High-level analysis scripts with low-level performance

Jim Pivarski

Princeton University – DIANA

June 20, 2016

This talk is about a future project I'm working toward.

Goals:

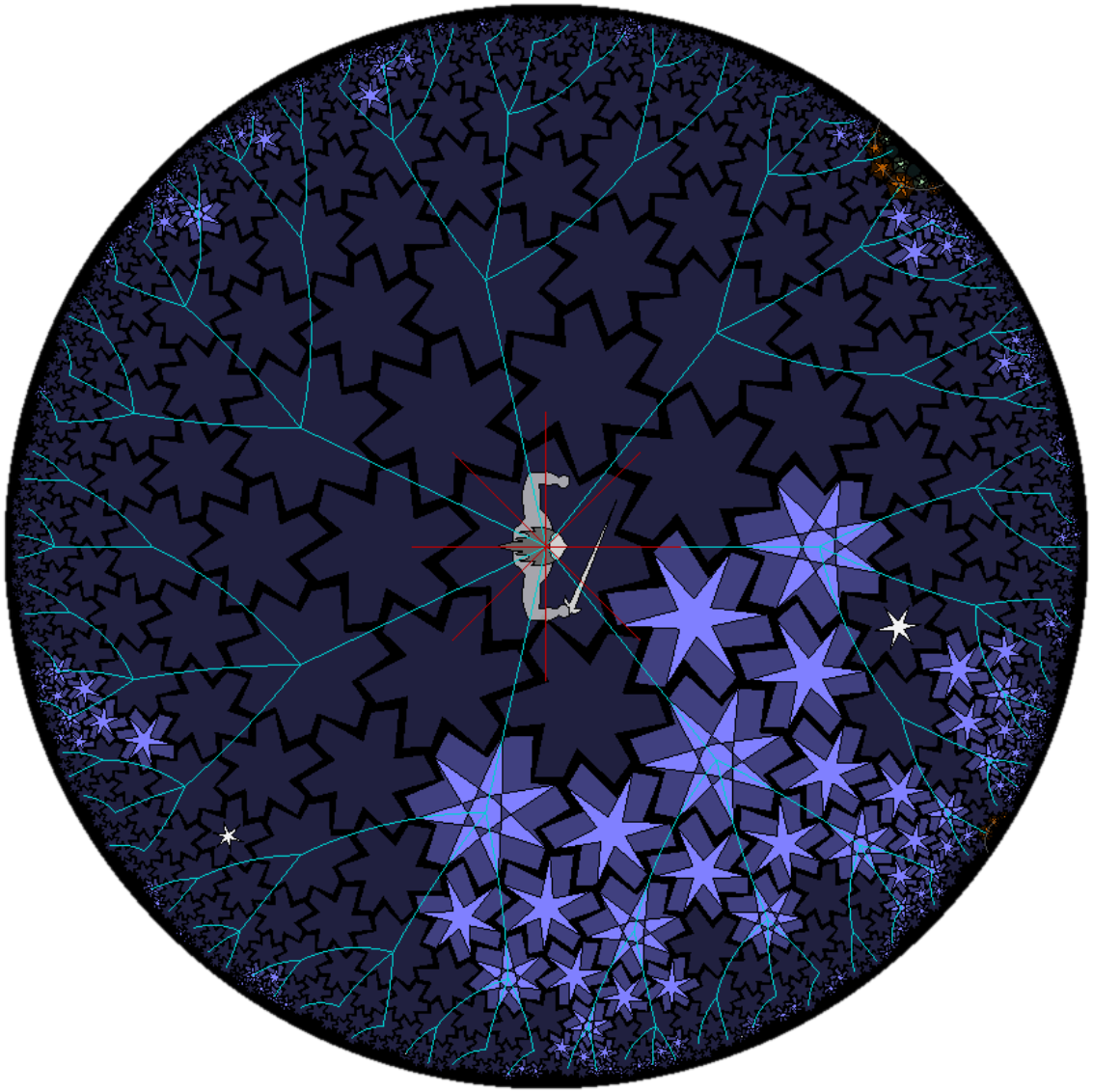
- to increase the separation between physics-relevant concepts and low-level computing details
- without sacrificing computational performance; in most cases, improving it.

