INTRODUCTION TO

GPU PROGRAMMING

EXPECTATIONS

EXPECTATIONS

- A very broad introduction to the subject.

EXPECTATIONS

- A very broad introduction to the subject.
- Not a specifically CUDA workshop.

EXPECTATIONS

- A very broad introduction to the subject.
- Not a specifically CUDA workshop.
- Not a machine learning workshop.

EXPECTATIONS

- A very broad introduction to the subject.
- Not a specifically CUDA workshop.
- Not a machine learning workshop.
- Not an ad for GPUs.

EXPECTATIONS

- A very broad introduction to the subject.
- Not a specifically CUDA workshop.
- Not a machine learning workshop.
- Not an ad for GPUs.
- Focusing on GPUs in high-performance computing (HPC).

INTRODUCTION

WHAT IS A GPU AND WHAT IS IT GOOD FOR?

WHAT IS A GPU AND WHAT IS IT GOOD FOR?

- An electronic circuit.

WHAT IS A GPU AND WHAT IS IT GOOD FOR?

- An electronic circuit.
- Originally a *graphics* processing unit.

WHAT IS A GPU AND WHAT IS IT GOOD FOR?

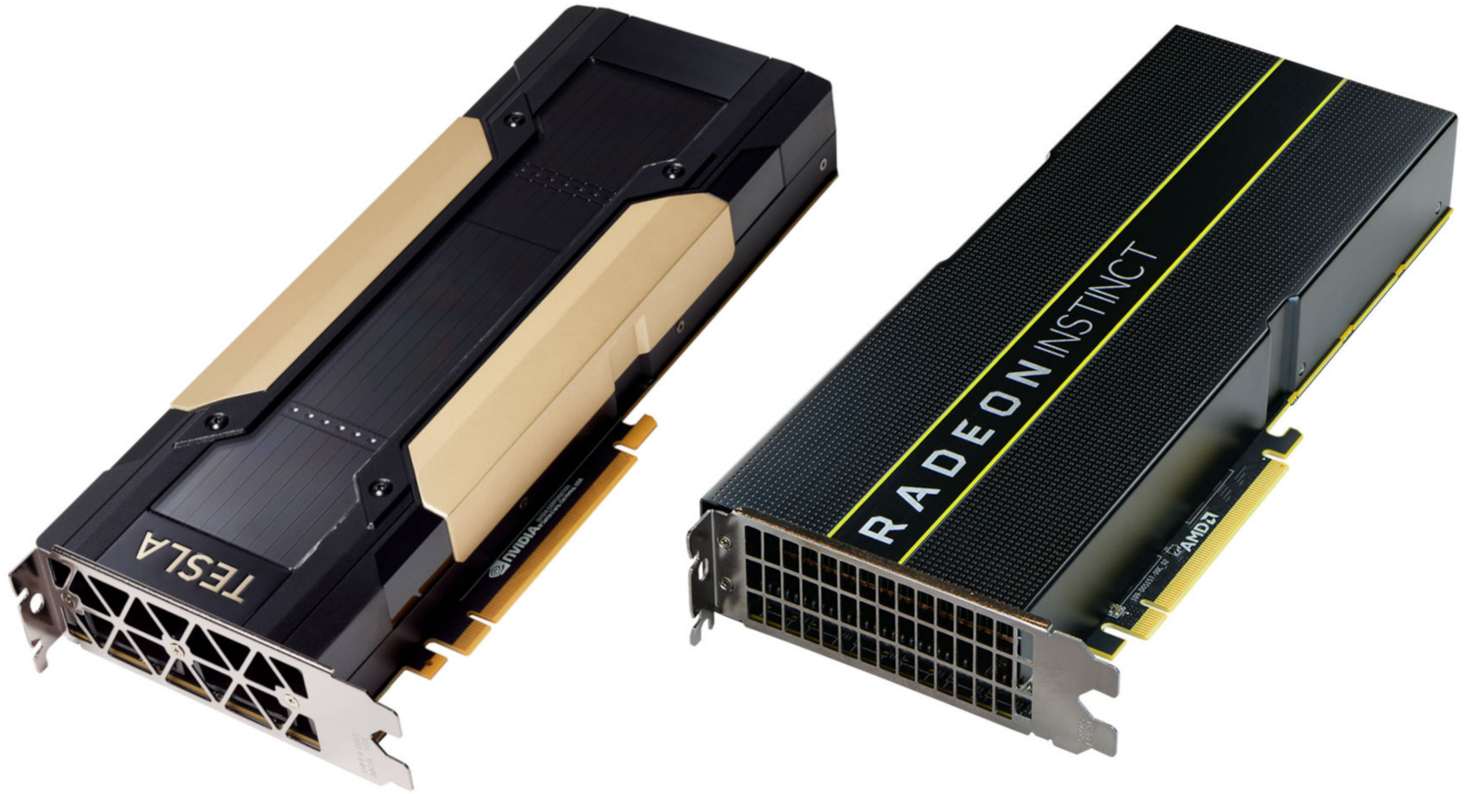
- An electronic circuit.
- Originally a *graphics* processing unit.
- Accelerates computation through *parallelization*.

WHAT IS A GPU AND WHAT IS IT GOOD FOR?

- An electronic circuit.
- Originally a *graphics* processing unit.
- Accelerates computation through *parallelization*.
- Supports *general purpose* computations.

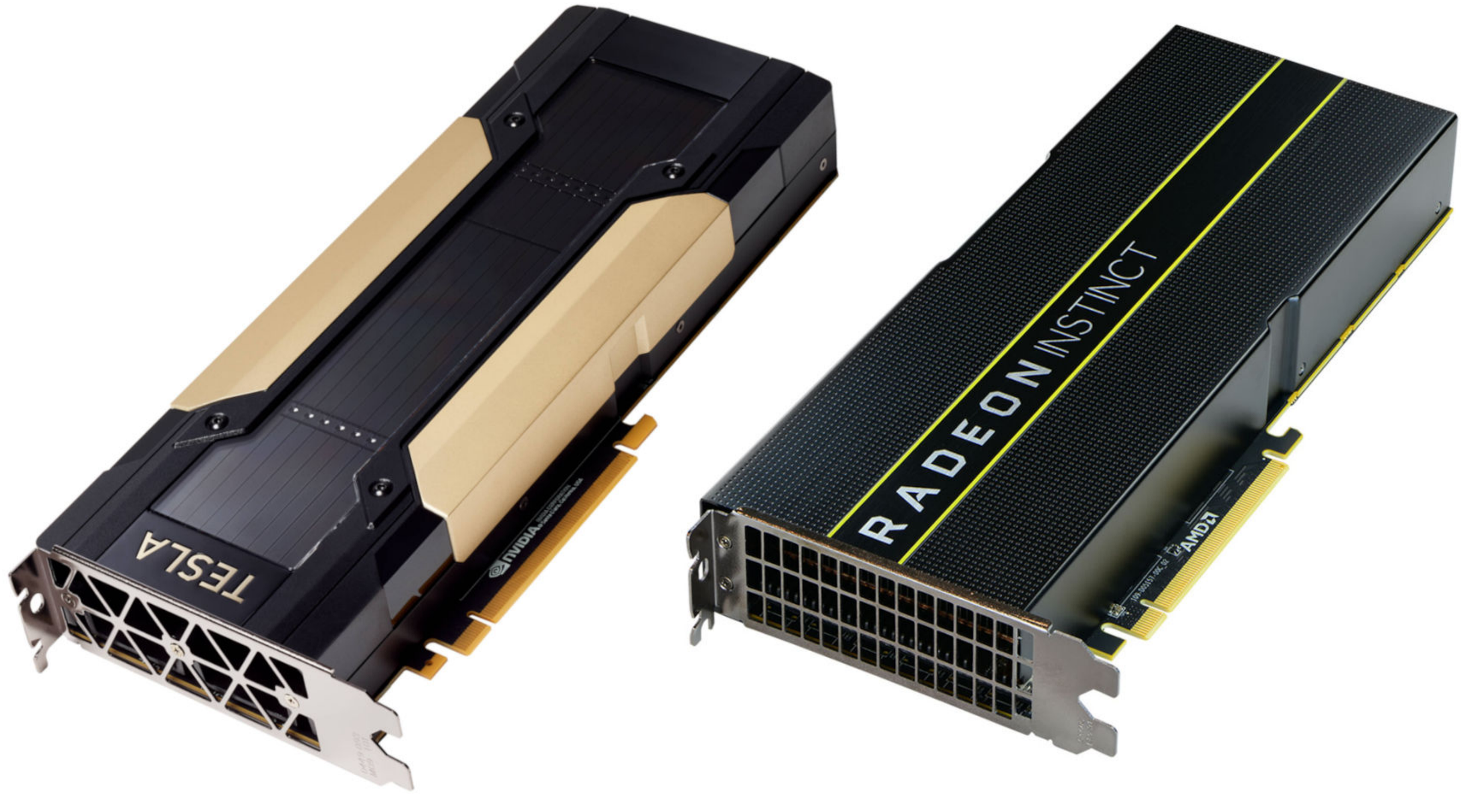
WHAT IS A GPU AND WHAT IS IT GOOD FOR?

- An electronic circuit.
- Originally a *graphics* processing unit.
- Accelerates computation through *parallelization*.
- Supports *general purpose* computations.



WHAT IS A GPU AND WHAT IS IT GOOD FOR?

- An electronic circuit.
- Originally a *graphics* processing unit.
- Accelerates computation through *parallelization*.
- Supports *general purpose* computations.



Wait, where are the fans? 🤨
…and the graphics output???

VECTORIZATION

VECTORIZATION

C CODE

```
void add(float* restrict a, float* restrict b)
{
  for (int i=0; i<16; i++)
    b[i] = a[i] + b[i];
}
```

How many *add* instructions?

VECTORIZATION

C CODE

```
void add(float* restrict a, float* restrict b)
{
  for (int i=0; i<16; i++)
    b[i] = a[i] + b[i];
}
```

How many *add* instructions?

Compile with: `gcc -O3 -mavx512f`

VECTORIZATION

C CODE

```
void add(float* restrict a, float* restrict b)
{
  for (int i=0; i<16; i++)
    b[i] = a[i] + b[i];
}
```

How many *add* instructions?

Compile with: `gcc -O3 -mavx512f`

GENERATED CPU INSTRUCTIONS

```
vmovups zmm0, ZMMWORD PTR [rdi]
vaddps  zmm0, zmm0, ZMMWORD PTR [rsi]
vmovups ZMMWORD PTR [rsi], zmm0
; ...
```

On **Niagara:** *one* add instruction!

VECTORIZATION

C CODE

```
void add(float* restrict a, float* restrict b)
{
  for (int i=0; i<16; i++)
    b[i] = a[i] + b[i];
}
```

How many *add* instructions?

