Physics, Algorithms and Computers:
A short tour of a few things I’ve done

Fernando Pérez

Department of Applied Mathematics
University of Colorado, Boulder.

Tradelink, Chicago
April 16, 2007
Outline

1. Physics and Mathematics
Outline

1. Physics and Mathematics

2. Numerical Multiresolution Algorithms
Outline

1. Physics and Mathematics
2. Numerical Multiresolution Algorithms
3. Computing with Python and IPython
Outline

1. Physics and Mathematics
2. Numerical Multiresolution Algorithms
3. Computing with Python and IPython
4. The future: IPython as a network-aware Python VM
Outline

1. Physics and Mathematics
2. Numerical Multiresolution Algorithms
3. Computing with Python and IPython
4. The future: IPython as a network-aware Python VM
5. Odds and Ends, Wrapup
Outline

1. Physics and Mathematics
2. Numerical Multiresolution Algorithms
3. Computing with Python and IPython
4. The future: IPython as a network-aware Python VM
5. Odds and Ends, Wrapup
Theoretical physics

- PhD in Lattice QCD (Quantum Chromo Dynamics): numerical simulations of the behavior of quarks and gluons.
- Previous work: Classical and Quantum Chaos.
- Classical chaos: numerical integration of the classical 3-body Coulomb system.
- Quantum chaos: defining a quantum version of Lyapunov exponents.
- Picked up computing along the way.
Physical problems formulated as PDEs

- The Laplace/Poisson equations
  \[-\Delta u = f\]

- The Schrödinger equation (for stationary states)
  \[\left(-\frac{1}{2}\Delta + V\right)\psi = E\psi\]

- The modified Stokes equation (time-stepping schemes for Navier-Stokes):
  \[\alpha v - \mu \Delta v + \nabla p = f\]
  \[\nabla \cdot v = 0\]

A good fraction of the world’s (scientific) computing time is devoted to the solution of this type of problem.
Broadly and loosely speaking, there are two main options; each has its own set of difficulties.

**Discretize differential operator and invert**

- Sparse matrices (fast solvers)
- That represent unbounded operators...
Numerical approaches

Broadly and loosely speaking, there are two main options; each has its own set of difficulties.

**Discretize differential operator and invert**

- Sparse matrices (fast solvers)
- That represent unbounded operators...
- And hence are **ill-conditioned** and require pre-conditioning.
Numerical approaches

Broadly and loosely speaking, there are two main options; each has its own set of difficulties.

Discretize differential operator and invert

- Sparse matrices (fast solvers)
- That represent unbounded operators...
- And hence are **ill-conditioned** and require pre-conditioning.
Numerical approaches

Broadly and loosely speaking, there are two main options; each has its own set of difficulties.

Discretize differential operator and invert

- Sparse matrices (fast solvers)
- That represent unbounded operators...
- And hence are **ill-conditioned** and require pre-conditioning.

Write integral formulation and apply integral operator (Green’s functions)

- Well-conditioned objects...
- That lead to **dense matrices**
Numerical approaches

Broadly and loosely speaking, there are two main options; each has its own set of difficulties.

Discretize differential operator and invert
- Sparse matrices (fast solvers)
- That represent unbounded operators...
- And hence are **ill-conditioned** and require pre-conditioning.

Write integral formulation and apply integral operator (Green’s functions)
- Well-conditioned objects...
- That lead to **dense matrices**
- And don’t easily generalize to multiple dimensions.
What are we after?

**Immediate Goals**

- Numerical algorithms with **finite but controlled precision**.
- **Multiscale**, fully **adaptive** algorithms.
- Approximations are a cousin of the Fast Multipole Method (**FMM**), but easier to generalize in dimension and kernel.
- Green’s functions: $G(r - r')$. Many fundamental physical processes are **2-body interactions** (Nature cooperates).
What are we after?

Immediate Goals

- Numerical algorithms with **finite but controlled precision**.
- **Multiscale**, fully **adaptive** algorithms.
- Approximations are a cousin of the Fast Multipole Method (**FMM**), but easier to generalize in dimension and kernel.
- Green’s functions: $G(r - r')$. Many fundamental physical processes are **2-body interactions** (Nature cooperates).