This talk is about programming languages because languages *are the user interface* of data analysis.

The same is true in industry:

- business intelligence speaks SQL,
- statisticians speak R and SAS,
- financial analysts write extensive Excel macros. . .

The choice of language matters



All programming languages fill the "space of possible programs" because they're Turing complete.

However, different languages are like different metrics on this space.

A small change in one is a big change in another.

Evolution of language choice

High energy physics

Decades of Fortran, transitioned to C++ in late '90s, may be leaning towards Python now.

Data science in industry

Big Data/Hadoop grew out of web development: distributed systems in Java.

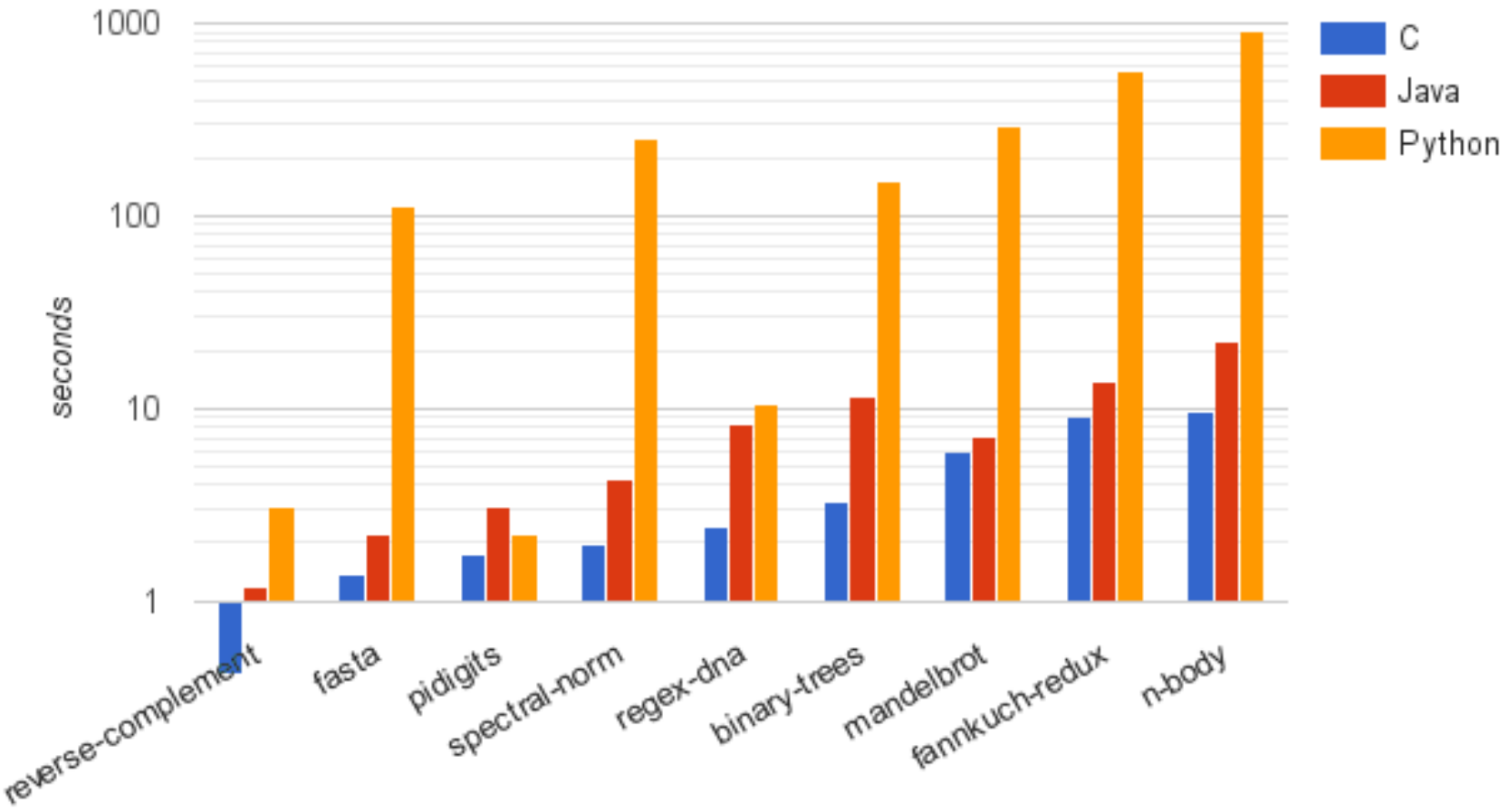
Now involves more machine learning and statistics, so shift toward Scala on the JVM, Python for Scikit-Learn, and R.

