THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Comparison of GPU programming models for computational fluid dynamics

Robert Watt , Peter Jacobs , Rowan Gollan , and Shahzeb Imran
The University of Queensland

Australasian Fluid Mechanics Conference
December 02, 2024

The fragmentation of GPUs

Gadi: Nvidia

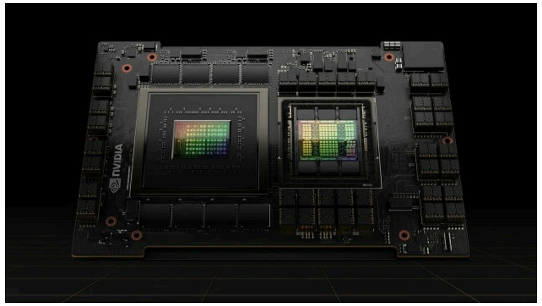

Figure 1: https://www.nvidia.com/en-au/data-center/h100/

Setonix: AMD Mi250X

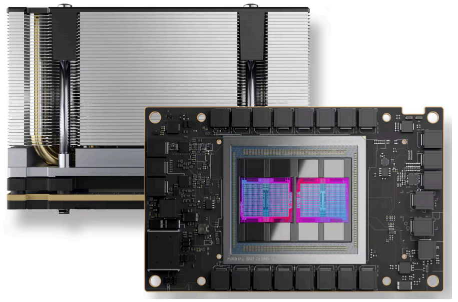

Figure 2: https://www.nvidia.com/en-au/data-center/h100/

The fragmentation of GPUs

Gadi: Nvidia

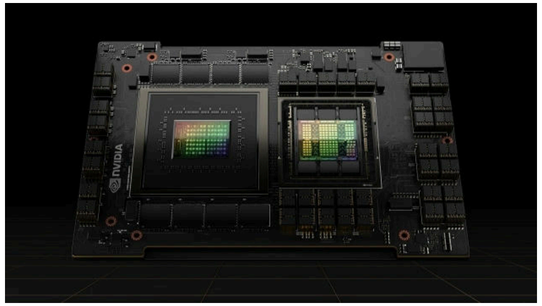
Figure 6: https://www.nvidia.com/en-au/data-center/h100/

Setonix: AMD Mi250X

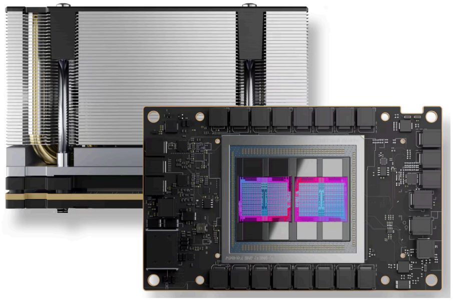
Figure 7: https://www.nvidia.com/en-au/data-center/h100/

Frontier: AMD Mi250X

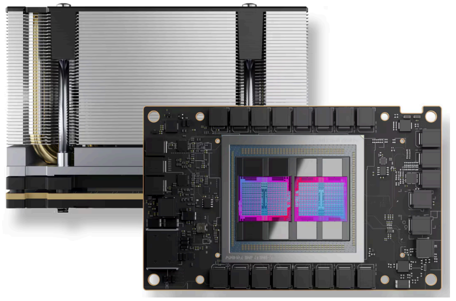
Figure 8: https://www.amd.com/en/products/accelerators/instinct/mi200/mi250x.html

El Capitan: AMD Mi300

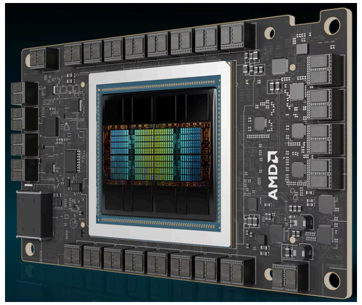
Figure 9: https://www.amd.com/en/products/accelerators/instinct/mi300.html

