The Exodus to GPU. The case of scientific analysis for the Euclid mission.

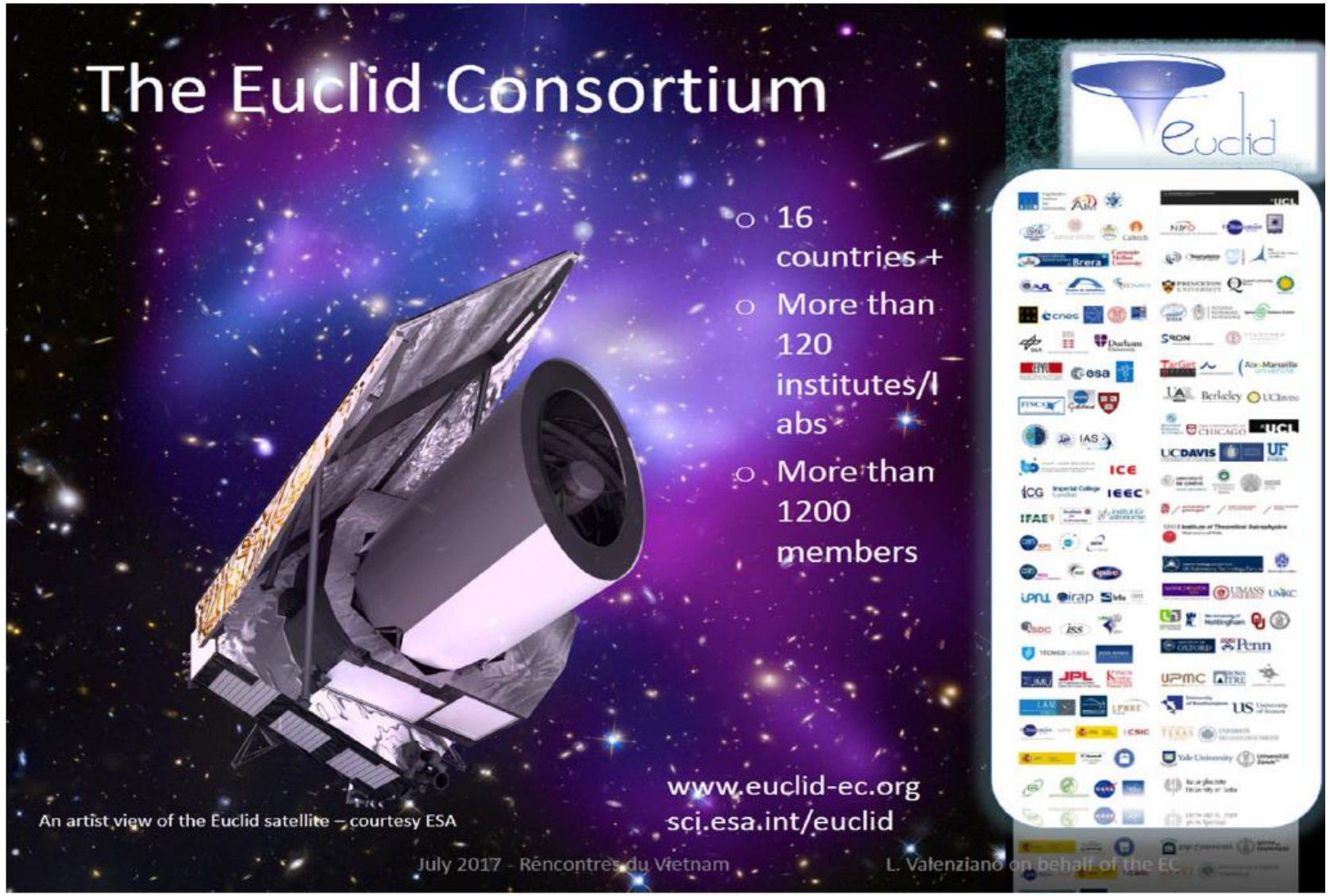
Alessandro Renzi (UniPD and INFN)