Compile with: `gcc -O3 -mavx512f`

GENERATED CPU INSTRUCTIONS

```
vmovups zmm0, ZMMWORD PTR [rdi]
vaddps  zmm0, zmm0, ZMMWORD PTR [rsi]
vmovups ZMMWORD PTR [rsi], zmm0
; ...
```

On **Niagara**: *one* add instruction!

single instruction, multiple data (*SIMD*)

VECTORIZATION

C CODE

```
void add(float* restrict a, float* restrict b)
{
  for (int i=0; i<16; i++)
    b[i] = a[i] + b[i];
}
```

How many *add* instructions?

Compile with: `gcc -O3 -mavx512f`

GENERATED CPU INSTRUCTIONS

```
vmovups zmm0, ZMMWORD PTR [rdi]
vaddps  zmm0, zmm0, ZMMWORD PTR [rsi]
vmovups ZMMWORD PTR [rsi], zmm0
; ...
```

On **Niagara**: *one* add instruction!

single instruction, multiple data (*SIMD*)

- CPU programming: have to *convince the compiler* to vectorize.
- GPU programming: vectorization *by default*.

CPU VS. GPU

Intel Xeon Gold 6148 ***CPU***	**Nvidia Tesla V100** ***GPU***
20 cores	84 SMs
2 × 512 bit SIMD units / core	4 × 512 bit SIMD units / SM
2.4 GHz base clock (3.7 GHz turbo)	0.9 GHz base clock (1.3 GHz boost)
128 GB/s max. bandwidth	900 GB/s max. bandwidth

PROGRAMMING WITH A CO-PROCESSOR

PROGRAMMING WITH A CO-PROCESSOR

- Examples of hardware that can be used as a co-processors:
  - GPU, MIC, FPGA, ASIC, VE...

PROGRAMMING WITH A CO-PROCESSOR

- Examples of hardware that can be used as a co-processors:
  - GPU, MIC, FPGA, ASIC, VE...
- Architecture agnostic code $\longrightarrow$ architecture specific instructions.
  - A program has *host code* and *device code*.

PROGRAMMING WITH A CO-PROCESSOR

- Examples of hardware that can be used as a co-processors:
  - GPU, MIC, FPGA, ASIC, VE...
- Architecture agnostic code ⟶ architecture specific instructions.
  - A program has *host code* and *device code*.
- Host and device memories are *separately addressed*.
  - *Copying* data between memory spaces is (usually) required.

THE THREAD MODEL

A *thread* is a serial stream of instructions.

Threads should <u>oversubscribe</u> the “CUDA cores” (stream processors).

THE THREAD MODEL

Multiple threads are arranged in a *block*.

Threads in one block run on the same SM (compute unit).

Within a block they can synchronize and share cache memory.

THE THREAD MODEL

Multiple blocks are arranged in a *grid*.

Threads in different blocks cannot synchronize.

THE THREAD MODEL

A *host* may have different grids running on separate GPUs.

The CPU cores can work simultaneously.

THE THREAD MODEL

The *cluster* has multiple hosts.

Use MPI or NCCL/RCCL for collective communication.

THE THREAD MODEL (SUMMARY)

- Threads running in lockstep in a warp.
- One or more warps in block.
- Multiple blocks in a grid.
- Multiple grids on a host.
- The host has CPU threads as well.
- Multiple hosts in the cluster.

SETUP

COMPUTING RESOURCES FOR THE EXERCISE

COMPUTING RESOURCES FOR THE EXERCISE

- ComputeCanada users: use one of the national systems.

COMPUTING RESOURCES FOR THE EXERCISE

- ComputeCanada users: use one of the national systems.
- Non-ComputeCanada users: use your guest account on Graham.

COMPUTING RESOURCES FOR THE EXERCISE

- ComputeCanada users: use one of the national systems.
- Non-ComputeCanada users: use your guest account on Graham.
- You can also use your own workstation,

COMPUTING RESOURCES FOR THE EXERCISE

- ComputeCanada users: use one of the national systems.
- Non-ComputeCanada users: use your guest account on Graham.
- You can also use your own workstation,
- or free GPU cloud resources like Colab, Kaggle, or SageMaker Studio Lab

COMPUTING RESOURCES FOR THE EXERCISE

- ComputeCanada users: use one of the national systems.
- Non-ComputeCanada users: use your guest account on Graham.
- You can also use your own workstation,
- or free GPU cloud resources like Colab, Kaggle, or SageMaker Studio Lab

Exact type of GPU generally doesn't matter, but **not all platforms support all frameworks**.

COMPUTING RESOURCES FOR THE EXERCISE

- ComputeCanada users: use one of the national systems.
- Non-ComputeCanada users: use your guest account on Graham.
- You can also use your own workstation,
- or free GPU cloud resources like Colab, Kaggle, or SageMaker Studio Lab

Exact type of GPU generally doesn't matter, but **not all platforms support all frameworks**.

Useful links:
https://docs.computecanada.ca/wiki/Using_GPUs_with_Slurm
https://docs.computecanada.ca/wiki/Python#Creating_and_using_a_virtual_environment
https://docs.scinet.utoronto.ca/index.php/Mist

SETTING UP THE ENVIRONMENT (PYTHON)

```
mkdir $SCRATCH/scinet-hpc133
cd $SCRATCH/scinet-hpc133
```

Graham

```
module load cuda python scipy-stack
virtualenv --no-download virtualenv
source virtualenv/bin/activate
pip install --no-index --upgrade pip numba cupy
```

Mist

```
module load anaconda3
conda create -p condaenv -c conda-forge \
  python=3.8.14 numba cupy cudatoolkit=11.0.3 -y
source activate ./condaenv
conda clean --all -y
rm -rf ~/.conda/pkgs/*
```

```
wget scinet.courses/mod/2253 -O src.tar.bz2
tar xf src.tar.bz2
```

Graham

Running from login node

```
srun --time=00:01:00 --gres=gpu:p100:1 \
  python src/vector_add/08_numba.py
```

Interactive job

```
salloc --time=00:15:00 --gres=gpu:p100:1
```

* you may have to specify `--account`

Mist

Running from login node

```
python src/vector_add/08_numba.py
```

Interactive job

```
debugjob -g 1
```

GPU PROGRAMMING FRAMEWORKS

THE FRAMEWORKS

+ programs & libraries!

THE FRAMEWORKS

NVIDIA CUDA · ROCm · OpenCL · OpenACC · OpenMP · SYCL · Numba

+ programs & libraries!

- Theoretically, it doesn't matter which one you choose.
  - In practice, performance can vary.

THE FRAMEWORKS

NVIDIA CUDA · ROCm · OpenCL · OpenACC (More Science, Less Programming) · OpenMP · SYCL · Numba

+ programs & libraries!

- Theoretically, it doesn't matter which one you choose.
  - In practice, performance can vary.
- Threads execute small programs called *kernels*.

THE FRAMEWORKS

+ programs & libraries!

- Theoretically, it doesn't matter which one you choose.
  - In practice, performance can vary.
- Threads execute small programs called *kernels*.
- *Memory transfer* may be explicit or implicit.

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

- Oldest and most mature.

```cpp
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

NVIDIA CUDA

- Oldest and most mature.
- Proprietary & Nvidia-specific.

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

SciNet

NVIDIA CUDA

- Oldest and most mature.
- Proprietary & Nvidia-specific.
- Extension of C++.

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

- Oldest and most mature.
- Proprietary & Nvidia-specific.
- Extension of C++.

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

- Oldest and most mature.
- Proprietary & Nvidia-specific.
- Extension of C++.

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

SciNet

NVIDIA CUDA

- Oldest and most mature.
- Proprietary & Nvidia-specific.
- Extension of C++.

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

- Oldest and most mature.
- Proprietary & Nvidia-specific.
- Extension of C++.

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

- Oldest and most mature.
- Proprietary & Nvidia-specific.
- Extension of C++.

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

- Oldest and most mature.
- Proprietary & Nvidia-specific.
- Extension of C++.

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

SciNet

NVIDIA CUDA

- Oldest and most mature.
- Proprietary & Nvidia-specific.
- Extension of C++.

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

- Oldest and most mature.
- Proprietary & Nvidia-specific.
- Extension of C++.

```cpp
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

```cpp
#include <algorithm>
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

int main()
{
  constexpr size_t n{1'000'000};
  thrust::host_vector<float> a(n, 2.), b(n, 3.);

  thrust::device_vector<float> a_dev{a}, b_dev{b};

  thrust::transform(a_dev.begin(), a_dev.end(), b_dev.begin(),
                    b_dev.begin(), thrust::plus<float>());

  b = b_dev;

  if (std::all_of(begin(b), end(b), [](const auto x){ return x == 5.; }))
  std::cout << "Success!\n";
  else std::cout << "Failure :(\n";
}
```

- Thrust is included in CUDA.

```cpp
#include <algorithm>
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

int main()
{
  constexpr size_t n{1'000'000};
  thrust::host_vector<float> a(n, 2.), b(n, 3.);

  thrust::device_vector<float> a_dev{a}, b_dev{b};

  thrust::transform(a_dev.begin(), a_dev.end(), b_dev.begin(),
                    b_dev.begin(), thrust::plus<float>());

  b = b_dev;

  if (std::all_of(begin(b), end(b), [](const auto x){ return x == 5.; }))
  std::cout << "Success!\n";
  else std::cout << "Failure :(\n";
}
```

- Thrust is included in CUDA.
- Provides STL-like abstractions.

```cpp
#include <algorithm>
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

int main()
{
  constexpr size_t n{1'000'000};
  thrust::host_vector<float> a(n, 2.), b(n, 3.);

  thrust::device_vector<float> a_dev{a}, b_dev{b};

  thrust::transform(a_dev.begin(), a_dev.end(), b_dev.begin(),
                    b_dev.begin(), thrust::plus<float>());

  b = b_dev;

  if (std::all_of(begin(b), end(b), [](const auto x){ return x == 5.; }))
  std::cout << "Success!\n";
  else std::cout << "Failure :(\n";
}
```

- Thrust is included in CUDA.
- Provides STL-like abstractions.
- Useful library functions for reduction, sorting, etc.

```cpp
#include <algorithm>
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

int main()
{
  constexpr size_t n{1'000'000};
  thrust::host_vector<float> a(n, 2.), b(n, 3.);

  thrust::device_vector<float> a_dev{a}, b_dev{b};

  thrust::transform(a_dev.begin(), a_dev.end(), b_dev.begin(),
                    b_dev.begin(), thrust::plus<float>());

  b = b_dev;

  if (std::all_of(begin(b), end(b), [](const auto x){ return x == 5.; }))
  std::cout << "Success!\n";
  else std::cout << "Failure :(\n";
}
```

- Thrust is included in CUDA.
- Provides STL-like abstractions.
- Useful library functions for reduction, sorting, etc.

```
#include <algorithm>
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

int main()
{
  constexpr size_t n{1'000'000};
  thrust::host_vector<float> a(n, 2.), b(n, 3.);

  thrust::device_vector<float> a_dev{a}, b_dev{b};

  thrust::transform(a_dev.begin(), a_dev.end(), b_dev.begin(),
                    b_dev.begin(), thrust::plus<float>());

  b = b_dev;

  if (std::all_of(begin(b), end(b), [](const auto x){ return x == 5.; }))
  std::cout << "Success!\n";
  else std::cout << "Failure :(\n";
}
```