Overall program

- The ‘**curse of dimensionality**’: the exponential rise in complexity of many algorithms with the underlying physical dimension.
- Multiparticle Schrödinger equation, Navier-Stokes.
- A toolbox of reliable algorithms for the efficient application of integral transforms in multiple dimensions.
Key mathematical ideas:

1. **Multiresolution analysis (wavelets):** sparse matrix representations for a large class of kernels.
2. **Separated representations:** reduction of dimensionality cost.
Multiresolution algorithms in multiple dimensions

Key mathematical ideas:

1. Multiresolution analysis (wavelets): sparse matrix representations for a large class of kernels.
2. Separated representations: reduction of dimensionality cost.

Group effort over many years (1988-today):

1. Gregory Beylkin, Lucas Monzón, Christopher Kurcz - CU Boulder
2. Martin Mohlenkamp - Ohio University
3. Robert Harrison, George Fann, Takeshi Yanai, Zhengting Gan - ORNL
4. Vani Cheruvu - (now at NCAR)
5. Robert Cramer - (now at Raytheon)
Outline

1. Physics and Mathematics
2. Numerical Multiresolution Algorithms
3. Computing with Python and IPython
4. The future: IPython as a network-aware Python VM
5. Odds and Ends, Wrapup
Multiresolution analysis, intuitively

Imagine a simple signal $f(t)$ you want to study:
Multiresolution analysis, intuitively

Imagine a simple signal $f(t)$ you want to study:

At each scale $n$, divide the unit interval $[0, 1]$ into $2^n$ binary subintervals:

- $n=0$: $l=0$
- $n=1$: $l=0$, $l=1$
- $n=2$: $l=0$, $l=1$, $l=2$, $l=3$
And compute:

- **Average** \((s)\) values of function at level \(n\): space \(V_n\).
- **Differences** \((d)\) between successive levels: space \(W_n = V_{n+1} - V_n\).
And compute:

- **Average** \( (s) \) values of function at level \( n \): space \( V_n \).
- **Differences** \( (d) \) between successive levels: space \( W_n = V_{n+1} - V_n \).

\( f(t) \) can be studied (compressed, denoised, ...) from \( \{V_0, W_0, W_1, \ldots\} \).

The \( d \) coefficients are **small** and **localized** around changes.

We’ll use **multiwavelets**: \( p \) coefficients per subinterval (‘high order Haar’).
Adaptive subdivision of the unit interval in $\mathbb{R}^d$

Simple recursive subdivision produces a $d$-binary tree on the unit interval, with $p^d$ coefficient blocks on the leaves:
Functions: adaptive, controlled accuracy decompositions

$N_{\text{nod}} = 12, \epsilon = 1.0 \times 10^{-10}, N_{\text{blocks}} = 21$

$N_{\text{nod}} = 10, \epsilon = 5.0 \times 10^{-11}, N_{\text{blocks}} = 634$
Functions: adaptive, controlled accuracy decompositions

- $N_{nod} = 12$, $\epsilon = 1.0 \times 10^{-10}$, $N_{blocks} = 21$
- $N_{nod} = 10$, $\epsilon = 5.0 \times 10^{-11}$, $N_{blocks} = 634$

![Graph and diagrams showing adaptive Gaussian grid and 3D plots with different parameters for nodal points and block counts.]
Operators ($d = 1$): sparse representations

Again, project the operator on each scale and use differences:

$$T_n = T_0 + (T_1 - T_0) + (T_2 - T_1) + \ldots = T_0 + \sum_{j=1}^{n} D^j.$$
Operators \((d = 1)\): sparse representations

Again, project the operator on each scale and use differences:

\[
T_n = T_0 + (T_1 - T_0) + (T_2 - T_1) + \ldots = T_0 + \sum_{j=1}^{n} D^j.
\]

\(T^j\)

\(D^j\)

\(j = 3\)

\(j = 7\)
The Modified NSF

A graphical illustration of what we gain for $T_j$ ($j = 5$ shown):
The Modified NSF

A graphical illustration of what we gain for $T_j$ ($j = 5$ shown):
Adaptive natural-scale application
A graphical representation

Redundant tree of input (output skeleton)

Terminal
Non-terminal
Numerical example in 1D

Consider the periodic analogue of the Hilbert transform.