Aurora: Intel Data Center GPU Max

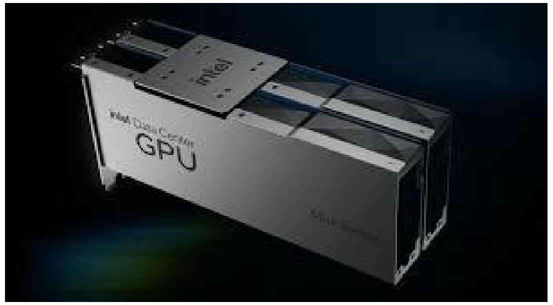
Figure 10: https://www.tweaktown.com/news/89438/intels-new-data-center-gpu-max-1100-uses-controversial-12vhpwr-connector/index.html

GPU programming models

CUDA

- Nvidia's language, for programming Nvidia GPUs
- Based on C/C++ or Fortran
- Runs on Nvidia GPUs

SYCL

- Provides an abstraction for heterogeneous programming in C++
- Implemented by various compilers (e.g. OneAPI by Intel)
- Runs on any (supported) accelerator

RAJA

- Similar to Kokkos, but split into more libraries
- Runs on any (supported) accelerator

OCCA

- Used in the first trillion degree of freedom simulation
- Runs on any (supported) accelerator

ROCm/HIP

- AMD's language for programming AMD GPUs
- Based on C/C++ or Fortran
- Runs on AMD and Nvidia GPUs

Kokkos

- C++ library providing abstractions for parallel programming
- Originally developed by Sandia National Lab, now a Linux Foundation project
- Runs on any supported accelerator
- Used in Trillinos

Chapel

- Programming language from Cray/HP for high-performance computing
- Runs on CPUs, Nvidia GPUs, with some AMD and Intel GPU support

OpenMP

- Parallelism library built into C++ compilers
- Allows for offload to many devices, including GPUs

OpenCL

- Maybe dead? Maybe not?

julia

- Programming language with libraries to execute on Nvidia and AMD GPUs

That’s a lot of options...

Which programming model should we use?

That’s a lot of options...

Which programming model should we use?

How well do they run across the different GPU brands?

That’s a lot of options…

Which programming model should we use?

How well do they run across the different GPU brands?