machine code, assembly	**"native"** C, C++, Fortran	**"managed"** Java, C#, Scala	**"interpreted"** Python, R, Javascript	regex, JSON, config files...

fast, powerful, "close to the metal" ⟷ flexible, dynamic, or convenient

Abstractions *versus* performance?

Experience tells us that low-level is fast and high-level is slow.



http://benchmarksgame.alioth.debian.org/

Domain-Specific Languages (DSLs)

But it doesn't have to be: intentionally restricting the scope of the language allows more optimization.

Prime example: SQL.

But also:

- Fortran's lack of (aliasable) pointers
- regular expressions for string manipulation
- Numpy in Python
- Histogrammar...

More accurate statement

High-level abstractions + complete generality is slow.

High-level abstractions + restricted domain

- can be as fast as a custom-tuned program (especially with JIT),
- but with better separation of domain knowledge from computing details.

More accurate statement

High-level abstractions + complete generality is slow.

High-level abstractions + restricted domain

- can be as fast as a custom-tuned program (especially with JIT),
- but with better separation of domain knowledge from computing details.

A well-designed DSL can encourage exploration of the problem space (physics) while the backend optimizes performance.

(A poorly designed DSL can make it impossible to get work done!)

Goals for my future project

I plan to design a domain specific language for end-user physics analysis scripts with the following properties:

- a subset of or based on Python syntax
- non-exclusive: mix with normal (slow) Python
- immutable, maybe total-functional (next slide)
- very strongly typed, but only through inference (next^2 slide)
- manual optimizations via CSS-style selectors (next^3 slide)
- supporting imperative idioms through patterns (next^4 slide)

Goals for my future project

I plan to design a domain specific language for end-user physics analysis scripts with the following properties:

- a subset of or based on Python syntax
- non-exclusive: mix with normal (slow) Python
- immutable, maybe total-functional (next slide)
- very strongly typed, but only through inference (next^2 slide)
- manual optimizations via CSS-style selectors (next^3 slide)
- supporting imperative idioms through patterns (next^4 slide)

Scope: only the data manipulation, not whole applications

Goals for my future project

I plan to design a domain specific language for end-user physics analysis scripts with the following properties:

- a subset of or based on Python syntax
- non-exclusive: mix with normal (slow) Python
- immutable, maybe total-functional (next slide)
- very strongly typed, but only through inference (next^2 slide)
- manual optimizations via CSS-style selectors (next^3 slide)
- supporting imperative idioms through patterns (next^4 slide)

Scope: only the data manipulation, not whole applications

Backends: convert to C, CUDA/OpenCL, or Verilog/HDL for a traditional compiler to compile ("*transpiling* to C")

Goals for my future project

I plan to design a domain specific language for end-user physics analysis scripts with the following properties:

- a subset of or based on Python syntax
- non-exclusive: mix with normal (slow) Python
- immutable, maybe total-functional (next slide)
- very strongly typed, but only through inference (next^2 slide)
- manual optimizations via CSS-style selectors (next^3 slide)
- supporting imperative idioms through patterns (next^4 slide)

Scope: only the data manipulation, not whole applications

Backends: convert to C, CUDA/OpenCL, or Verilog/HDL for a traditional compiler to compile ("*transpiling* to C")

Stepping stone: Histogrammar, my histogram-aggregation DSL, is being used to test some of the basic ideas.

