# High Performance Computing in Python

Andreas Kloeckner

University of Illinois

December 9, 2019

# Outline

Python+GPU

Arrays

Parallel Patterns

Performance: Expectations and Measurement

# Why GPUs?

- IPC and Clock frequency have not shown improvements
  - Parallel computers: only way to more work per second
- Not just parallelism: concurrency!
- Need to expose concurrency to the programming model
- "GPU:" a name for a processor architecture focused on
  - Concurrency
  - Throughput

# GPU Programmability

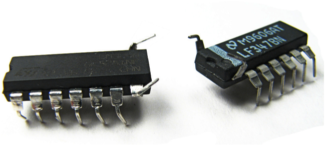
The 'nightmare limit':

- "Infinitely" many cores
- "Infinite" vector width
- "Infinite"memory/comm.\ latency

Further complications:

- Commodity chips
  - Compute only one design driver of many
- Bandwidth only achievable by *homogeneity*
- Compute bandwidth $\gg$ Memory bandwidth

$\rightarrow$ Programmability is key.

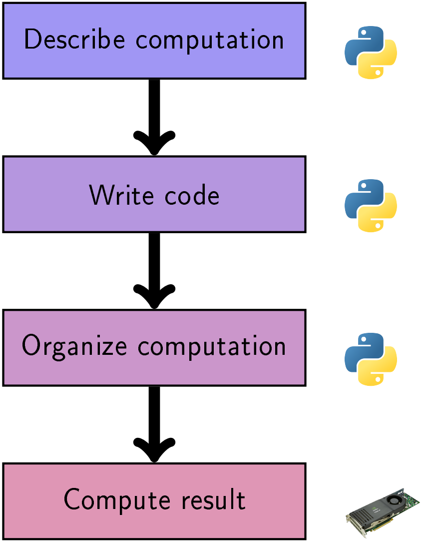


# Why Python for HPC

- Mature, large and active community
- Emphasizes readability
- Written in widely-portable C
  - Easy coupling to C/C++ (pybind11) / Fortran (f2py)
- A 'multi-paradigm' language
- Rich ecosystem of sci-comp related software
- Great as a 'glue language'

[Python logo: python.org]

# Addressing the Elephant in the Room: Slowness



# Python + GPUs

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

  $\rightarrow$ complement each other
- CPU: largely restricted to control tasks $\sim$1000/sec
  - Scripting fast enough
- Python + OpenCL = **PyOpenCL**
- Python + CUDA = **PyCUDA**


[GPU: Nvidia Corp.]

# What is OpenCL?

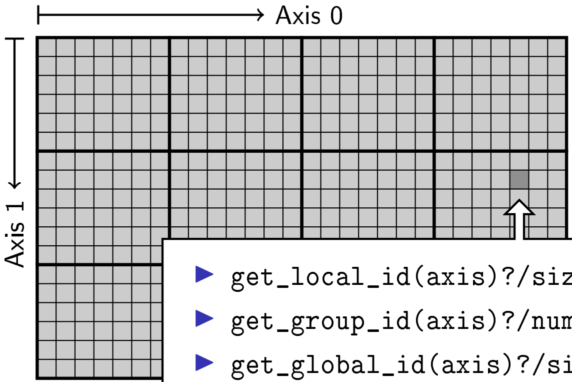
OpenCL (Open Computing Language) is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. [OpenCL 1.1 spec]

- Device-neutral (Nv GPU, AMD GPU, Intel/AMD CPU)
- Vendor-neutral
- JIT built into the standard

Defines:

- Host-side programming interface (library)
- Device-side programming language (!)

# Wrangling the Grid




- `get_local_id(axis)?/size(axis)?`
- `get_group_id(axis)?/num_groups(axis)?`
- `get_global_id(axis)?/size(axis)?`

`axis=0,1,2,...`

# Machine Abstractions

Is OpenCL only for GPUs?

No. Implementations for CPUs exist.

How does OpenCL map onto CPUs?

- Two levels of concurrency, one for cores, one for vector lanes
- Use the same mapping idea for CPUs
- Realize that you're not programming the hardware: you're programming an abstract model of the hardware.

What is essential about programming in OpenCL, what is arbitrary?

