# Application of lattice techniques to Dark Matter: Axion simulations in HPC

Alejandro Vaquero

University of Utah

March 9$^{th}$, 2020

# Motivation

- The axion as a dark matter candidate eludes detection
- Need theoretical input to enhance our chances
  - Microlensing (cluster formation)
  - Resonance cavities (mass)
  - ...
- Two possible scenarios

**Pre-inflationary**

- PQ symmetry broken before inflation
- Axion field homogeneous at horizon scales
- Axion production coming mainly from misalignment

**Post-inflationary**

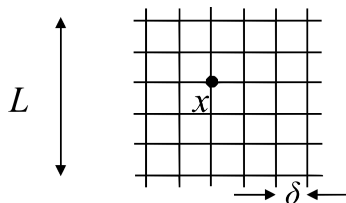
- PQ symmetry **restored** after inflation
- $U(1)$ topological defects appear
- Axion production driven by topological structures

- Assuming the post-inflationary scenario allows to make predictions about the axion properties

# The axion as a DM candidate

- The analytical treatment of topological defects is non-perturbative
  - **Solution:** Put the axion field in a computer
- Axion eom in comoving coordinates

$$\phi_{\tau\tau} - \nabla^2\phi + \frac{\lambda}{2}\phi\left(|\phi|^2 - \tau^2\right) - \tau^{n+3} = 0, \quad \chi(T) = \chi_0 T^{-n}$$




- The axion field lives on the sites
- Discretize space and time in a computer
  - Finite lattice spacing $\delta$ (UV cutoff)
  - Finite spatial volume $L$ (IR cutoff)
  - Finite time step $d\tau$

- We perform simulations in an unphysical setup and approach the physical limit
  - Enlarge the volume and reduce $\delta$ and $d\tau$
  - Take $\lambda$ to its physical value

# Numerical integrators

- The eom are solved using a numerical integrator (propagator)
- Several kind of integrators available
  - Symplectic vs Non-symplectic
    - Symplectic integrators preserve the phase space structure
    - More accurate and stable than non-symplectic in non-dissipative systems
  - Reversible vs Non-reversible
    - Reversible integrators preserve the constants of motion

$$\mathcal{I}_\phi(\varepsilon): \quad \phi_t, \dot{\phi}_t \to \phi_{t+\varepsilon}, \dot{\phi}_t$$

$$\mathcal{I}_v(\varepsilon): \quad \phi_t, \dot{\phi}_t \to \phi_t, \dot{\phi}_{t+\varepsilon}$$

$$\mathcal{J}(\varepsilon, N) = \left[\mathcal{I}_v(\varepsilon/2)\mathcal{I}_\phi(\varepsilon)\mathcal{I}_v(\varepsilon/2)\right]^N$$

    - Any integrator with a symmetric structure is reversible
- Integrator order
  - Reduces time-discretization effects while keeping the time-step constant ($d\tau$)
  - Larger order $\longrightarrow \begin{cases} \text{Larger stepsize} \\ \text{Larger computational cost per stepsize} \end{cases}$

# Discretization errors

- UV cutoff
  - The axion spectrum at large energies is cut off abruptly
  - High spatial/temporal modes (wrt $\delta$, $d\tau$) can not be resolved
  - Derivative in the eom
    - The derivative can be expanded in powers of $\delta$

$$\phi(x+\delta) = \phi(x) + \delta\partial_j\phi(x) + \frac{\delta^2}{2}\partial_{jj}\phi(x) + \dots$$

    - From here (no summation on $j$):

$$\partial_j\partial_j\phi(x) = \begin{cases} \frac{\phi_{x+\delta}+\phi_{x-\delta}-2\phi_x}{\delta^2} + O(\delta^2) \\ \frac{16}{12}\frac{\phi_{x+\delta}+\phi_{x-\delta}-2\phi_x}{\delta^2} - \frac{\phi_{x+2\delta}+\phi_{x-2\delta}-2\phi_x}{\delta^2} + O(\delta^4) \\ \dots \end{cases}$$