Immutable, maybe total-functional

"Functional programming" eliminates mutable program state:

- output of functions depend strictly on their inputs
- `x = x + 1` is a false mathematical statement
- assignments form a time-independent graph, may be written in any order and backend may execute in any order
- backend may substitute mutable data structures by analyzing (or temporally rearranging) the assignment graph
- good for concurrency (no locks)

Immutable, maybe total-functional

"Functional programming" eliminates mutable program state:

- output of functions depend strictly on their inputs
- `x = x + 1` is a false mathematical statement
- assignments form a time-independent graph, may be written in any order and backend may execute in any order
- backend may substitute mutable data structures by analyzing (or temporally rearranging) the assignment graph
- good for concurrency (no locks)

"Total functional programming" also eliminates unbounded loops and exceptions:

- programs are known to halt (not Turing complete), maybe even with time estimates from static analysis
- exactly model mathematical functions: $f : \mathcal{D} \to \mathcal{R}$

Strongly typed through inference

Type check is a formal proof that program is free of certain errors.

Scala example (eliminates runtime null pointer exceptions):

```
val numberOrNone: Option[Double] = Some(3.14)
val cosx = numberOrNone match {
  case Some(x) => cos(x)
  case None => -999.0
}

// or better
val cosxOrNone = numberOrNone.map(cos(_))

// but cos(numberOrNone) would be a compiler error
```

Strongly typed through inference

- `numberOrNone` is a value from the set $\mathbb{R} \cup \{\texttt{None}\}$.

Strongly typed through inference

- `numberOrNone` is a value from the set $\mathbb{R} \cup \{\texttt{None}\}$.
- One could catch division-by-zero errors in the same way by considering sets like $\mathbb{R} \cup \{-\infty, \infty\}$.

Strongly typed through inference

- `numberOrNone` is a value from the set $\mathbb{R} \cup \{\texttt{None}\}$.
- One could catch division-by-zero errors in the same way by considering sets like $\mathbb{R} \cup \{-\infty, \infty\}$.
- For physics applications, it could be useful to consider any interval, like $[-3, 8] \cap \mathbb{Z}$ or $[0, \infty)$, as "data types."

Strongly typed through inference

- `numberOrNone` is a value from the set $\mathbb{R} \cup \{\texttt{None}\}$.
- One could catch division-by-zero errors in the same way by considering sets like $\mathbb{R} \cup \{-\infty, \infty\}$.
- For physics applications, it could be useful to consider any interval, like $[-3, 8] \cap \mathbb{Z}$ or $[0, \infty)$, as "data types."
  - This is like an extreme form of `int` versus `unsigned int`.
  - Useful feedback to the data analyst: "Why does my function output have such a large range?"
  - Could even be used to set bit widths for an FPGA backend.
  - I have implemented this for $+$, $-$, $*$, $/$, $**$, and modular arithmetic with 6k lines of unit tests. Extending to continuous functions will involve searches for inflection points.

Strongly typed through inference

- `numberOrNone` is a value from the set $\mathbb{R} \cup \{\texttt{None}\}$.
- One could catch division-by-zero errors in the same way by considering sets like $\mathbb{R} \cup \{-\infty, \infty\}$.
- For physics applications, it could be useful to consider any interval, like $[-3, 8] \cap \mathbb{Z}$ or $[0, \infty)$, as "data types."
  - This is like an extreme form of `int` versus `unsigned int`.
  - Useful feedback to the data analyst: "Why does my function output have such a large range?"
  - Could even be used to set bit widths for an FPGA backend.
  - I have implemented this for $+$, $-$, $*$, $/$, $**$, and modular arithmetic with 6k lines of unit tests. Extending to continuous functions will involve searches for inflection points.
- Inference only: intervals specified on input arguments, everything else inferred. The compiler should be telling the user what the domains are, not the other way around. (Being purely functional helps this.)

Optimizations via CSS-style selectors

High-level code is frustrating when it *takes away* the ability to manually optimize.

The point is not to make the physicist unaware of the low-level details, just to remove the necessity of thinking about both at the same time.

Optimizations via CSS-style selectors

High-level code is frustrating when it *takes away* the ability to manually optimize.

The point is not to make the physicist unaware of the low-level details, just to remove the necessity of thinking about both at the same time.

(You don't have to think about nuclear physics when studying atomic structure, but that doesn't mean you can't *know about* nuclear physics!)