- Essential: the semantics of the programming model (what does the program mean?)
- Arbitrary: the spelling of the programm ($\rightarrow$ OCCA)

# Demo

[DEMO: 01-hello-pyopencl]

To follow along: `https://bit.ly/vtgpu19`

# Programming Approaches

Decisions that determine your approach to throughput computing:

- AOT vs JIT
- Meta vs not
- In-language vs Hybrid

# Outline

- Python+GPU
- Arrays
- Parallel Patterns
- Performance: Expectations and Measurement

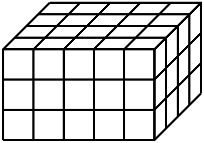
# Why Arrays?

- Parallelism: best if applied in large quantities
- Arrays: The only data structure that supports large-scale concurrency
- Structured
- $O(1)$ element access
- Static (i.e. known-from-the-outset) control flow for traversal

# Arrays in Numpy

Core attributes of an array:

- Shape
- `dtype` (data type)
- Strides
- Pointer
- (Lifetime relationship)



# Demo: Host Arrays

[DEMO: 02-numpy-arrays]

# Device Arrays

Want: An array object that works just like `numpy` arrays, but on the GPU

Issues:

- Which command queue? (Which context?)
- Synchronization?
- When to generate code? For which data types?

# Demo: Device Arrays

[DEMO: 03-pyopencl-arrays]

# Outline

- Python+GPU
- Arrays
- Parallel Patterns
- Performance: Expectations and Measurement

# Map

$$y_i = f_i(x_i)$$

where $i \in \{1, \dots, N\}$.

Notation: (also for rest of this lecture)

- $x_i$: inputs
- $y_i$: outputs
- $f_i$: (pure) functions (i.e. *no side effects*)

# Map

When does a function have a "side effect"?

In addition to producing a value, it

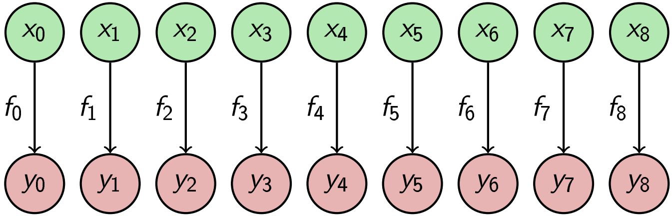
- modifies non-local state, or
- has an observable interaction with the outside world.

where $i \in \{1, \ldots, N\}$.

Notation: (also for rest of this lecture)

- $x_i$: inputs
- $y_i$: outputs
- $f_i$: (pure) functions (i.e. *no side effects*)

# Map: Graph Representation



# Embarrassingly Parallel: Examples

Surprisingly useful:

- Element-wise linear algebra:
  Addition, scalar multiplication (*not* inner product)
- Image Processing: Shift, rotate, clip, scale, ...
- Monte Carlo simulation
- (Brute-force) Optimization
- Random Number Generation
- Encryption, Compression
  (after blocking)

# Demo

[DEMO: 04-elementwise]

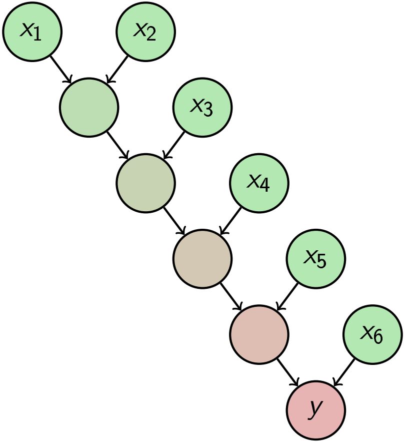
# Reduction

$$y = f(\cdots f(f(x_1, x_2), x_3), \ldots, x_N)$$

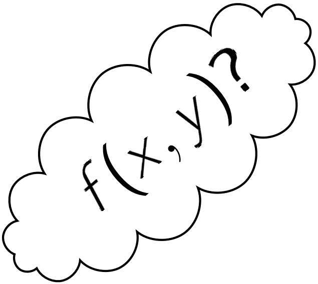
where $N$ is the input size.
Also known as

- Lisp/Python function `reduce` (Scheme: `fold`)
- C++ STL `std::accumulate`

# Reduction: Graph



# Approach to Reduction




Can we do better?

"Tree" very imbalanced. What property of $f$ would allow 'rebalancing'?