\[(Cf)(y) = \text{p.v.} \int_0^1 \cot(\pi(y - x)) f(x) \, dx,\]

applied to the periodic function

\[f(x) = \sum_{k \in \mathbb{Z}} e^{-a(x+k-1/2)^2} \rightarrow (Cf)(y) = i \sqrt{\frac{\pi}{a}} \sum_{n \in \mathbb{Z}} \text{sign}(n) e^{-n^2\pi^2/a} e^{2\pi iny} \]
Numerical example in 1D

Consider the periodic analogue of the Hilbert transform.

\[(Cf)(y) = \text{p.v.} \int_0^1 \cot(\pi(y - x)) f(x) \, dx,\]

applied to the periodic function

\[f(x) = \sum_{k \in \mathbb{Z}} e^{-a(x+k-1/2)^2} \rightarrow (Cf)(y) = i \sqrt{\frac{\pi}{a}} \sum_{n \in \mathbb{Z}} \text{sign}(n) e^{-n^2\pi^2/4a} e^{2\pi iny}.\]
**d = 1, lessons learned**

### The Good

1. Sparse representations of operators via multiwavelets lead to fast algorithms.
2. Accuracy is guaranteed by construction.
3. We can efficiently handle multi-scale interactions.
4. While the example was a convolution (fast via FFT?), we have an automatically adaptive algorithm.
$d = 1$, lessons learned

The Good

1. Sparse representations of operators via multiwavelets lead to fast algorithms.
2. Accuracy is guaranteed by construction.
3. We can efficiently handle multi-scale interactions.
4. While the example was a convolution (fast via FFT?), we have an automatically adaptive algorithm.

The Bad

This approach does not directly extend to $d > 1$. 
The ‘curse of dimensionality’

A simple observation: numerical algorithms \((C = AB, y = Ax)\) in \(d\) physical dimensions scale exponentially with \(d\) in complexity. Not good.

A typical example: Poisson’s equation (electromagnetics, gravity, . . .):

\[
\nabla^2 \phi(\mathbf{r}) = \rho(\mathbf{r})
\]
A simple observation: numerical algorithms \((C = AB, y = Ax)\) in \(d\) physical dimensions scale exponentially with \(d\) in complexity. Not good.

A typical example: Poisson’s equation (electromagnetics, gravity, . . . ):

\[
\nabla^2 \phi(\mathbf{r}) = \rho(\mathbf{r})
\]

A Green’s function solution (free space, \(d = 3\), ignore constants):

\[
\phi(\mathbf{r}) = \int G(\mathbf{r} - \mathbf{r}')\rho(\mathbf{r}')d^3\mathbf{r}' = \int \frac{1}{|\mathbf{r} - \mathbf{r}'|}\rho(\mathbf{r}')d^3\mathbf{r}'.
\]
The ‘curse of dimensionality’

**A simple observation**: numerical algorithms \((C = AB, y = Ax)\) in \(d\) physical dimensions scale exponentially with \(d\) in complexity. Not good.

**A typical example**: Poisson’s equation (electromagnetics, gravity,\ldots):

\[
\nabla^2 \phi(r) = \rho(r)
\]

A Green’s function solution (free space, \(d = 3\), ignore constants):

\[
\phi(r) = \int G(r - r')\rho(r')d^3r' = \int \frac{1}{|r - r'|}\rho(r')d^3r'.
\]

If we discretize using a global basis, this becomes:

\[
\phi_{ijk} = \sum_{i'j'k'=1}^N G_{ii',jj',kk'}\rho_{i'j'k'}
\]

Applying an integral kernel is a matrix-vector multiplication.
Can we do this efficiently for $d > 1$?

- What if we could write:

$$G_{i'i'j'j'k'k'} = \sum_{m=1}^{M} w_{m} F_{ii'}^{m} F_{jj'}^{m} F_{kk'}^{m}.$$
Can we do this efficiently for $d > 1$?

- What if we could write:

\[ G_{ii', jj', kk'} = \sum_{m=1}^{M} w_m F_{ii'}^m F_{jj'}^m F_{kk'}^m. \]

- We could separate the different dimensions:

\[ \phi_{ijk} = \sum_{m=1}^{M} w_m \sum_{i'} F_{ii'}^m \sum_{j'} F_{jj'}^m \sum_{k'} F_{kk'}^m p_{i' j' k'}. \]

The problem partially factorizes.
Can we do this efficiently for $d > 1$?

