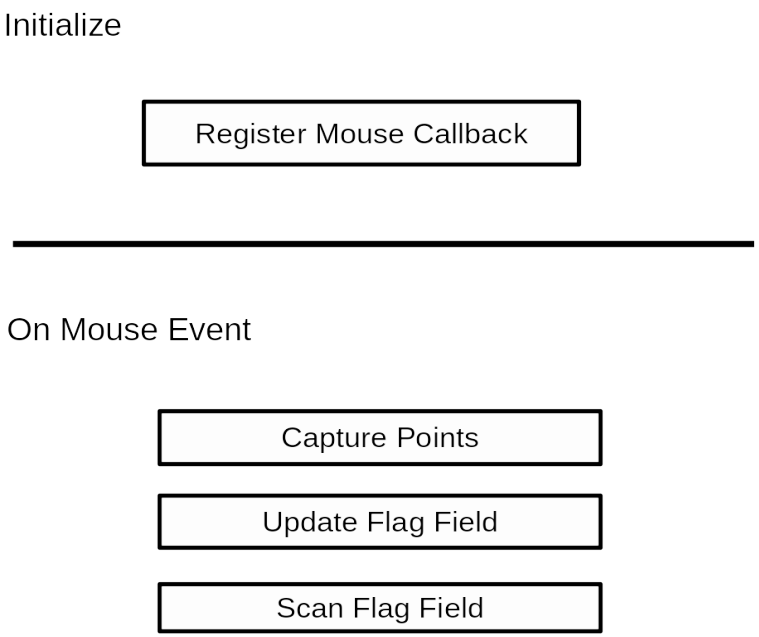
We use GLFW mouse call backs to interactively add obstacles to and remove from the domain.

- Press and drag left mouse button to add obstacles.
- Press and drag right mouse button to erase obstacles.

Adding Interactivity

We use GLFW mouse call backs to interactively add obstacles to and remove from the domain.

- Press and drag left mouse button to add obstacles.
- Press and drag right mouse button to erase obstacles.



Bibliography

References I

- Lattice Boltzmann Methods (June 8,2018).
  Retrieved from https://en.wikipedia.org/wiki/Lattice_Boltzmann_methods
- Lattice Boltzmann Method (June 13,2018).
  Retrieved from
  http://andrew.gibiansky.com/blog/physics/lattice-boltzmann-method/
- Chen Peng, The Lattice Boltzmann Method for Fluid Dynamics: Theory and Applications
  https://cmcs.epfl.ch/files/content/sites/cmcs/files/People/Peng
- CUDA C Programming Guide (June 13, 2018).
  Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions
- Pascal Tuning Guide (June 13, 2018).
  Retrieved from
  https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#cuda-best-practices
- Memory Coalescing (June 13, 2018).
  Retrieved from https://cvw.cac.cornell.edu/gpu/coalesced

References II

CUDA - Wikipedia (June 13, 2018).
Retrieved from https://en.wikipedia.org/wiki/CUDA