- $\lambda$, string core size $r_s$ and saxion mass $m_s$
  - String core size $r_s \simeq (\sqrt{\lambda}\tau)^{-1}$
  - When $m_s \approx 1/\delta$ and $r_s \lesssim \delta$ discretization effects become **large**

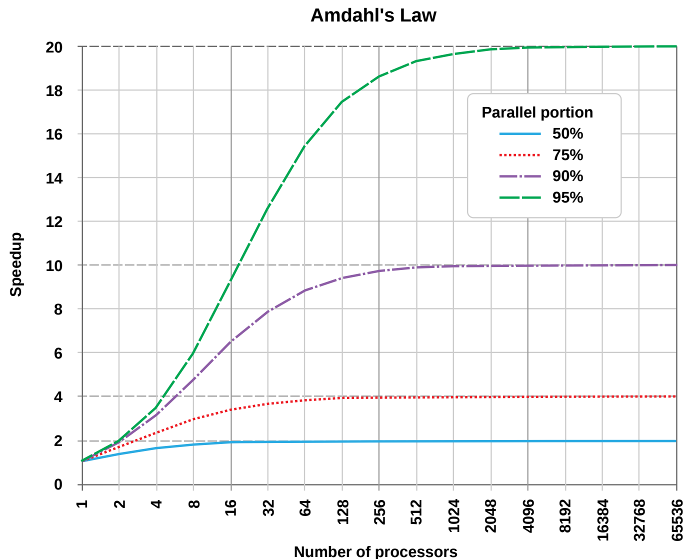
# Dynamical range

- Ideal value for $\lambda \approx 10^{51} - 10^{60} \implies r_s \approx 10^{-25} - 10^{-30}$
- Also we want $m_s^2 \gg m_a^2 \quad \lambda \gg 1$
- Assuming a reasonably large volume $V \sim O(1)$, we need $N \sim O(10^{70} - 10^{80} \sim (10^{25})^3)$ to resolve the strings
- Largest volume ever simulated $N \sim O(10^{12} \sim (10^4)^3)$ — Vaquero, Redondo, Stadler, 2018
- Proposals:
  - Fat (PRS) strings — Press, Ryden, Spergel, 1989
  - Effective theory with local strings — Dabholkar, Quashnock, 1990; Moore, Klaer, 2017
    - Attach a local string (gauge) to every global string
    - Local strings increase the tension
  - Adaptative Mesh Refinement (AMR) — Drew, Shellard, 2019
    - Only strings need ultrafine lattice spacing

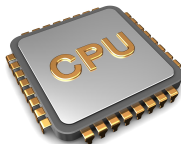
# Parallelization strategies

- Lattice simulations are embarrasingly parallelizable
  - Each independent core treats a different point of the lattice
  - All cores do the same operations
  - Easy to port to GPUs / vector processors
- Larger lattices available with MPI and a halo mechanism exchange
- Parallelization introduced to
  - Simulate a **larger** volume
  - Increase performance, i.e. **saturate** flops / memory bandwidth
- Hierarchy in the parallelization
  - Nodes $\rightarrow$ Blocks/Threads $\rightarrow$ Threads/SIMD lanes
- Hierarchy in memory bandwidth
  - IB $\sim 10$ GB/s $\rightarrow$ DDR4 $\sim O(100)$ GB/s $\rightarrow$ Cache $O(10)$ TB/s

# Parallelization limits: Amdahl's law



## CPU or GPU?

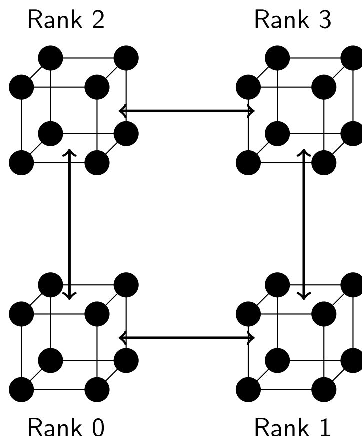


| | CPU | GPU |
|---|---|---|
| Typical memory size | $\sim 128$ GB | $\sim 16$ GB |
| Typical memory bandwidth | $\sim 100$ GB/s | $\sim 1$ TB/s |