- Thrust is included in CUDA.
- Provides STL-like abstractions.
- Useful library functions for reduction, sorting, etc.

```cpp
#include <algorithm>
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

int main()
{
  constexpr size_t n{1'000'000};
  thrust::host_vector<float> a(n, 2.), b(n, 3.);

  thrust::device_vector<float> a_dev{a}, b_dev{b};

  thrust::transform(a_dev.begin(), a_dev.end(), b_dev.begin(),
                    b_dev.begin(), thrust::plus<float>());

  b = b_dev;

  if (std::all_of(begin(b), end(b), [](const auto x){ return x == 5.; })
  std::cout << "Success!\n";
  else std::cout << "Failure :(\n";
}
```

- Thrust is included in CUDA.
- Provides STL-like abstractions.
- Useful library functions for reduction, sorting, etc.

```cpp
#include <algorithm>
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

int main()
{
  constexpr size_t n{1'000'000};
  thrust::host_vector<float> a(n, 2.), b(n, 3.);

  thrust::device_vector<float> a_dev{a}, b_dev{b};

  thrust::transform(a_dev.begin(), a_dev.end(), b_dev.begin(),
                    b_dev.begin(), thrust::plus<float>());

  b = b_dev;

  if (std::all_of(begin(b), end(b), [](const auto x){ return x == 5.; }))
  std::cout << "Success!\n";
  else std::cout << "Failure :(\n";
}
```

- Thrust is included in CUDA.
- Provides STL-like abstractions.
- Useful library functions for reduction, sorting, etc.

```cpp
#include <algorithm>
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

int main()
{
  constexpr size_t n{1'000'000};
  thrust::host_vector<float> a(n, 2.), b(n, 3.);

  thrust::device_vector<float> a_dev{a}, b_dev{b};

  thrust::transform(a_dev.begin(), a_dev.end(), b_dev.begin(),
                    b_dev.begin(), thrust::plus<float>());

  b = b_dev;

  if (std::all_of(begin(b), end(b), [](const auto x){ return x == 5.; }))
  std::cout << "Success!\n";
  else std::cout << "Failure :(\n";
}
```

HIP

- AMD's *clone* of CUDA (mutually intelligible).
- But it's *open source*!
- Automatic conversion tools provided.

```cpp
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

```cpp
#include "hip/hip_runtime.h"
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  hipMalloc((void**)&a_dev, n*sizeof(float));
  hipMalloc((void**)&b_dev, n*sizeof(float));
  hipMemcpy(a_dev, a, n*sizeof(float), hipMemcpyHostToDevice);
  hipMemcpy(b_dev, b, n*sizeof(float), hipMemcpyHostToDevice);

  hipLaunchKernelGGL(add, dim3(n/64), dim3(64), 0, 0, a_dev, b_dev);

  hipMemcpy(b, b_dev, n*sizeof(float), hipMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  hipFree(a_dev);
  hipFree(b_dev);
}
```

```
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  cudaMalloc((void**)&a_dev, n*sizeof(float));
  cudaMalloc((void**)&b_dev, n*sizeof(float));
  cudaMemcpy(a_dev, a, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, n*sizeof(float), cudaMemcpyHostToDevice);

  add<<<n/64,64>>>(a_dev, b_dev);

  cudaMemcpy(b, b_dev, n*sizeof(float), cudaMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  cudaFree(a_dev);
  cudaFree(b_dev);
}
```

```
#include "hip/hip_runtime.h"
#include <algorithm>
#include <iostream>

__global__ void add(float *a, float *b)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = a[i] + b[i];
}

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  float *a_dev, *b_dev;
  hipMalloc((void**)&a_dev, n*sizeof(float));
  hipMalloc((void**)&b_dev, n*sizeof(float));
  hipMemcpy(a_dev, a, n*sizeof(float), hipMemcpyHostToDevice);
  hipMemcpy(b_dev, b, n*sizeof(float), hipMemcpyHostToDevice);

  hipLaunchKernelGGL(add, dim3(n/64), dim3(64), 0, 0, a_dev, b_dev);

  hipMemcpy(b, b_dev, n*sizeof(float), hipMemcpyDeviceToHost);

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  hipFree(a_dev);
  hipFree(b_dev);
}
```

OpenCL™

- A *standard* rather than a piece of software.
- Depends on *vendor implementations* (and that's a mess).
- Targets *multiple* "platforms", not just GPUs.

```
#define CL_TARGET_OPENCL_VERSION 120
#include <algorithm>
#include <CL/opencl.h>
#include <iostream>

const char *kernel_source =
"__kernel void add(__global float *a, __global float *b) \n"
"{                                                        \n"
"  int i = get_global_id(0);                              \n"
"  b[i] = a[i] + b[i];                                    \n"
"}                                                        \n";

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  cl_platform_id platform;
  clGetPlatformIDs(1, &platform, NULL);
  cl_device_id device_id;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NUL
  cl_int err;
  cl_context context = clCreateContext(0, 1, &device_id, NULL, NU
  cl_command_queue queue = clCreateCommandQueue(context, device_i

  cl_mem a_dev;
  cl_mem b_dev;
  a_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,  n*sizeof(flo
  b_dev = clCreateBuffer(context, CL_MEM_READ_WRITE, n*sizeof(flo
  clEnqueueWriteBuffer(queue, a_dev, CL_TRUE, 0, n*sizeof(float),
  clEnqueueWriteBuffer(queue, b_dev, CL_TRUE, 0, n*sizeof(float),

  cl_program program = clCreateProgramWithSource(context, 1, &ker
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  cl_kernel kernel = clCreateKernel(program, "add", &err);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_dev);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_dev);
  const size_t global_size{n}, local_size{64};
```

OpenCL™

- A *standard* rather than a piece of software.
- Depends on *vendor implementations* (and that's a mess).
- Targets *multiple* "platforms", not just GPUs.
- Has tonnes of *boilerplate* and a steep learning curve.

```
#define CL_TARGET_OPENCL_VERSION 120
#include <algorithm>
#include <CL/opencl.h>
#include <iostream>

const char *kernel_source =
"__kernel void add(__global float *a, __global float *b) \n"
"{                                                        \n"
"  int i = get_global_id(0);                              \n"
"  b[i] = a[i] + b[i];                                    \n"
"}                                                        \n";

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  cl_platform_id platform;
  clGetPlatformIDs(1, &platform, NULL);
  cl_device_id device_id;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NUL
  cl_int err;
  cl_context context = clCreateContext(0, 1, &device_id, NULL, NU
  cl_command_queue queue = clCreateCommandQueue(context, device_i

  cl_mem a_dev;
  cl_mem b_dev;
  a_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,  n*sizeof(flo
  b_dev = clCreateBuffer(context, CL_MEM_READ_WRITE, n*sizeof(flo
  clEnqueueWriteBuffer(queue, a_dev, CL_TRUE, 0, n*sizeof(float),
  clEnqueueWriteBuffer(queue, b_dev, CL_TRUE, 0, n*sizeof(float),

  cl_program program = clCreateProgramWithSource(context, 1, &ker
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  cl_kernel kernel = clCreateKernel(program, "add", &err);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_dev);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_dev);
  const size_t global_size{n}, local_size{64};
```

OpenCL™

- A *standard* rather than a piece of software.
- Depends on *vendor implementations* (and that's a mess).
- Targets *multiple* "platforms", not just GPUs.
- Has tonnes of *boilerplate* and a steep learning curve.
- Strongly inspired by *CUDA*.

```
#define CL_TARGET_OPENCL_VERSION 120
#include <algorithm>
#include <CL/opencl.h>
#include <iostream>

const char *kernel_source =
"__kernel void add(__global float *a, __global float *b) \n"
"{                                                        \n"
"  int i = get_global_id(0);                              \n"
"  b[i] = a[i] + b[i];                                    \n"
"}                                                        \n";

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  cl_platform_id platform;
  clGetPlatformIDs(1, &platform, NULL);
  cl_device_id device_id;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NUL
  cl_int err;
  cl_context context = clCreateContext(0, 1, &device_id, NULL, NU
  cl_command_queue queue = clCreateCommandQueue(context, device_i

  cl_mem a_dev;
  cl_mem b_dev;
  a_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,  n*sizeof(flo
  b_dev = clCreateBuffer(context, CL_MEM_READ_WRITE, n*sizeof(flo
  clEnqueueWriteBuffer(queue, a_dev, CL_TRUE, 0, n*sizeof(float),
  clEnqueueWriteBuffer(queue, b_dev, CL_TRUE, 0, n*sizeof(float),

  cl_program program = clCreateProgramWithSource(context, 1, &ker
  clBuildProgram(program, 0, NULL, NULL, NULL);
  cl_kernel kernel = clCreateKernel(program, "add", &err);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_dev);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_dev);
  const size_t global_size{n}, local_size{64};
```

OpenCL™

- A *standard* rather than a piece of software.
- Depends on *vendor implementations* (and that's a mess).
- Targets *multiple* "platforms", not just GPUs.
- Has tonnes of *boilerplate* and a steep learning curve.
- Strongly inspired by *CUDA*.
- Kernels compiled *just in time* from strings.

```
#define CL_TARGET_OPENCL_VERSION 120
#include <algorithm>
#include <CL/opencl.h>
#include <iostream>

const char *kernel_source =
"__kernel void add(__global float *a, __global float *b) \n"
"{                                                        \n"
"  int i = get_global_id(0);                              \n"
"  b[i] = a[i] + b[i];                                    \n"
"}                                                        \n";

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  cl_platform_id platform;
  clGetPlatformIDs(1, &platform, NULL);
  cl_device_id device_id;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NUL
  cl_int err;
  cl_context context = clCreateContext(0, 1, &device_id, NULL, NU
  cl_command_queue queue = clCreateCommandQueue(context, device_i

  cl_mem a_dev;
  cl_mem b_dev;
  a_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,  n*sizeof(flo
  b_dev = clCreateBuffer(context, CL_MEM_READ_WRITE, n*sizeof(flo
  clEnqueueWriteBuffer(queue, a_dev, CL_TRUE, 0, n*sizeof(float),
  clEnqueueWriteBuffer(queue, b_dev, CL_TRUE, 0, n*sizeof(float),

  cl_program program = clCreateProgramWithSource(context, 1, &ker
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  cl_kernel kernel = clCreateKernel(program, "add", &err);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_dev);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_dev);
  const size_t global_size{n}, local_size{64};
```

OpenCL™

- A *standard* rather than a piece of software.
- Depends on *vendor implementations* (and that's a mess).
- Targets *multiple* "platforms", not just GPUs.
- Has tonnes of *boilerplate* and a steep learning curve.
- Strongly inspired by *CUDA*.
- Kernels compiled *just in time* from strings.
- Before HIP, was the only way to program *AMD* GPUs.