- What if we could write:

$$G_{i'i'j'j',kk'} = \sum_{m=1}^{M} w_m F_{ii'}^m F_{jj'}^m F_{kk'}^m.$$ 

- We could separate the different dimensions:

$$\phi_{ijk} = \sum_{m=1}^{M} w_m \sum_{i'} F_{ii'}^m \sum_{j'} F_{jj'}^m \sum_{k'} F_{kk'}^m \rho_{i'j'k'}.$$ 

The problem partially factorizes.

- And if we can construct sparse $F_{ii'}^m$ representations, we may have a fast, multidimensional algorithm.
Operators ($d > 1$): Gaussians to the rescue

We can approximate a wide class of kernels as sums of Gaussians:

$$
\frac{1}{\| r - r' \|} \approx \sum_{m=1}^{M} w_m e^{-\tau_m \| r - r' \|^2},
$$

with controlled accuracy $\varepsilon$ over a large dynamic range [$M \approx O(-\log \varepsilon)$]:

This gives us the factorization we wanted for our kernel:

$$
G_{ii',jj',kk'} = \sum_{m=1}^{M} w_m F_{ii'}^m F_{jj'}^m F_{kk'}^m.
$$
Sparsity also in the $m$ ‘direction’?

- The Gaussian expansion gave us separation of directions...
Sparsity also in the $m$ ‘direction’?

- The Gaussian expansion gave us separation of directions...
- at the cost of a new ‘internal’ degree of freedom, the separation index $m$. 
Sparsity also in the $m$ ‘direction’?

- The Gaussian expansion gave us separation of directions...
- at the cost of a new ‘internal’ degree of freedom, the separation index $m$. 
Sparsity also in the $m$ ‘direction’?

- The Gaussian expansion gave us separation of directions...
- at the cost of a new ‘internal’ degree of freedom, the separation index $m$.

*Do we really need all these terms?*
The 2-scale differences cancel most terms

Full (inc. weights) norms (all shifts)

Norm vs. Separation index
Poisson’s equation: an example

Let’s solve Poisson’s equation for a simple case with a known solution, a sum of Gaussians:

\[ \rho(\mathbf{r}; \alpha) = \sum_{i=1}^{3} (6\alpha - 4\alpha^2 r_i^2) e^{-\alpha r_i^2} \implies \phi(\mathbf{r}; \alpha) = \sum_{i=1}^{3} e^{-\alpha r_i^2}, \quad \alpha = 300. \]

We compute

\[ \phi(\mathbf{r}) = \frac{1}{4\pi} \int_{\mathbb{R}^3} \frac{\rho(\mathbf{r}')}{||\mathbf{r} - \mathbf{r}'||} d\mathbf{r}'. \]

Timings: done on a Pentium 4, 2.8 GHz machine

<table>
<thead>
<tr>
<th>Tolerance</th>
<th>Resulting (\varepsilon_{L^2})</th>
<th>Basis order</th>
<th>App. time (s)</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10^{-4})</td>
<td>(2.3 \times 10^{-5})</td>
<td>12</td>
<td>2.1</td>
<td>1770</td>
</tr>
<tr>
<td>(10^{-6})</td>
<td>(8.4 \times 10^{-7})</td>
<td>12</td>
<td>12</td>
<td>1670</td>
</tr>
<tr>
<td>(10^{-8})</td>
<td>(2.0 \times 10^{-9})</td>
<td>14</td>
<td>33</td>
<td>1880</td>
</tr>
</tbody>
</table>
Poisson’s equation: an example

Let's solve Poisson's equation for a simple case with a known solution, a sum of Gaussians:

\[
\rho(r; \alpha) = \sum_{i=1}^{3} (6\alpha - 4\alpha^2 r_i^2) e^{-\alpha r_i^2} \implies \phi(r; \alpha) = \sum_{i=1}^{3} e^{-\alpha r_i^2}, \quad \alpha = 300.
\]

We compute

\[
\phi(r) = \frac{1}{4\pi} \int_{\mathbb{R}^3} \frac{\rho(r')}{||r - r'||} dr'.
\]