- Simulations faster in GPUs due to larger memory bandwidth
- To run a $N = 8192^3$ simulation we roughly need $13$ TB of memory (plus ghosts)
  - $\sim$ 120 typical HPC CPU nodes
  - $\sim$ 480 typical HPC GPU nodes (assuming 2 GPUs / node)
- MPI overhead can kill performance advantages
  - Overlap computations and communications

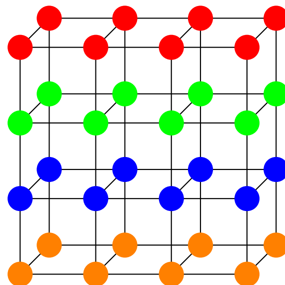
# Parallelization over MPI

- **M**essage **P**assing **I**nterface, inter-process communication API
- Efficient for **weakly** coupled systems in distributed memory
- Launch $N$ copies of a process, assign a **rank**
- **Explicit** exchange of data
- Connection **topology** might be non-trivial



# Parallelization over threads in CPUs

- **Threads** are used in **shared** memory configuration
- Efficient for **strongly** coupled systems in shared memory
- Threads are created **dynamically** on demand
- No need to exchange data, **shared**
- Several implementations:
  - Intel TBB
  - OpenMP
  - OpenACC
  - pthreads
  - C++ threads...



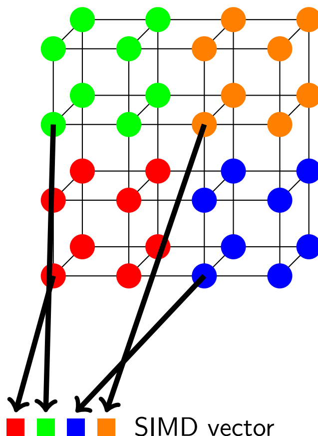
Thread 1
Thread 2
Thread 3
Thread 4

# Vectorization in CPUs

**S**ingle **I**nstruction **M**ultiple **D**ata

- Split the lattice in **virtual** nodes
- Each SIMD lane deals with a **different** virtual node
- Optimal data layout: **Structure of Arrays (of Vectors)**
- **Alignment** important!

$$\left(\phi_x^r \phi_x^g \phi_x^b \phi_x^o \quad \phi_{x+1}^r \phi_{x+1}^g \phi_{x+1}^b \phi_{x+1}^o \quad \ldots\right)$$

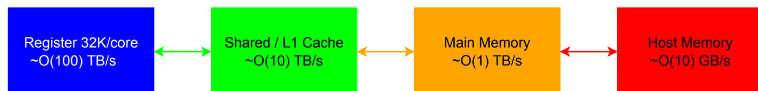



**Reference:** Intel intrinsics guide

software.intel.com/sites/landingpage/IntrinsicsGuide/

# Parallelization on GPUs

- Uses **S**ingle **I**nstruction **M**ultiple **T**hread paradigm
- Each thread executes the **same instruction** in a synchronized fashion
- Each thread reads a **different** memory location
- **Divergences** in the execution flow incurr in a **penalty**
- **Random memory access** incurrs in a **penalty**
  - Use SoA (again!)
- **Hierarchy** of memories



# Coding for GPUs

**Directive based (OpenACC/OpenMP)**

- No major rewrites of the code needed (if you have the right layout)
- Portability

- Difficult to write performant code
- Explicit memory management
- No deep copy!!!