```
#define CL_TARGET_OPENCL_VERSION 120
#include <algorithm>
#include <CL/opencl.h>
#include <iostream>

const char *kernel_source =
"__kernel void add(__global float *a, __global float *b) \n"
"{                                                        \n"
"  int i = get_global_id(0);                              \n"
"  b[i] = a[i] + b[i];                                    \n"
"}                                                        \n";

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  cl_platform_id platform;
  clGetPlatformIDs(1, &platform, NULL);
  cl_device_id device_id;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NUL
  cl_int err;
  cl_context context = clCreateContext(0, 1, &device_id, NULL, NU
  cl_command_queue queue = clCreateCommandQueue(context, device_i

  cl_mem a_dev;
  cl_mem b_dev;
  a_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,  n*sizeof(flo
  b_dev = clCreateBuffer(context, CL_MEM_READ_WRITE, n*sizeof(flo
  clEnqueueWriteBuffer(queue, a_dev, CL_TRUE, 0, n*sizeof(float),
  clEnqueueWriteBuffer(queue, b_dev, CL_TRUE, 0, n*sizeof(float),

  cl_program program = clCreateProgramWithSource(context, 1, &ker
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  cl_kernel kernel = clCreateKernel(program, "add", &err);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_dev);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_dev);
  const size_t global_size{n}, local_size{64};
```

OpenCL™

- A *standard* rather than a piece of software.
- Depends on *vendor implementations* (and that's a mess).
- Targets *multiple* "platforms", not just GPUs.
- Has tonnes of *boilerplate* and a steep learning curve.
- Strongly inspired by *CUDA*.
- Kernels compiled *just in time* from strings.
- Before HIP, was the only way to program *AMD* GPUs.
- Nvidia's OpenCL implementation:
  - is historically slow compared to the CUDA runtime.
  - is historically behind the standard.
  - doesn't even work on PPC64LE (Mist).

```
#define CL_TARGET_OPENCL_VERSION 120
#include <algorithm>
#include <CL/opencl.h>
#include <iostream>

const char *kernel_source =
"__kernel void add(__global float *a, __global float *b) \n"
"{                                                        \n"
"  int i = get_global_id(0);                              \n"
"  b[i] = a[i] + b[i];                                    \n"
"}                                                        \n";

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  cl_platform_id platform;
  clGetPlatformIDs(1, &platform, NULL);
  cl_device_id device_id;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NUL
  cl_int err;
  cl_context context = clCreateContext(0, 1, &device_id, NULL, NU
  cl_command_queue queue = clCreateCommandQueue(context, device_i

  cl_mem a_dev;
  cl_mem b_dev;
  a_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,  n*sizeof(flo
  b_dev = clCreateBuffer(context, CL_MEM_READ_WRITE, n*sizeof(flo
  clEnqueueWriteBuffer(queue, a_dev, CL_TRUE, 0, n*sizeof(float),
  clEnqueueWriteBuffer(queue, b_dev, CL_TRUE, 0, n*sizeof(float),

  cl_program program = clCreateProgramWithSource(context, 1, &ker
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  cl_kernel kernel = clCreateKernel(program, "add", &err);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_dev);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_dev);
  const size_t global_size{n}, local_size{64};
```

OpenCL™

- A *standard* rather than a piece of software.
- Depends on *vendor implementations* (and that's a mess).
- Targets *multiple* "platforms", not just GPUs.
- Has tonnes of *boilerplate* and a steep learning curve.
- Strongly inspired by *CUDA*.
- Kernels compiled *just in time* from strings.
- Before HIP, was the only way to program *AMD* GPUs.
- Nvidia's OpenCL implementation:
  - is historically slow compared to the CUDA runtime.
  - is historically behind the standard.
  - doesn't even work on PPC64LE (Mist).
- Not recommended for beginners (or anyone) in 2023.

```
#define CL_TARGET_OPENCL_VERSION 120
#include <algorithm>
#include <CL/opencl.h>
#include <iostream>

const char *kernel_source =
"__kernel void add(__global float *a, __global float *b) \n"
"{                                                       \n"
"  int i = get_global_id(0);                             \n"
"  b[i] = a[i] + b[i];                                   \n"
"}                                                       \n";

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  cl_platform_id platform;
  clGetPlatformIDs(1, &platform, NULL);
  cl_device_id device_id;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NUL
  cl_int err;
  cl_context context = clCreateContext(0, 1, &device_id, NULL, NU
  cl_command_queue queue = clCreateCommandQueue(context, device_i

  cl_mem a_dev;
  cl_mem b_dev;
  a_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,  n*sizeof(flo
  b_dev = clCreateBuffer(context, CL_MEM_READ_WRITE, n*sizeof(flo
  clEnqueueWriteBuffer(queue, a_dev, CL_TRUE, 0, n*sizeof(float),
  clEnqueueWriteBuffer(queue, b_dev, CL_TRUE, 0, n*sizeof(float),

  cl_program program = clCreateProgramWithSource(context, 1, &ker
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  cl_kernel kernel = clCreateKernel(program, "add", &err);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_dev);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_dev);
  const size_t global_size{n}, local_size{64};
```

```cpp
#include <algorithm>
#include <iostream>

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  #pragma acc kernels
  #pragma acc loop independent
  for (int i = 0; i < n; i++)
    b[i] = a[i] + b[i];

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
}
```

Example of OpenACC.

- *Directive*-based approaches.

```cpp
#include <algorithm>
#include <iostream>

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  #pragma acc kernels
  #pragma acc loop independent
  for (int i = 0; i < n; i++)
    b[i] = a[i] + b[i];

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
}
```

Example of OpenACC.

- *Directive*-based approaches.
- "Easy" to accelerate existing CPU code.

```cpp
#include <algorithm>
#include <iostream>

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  #pragma acc kernels
  #pragma acc loop independent
  for (int i = 0; i < n; i++)
    b[i] = a[i] + b[i];

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
}
```

Example of OpenACC.

OpenMP OpenACC

More Science, Less Programming

- *Directive*-based approaches.
- “Easy” to accelerate existing CPU code.
- Single code base for CPU and GPU.

```
#include <algorithm>
#include <iostream>

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  #pragma acc kernels
  #pragma acc loop independent
  for (int i = 0; i < n; i++)
    b[i] = a[i] + b[i];

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
}
```

Example of OpenACC.

- *Directive*-based approaches.
- “Easy” to accelerate existing CPU code.
- Single code base for CPU and GPU.
- Shared origin and mutually intelligible.

```cpp
#include <algorithm>
#include <iostream>

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  #pragma acc kernels
  #pragma acc loop independent
  for (int i = 0; i < n; i++)
    b[i] = a[i] + b[i];

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
}
```

Example of OpenACC.

OpenMP OpenACC

More Science, Less Programming

- *Directive*-based approaches.
- “Easy” to accelerate existing CPU code.
- Single code base for CPU and GPU.
- Shared origin and mutually intelligible.
- Kernels are generated *from loops*.

```cpp
#include <algorithm>
#include <iostream>

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  #pragma acc kernels
  #pragma acc loop independent
  for (int i = 0; i < n; i++)
    b[i] = a[i] + b[i];

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
}
```

Example of OpenACC.

OpenMP

OpenACC
More Science, Less Programming

- *Directive*-based approaches.
- “Easy” to accelerate existing CPU code.
- Single code base for CPU and GPU.
- Shared origin and mutually intelligible.
- Kernels are generated *from loops*.
- Memory copies [kinda] automatic.

```cpp
#include <algorithm>
#include <iostream>

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  #pragma acc kernels
  #pragma acc loop independent
  for (int i = 0; i < n; i++)
    b[i] = a[i] + b[i];

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
}
```

Example of OpenACC.

More Science, Less Programming

- *Directive*-based approaches.
- “Easy” to accelerate existing CPU code.
- Single code base for CPU and GPU.
- Shared origin and mutually intelligible.
- Kernels are generated *from loops*.
- Memory copies [kinda] automatic.

```cpp
#include <algorithm>
#include <iostream>

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  #pragma omp target data map(to: a[:n]) map(tofrom: b[:n])
  #pragma omp target parallel for
  for (int i = 0; i < n; i++)
    b[i] = a[i] + b[i];

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
}
```

Example of OpenMP offloading.

```cpp
#include <algorithm>
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl;

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  sycl::device device{sycl::default_selector()};
  sycl::context context{device};
  sycl::queue queue{context, device};

  float* a_dev{sycl::malloc_device<float>(n, device, context)};
  float* b_dev{sycl::malloc_device<float>(n, device, context)};

  queue.copy(a, a_dev, n);
  queue.copy(b, b_dev, n);
  queue.wait();

  queue.parallel_for(sycl::range<1>{n},
    [=](const auto i) { b_dev[i] = a_dev[i] + b_dev[i]; }
  );
  queue.wait();

  queue.copy(b_dev, b, n);
  queue.wait();

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  sycl::free(a_dev, context);
  sycl::free(b_dev, context);
}
```

- Another standard developed by Khronos Group.

```cpp
#include <algorithm>
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl;

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  sycl::device device{sycl::default_selector()};
  sycl::context context{device};
  sycl::queue queue{context, device};

  float* a_dev{sycl::malloc_device<float>(n, device, context)};
  float* b_dev{sycl::malloc_device<float>(n, device, context)};

  queue.copy(a, a_dev, n);
  queue.copy(b, b_dev, n);
  queue.wait();

  queue.parallel_for(sycl::range<1>{n},
    [=](const auto i) { b_dev[i] = a_dev[i] + b_dev[i]; }
  );
  queue.wait();

  queue.copy(b_dev, b, n);
  queue.wait();

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  sycl::free(a_dev, context);
  sycl::free(b_dev, context);
}
```

SYCL™

- Another standard developed by Khronos Group.
- oneAPI Favoured by *Intel* for their future GPUs.

```
#include <algorithm>
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl;

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  sycl::device device{sycl::default_selector()};
  sycl::context context{device};
  sycl::queue queue{context, device};

  float* a_dev{sycl::malloc_device<float>(n, device, context)};
  float* b_dev{sycl::malloc_device<float>(n, device, context)};

  queue.copy(a, a_dev, n);
  queue.copy(b, b_dev, n);
  queue.wait();

  queue.parallel_for(sycl::range<1>{n},
    [=](const auto i) { b_dev[i] = a_dev[i] + b_dev[i]; }
  );
  queue.wait();

  queue.copy(b_dev, b, n);
  queue.wait();

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  sycl::free(a_dev, context);
  sycl::free(b_dev, context);
}
```

SYCL

- Another standard developed by Khronos Group.
- oneAPI Favoured by *Intel* for their future GPUs.
- Based on *standard* C++17.