A discourse about the challenge for cosmology data analysis using HPC

Outline

- The Euclid Mission
- Status of Software in the scientific analysis of Euclid mission data
- Why GPU matters? – The problem of modern HPC
- My solution: the Julia Programming Language
- Advantages/Disadvantages of Julia
- Julia vs commonly used languages for numerical computing
- A couple of examples of Julia usage and philosophy
- Training to GPU usage with Julia

The Euclid Mission



The mission would map the Large-Scale Structures (LSS) of the Universe and can be used for study fundamental physics from cosmology, especially the Dark Energy (which is related to expansion acceleration of the Universe) and some properties of neutrinos (sum of masses, number of species, hierarchy).

The Status of Computing in Euclid

- The Science Ground Segment (SGS) is responsible for:
  - archiving and public dissemination of data collected by Euclid
  - development and run of the pipeline for the extraction of products for science
- The scientific exploitation of Euclid products is entrusted to the individual Science Working Groups (SWG) who, at the moment, must use (own) computing resources external to SGS for the scientific analysis
- I am part of the INFN group in Euclid, and we have organized ourselves for the scientific analysis of the Euclid data
- Euclid's data are images, and their study, from the point of view of cosmology, is basically an analysis of the correlations in these images compared to the model's simulated cosmology. These considerations strongly constrain the hardware needed for the analysis of these data
  - We need HPC for Euclid data analysis!

Status of Software for the scientific analysis of Euclid mission data

- Simulations [rarely used/modified by “standard user”] -> N-body codes (e.g. Gadget, Ramses, etc,)
  - Very stable
  - Vanilla versions in general publicly available with MPI+threads
  - Heavily optimized -> hard to modify
  - Extremely high computational demands
  - Most used software have no public GPU code
- Software to analyze maps and catalogs [heavily used rarely modified] (e.g. nbodykit, healpix, etc.)
  - Medium stable (reduced stability due to algorithmic or theoretical advancement)
  - Many threads-only parallelism, sometimes MPI-only or MPI+threads
  - Medium optimization (mostly homebrewed), small needs of modification
  - From high to low computational demands (depends on the analysis)
  - Most used software have no public GPU code
- “Theory” software [heavily used/modified] (CAMB and CLASS)
  - Vanilla version very stable
  - Threads-only
  - Highly optimized for use with MCMC -> hard to modify
  - High computational demands in use with MCMC
  - No public GPU code
- Right now, new codes are produced to analyze data, and we (my group) are producing new codes too!

Why GPU matters? – The problem of modern HPC → The CINECA's Leonardo use case

Technical Information

Leonardo will be built from **Atos' BullSequana XH2000 supercomputer nodes, each with four NVIDIA Tensor Core GPUs and a single Intel CPU**. It will also use NVIDIA Mellanox HDR 200Gb/s InfiniBand connectivity, with smart in-network computing acceleration engines that enable extremely low latency and high data throughput to provide the highest AI and HPC application performance and scalability.

Leonardo will feature nearly 14 000 NVIDIA Ampere architecture-based GPUs. It will deliver 10 exaflops of FP16 AI performance. NVIDIA Ampere architecture GPUs can accelerate over 1,800 applications such as Quantum Espresso for material science, SPECFEM3D for geoscience and MILC for quantum physics by up to 70x, making previous big challenge simulations almost real-time tasks.

3rd Gen Intel Xeon Scalable processors (Ice Lake) are optimized to perform computationally intensive workloads in high-performance computing systems like Leonardo. The follow-on processor to Intel's Ice Lake server processors is Sapphire Rapids, which will enable exascale computing with advanced built-in AI acceleration capabilities.