**Dedicated CUDA/HIP code**

- Easier to achieve good performance
- Can accomodate any data structures

- Requires major rewiting of the code
- Explicit memory management
- Lack of portability

# Coding for GPUs

- GPU execution model
  - Explicitly divide execution in a **grid** of **blocks** of **threads**
- Example kernel launch

```
dim3 gridSize((Sf+xBlock-1)/xBlock, (Lz+yBlock-1)/yBlock, 1);
dim3 blockSize(xBlock, yBlock, 1);

propagateKernel<float, VQCD_1><<<gridSize,blockSize>>> (m, v, m2, dz);
```

- Example kernel

```
template<typename Float, const VqcdType VQcd>
__global__ void propagateKernel(const complex<Float> * __restrict__ m, complex<Float> * __restrict__ v, complex<Float> *
        __restrict__ m2, Float dz)
{
  uint idx = threadIdx.x + blockDim.x*blockIdx.x + sliceSize*(threadIdx.y + blockDim.y*blockIdx.y);

  if(idx >= numPoints)
    return;

  propagateCore<Float, VQcd>(idx, m, v, m2, dz);
}
```

# Coding for GPUs

```
template<typename Float, const VqcdType VQcd>
static __device__ void propagateCore(uint idx, const complex<Float> * __restrict__ m, complex<Float> * __restrict__ v, complex<
        Float> * __restrict__ m2, Float dz)
{
  uint X[3], idxPx, idxPy, idxMx, idxMy;

  complex<Float> mel, a, tmp;

  idx2Vec(idx, X, Lx);

  if(X[0] == Lx-1)
    idxPx = idx - Lx+1;
  else
    idxPx = idx+1;
  ...
  tmp = m[idx];
  mel = m[idxMx] + m[idxPx] + m[idxPy] + m[idxMy] + m[idxPz] + m[idxMz];

  Float pot = tmp.real()*tmp.real() + tmp.imag()*tmp.imag();

  a      = (mel-((Float) 6.)*tmp)*ood2 + zQ - tmp*(((Float) lambda)*(pot - z2));
  mel     = v[idx];
  mel    += a*dz;
  v[idx] = mel;
  mel    *= dz;
  tmp    += mel;
  m2[idx] = tmp;
}
```

# Coding for GPUs

- Kernel launches are asynchronous

```
dim3 gridSize((Sf+xBlock-1)/xBlock, (Lz+yBlock-1)/yBlock, 1);
dim3 blockSize(xBlock, yBlock, 1);

propagateKernel<float, VQCD_1><<<gridSize,blockSize>>> (m, v, m2, dz);
doStuffOnCpu();
```

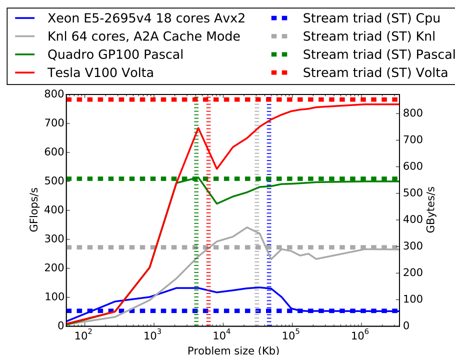
- If `doStuffOnCpu()` depends on GPU data, there will be a **race condition**
  - Force synchronization (**large penalty**)
- Asynchronous launches allow us to overlap data transfer and computations
  - Use streams

```
sendStuffToGpu(STREAM0);
launchKernel<<<Blocks,Threads,0,STREAM0>>>
getStuffFromGpu(STREAM0);
sendMoreStuffToGpu(STREAM1);
launchKernel<<<Blocks,Threads,0,STREAM1>>>
getStuffFromGpu(STREAM1);

synchronize(STREAM0):
doStuffOnCpuWithStream0():

synchronize(STREAM1):
doStuffOnCpuWithStream1():
```

GPU execution time

# Optimization and profiling

- When optimizing we need to check the **arithmetic intensity** of our code
- IC codes have **low** arithmetic intensity. Example: our IC code $\approx 0.5$
- Current processors/accelerators feature a **large** flops/bandwidth ratio
  - NVidia V100 $\approx 15$, Intel Xeon $\approx 30$, KNL $\approx 40$
- Performance bounded by **memory bandwidth**, not by computational power
  - How to tell if my code is doing well? Measure GB/s, compare to roofline