```
#include <algorithm>
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl;

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  sycl::device device{sycl::default_selector()};
  sycl::context context{device};
  sycl::queue queue{context, device};

  float* a_dev{sycl::malloc_device<float>(n, device, context)};
  float* b_dev{sycl::malloc_device<float>(n, device, context)};

  queue.copy(a, a_dev, n);
  queue.copy(b, b_dev, n);
  queue.wait();

  queue.parallel_for(sycl::range<1>{n},
    [=](const auto i) { b_dev[i] = a_dev[i] + b_dev[i]; }
  );
  queue.wait();

  queue.copy(b_dev, b, n);
  queue.wait();

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  sycl::free(a_dev, context);
  sycl::free(b_dev, context);
}
```

SciNet

SYCL™

- Another standard developed by Khronos Group.
- oneAPI Favoured by *Intel* for their future GPUs.
- Based on *standard* C++17.
- *DPC++* and *OpenSYCL* support multiple backends.

```
#include <algorithm>
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl;

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  sycl::device device{sycl::default_selector()};
  sycl::context context{device};
  sycl::queue queue{context, device};

  float* a_dev{sycl::malloc_device<float>(n, device, context)};
  float* b_dev{sycl::malloc_device<float>(n, device, context)};

  queue.copy(a, a_dev, n);
  queue.copy(b, b_dev, n);
  queue.wait();

  queue.parallel_for(sycl::range<1>{n},
    [=](const auto i) { b_dev[i] = a_dev[i] + b_dev[i]; }
  );
  queue.wait();

  queue.copy(b_dev, b, n);
  queue.wait();

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  sycl::free(a_dev, context);
  sycl::free(b_dev, context);
}
```

SYCL™

- Another standard developed by Khronos Group.
- oneAPI Favoured by *Intel* for their future GPUs.
- Based on *standard* C++17.
- *DPC++* and *OpenSYCL* support multiple backends.
- *Still niche* as of 2023.

```cpp
#include <algorithm>
#include <CL/sycl.hpp>
#include <iostream>
using namespace cl;

int main()
{
  constexpr size_t n{1'000'000};
  float *a{new float[n]}, *b{new float[n]};
  std::fill(a, a+n, 2.);
  std::fill(b, b+n, 3.);

  sycl::device device{sycl::default_selector()};
  sycl::context context{device};
  sycl::queue queue{context, device};

  float* a_dev{sycl::malloc_device<float>(n, device, context)};
  float* b_dev{sycl::malloc_device<float>(n, device, context)};

  queue.copy(a, a_dev, n);
  queue.copy(b, b_dev, n);
  queue.wait();

  queue.parallel_for(sycl::range<1>{n},
    [=](const auto i) { b_dev[i] = a_dev[i] + b_dev[i]; }
  );
  queue.wait();

  queue.copy(b_dev, b, n);
  queue.wait();

  if (std::all_of(b, b+n, [](const auto x){ return x == 5.; }))
    std::cout << "Success!\n";
  else std::cout << "Failure :(\n";

  delete[] a;
  delete[] b;
  sycl::free(a_dev, context);
  sycl::free(b_dev, context);
}
```

```python
import numpy as np
from numba import vectorize

@vectorize('float32(float32, float32)', target='cuda')
def add(a, b):
  return a + b

if __name__ == '__main__':
  n = 1_000_000
  a = np.ones(n, dtype=np.float32)*2
  b = np.ones(n, dtype=np.float32)*3

  b = add(a, b)

  if all([x == 5. for x in b]):
    print('Success!')
  else: print('Failure :(')
```

Numba

- *Just-in-time* (JIT) compiler for Python.

```
import numpy as np
from numba import vectorize

@vectorize('float32(float32, float32)', target='cuda')
def add(a, b):
  return a + b

if __name__ == '__main__':
  n = 1_000_000
  a = np.ones(n, dtype=np.float32)*2
  b = np.ones(n, dtype=np.float32)*3

  b = add(a, b)

  if all([x == 5. for x in b]):
    print('Success!')
  else: print('Failure :(')
```

Numba

- *Just-in-time* (JIT) compiler for Python.
- Supports Nvidia ~~and AMD~~.

```python
import numpy as np
from numba import vectorize

@vectorize('float32(float32, float32)', target='cuda')
def add(a, b):
  return a + b

if __name__ == '__main__':
  n = 1_000_000
  a = np.ones(n, dtype=np.float32)*2
  b = np.ones(n, dtype=np.float32)*3

  b = add(a, b)

  if all([x == 5. for x in b]):
    print('Success!')
  else: print('Failure :(')
```

Numba

- *Just-in-time* (JIT) compiler for Python.
- Supports Nvidia ~~and AMD~~.
- Kernels, ufuncs, and reductions.

```python
import numpy as np
from numba import vectorize

@vectorize('float32(float32, float32)', target='cuda')
def add(a, b):
  return a + b

if __name__ == '__main__':
  n = 1_000_000
  a = np.ones(n, dtype=np.float32)*2
  b = np.ones(n, dtype=np.float32)*3

  b = add(a, b)

  if all([x == 5. for x in b]):
    print('Success!')
  else: print('Failure :(')
```

Numba

- *Just-in-time* (JIT) compiler for Python.
- Supports Nvidia ~~*and AMD*~~.
- Kernels, ufuncs, and reductions.
- Device code is a *restricted subset* of Python.

```python
import numpy as np
from numba import vectorize

@vectorize('float32(float32, float32)', target='cuda')
def add(a, b):
  return a + b

if __name__ == '__main__':
  n = 1_000_000
  a = np.ones(n, dtype=np.float32)*2
  b = np.ones(n, dtype=np.float32)*3

  b = add(a, b)

  if all([x == 5. for x in b]):
    print('Success!')
  else: print('Failure :(')
```

Numba

- *Just-in-time* (JIT) compiler for Python.
- Supports Nvidia ~~and AMD~~.
- Kernels, ufuncs, and reductions.
- Device code is a *restricted subset* of Python.
- Strongly associated with *NumPy*.

```python
import numpy as np
from numba import vectorize

@vectorize('float32(float32, float32)', target='cuda')
def add(a, b):
  return a + b

if __name__ == '__main__':
  n = 1_000_000
  a = np.ones(n, dtype=np.float32)*2
  b = np.ones(n, dtype=np.float32)*3

  b = add(a, b)

  if all([x == 5. for x in b]):
    print('Success!')
  else: print('Failure :(')
```

EXERCISE

HOMEWORK EXERCISE

2D DIFFUSION (HEAT) EQUATION

$$\frac{\partial T}{\partial t} = D\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

Implement an explicit *finite difference*, *time marching* solution using a GPU programming framework.

HOMEWORK EXERCISE

2D DIFFUSION (HEAT) EQUATION

$$\frac{\partial T}{\partial t} = D\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

Implement an explicit *finite difference*, *time marching* solution using a GPU programming framework.

Assumptions:

- Domain is a square.
- Arbitrary initial conditions: $T_0(x, y)$.
- Simple boundary conditions: $T = 0$ on edges.

FINITE DIFFERENCE METHOD

Discretize the domain: $(x_j, y_i) = (j\Delta x, i\Delta y)$ for integer j and i.

FINITE DIFFERENCE METHOD

Discretize the domain: $(x_j, y_i) = (j\Delta x, i\Delta y)$ for integer j and i.

Discrete first and second derivatives of some function $f(x, \ldots)$:

$$\left.\frac{\partial f}{\partial x}\right|_{x_j} \approx \frac{f(x_{j+1}) - f(x_j)}{\Delta x}$$

$$\left.\frac{\partial^2 f}{\partial x^2}\right|_{x_j} \approx \frac{f(x_{j+1}) - 2f(x_j) + f(x_{j-1})}{\Delta x^2}$$

FINITE DIFFERENCE METHOD

Discretize the domain: $(x_j, y_i) = (j\Delta x, i\Delta y)$ for integer j and i.

Discrete first and second derivatives of some function $f(x, \ldots)$:

$$\left.\frac{\partial f}{\partial x}\right|_{x_j} \approx \frac{f(x_{j+1}) - f(x_j)}{\Delta x}$$

$$\left.\frac{\partial^2 f}{\partial x^2}\right|_{x_j} \approx \frac{f(x_{j+1}) - 2f(x_j) + f(x_{j-1})}{\Delta x^2}$$

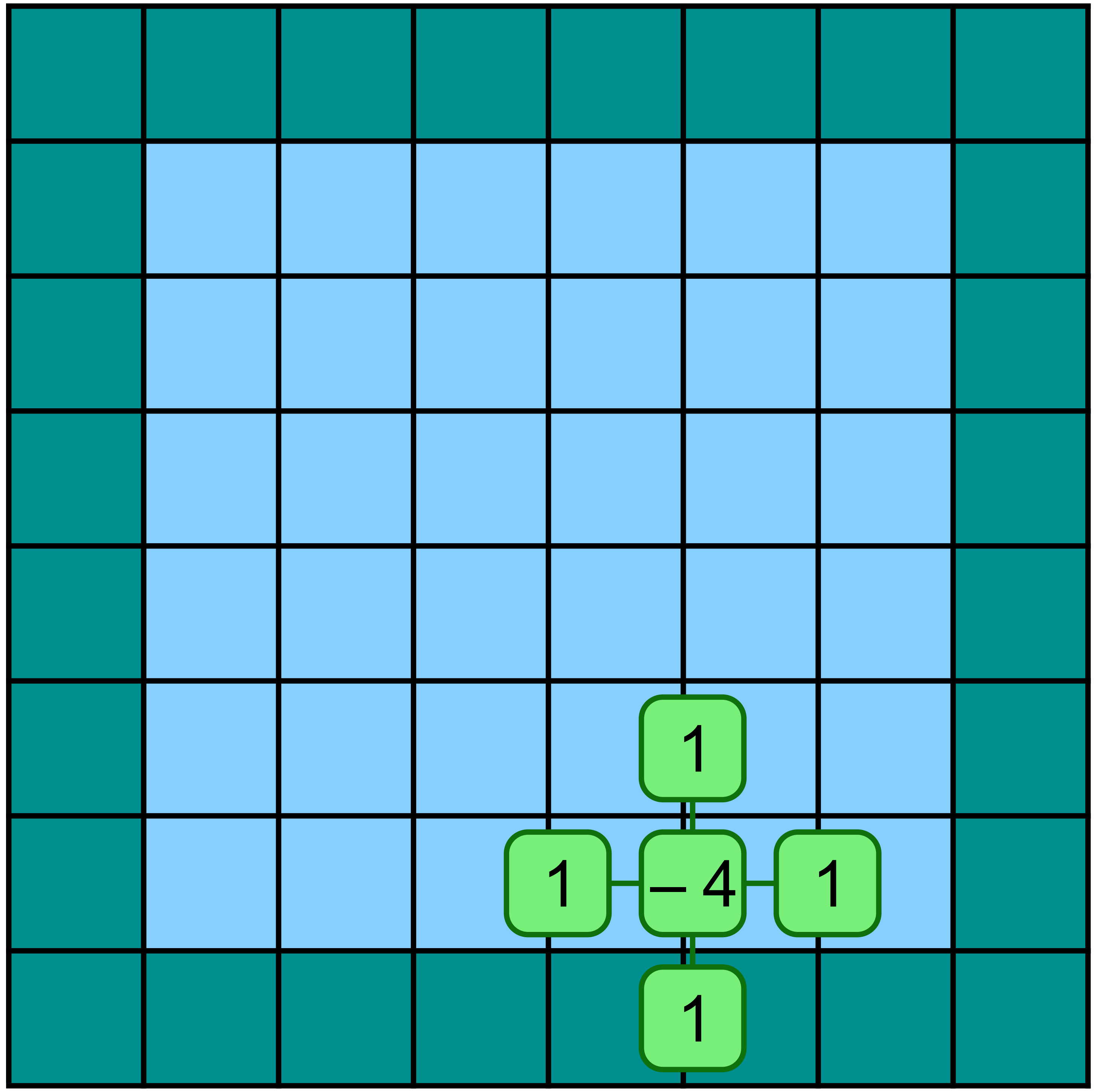
A single *step* of the diffusion equation ($t_k \rightarrow t_{k+1} \equiv t_k + \Delta t$, also assuming $\Delta y = \Delta x$):

$$\begin{aligned} T(x_j, y_i, t_{k+1}) &= T(x_j, y_i, t_k) + \frac{D\Delta t}{\Delta x^2}\Big[\quad T(x_{j+1}, y_i, t_k) + T(x_{j-1}, y_i, t_k) + \\ &\qquad T(x_j, y_{i+1}, t_k) + T(x_j, y_{i-1}, t_k) - \\ &\qquad 4T(x_j, y_i, t_k) \quad \Big] \end{aligned}$$

LAPLACIAN AS A 5-POINT STENCIL

$$\nabla^2 f(x,y) \equiv \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$
$$\approx f(x_{j+1}, y_i) + f(x_{j-1}, y_i)$$
$$+ f(x_j, y_{i+1}) + f(x_j, y_{i-1})$$
$$-4 f(x_j, y_i)$$