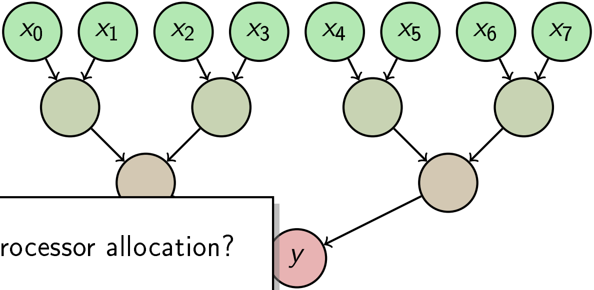
$$f(f(x,y),z) = f(x, f(y,z))$$

Looks less improbable if we let $x \circ y = f(x,y)$:

$$x \circ (y \circ z)) = (x \circ y) \circ z$$

Has a very familiar name: *Associativity*

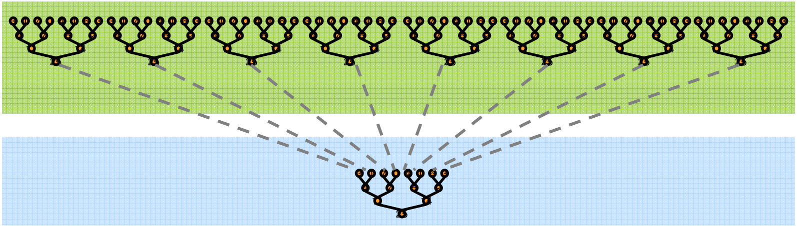
# Reduction: A Better Graph




Processor allocation?

# Mapping Reduction to SIMD/GPU

- Obvious: Want to use tree-based approach.
- Problem: Two scales, Work group and Grid
  - to occupy both to make good use of the machine.
- In particular, need synchronization after each tree stage.
- Solution: Use a two-scale algorithm.



*In particular:* Use multiple grid invocations to achieve inter-workgroup synchronization.

# Demo

[DEMO: 05-reduction]

# Scan

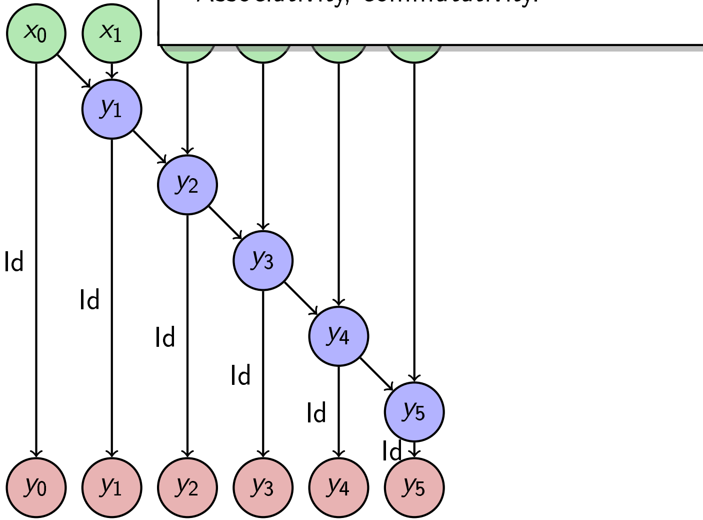
$$
\begin{aligned}
y_1 &= x_1 \\
y_2 &= f(y_1, x_2) \\
\vdots &= \vdots \\
y_N &= f(y_{N-1}, x_N)
\end{aligned}
$$

where $N$ is the input size. (Think: $N$ large, $f(x, y) = x + y$)

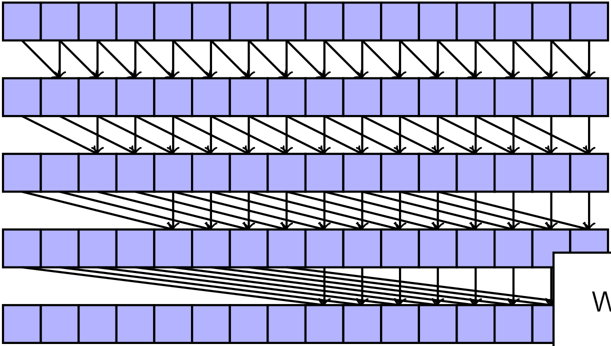
- Prefix Sum/Cumulative Sum
- Abstract view of: loop-carried dependence
- Also possible: Segmented Scan

# Scan: Graph

Again: Need assumptions on $f$.
Associativity, commutativity.