**Timings:** done on a Pentium 4, 2.8 GHz machine

<table>
<thead>
<tr>
<th>Tolerance</th>
<th>Resulting $\varepsilon_{L^2}$</th>
<th>Basis order</th>
<th>App. time (s)</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-4}$</td>
<td>$2.3 \times 10^{-5}$</td>
<td>12</td>
<td>2.1</td>
<td>1770</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>$8.4 \times 10^{-7}$</td>
<td>12</td>
<td>12</td>
<td>1670</td>
</tr>
<tr>
<td>$10^{-8}$</td>
<td>$2.0 \times 10^{-9}$</td>
<td>14</td>
<td>33</td>
<td>1880</td>
</tr>
</tbody>
</table>
1. Physics and Mathematics
2. Numerical Multiresolution Algorithms
3. Computing with Python and IPython
4. The future: IPython as a network-aware Python VM
5. Odds and Ends, Wrapup
**IPython: 2001-Today**

The interactive prompt: one of Python’s greatest strengths. But: it feels like a half-implemented idea (vs. the Unix shell, or Mathematica’s prompt)

In its simplest form, IPython is a BSD-licensed Python shell replacement. In broader terms, it tries to be:

1. **A better Python shell**: object introspection, system access, ’magic’ command system for adding functionality when working interactively, . . .

2. **An embeddable interpreter**: useful for debugging and for mixing batch-processing with interactive work.

3. **A flexible component**: you can use it as the base environment for other systems with Python as the underlying language. It is very configurable in this direction.

4. **A system for interactive control of distributed/parallel computing systems.**

5. **An interactive component** we can plug into GUIs, browser-based shells, etc.
IPython and scientific computing

Scientific computing is *exploratory*: a good interactive environment is a *necessity*.

What is IPython?

- Besides a (much) better interactive shell, with threading support (GTK, WX and Qt), system access, etc...

- It is a base layer for building customized interactive environments.
IPython and scientific computing
Scientific computing is *exploratory*: a good interactive environment is a *necessity*.

What is IPython?

- Besides a (much) better interactive shell, with threading support (GTK, WX and Qt), system access, etc...
- It is a base layer for building customized interactive environments.

How?

- Input preprocessing: $\frac{2}{3} \rightarrow \mathbb{Z}/(2)/\mathbb{Z}(3)$
- Output preprocessing.
- Customized tab-completers: PyMad (Institut Laue Langevin-CEA Grenoble), tab-completion over the network for proxied objects.
- Customized exception handlers. And a lot more...
IPython and scientific computing

Scientific computing is *exploratory*: a good interactive environment is a *necessity*.

What is IPython?

- Besides a (much) better interactive shell, with threading support (GTK, WX and Qt), system access, etc...
- It is a base layer for building customized interactive environments.

How?

- Input preprocessing: $2/3 \rightarrow \mathbb{Z}/2\mathbb{Z}/\mathbb{Z}/3$
- Output preprocessing.
- Customized tab-completers: PyMad (Institut Laue Langevin-CEA Grenoble), tab-completion over the network for proxied objects.
- Customized exception handlers. And a lot more...

**Requirements**: IPython depends only on the Python standard library (Python $\geq 2.3$) and is written in pure Python (no extension C code).
Who uses IPython?

- Available for all Linux distributions and Fink, distributed by Enthought for Windows.
- **SAGE**: a system for mathematical research and teaching with a focus on algebra, geometry and number theory.
- **PyRAF**: environment for astronomical image analysis, from the Space Telescope Science Institute.
- **CASA**: The interactive shell for CASA (Common Astronomy Software Applications), from the National Radio Astronomy Observatory.
- **Ganga**: system developed at CERN for Grid job control for the LHCb and ATLAS experiments, uses IPython for its command-line CLIP interface.
- **PyMAD**: IPython is used to control a neutron spectrometer at the CEA-Grenoble and the Institut Laue Langevin in France.
- **Pymerase**: project for microarray gene expression databases, exposes an IPython shell in its interactive iPymerase mode.
A few useful interactive tools

- `%run`: execution and testing of code, with *lots* of bells and whistles.
- `%edit`: call your favorite `$EDITOR` on the spot.
- `%macro`: interactively recall groups of lines quickly (you can `%edit` them)
- `%save`: save a group of lines to a named file.
- `%store`: lightweight persistence for any variable (including macros).
- `%debug` and `%pdb`: automatic invocation of a debugger (IPython-enhanced pdb)
- Embedding IPython: open an interactive shell inside any program you want
- Shell access: direct access to the underlying OS. Use Python for shell-like tasks (*much* nicer syntax than bash).
Outline