Todays talk: Compare Kokkos with pure CUDA

Kokkos overview

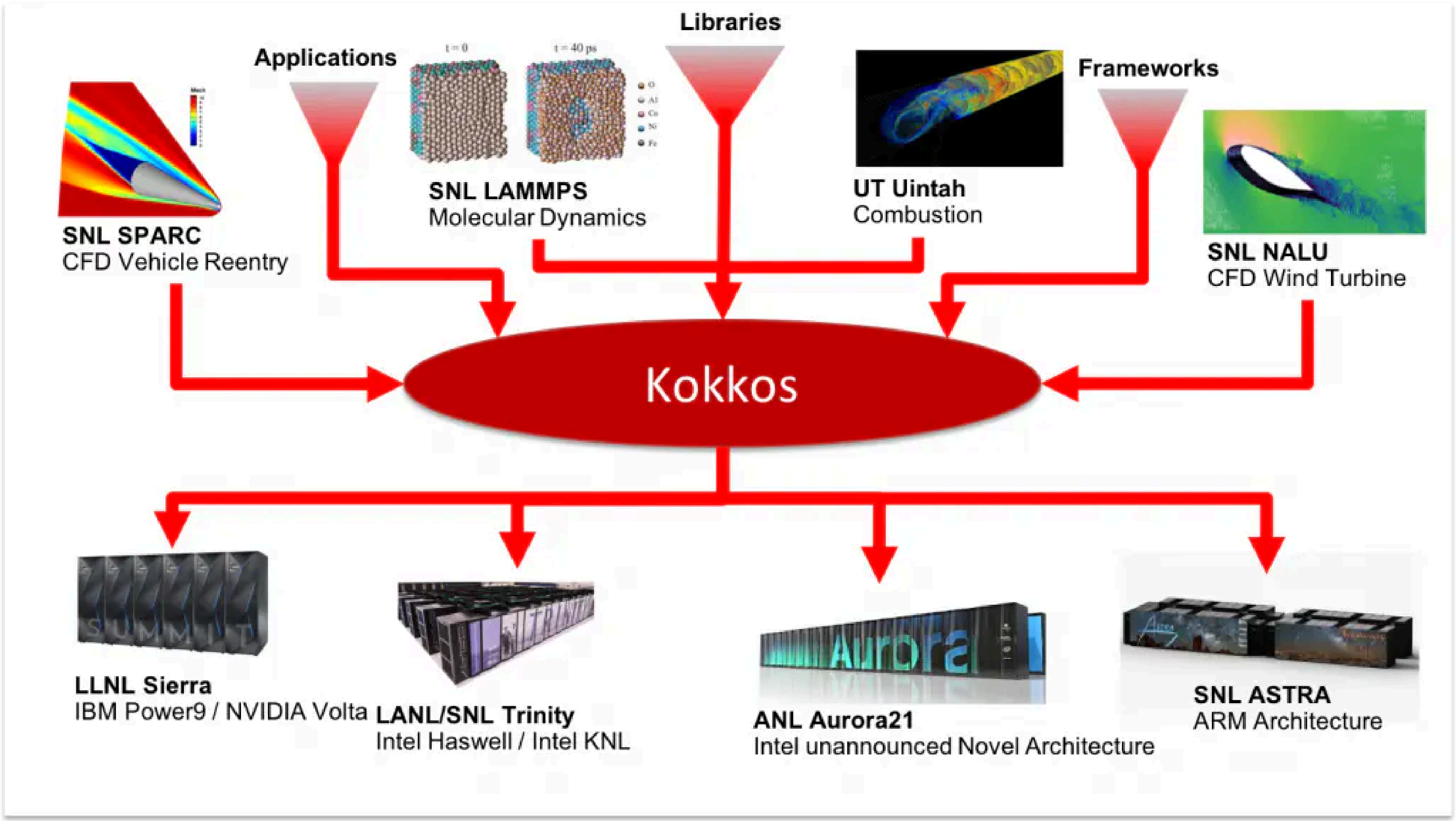

Figure 11: Kokkos architecture (https://kokkos.org/about/abstract/)

Kokkos overview

- C++ library managing mapping of parallelism of the algorithm to the parallelism of the hardware
- Can compile using vendor-provided compilers (e.g. CUDA or HIP), or can use SYCL, OpenMP, C++ threads etc.
- Kokkos manages shared memory parallelisation
  - Compatible with MPI, HPX, PGAS etc.

Advantages

- Automatic memory managment
- Compile-time configurable memory layout (row major vs column major arrays)
  - Helps get the best memory layout for a particular piece of hardware
- Transparent managment of different memory spaces
- Can perform device specific optimisations
- Free and open-source

Disadvantages

- Still C++
- Relies heavily on template meta-programming
  - may be confusing for C++ beginners
  - Compile times are longer, but not necessarily prohibitive

Example program

CUDA

```
#ifdef HIP
#define cudaMalloc hipMalloc
#define cudaFree hipFree
#endif

__global__
void initialise(double* a, int size) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < size) {
        a[i] = 0.0;
    }
}

__global__
void add_one(double* a, int size) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < size) {
        a[i] += 1.0;
    }
}

int main() {
    // allocate an array of 10 doubles on the GPU
    double* a;
    cudaMalloc(&a, 10*sizeof(double));
    initialise<<<1, 32>>>(a, 10);
    cudaDeviceSyncronize();
    add_one<<<1, 32>>>(a, 10);
    cudaDeviceSyncronize();
    cudaFree(a);
}
```

Kokkos

```
#include <Kokkos_Core.hpp>
int main() {
  // allocate an array of 10 doubles
  // on the default device
  // (these will be zero initialised)
  Kokkos::View<double*, KokkosDefaultMemSpace> a(10);

  // Add one to each double in parallel
  Kokkos::parallel_for(10, KOKKOS_LAMBDA(const int i){
      a(i) += 1.0;
  });
}
```