- **More than 136 BullSequana XH2000 Direct Liquid cooling racks**
- **250 PFLOPs HPL Linpack Performance (Rmax)**
- **10 ExaFLOPS of FP16 AI performance**
- **3456 servers equipped with Intel Xeon Ice Lake and NVIDIA Ampere architecture GPUs**
- **1536 servers with Intel Xeon Sapphire processors**
- **5PB of High Performance storage**
- **100PB of Large Capacity Storage**

Why GPU matters? – The problem of modern HPC → Machine Learning optimized hardware

- Even the CPU partition in Leonardo is optimized for ML!
- The transition to GPU and ML optimized hardware is happening for all the big computational centers in the world (see the Top500 list for an overview of the new computational systems)
- Energy power issues are real: the switch to energy efficient hardware will last for many year to come
  - Software right now must be modified and optimized to gain the most from those new computing hardware
  - GAIN from using GPU-optimized code
  - GAIN from using ML-oriented algorithms

My solution: the Julia Programming Language

- This is the time where codes must be heavily modified to follow the GPU transition
  - We can catch the ball and choose the best tools «today» to deal with the difficulty of optimizing new hardware
  - Julia is the right tool the right moment
  - With Julia we don’t have to choose any more between minimizing human vs CPU time, we can have both!
- Nowadays PhD students and postdoc learn Python and use external tools to increase the computational efficiency of it
  - Most of the time Python is used as glue, so a developer of a particular code should learn multiple languages or rely on code made by other more expert groups
  - Even considering only compiled languages, every field has its own set of libraries and solutions (reinventing the wheel) that maybe are not even “the best”
- Think about algorithms and not about hardware or optimization!
  - With Julia we could concentrate on producing good code, and later think on how optimize
  - An unoptimized code in Julia is still as fast as a standard compiled language
- Julia has built-in “environments” capabilities (use of containers only for very specialized operations)

Julia - A due warning

- I am not a Julia evangelist
  - I decided to use Julia after a couple of years of thoughts and trials
  - About 2 years ago (version 1.5-1.6, and later 1.7) for me Julia was promising but not ready for real world, stable, HPC use cases!
  - I have still some concerns
- My goal for this talk is to show you some interesting features of Julia and suggest that it is ready for HPC power use
- I'll try to be as objective as possible
- In my opinion the advantages of adopting Julia for numerical and data computing outweighs the disadvantages
- All my considerations came from the use case of cosmological data analysis (for Euclid), different fields could reach different conclusions based on their specific case

Julia - Advantages

- High-performance GPU programming in a high-level language (https://juliagpu.org/) -> Multiple GPU support already present!
- Fast
  - Julia was designed from the beginning for high performance. Julia programs compile to efficient native code for multiple platforms via LLVM
- Reproducible
  - Reproducible environments make it possible to recreate the same Julia environment every time, across platforms, with pre-built binaries
- Composable
  - Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns
- General
  - Julia provides asynchronous I/O, metaprogramming, debugging, logging, profiling, a package manager, and more.
- Built-in linear algebra:
  - Let Uppercase arrays and lowercase scalars
    - define a simple function f(x,y)= x + y
    - c = a * b [scalar]
    - C = A .* b [array]
    - c = f(a,b) [scalar]
    - C = f(A,b) [error]
    - C = f(A,B) [array]
    - C = f.(A,b) [array]
    - C = f.(a,b) [array]
- State-of-the-art capable ML algorithms!
- As easy to learn and use as python!
- The interface standardization of Julia is stable with respect to change of hardware (-> What happens if in 10 years from now we will need to change again hardware?)