1. Physics and Mathematics
2. Numerical Multiresolution Algorithms
3. Computing with Python and IPython
4. The future: IPython as a network-aware Python VM
5. Odds and Ends, Wrapup
Lessons from IPython

- IPython has been pushed by some almost to replace Python as the user-visible system (better exceptions, control via `%magics`, etc.)
Lessons from IPython

- IPython has been pushed by some almost to replace Python as the user-visible system (better exceptions, control via `%magics`, etc.)

- If everyone wants it so bad, let’s give it to them!
Lessons from IPython

- IPython has been pushed by some almost to replace Python as the user-visible system (better exceptions, control via $\%$magics, etc.)

- If everyone wants it so bad, let’s give it to them!

- Any good interactive system should have *two* levels of operation:
  - The actual execution language: in our case, Python.
  - A control mechanism: the $\%$magics.

- But we’ll make all of this available *over the network*.

- And it will be *non-blocking* (so your extension C code doesn’t freeze your sessions).

- We’re going to clarify, organize and improve all the public extension points, so extension authors have an easier time.
The kernel is an IPython instance that listens on a network port rather than to an interactive prompt.

- It has a control protocol for commands.
- And it can also pass any object which can be serialized (pickle for now).
- Developed using Twisted and non-blocking sockets.
- Can be started at any time using various means (SSH, Xgrid, GridEngine, Condor, etc.)
- Eventually, this kernel will be the core of IPython.
IPython as we know it will continue to exist
But better, cleaner, and embeddable in GUIs

Today's IPython

- Terminal controller
  - In-process
  - Single-line (readline)
  - Multiline (curses)

IPython Kernel engine
- Single process
- User namespace
- No networking

What we wish we could do with today's IPython

- GUI environment
  - IDLE, Envisage,…
  - Still in-process
  - Different I/O

IPython Kernel engine
- Single process
- User namespace
- No networking
Why do we need this?

- The Python VM has a global lock (the Global Interpreter Lock – GIL).
- It protects the global state of the interpreter
  - Only one thread can execute Python code at the same time.
  - No Python variables may be modified without holding the GIL.
- Python *does* have threads: they work well for non-CPU bound tasks.

**BUT**

- Extensions (C, Fortran) can fully block the VM.
- And poof goes all hope of the ability to control a cluster
A 2-process kernel (2)

The "IPython VM"

Kernel Controller
- Non-blocking
- World-visible

Kernel Engine
*MAY BLOCK!*

Twisted

Networks
Distributed/parallel computing

- Think of Python as 'the CPU’
- But these souped-up kernels let you talk to it conveniently.

![Diagram of IPython InteractiveCluster](image_url)
Outline

1. Physics and Mathematics
2. Numerical Multiresolution Algorithms
3. Computing with Python and IPython
4. The future: IPython as a network-aware Python VM
5. Odds and Ends, Wrapup
FluidLab: a MayaVi based CFD visualization tool
Volumetric rendering with FluidLab
GPUs for numerics

- Project in 2006: implement an unequally spaced FFT (Fast Fourier Transform) on a GPU.
- Done using NVidia G6800-series hardware.
- Very painful, but we got our part working.
- Current NVidia hardware (G80) has significant improvements that make it far more attractive for development.
- Much less of a square peg (your numerical algorithm) into a round hole (a 3d graphics language like OpenGL/Cg).
- This can be a very interesting technology in the years to come, and in a sense it is inevitable.
Reflections on technical computing

- More and more fields are becoming computationally driven (biology, medical sciences, finance, ...)
- Better tools are needed to explore problems, try algorithms, play with data, ...
- Python is a good fit in this context.
  - And I think IPython fills an important need there.
- As computers get faster, better algorithms are needed ($N \log N$ is really better than $N^2$ only if $N$ gets large).
  - Python is great for algorithmic exploration and development.
- Parallelism is here, like it or not.
  - Multicore chips.
  - Cheaper clusters.
  - Hybrid CPU/GPU.
  - We’re also working hard on IPython so it contributes on this front.