Optimizations via CSS-style selectors

Take a hint from HTML+CSS, which separates structure from style by putting them in two separate files:

HTML file

```html
<html>
  <body>
    <ul id="bulleted-list">
      <li class="first">one</li>
      <li class="rest">two</li>
      <li class="rest">three</li>
    </ul>
    <ol>
      <li>unaffected</li>
    </ol>
  </body>
</html>
```

CSS file

```css
#id { border: solid 1px
    red; }

ul li { color: blue; }

li.first { font-weight:
    bold; }

.rest { text-decoration:
    underline; }
```

Optimizations via CSS-style selectors

Consider a variant of CSS selectors that picks program elements and applies optimization hints:

Correctness

```
# type declarations as Python3
    argument decorations
def doWeirdStuff(
      x: [-10, 10],
      xs: list(size=[1, inf],
               data=[-5, 5])):

    # function body
    xs2 = xs.appended(xs[0])

    return xs2.appended(x / 2)
```

Performance

```
/* only affects x in doWeirdStuff,
    not other functions */
doWeirdStuff x {
  data-type: signed char;
}

/* implies that xs2 is also a
    mutable linked list */
xs {
  data-type: mutable linked list;
  storage: contiguous obstack;
}
```

Status: not deeply thought-through yet.

Imperative idioms through patterns

dianahep

Problem with Python

Large-scale syntax isn't suited for functional programming:

- control in statements, not expressions
- cannot put statements in lambda functions

Problem with anything else

Unfamiliar to physicists: yet another language!

Besides, Python's expression syntax is excellent, want to keep that.

Imperative idioms through patterns

Imperative code, the way Python was meant to be used:

```
def function(x: (-inf, inf)):
    if x > 0:
        y = 1
    elif x < 0:
        y = -1
    else:
        y = 0

    tenOfThem = []
    for i in range(10):
        tenOfThem.append(y)
    return tenOfThem
```

Functional code, the way I'd want to use it:

```
def function(x: (-inf, inf)):
    y =  1 if x > 0 else
        -1 if x < 0 else
         0

    tenOfThem = range(10) \
        .map(lambda i: y)

    return tenOfThem
```

I have to think backwards to read the one on the right.

Imperative idioms through patterns

Imperative code, the way Python was meant to be used:

```
def function(x: (-inf, inf)):
    if x > 0:
        y = 1
    elif x < 0:
        y = -1
    else:
        y = 0

    tenOfThem = []
    for i in range(10):
        tenOfThem.append(y)
    return tenOfThem
```

Functional code, the way I'd want to use it:

```
def function(x: (-inf, inf)):
    y =  1 if x > 0 else
        -1 if x < 0 else
         0

    tenOfThem = range(10) \
        .map(lambda i: y)

    return tenOfThem
```

But suppose the left is recognized as "idioms" and translated?

- `if` statements where every branch defines the same symbol
- `for` loops that only append to a list

Transpiling in Histogrammar

Histogrammar is a DSL with a much smaller scope (making histograms in distributed systems).

Deeply nested structure is a nice abstraction, but it's surely slower than filling an array.

```
directory_of_histograms =
    Label(
        one = Select(lambda d: d.trigger > 5,
                     Bin(100, 0, 80, lambda d: d.pt, Count())),
        two = Select(lambda d: d.pt > 30,
                     Bin(100, 0, 120, lambda d: d.met, Count()))
    )
```

Transpiling in Histogrammar

Histogrammar is a DSL with a much smaller scope (making histograms in distributed systems).

Deeply nested structure is a nice abstraction, but it's surely slower than filling an array.

```
directory_of_histograms =
    Label(
        one = Select(lambda d: d.trigger > 5,
                     Bin(100, 0, 80, lambda d: d.pt, Count())),
        two = Select(lambda d: d.pt > 30,
                     Bin(100, 0, 120, lambda d: d.met, Count()))
    )
```

Transparent speed-up: JIT compilation

http://github.com/diana-hep/histogrammar/scala-jit transpiles the histogram structure into explicit C code, compiles it, and runs it.