The codes

Chicken 🐔

- CUDA/HIP
- Block-structured grids
- Finite volume
  - Selection of upwind flux caculators
  - 2nd order accuracy via MUSCL style reconstruction
  - Van-Albada limiter
- Viscous gradients calculated with least-squares
- Time integration:
  - 3rd order Runge-Kutta time integration

The codes

Chicken 🐔

- CUDA/HIP
- Block-structured grids
- Finite volume
  - Selection of upwind flux caculators
  - 2nd order accuracy via MUSCL style reconstruction
  - Van-Albada limiter
- Viscous gradients calculated with least-squares
- Time integration:
  - 3rd order Runge-Kutta time integration

```
     MM      --
    <' \___/| --
      \_  _/  --
        ][   --
-----------------
     CHICKEN
-----------------
```

The codes

Chicken 🐔

- CUDA/HIP
- Block-structured grids
- Finite volume
  - Selection of upwind flux caculators
  - 2nd order accuracy via MUSCL style reconstruction
  - Van-Albada limiter
- Viscous gradients calculated with least-squares
- Time integration:
  - 3rd order Runge-Kutta time integration

```
     MM      --
    <' \___/| --
      \_  _/  --
        ][   --
-----------------
     CHICKEN
-----------------
```

Ibis 🗑️🐔

- Kokkos
- Unstructured grids
- Finite volume
  - Selection of upwind flux caculators
  - 2nd order accuracy via MUSCL style reconstruction
  - Barth-Jespersen limiter
- Inviscid and viscous gradients calculated with least-squares
- Time integration:
  - Any explicit Runge-Kutta
  - Jacobian-Free Newton-Krylov

The codes

Chicken 🐔

- CUDA/HIP
- **Block-structured grids**
- Finite volume
  - Selection of upwind flux caculators
  - 2nd order accuracy via MUSCL style reconstruction
  - Van-Albada limiter
- Viscous gradients calculated with least-squares
- Time integration:
  - 3rd order Runge-Kutta time integration

```
     MM       --
    <' \___/| --
      \_  _/   --
        ][    --
-----------------
     CHICKEN
-----------------
```

Ibis 🗑️🐔

- Kokkos
- **Unstructured grids**
- Finite volume
  - Selection of upwind flux caculators
  - 2nd order accuracy via MUSCL style reconstruction
  - Barth-Jespersen limiter
- **Inviscid** and viscous **gradients** calculated with least-squares
- Time integration:
  - Any explicit Runge-Kutta
  - Jacobian-Free Newton-Krylov

The codes

Chicken 🐔

- CUDA/HIP
- **Block-structured grids**
- Finite volume
  - Selection of upwind flux caculators
  - 2nd order accuracy via MUSCL style reconstruction
  - Van-Albada limiter
- Viscous gradients calculated with least-squares
- Time integration:
  - 3rd order Runge-Kutta time integration

```
     MM      --
    <' \___/| --
      \_  _/  --
        ][   --
-----------------
     CHICKEN
-----------------
```

Ibis 🗑️🐔

- Kokkos
- **Unstructured grids**
- Finite volume
  - Selection of upwind flux caculators
  - 2nd order accuracy via MUSCL style reconstruction
  - Barth-Jespersen limiter
- **Inviscid** and viscous **gradients** calculated with least-squares
- Time integration:
  - Any explicit Runge-Kutta
  - Jacobian-Free Newton-Krylov

```
   MM
  <' \___/|
   \_  _/
     ][
 ___/-\___
|---------|
 | | | | |
 | | | | |
 | | | | |
 | | | | |
 |_______|
```