```
laplacian[i][j] =    f[ i ][j+1] + f[ i ][j-1]
                 +  f[i+1][ j ] + f[i-1][ j ]
                 -4*f[ i ][ j ]
```



HOME EXERCISE INSTRUCTIONS

Submit your attempt by 2023 March 11 00:00.

- Serial CPU-based solutions are provided in Python and C++.
  You can start by modifying the one in your language, but don't have to.
- You need to identify the *bottleneck* and accelerate it using the GPU.
- There is more than one right answer.
- Bonus (1): the smaller Δx, the more *accurate* and *computationally heavy* the solution. Plot the timing for your solution and of the serial CPU-based solution (and possibly improved CPU-based solutions) as a function of Δx.

Bonus 2 & 3 are beyond the scope of this workshop:

- Bonus (2): decompose the domain and solve the problem with multiple GPUs on the same node.
- Bonus (3): use a distributed memory library to deploy your solution on multiple nodes.

Hint: for a single node you could use `multiprocessing` in Python and `thread` or *OpenMP* in C++.
For multiple nodes you could use `mpi4py` (Python) or *MPI* (C++).

CLASS EXERCISE

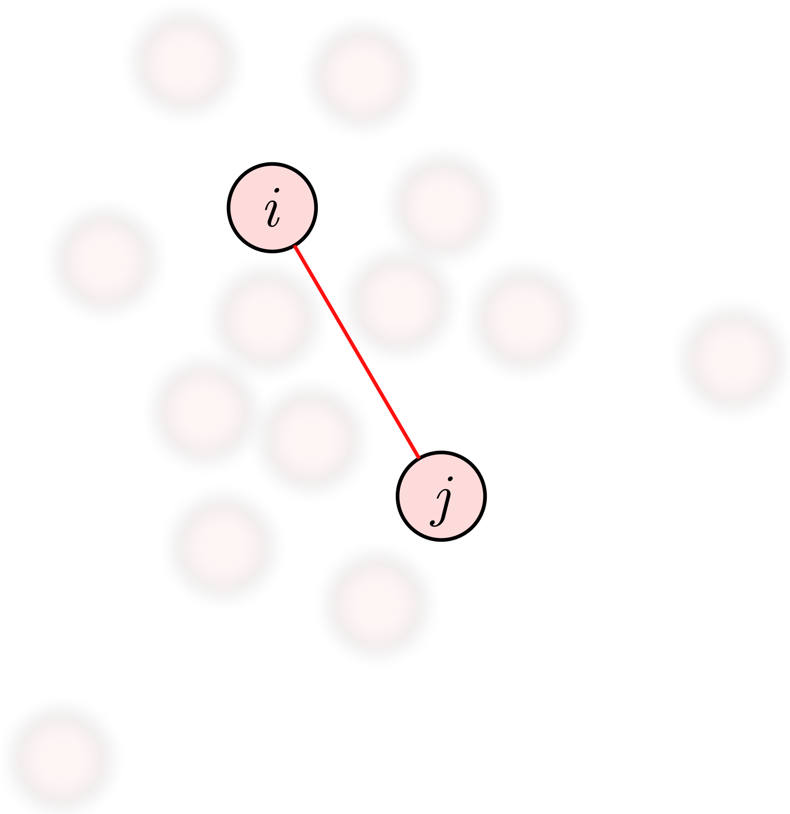
1. Problem overview.
2. A naïve solution in Python.
3. Successively improving the solution.
4. A GPU-accelerated solution with Numba.
5. Comparing with a professional *N*-body library.

PROBLEM OVERVIEW

Given a system of N particles, calculate the *gravitational potential* on each one.

PROBLEM OVERVIEW

Given a system of *N* particles, calculate the *gravitational potential* on each one.

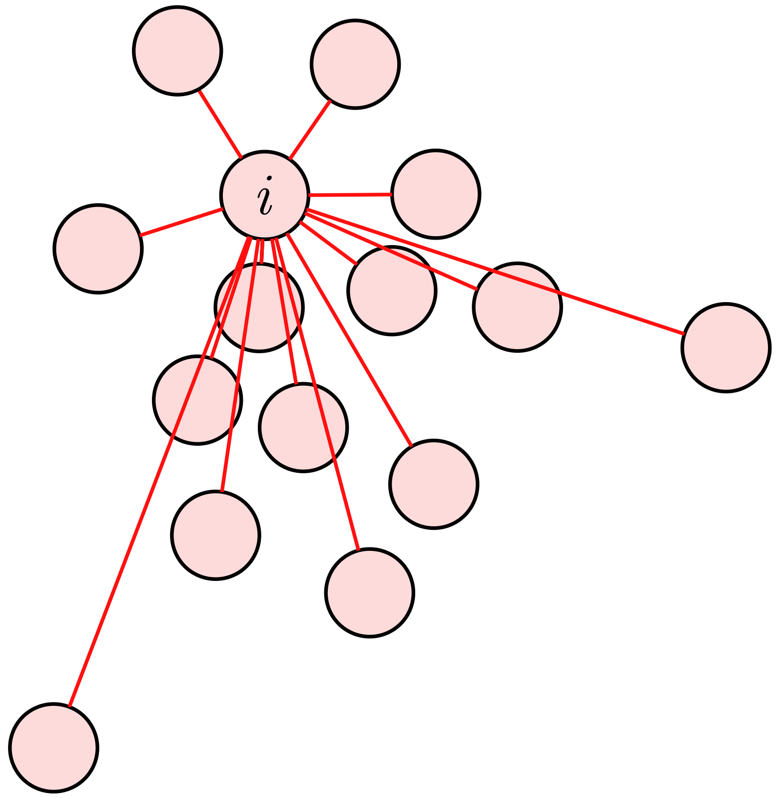


Two-body potential

$$\Phi_i = -\frac{Gm_j}{r_{ij}}$$

PROBLEM OVERVIEW

Given a system of N particles, calculate the *gravitational potential* on each one.

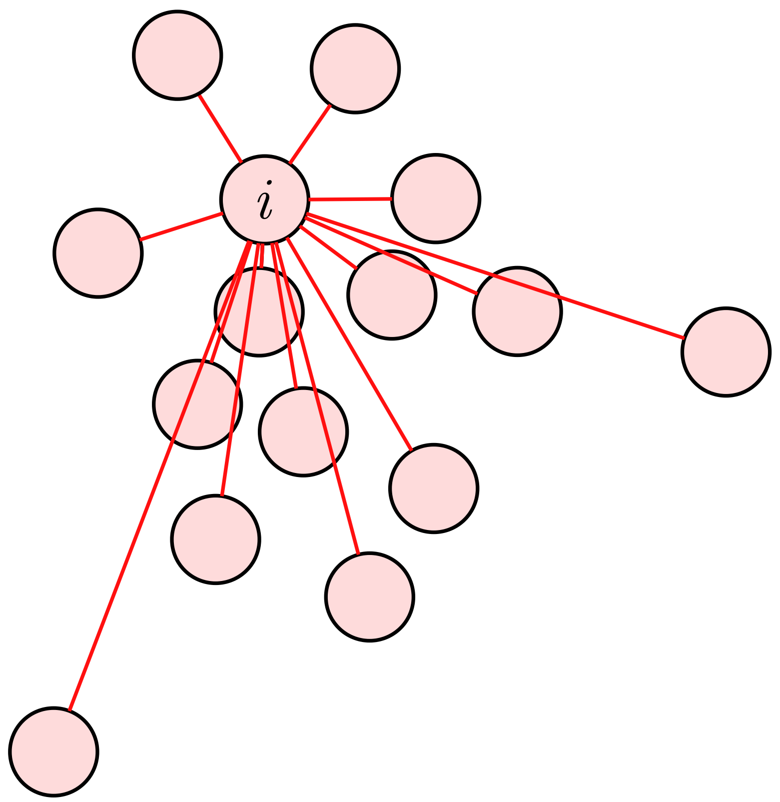


Many-body potential

$$\Phi_i = -\sum_{i \neq j} \frac{Gm_j}{r_{ij}}$$

PROBLEM OVERVIEW

Given a system of N particles, calculate the *gravitational potential* on each one.

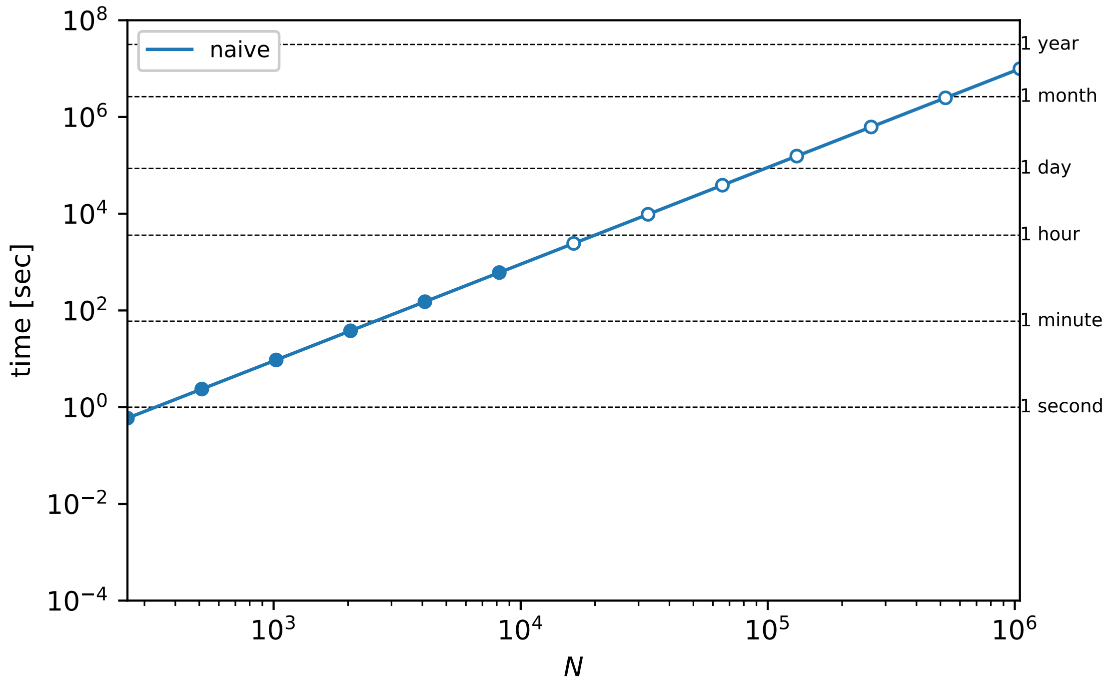


Many-body potential

$$\Phi_i = -\sum_{i \neq j} \frac{Gm_j}{r_{ij}}$$

- Calculate Φ_i *for every* i.
- The number of pairs is $N(N-1)/2$.
- The complexity is $\mathcal{O}(N^2)$.
- Note on alternative algorithms.
- We'll assume $G = 1$ and $m_j = 1/N$.

NAÏVE SOLUTION

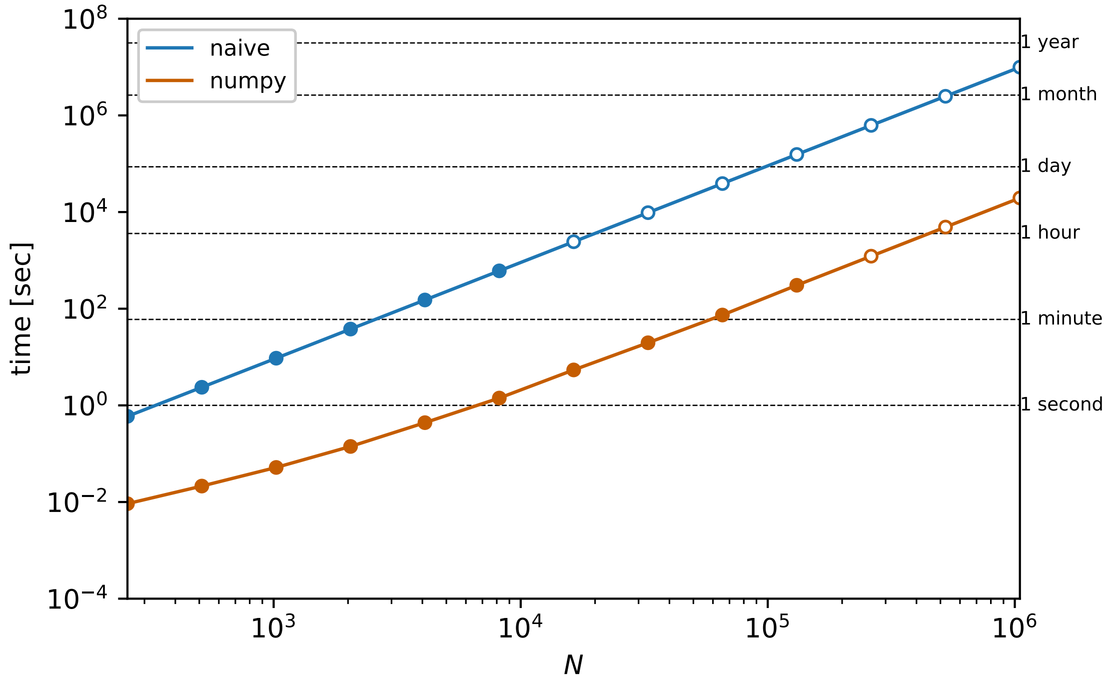


Naïve	10 000 000	sec	(~4 months)