Julia - Disadvantages

- Language features stability (from the download page):
  - Almost everyone should be downloading and using the latest stable release of Julia. Great care is taken not to break compatibility with older Julia versions, so older code should continue to work with the latest stable Julia release. You should only be using the long-term support (LTS) version of Julia if you work at an organization which implementing or certifying upgrades is prohibitively expensive and there is no need for new language features
- Julia is continuously evolving (see previous point)
- Backward-compatibility is strongly encouraged but not forced on packages of language (keep in mind however that the python 2 -> 3 switch was a lesson for everyone!)
- Julia support on big computational centers could be low or absent right now
  - Particular care is needed with MPI or GPU drivers
- Julia is very mature on CUDA, but less so on Intel, AMD and Apple hardware
- People in general (students in particular) don’t like to learn new/multiple languages…

Julia vs Modern Fortran

- Julia is Modern Fortran, but better (or at least, for many different reasons, do the same things as Fortran, but has a larger and growing community)
- Even if Fortran is rapidly catching-up (in my opinion) it is late on the new paradigm shift (LFortran is very promising but it is still in heavy development)

Julia vs C++

- There are no classes in Julia. Instead, there are structures (mutable or immutable), containing data but no methods.
- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based (in general), they can even be customized!
- Julia arrays are column major (Fortran ordered)
- Julia values are not copied when assigned or passed to a function
  - However, by convention, functions that modify their arguments have a ! at the end of the name
- Arrays “framework” is a feature of the language -> no reasons to work with flattened array
- Effort to standardize packages through common interfaces (e.g. see later AbstractFFTs)
- A note about C++: so far so “good”, but new competitors are coming, i.e. Carbon and Rust

Julia vs Python

- Julia has performances comparable to those of C/C++/Fortran
- Don’t need to rely on external packages to be speed-effective (see python numpy, numba, etc.) – Standard language and decorators are usable out of the box
- Array “framework” is Julia standard core features (including parallel operations)
- In Julia you don’t need to write vectorized code for performance reasons!
- Julia can be used as an interpreted language exactly as python but without losing performances (uses LLVM to compile)
- Julia could be a “glue language“ but it is not necessary
- Most of Julia (standard) packages are written in pure Julia (debugging and profiling is easier!)
- Plots have a common interface with different backends (including matplotlib)
- In general, there is a strong effort in Julia to have common interface for different packages doing seme thing (in contrast, the Python math module has asin, acos, and atan methods. NumPy has arcsin, arccos, and arctan.)
- Machine learning and general modern data analysis tools as rich as python

Julia example 1 – Custom array indeces

A final application: Fourier transforms

There are many more things you can do with custom indices. As one last illustration, consider the Discrete Fourier Transform, which is defined on a periodic domain. Typically, it's rather difficult to emulate a periodic domain with arrays, because arrays have finite size. However, it's possible to define indexing objects which possess periodic behavior. Here we use the FFTViews package, demonstrating the technique on a simple sinusoid:

```
julia> using FFTViews

julia> a = [sin(2π*x)+0.1 for x in linspace(0,1,16)];

julia> afft = FFTView(fft(a))
FFTViews.FFTView{Complex{Float64},1,Array{Complex{Float64},1}} with indices FFTViews.URange(0,15):
       1.6+0.0im
     1.498-7.53098im
 -0.288537+0.69659im
 -0.236488+0.35393im
 -0.222614+0.222614im
 -0.216932+0.14495im
 -0.214217+0.0887316im
 -0.212937+0.0423558im
 -0.212557+0.0im
 -0.212937-0.0423558im
 -0.214217-0.0887316im
 -0.216932-0.14495im
 -0.222614-0.222614im
 -0.236488-0.35393im
 -0.288537-0.69659im
     1.498+7.53098im
```

Now, as every student of Fourier transforms learns, the 0-frequency bin holds the sum of the values in `a`:

```
julia> afft[0]
1.6000000000000003 + 0.0im
```

Taken from https://julialang.org/blog/2017/04/offset-arrays/

We can also check the amplitude at the Fourier-peak, and explore the periodicity of the result:

```
julia> afft[1]
1.4980046017247872 - 7.53097769363728im

julia> afft[-1]      # negative frequencies are OK
1.4980046017247872 + 7.53097769363728im

julia> afft[64+1]    # look Ma, it's periodic!
1.4980046017247872 - 7.53097769363728im

julia> length(indices(afft,1))   # but we still know how big it is
16
```

Given the periodicity of `afft`, the commonly-used `fftshift` function (e.g., `fftshift(fft(a))`) can be replaced by `afft[-8:7]`. While very simple, these techniques make it surprisingly more pleasant to deal with what can sometimes become complex and error-prone index gymnastics.