The test problem

The problem

- Gaseous injection into Mach 4 cross flow

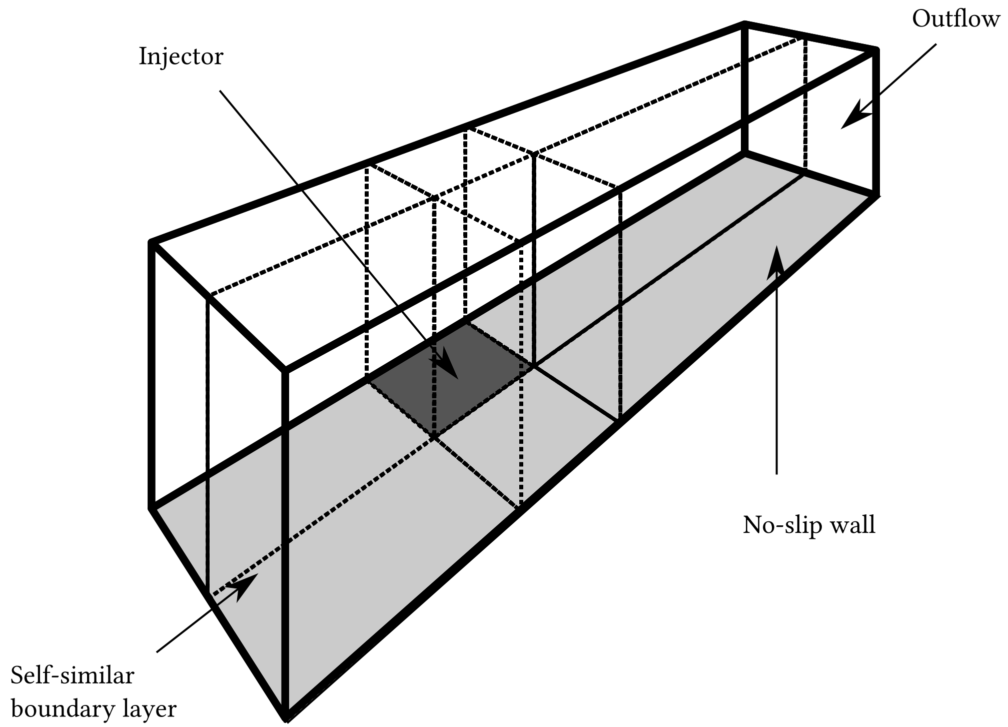

Figure 12: Domain and boundary conditions

Conditions

	Pressure (kPa)	Temperature (K)	Velocity (m/s)
Free-stream	1.013	300	1390
Injector	10.013	300	350

Grid

- Various grid resolutions from 0.5-3.5 million cells
- `Chicken` used blocking structure indicated in Figure 12
- `Ibis` merged all blocks into one block

Numerics

- AUSMDV flux calculator
- 2nd order accurate spatially
- 3rd order Runge-Kutta time-stepping