```python
import numpy as np

def calculate_potential(position : np.ndarray) -> np.ndarray:
    N = len(position)
    mass = 1 / N
    potential = np.empty(N)
    for i in range(N):
        potential[i] = 0
        for j in range(N):
            if j == i: continue
            dx, dy, dz = position[i,:] - position[j,:]
            r = np.sqrt(dx**2 + dy**2 + dz**2)
            potential[i] += - mass / r
    return potential
```

USING NUMPY

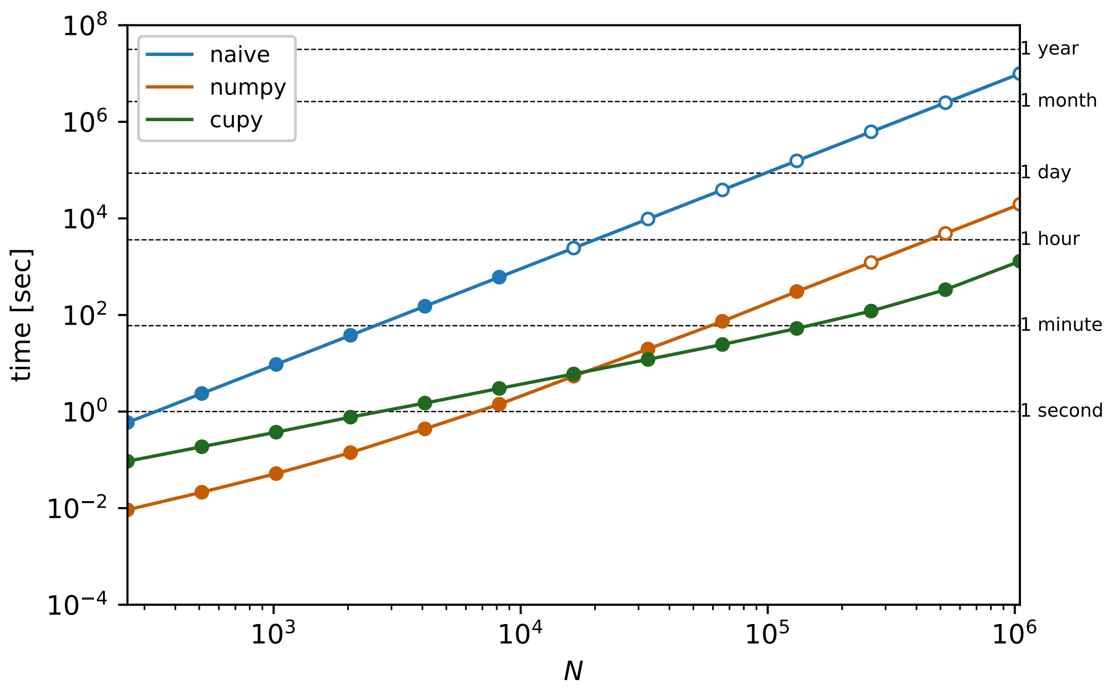



```python
import numpy as np

def calculate_potential(position : np.ndarray) -> np.ndarray:
    N = len(position)
    mass = 1 / N
    potential = np.empty(N)
    for i in range(N):
        dx, dy, dz = (position[i,:] - position).T
        r = np.sqrt(dx**2 + dy**2 + dz**2)
        r[i] = np.inf
        potential[i] = np.sum(- mass / r)
    return potential
```

Naïve	10 000 000	sec	(~4 months)
NumPy	20 000	sec	(~5 hours)

REPLACING NUMPY WITH CUPY

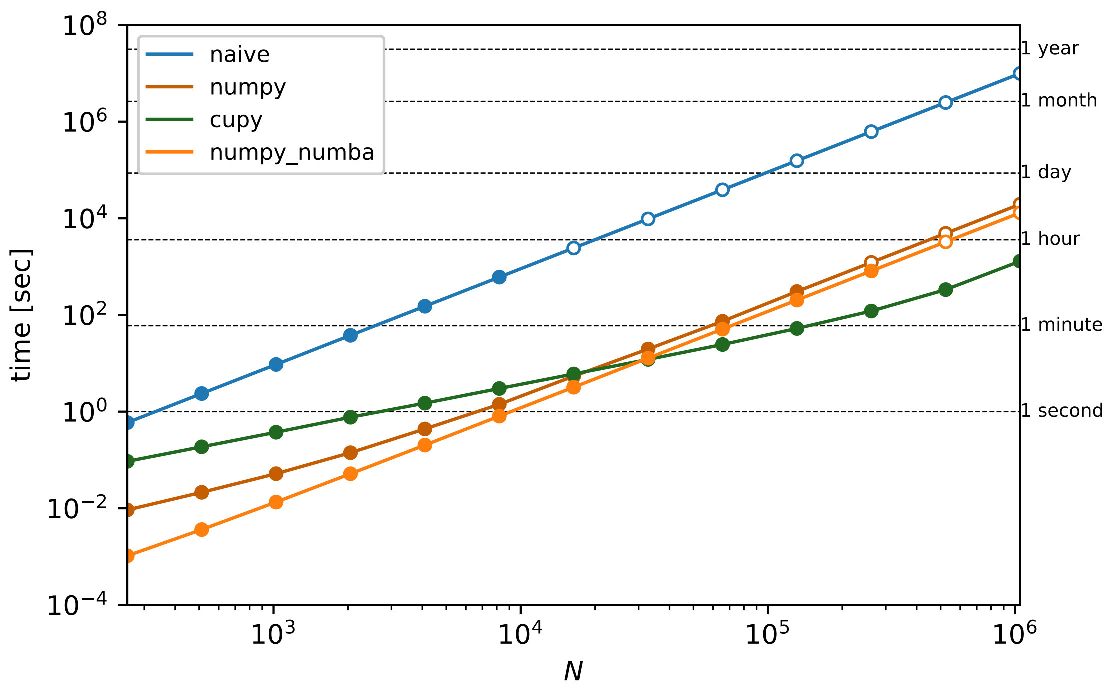


Naïve	10 000 000	sec	(~4 months)
NumPy	20 000	sec	(~5 hours)
CuPy	1 300	sec	(~20 minutes)