# Scan: Implementation



Work-efficient?

# Scan: Implementation II

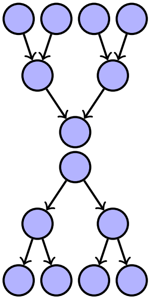
Two sweeps: Upward, downward, both tree-shape

On upward sweep:

- Get values `L` and `R` from left and right child
- Save $L$ in local variable `Mine`
- Compute `Tmp = L + R` and pass to parent

On downward sweep:

- Get value `Tmp` from parent
- Send `Tmp` to left child
- Sent `Tmp+Mine` to right child



# Scan: Examples

Name examples of Prefix Sums/Scans:

- Anything with a loop-carried dependence
- One row of Gauss-Seidel
- One row of triangular solve
- Segment numbering if boundaries are known
- Low-level building block for many higher-level algorithms algorithms, e.g. predicate filter, sort
- FIR/IIR Filtering
- Blelloch '93

# Demo

[DEMO: 06-scan]

# Assignment

Use PyOpenCL scan to

- Generate 10,000,000 uniformly distributed single-precision random numbers in $[0, 1)$
- Make a new array that retains only the ones $\leq 1/2$

# Practice

[DEMO: 07-scan-practice]

# Outline

Python+GPU

Arrays

Parallel Patterns

Performance: Expectations and Measurement

# Qualifying Performance

- What is *good* performance?
- Is speed-up (e.g. GPU vs CPU? C vs Matlab?) a meaningful way to assess performance?
- How else could one *form an understanding* of performance?

> Modeling: how understanding works in science

[Hager et al. '17](#)
[Hockney et al. '89](#)

# A Story of Bottlenecks

Imagine:

- A bank with a few service desks
- A revolving door at the entrance

What situations can arise at *steady-state*?

- Line inside the bank (good)
- Line at the door (bad)

What numbers do we need to characterize performance of this system?

- $P_{\text{peak}}$: [task/sec] Peak throughput of the service desks
- $I$: [tasks/customer] Intensity
- $b$: [customers/sec] Throughput of the revolving door

# A Story of Bottlenecks (cont'd)

- $P_{\text{peak}}$: [task/sec] Peak throughput of the service desks
- $I$: [tasks/customer] Intensity
- $b$: [customers/sec] Throughput of the revolving door

What is the aggregate throughput?

Bottleneck is either

- the service desks (good) or
- the revolving door (bad).

$$P \leq \min(P_{\text{peak}}, I \cdot b)$$

Hager et al. '17

# Application in Computation

Translate the bank analogy to computers:

- Revolving door: typicallly: Memory interface
- Revolving door throughput: Memory bandwidth [bytes/s]
- Service desks: Functional units (e.g. floating point)
- $P_{\text{peak}}$: Peak FU throughput (e.g.: [flops/s])
- Intensity: e.g. [flops/byte]

Which parts of this are task-dependent?

- All of them! This is not *a* model, it's a *guideline* for making models.
- Specifically $P_{\text{peak}}$ varies substantially by task

$$P \leq \min(P_{\text{peak}}, I \cdot b)$$

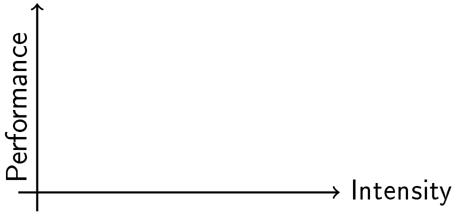
# A Graphical Representation: 'Roofline'

Plot (often log-log, but not necessarily):

- X-Axis: Intensity
- Y-Axis: Performance

What does our inequality correspond to graphically?

$$P \leq \min(P_{\text{peak}}, I \cdot b)$$




What does the shaded area mean?

Achievable performance

[Hager et al. '17](#)

# Example: Vector Addition

```
double r, s, a[N];
for (i=0; i<N; ++i)
  a[i] = r + s * a[i];}
```

Find the parameters and make a prediction.

Machine model:

- Memory Bandwidth: e.g. $b = 10$ GB/s
- $P_{\text{peak}}$: e.g. 4 GF/s

Application model:

- $I$ = 2 flops / 16 bytes = 0.125 flops/byte

# Demo

[DEMO: 08-perf-model]