Julia example 2 – Interface standardization of common numerical computations

- **AbstractFFTs.jl**
  - A general framework for fast Fourier transforms (FFTs) in Julia.
  - From the package description: This package is mainly not intended to be used directly. Instead, developers of packages that implement FFTs (such as FFTW.jl or FastTransforms.jl) extend the types/functions defined in AbstractFFTs. This allows multiple FFT packages to co-exist with the same underlying fft(x) and plan_fft(x) interface.

> The normalization convention for your FFT should be that it computes $y_k = \sum_j \exp(-2\pi i \cdot \frac{jk}{n})$ for a transform of length n, and the "backwards" (unnormalized inverse) transform computes the same thing but with $\exp(+2\pi i \cdot \frac{jk}{n})$.

Julia example 3 - Using CUDA

```
using CUDA
```

You could install CUDA with Julia artifacts!!!

```
julia> a = CuArray([1,2])
2-element CuArray{Int64, 1, CUDA.Mem.DeviceBuffer}:
 1
 2

julia> b = Array(a)
2-element Vector{Int64}:
 1
 2

julia> copyto!(b, a)
2-element Vector{Int64}:
 1
 2
```

```
julia> a = CuArray{Float32}(undef, (1,2));

julia> a .= 5
1×2 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 5.0  5.0

julia> map(sin, a)
1×2 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 -0.958924  -0.958924
```

```
julia> a = CUDA.ones(2,3)
2×3 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 1.0  1.0  1.0
 1.0  1.0  1.0

julia> reduce(+, a)
6.0f0

julia> mapreduce(sin, *, a; dims=2)
2×1 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 0.59582335
 0.59582335
```

Training to GPU usage with Julia

- Nice thing: you don’t need it! (knowing that device memory and node memory are different is enough!)

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>

#define N 512

__global__ void add(int *a, int *b, int *c){
   int tid = blockIdx.x;  // handle the data at this index

   if(tid < N)
      c[tid] = a[tid] + b[tid];
}

int main()
{
   int a[N], b[N], c[N], i;
   int *dev_a, *dev_b, *dev_c;

   cudaMalloc((void**)&dev_c, N*sizeof(int));
   cudaMalloc((void**)&dev_b, N*sizeof(int));
   cudaMalloc((void**)&dev_a, N*sizeof(int));
   for(i=0; i < N; i++)
   {
      a[i] = -i;
      b[i] = i*i*i;
   }
   cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);

   add <<<N, 1>>>(dev_a, dev_b, dev_c);
   cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
   for(i=0; i < N; i++)
      printf("%d + %d = %d\n", a[i], b[i], c[i]);

   cudaFree(dev_c);
   cudaFree(dev_b);
   cudaFree(dev_a);
   return 0;
}
```

```
using CUDA

x_d = CUDA.fill(1.0f0, N)  # a vector stored on the GPU filled with 1.0 (Float32)
y_d = CUDA.fill(2.0f0, N)  # a vector stored on the GPU filled with 2.0
```

```
y_d .+= x_d
```

```
CUDA.copyto!(y_d, y)
```

If you are not convinced, try to sum some arrays using multiple GPU with any language of your choice vs Julia!

Training to GPU usage with Julia

Sum using CUDA kernel:

```
function gpu_add1!(y, x)
    for i = 1:length(y)
        @inbounds y[i] += x[i]
    end
    return nothing
end

fill!(y_d, 2)
@cuda gpu_add1!(y_d, x_d)
@test all(Array(y_d) .== 3.0f0)
```

Thanks for the attention!