# Optimization and profiling

- Profiling is extremely important to optimize the code

```
totalGB = 0; totalGF = 0;

oTime = std::chrono::high_resolution_clock::now(); // Timer --> Start

doComputationallyExpensiveStuff();

totalGB += computationallyExpensiveStuffGBytes;
totalGF += computationallyExpensiveStuffGFlops;

eTime = std::chrono::high_resolution_clock::now(); // Timer --> Stop

dTime = std::chrono::duration_cast<std::chrono::microseconds>(eTime - oTime).
     count()*1e-6;

Log("ComputationallyExpensiveStuff␣took␣\%.2f␣ms", dTime*1e3);
Log("ComputationallyExpensiveStuff␣GFlops/s␣\%.2f", totalGF/dTime);
Log("ComputationallyExpensiveStuff␣GBytes/s␣\%.2f", totalGB/dTime);
```

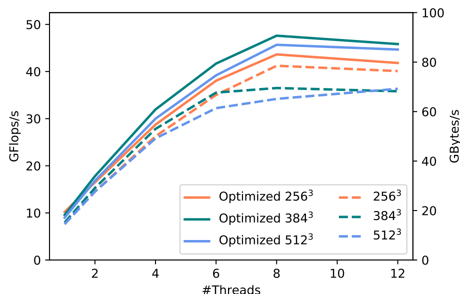
# Optimization and profiling

**Example** running on Intel 5300U ($\sim 14$ GB/s stream, $25.6$ GB/s peak)

```
162606ms: Logger ==> Profiler Scalar:
162606ms: Logger ==>    Function Init Allocation   GFlops/s 0.0000   GBytes/s 0.0000 Total time 0.06s (0.03%)
162606ms: Logger ==>    Function Init FFT          GFlops/s 0.0000   GBytes/s 0.0000 Total time 0.02s (0.01%)
162606ms: Logger ==>    Function Normalise         GFlops/s 1.3131   GBytes/s 2.1009 Total time 0.06s (0.04%)
162606ms: Logger ==>    Function Normalise Core    GFlops/s 2.3268   GBytes/s 1.4049 Total time 0.76s (0.47%)
162606ms: Logger ==>    Function Scale             GFlops/s 2.6462   GBytes/s 10.5850 Total time 0.01s (0.01%)
162606ms: Logger ==> Total Scalar: 0.91
162606ms: Logger ==>
162606ms: Logger ==> Profiler Genconf:
162606ms: Logger ==>    Function Random            GFlops/s 0.0000   GBytes/s 0.8234 Total time 0.16s (0.10%)
162606ms: Logger ==>    Function Smoother          GFlops/s 1.8975   GBytes/s 6.7467 Total time 2.55s (1.57%)
162606ms: Logger ==> Total Genconf: 2.71
162606ms: Logger ==>
162606ms: Logger ==> Profiler Propagator:
162606ms: Logger ==>    Function RKN4 Saxion       GFlops/s 22.2868  GBytes/s 42.4510 Total time 126.47s (77.78%)
162606ms: Logger ==> Total Propagator: 126.47
162606ms: Logger ==>

...

162606ms: Logger ==> Profiler Hdf5 I/O:
162606ms: Logger ==>    Function Write Map         GFlops/s 0.0000   GBytes/s 0.1553 Total time 0.07s (0.04%)
162606ms: Logger ==>    Function Write configuration GFlops/s 0.0000 GBytes/s 2.2699 Total time 0.24s (0.15%)
162606ms: Logger ==>    Function Write density     GFlops/s 0.0000   GBytes/s 0.6829 Total time 0.12s (0.08%)
162606ms: Logger ==>    Function Write energy      GFlops/s 0.0000   GBytes/s 0.0002 Total time 0.00s (0.00%)
162606ms: Logger ==>    Function Write energy map  GFlops/s 0.0000   GBytes/s 1.8055 Total time 0.37s (0.23%)
162606ms: Logger ==>    Function Write strings     GFlops/s 0.0000   GBytes/s 0.0052 Total time 20.92s (12.87%)
162606ms: Logger ==> Total Hdf5 I/O: 21.72
162606ms: Logger ==>
162606ms: Logger ==> Unaccounted time 1.22s of 162.61s (0.75%)
```