Transpiling in Histogrammar

Auto-generated C code operates on batches of data, batches of histograms, by casting pointers in a contiguous malloc block.

```
#include <inttypes.h>
#include <math.h>

uint64_t loop(void *dataBatch, void *storageBatch, int32_t inputBufferFill) {
  uint64_t storagePointer;
  uint64_t BinningUnwind_0;
  double BinningQuantity_0;
  int32_t BinningBin_0;
  for (int32_t rowIndex = 0;  rowIndex < inputBufferFill;  ++rowIndex) {
    storagePointer = (uint64_t)storageBatch;
    // Binning unwind-protect
    BinningUnwind_0 = storagePointer;
    BinningQuantity_0 = (*((double*)(dataBatch + 0 + rowIndex*8)));
    if (BinningQuantity_0 != BinningQuantity_0) {
      // Binning.nanflow
      storagePointer += 816;
      // Counting.entries without weight
      ++(*((int32_t*)storagePointer));
      storagePointer += 8;
    }
    else {
      BinningBin_0 = (int32_t)floor(100 * (BinningQuantity_0 - 0.0) * 0.0125);
      if (BinningQuantity_0 == -INFINITY  ||  BinningBin_0 < 0) {
        // Binning.underflow
        storagePointer += 800;
        // Counting.entries without weight
        ++(*((int32_t*)storagePointer));
```

Transpiling in Histogrammar

The whole workflow (calculating data in Scala, copying to off-heap, filling histograms, bringing back results):

#histograms	#entries	naive Scala	JIT C
1	10,000,000	13 seconds	0.81 seconds
100	100,000	27 seconds	0.75 seconds

Just the tight loop around filling (pull data from an array) and using cling instead of tcc:

#histograms	#entries	JIT C	ROOT
1	100,000,000	0.74 seconds	2.2 seconds
100	1,000,000	0.44 seconds	2.2 seconds

Prior art: tools to use & interoperate with

dianahep

Cython: adds performance hints to Python so that it can be more easily compiled into extension modules (C code for gcc).

```
cpdef int myfunction(int x, int y=2):
    a = x-y
    return a + x * y

cdef double _helper(double a):
    return a + 1

cdef class A:
    cdef public int a,b
    def __init__(self, b=0):
        self.a = 3
        self.b = b

    cpdef foo(self, double x):
        print x + _helper(1.0)
```

Prior art: tools to use & interoperate with

dianahep

Numba: propagates Numpy types through a Python function to produce LLVM bytecode for JIT compilation.

```
from numba import jit
from numpy import arange

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

Prior art: tools to use & interoperate with

SymPy: symbolic algebra system in Python (expression graph is a functional program, which can be simplified).

Theano: matrix expression compiler with CPUs and GPUs.

SymPy has a strong connection with Theano, a mathematical array compiler. SymPy expressions can be easily translated to Theano graphs and then compiled using the Theano compiler chain.

Run code block in SymPy Live

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
```

Run code block in SymPy Live

```
>>> from sympy.printing.theanocode import theano_function
>>> f = theano_function([x], [expr])
```

Prior art: tools to use & interoperate with

dianahep

PyCUDA/PyOpenCL: dispatches kernels to GPU.

CodePy: AST for CUDA/OpenCL (same author).

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

Recap

High-level abstractions are not inconsistent with computational performance when the problem domain is sufficiently restricted.

I'm planning to expand my current work from histogramming abstractions to a domain specific language for physics analysis that is:

- familiar enough to be used by physicists
- designed around the specific needs of physics analysis
- transpiled to highly performant code.

Greg Owen's talk on JIT in Spark 2.0

Volcano Iterator Model

Standard for 30 years: almost all databases do it

Each operator is an "iterator" that consumes records from its input operator

```
class Filter {
  def next(): Boolean = {
    var found = false
    while (!found && child.next()) {
      found = predicate(child.fetch())
    }
    return found
  }

  def fetch(): InternalRow = {
    child.fetch()
  }
  …
}
```

databricks

23

Greg Owen's talk on JIT in Spark 2.0

What if we hire a college freshman to implement this query in Java in 10 mins?