The test problem

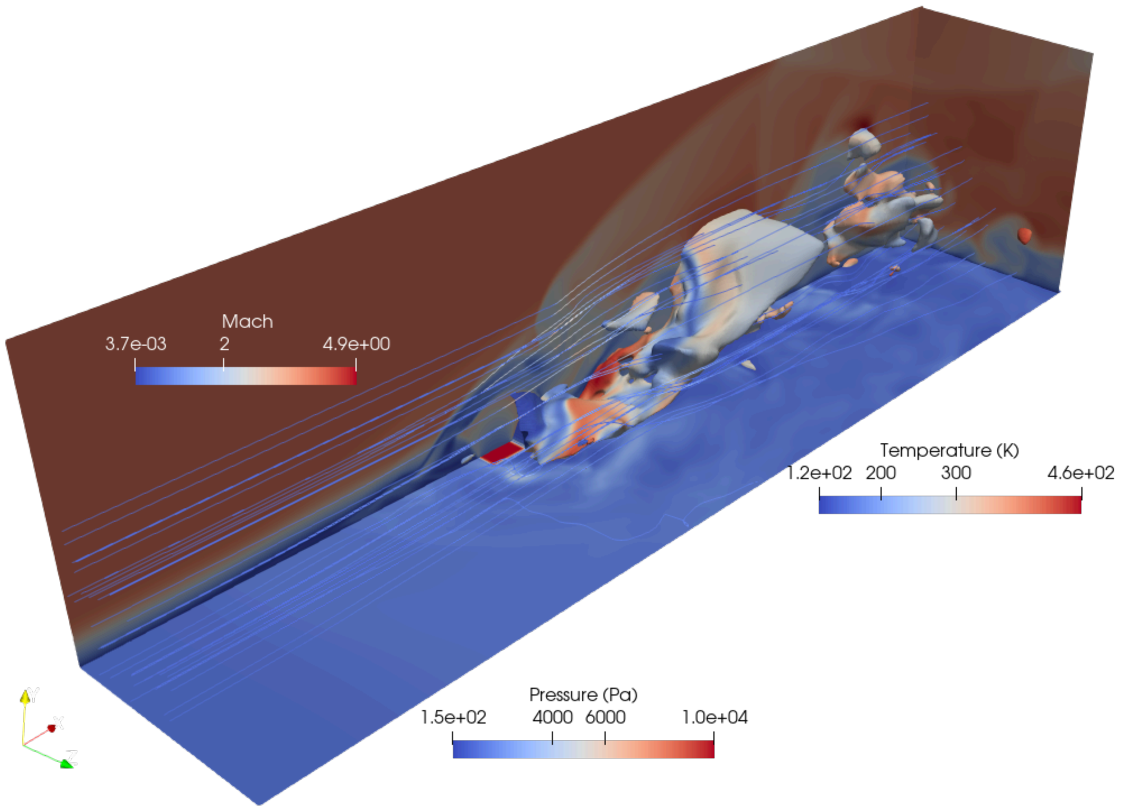

Figure 13: Flow field after 5ms

Code performance with grid resolution

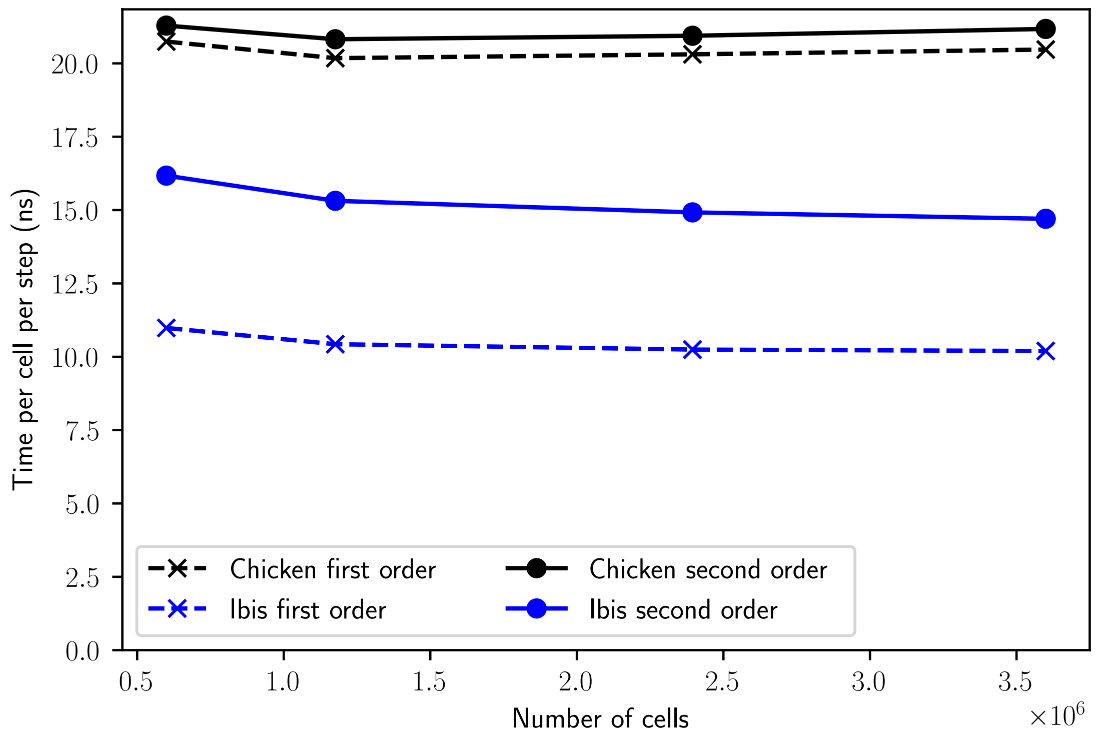

Figure 14: Time to update a cell at various grid resolutions on an Nvidia H100

Motivation

- Check to make sure the GPU is saturated
  - `Chicken` may require more cells to saturate the GPU
- Get base-line performance of each code