# Optimization and profiling

- Cache BW is much **higher** than main memory BW
- **Locality** of data greatly improves performance
  - Data divided in **small** bunches
  - Preload data **fits** cache
- More threads is not always better $\rightarrow$ avoid **cache thrashing**



# Performance portability

- Targeting different architectures can be complicated
  - Specific code per architecture
  - Every feature needs to be added multiple times
  - Difficult to maintain
- Performance portability
  - Same code, good performance everywhere
- How can I achieve performance portability?
  - Using backends
  - Using directives
    - OpenMP (C/C++/Fortran)
    - OpenACC (C/C++/Fortran)
  - Using performace portability libraries
    - Kokkos (C++)
    - Raja (C++)
    - SyCL (...)

# Programming languages

**High level**

- Fast and easy to learn and use
- Low performance
- Good for analysis / visualization

- Julia
- Python
- Matlab
- R / Octave

**Not so high level**

- High performance if properly written
- Steeper learning curve
- Good to develop simulation code

- C / C++
- Fortran

- Scripting languages (bash, zsh, sh...) are normally used to rename/concatenate files, etc

# Where can I run?

| | Size | Volume IC | Total Memory (CPU/GPU) | Total core-h/gpu-h |
|---|---|---|---|---|
| Your desktop | One node | $\sim 1024^3$ (CPU) $\sim 768^3$ (GPU) | 64/16 GB | 50/3 |
| Local facility | $O(10)$ nodes | $\sim 3072^3$ (CPU) $\sim 2048^3$ (GPU) | 1024/256 GB | 1300/100 |
| Medium HPC cluster | $O(100)$ nodes | $\sim 6144^3$ (CPU) $\sim 4096^3$ (GPU) | 8192/2048 GB | 10500/ - |
| Large HPC cluster | $O(1000)$ nodes | $\sim 12288^3$ (CPU) $\sim 8192^3$ (GPU) | 65536/16384 GB | 85000/ - |

- Medium-large HPC clusters usually require a detailed proposal to award computer time

**PRACE** (Europe)

- PizDaint (NVidia GPUs + Intel Xeon)
- Joliot-Curie (KNL + Intel Xeon)
- Marconi (KNL + Intel Xeon)
- SuperMUC (Intel Xeon)
- Marenostrum 4 (Intel Xeon)
- Juwels (Intel Xeon)
- Hawk (AMD EPYC)

**DOE and national labs** (USA)

- PerlMutter (Forthcoming, NVidia GPUs + AMD CPUs)
- Summit (NVidia GPUs + Power9)
- Sierra (NVidia GPUs + Power9)
- Stampede 2 (KNL + Intel Xeon)
- Frontera (Intel Xeon)
- Trinity (KNL)
- Theta (KNL)

# Future systems

Future exascale systems are all **GPU based**

- Aurora
  - Intel XE GPUs, coding in OpenCL and SyCL
  - Expected $\gtrsim 10$ PB and $\gtrsim 1.0$ EFLOP
  - Using $10-20\%$ of the machine $\rightarrow 32768^3$ GPU
- Frontier
  - AMD CPUs and GPUs, coding with HIP and ROCm
  - Expected $O(10)$ TB and $\gtrsim 1.5$ EFLOPS
  - Using $10-20\%$ of the machine $\rightarrow 16384^3$ GPU
- El Capitan
  - AMD CPUs and GPUs, coding with HIP and ROCm
  - Expected $O(10)$ TB and $\sim 2.0$ EFLOPS
  - Using $10-20\%$ of the machine $\rightarrow 16384^3$ GPU
- EuroHPC in Europe

# Our current code

- Runs IC simulations with a variety of initial conditions
- Performs consistently **above** the memory BW in Intel systems
  - Even when we use $O(10000)$ cores
- Runs efficiently in GPUs as well
- A lot of interesting features
  - Online analysis
  - Lattice reduction for output
  - Parallel I/O with HDF5
  - Includes scripts to make cool movies out of the data

`github.com/veintemillas/jaxions`

**Everybody** is invited to contribute

# Open discussion

Thanks for your attention