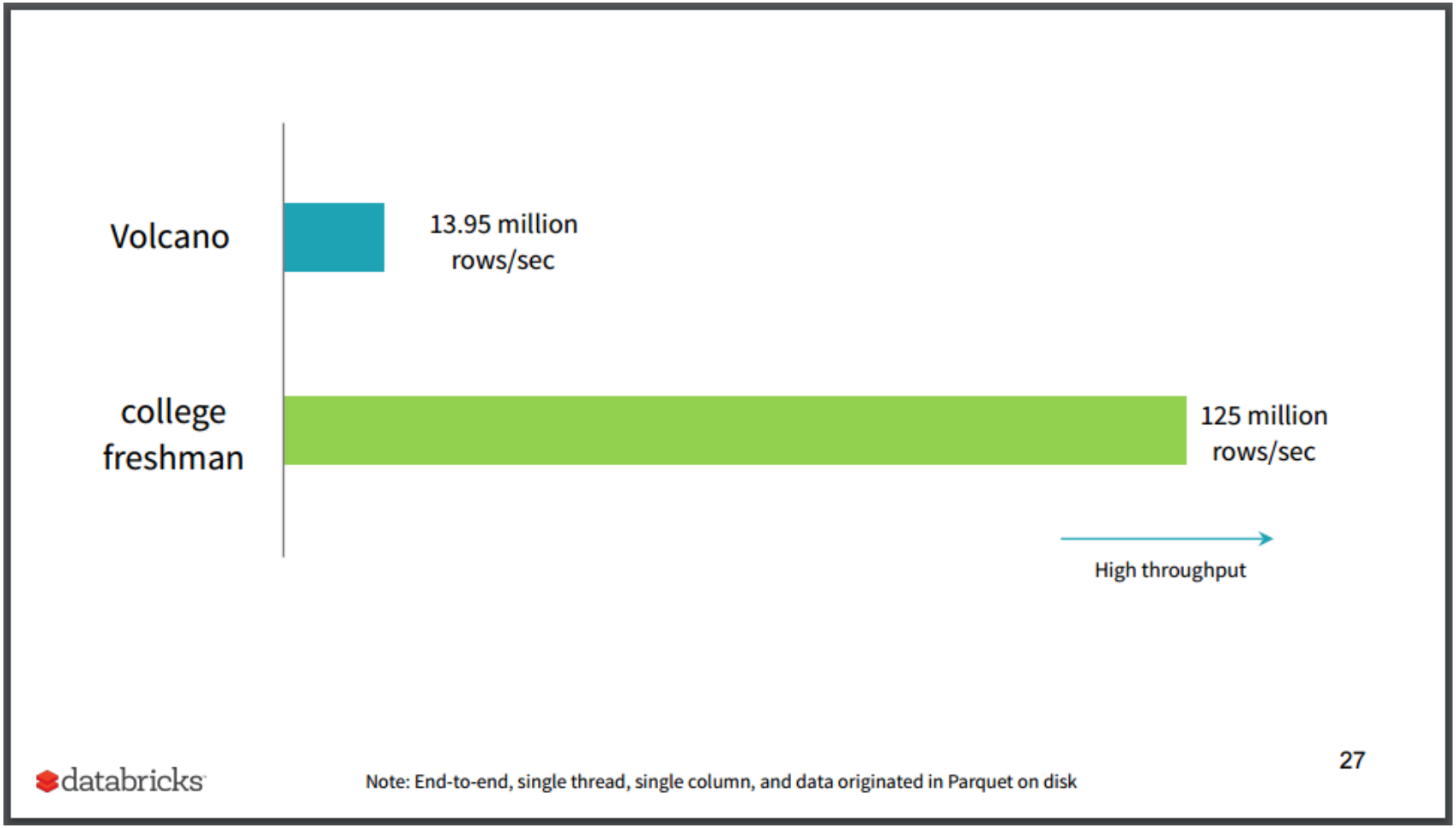
```
select count(*) from store_sales
where ss_item_sk = 1000
```

```
var count = 0
for (ss_item_sk in store_sales)
{
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

databricks

25

Greg Owen's talk on JIT in Spark 2.0



Greg Owen's talk on JIT in Spark 2.0

How does a student beat 30 years of research?

Volcano

1. Many virtual function calls
2. Data in memory (or cache)
3. No loop unrolling, SIMD, pipelining

Hand-written code

1. No virtual function calls
2. Data in CPU registers
3. Compiler loop unrolling, SIMD, pipelining

Take advantage of all the information that is known after query compilation

databricks

28