Reasons for differences

- `Chicken` moves memory for 2nd order around even when doing 1st order, so 1st order is slow
- `Chicken` calculates all fluxes in one large kernel
  - This may be causing register pressure, leading to lower occupancy than `Ibis`
  - The kernel with the lowest occupancy in `Ibis` had twice the occupancy of `Chicken`'s fused kernel
- `Chicken` uses array-of-structures, `Ibis` uses structure-of-arrays

Acceleration on different hardware

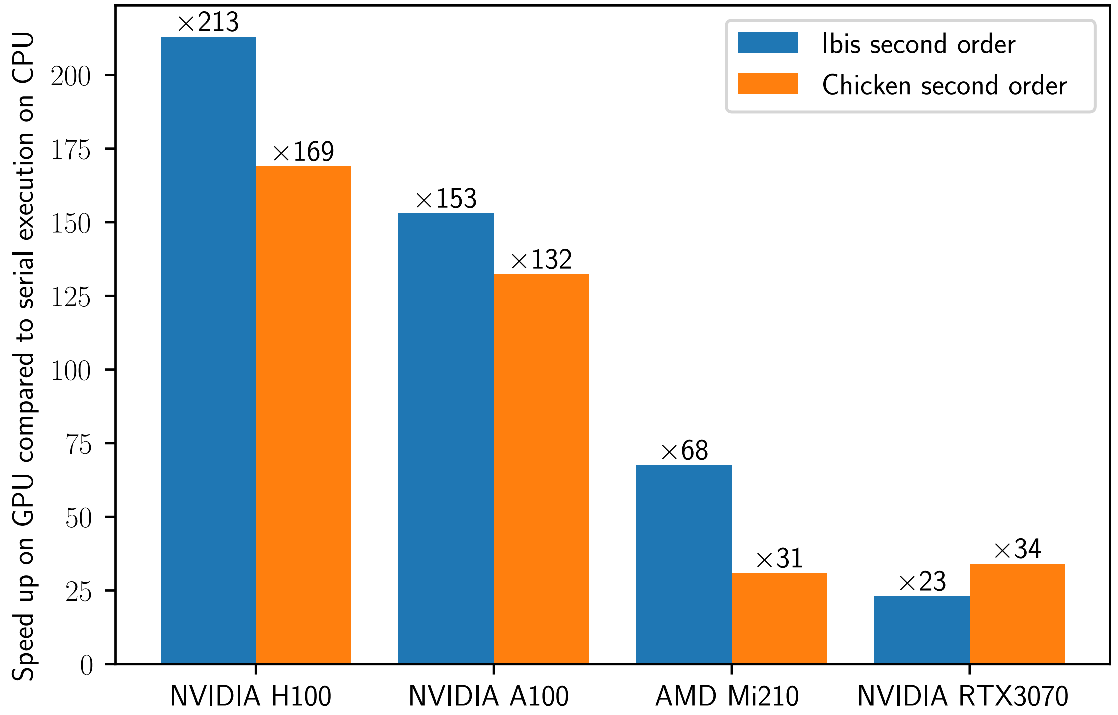

Figure 15: Acceleration compared to serial CPU performance on various GPUs

- `Ibis` (Kokkos) had greater acceleration on all HPC grade GPUs
  - Likely due to `Ibis` having greater occupancy owing to smaller kernels, smaller register count, higher occupancy, and better memory access patters
- `Chicken` (CUDA) had greater acceleration on consumer grade GPU
  - Likely due to few FP64 cores on consumer grade GPUs