```python
import numpy as np, cupy as cp

def calculate_potential(position : np.ndarray) -> np.ndarray:
    position = cp.array(position)
    N = len(position)
    mass = 1 / N
    potential = np.empty(N)
    for i in range(N):
        dx, dy, dz = (position[i,:] - position).T
        r = cp.sqrt(dx**2 + dy**2 + dz**2)
        r[i] = cp.inf
        potential[i]  = cp.sum(- mass / r)
    return potential
```

NUMPY + NUMBA

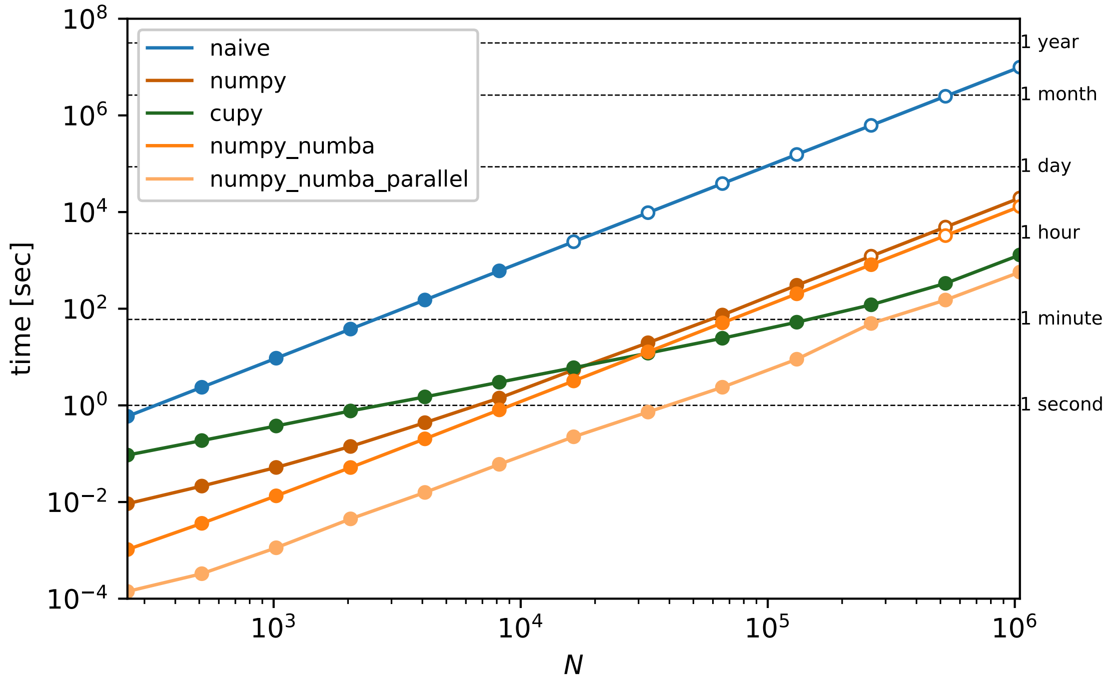


Naïve	10 000 000	sec	(~4 months)
NumPy	20 000	sec	(~5 hours)
CuPy	1 300	sec	(~20 minutes)
NumPy + Numba	13 000	sec	(~4 hours)

```python
import numpy as np
import numba

@numba.njit(numba.float64[:](numba.float64[:,:]))
def calculate_potential(position : np.ndarray) -> np.ndarray:
    N = len(position)
    mass = 1 / N
    potential = np.empty(N)
    for i in range(N):
        dx, dy, dz = (position[i,:] - position).T
        r = np.sqrt(dx**2 + dy**2 + dz**2)
        r[i] = np.inf
        potential[i] = np.sum(- mass / r)
    return potential
```

NUMPY + NUMBA (PARALLEL)

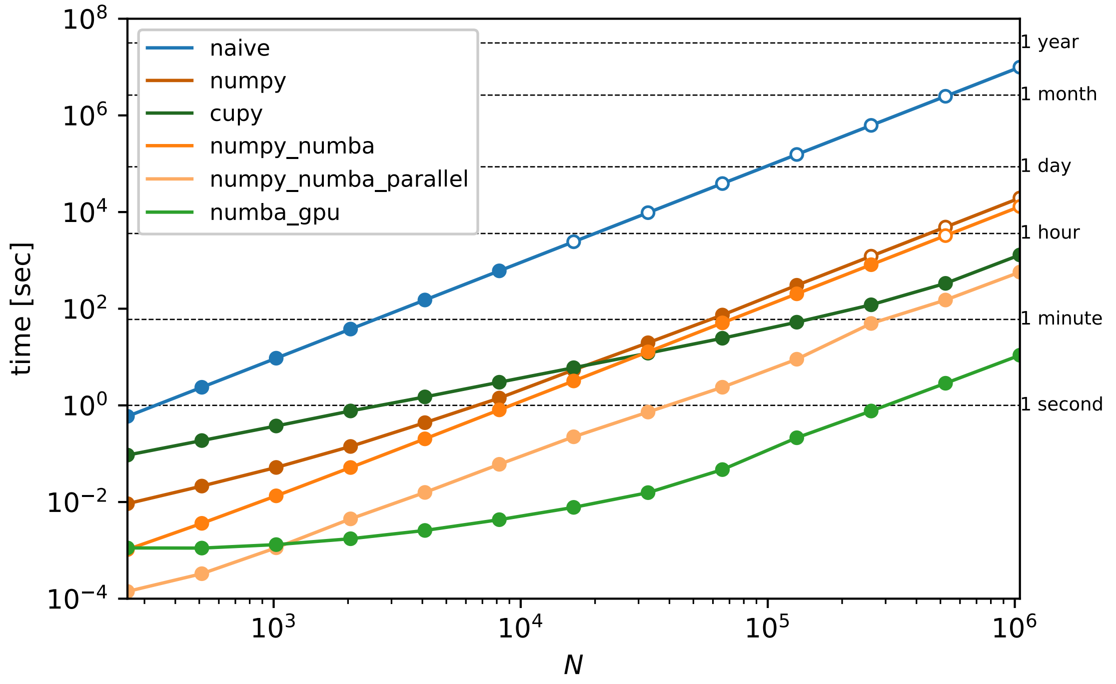


Naïve	10 000 000	sec	(~4 months)
NumPy	20 000	sec	(~5 hours)
CuPy	1 300	sec	(~20 minutes)
NumPy + Numba	13 000	sec	(~4 hours)
NumPy + Numba (parallel)	560	sec	(~9 minutes)

```python
import numpy as np
import numba

@numba.njit(numba.float64[:](numba.float64[:,:]), parallel=True)
def calculate_potential(position : np.ndarray) -> np.ndarray:
    N = len(position)
    mass = 1 / N
    potential = np.empty(N)
    for i in numba.prange(N):
        dx, dy, dz = (position[i,:] - position).T
        r = np.sqrt(dx**2 + dy**2 + dz**2)
        r[i] = np.inf
        potential[i] = np.sum(- mass / r)
    return potential
```

NUMBA ON THE GPU



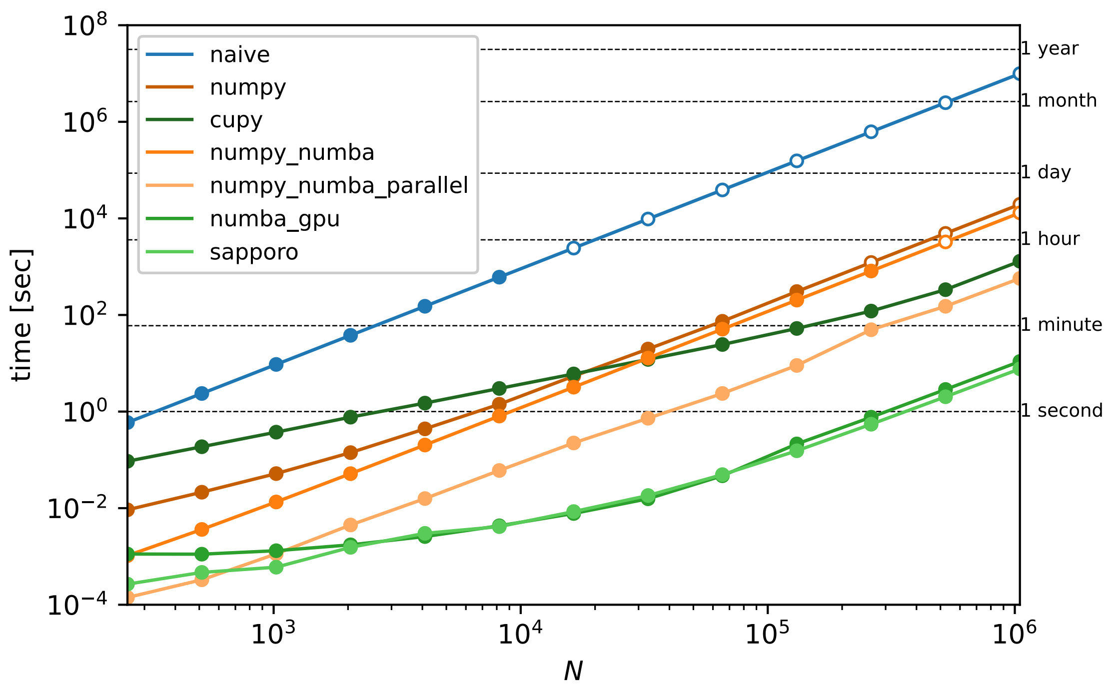
Naïve	10 000 000	sec	(~4 months)
NumPy	20 000	sec	(~5 hours)
CuPy	1 300	sec	(~20 minutes)
NumPy + Numba	13 000	sec	(~4 hours)
NumPy + Numba (parallel)	560	sec	(~9 minutes)
Numba (GPU)	11	sec	

```
import numpy as np, cupy as cp
from numba import cuda
import math

@cuda.jit
def kernel(position, potential):
    i = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    N = len(position)
    if i >= N: return
    mass = 1 / N
    potential_i = 0
    for j in range(N):
        if i != j:
            dx = position[i,0] - position[j,0]
            dy = position[i,1] - position[j,1]
            dz = position[i,2] - position[j,2]
            r = math.sqrt(dx**2 + dy**2 + dz**2)
            potential_i += - mass / r
    potential[i] = potential_i

def calculate_potential(position : np.ndarray) -> np.ndarray:
    threads_per_block = 32
    N = len(position)
    blocks_per_grid = int(np.ceil(N/threads_per_block))
    position  = cp.array(position)
    potential = cp.empty(N)
    kernel[blocks_per_grid, threads_per_block](position, potentia
    return cp.asnumpy(potential)
```

LIBRARY FUNCTION



Naïve	10 000 000	sec	(~4 months)
NumPy	20 000	sec	(~5 hours)
CuPy	1 300	sec	(~20 minutes)
NumPy + Numba	13 000	sec	(~4 hours)
NumPy + Numba (parallel)	560	sec	(~9 minutes)
Numba (GPU)	11	sec	
Sapporo	7.7	sec	

🍎 COMPARISON WITH SAPPORO 🍊

Our solution	**Sapporo**
full double precision	“double-single”
`sqrt` and division	`rsqrt` and multiplication
potential only	potential, acceleration, & jerk
optimized for large n_i	also for small n_i

TIPS FOR HOMEWORK EXERCISE

- Adding types to decorator can help:

```
@cuda.jit('void(float64[:,:], float64[:])')
```

- Instead of a CuPy array, we could do

```
position = cuda.to_device(position)
potential = cuda.device_array(N, dtype=np.float64)
...
return potential.copy_to_host()
```

- Remember that threads and blocks can be indexed in 2D.