GPU utilisation

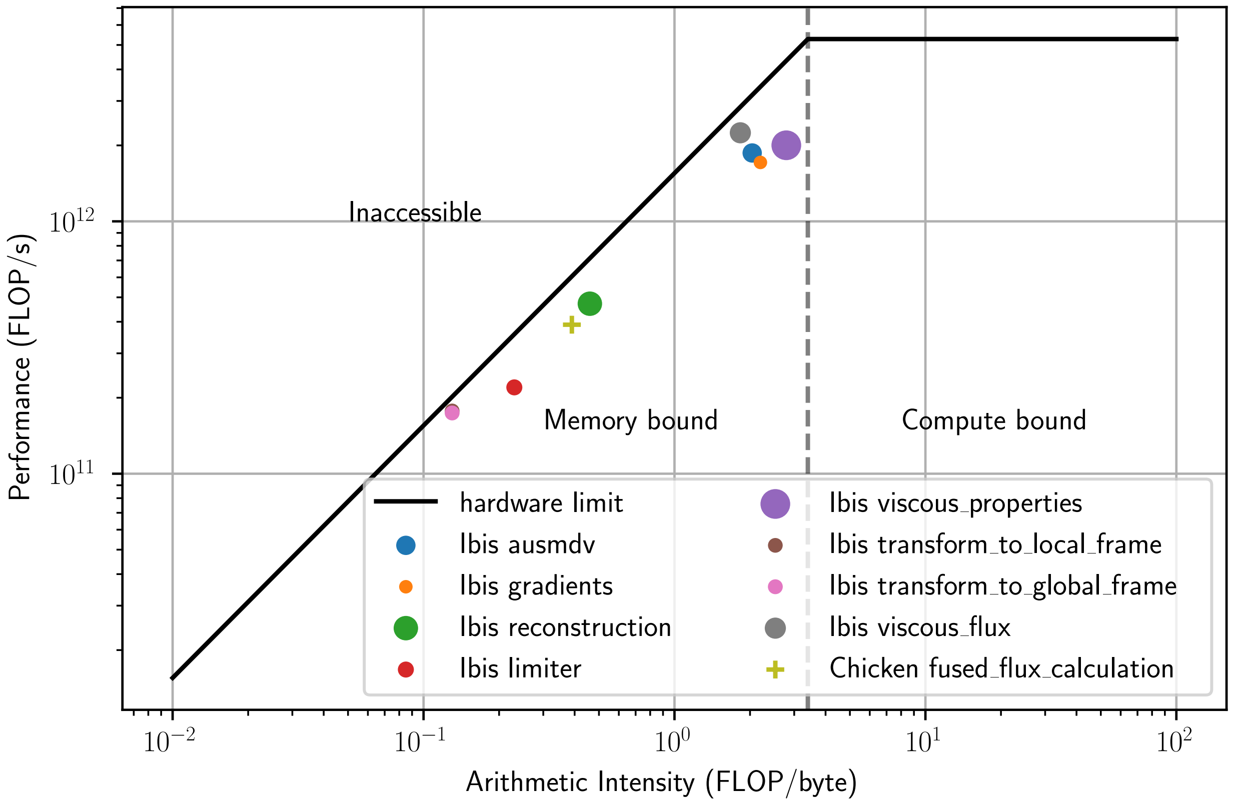

Figure 16: Roofline plot of the two codes on an Nvidia A100. Size of the circles represents time spent in that kernel

- Closer to the black bline is better (above is impossible)
- Some long-running kernels in `Ibis` benefit from being their own kernel with lower register pressure
- `Ibis` does more floating point operations per byte of data transferred partially due to better memory layout.

Conclusions

- Writing a code in CUDA doesn’t mean its fast
  - Leave the details of the parallelism to the experts, and we’ll write the physics
- Code design and memory access patterns had a larger impact on performance than overhead from abstractions
- Kokkos seems to be a good option for performance portable code
- But maybe one of the others is better?

```
    MM
  <' \___/|
   \_  _/
     ][
 ___/-\___
|---------|
 | | | | |
 | | | | |
 | | | | |
 | | | | |
 |_______|
```

```
    MM      --
  <' \___/| --
    \_  _/  --
      ][   --
----------------
    CHICKEN
----------------
```