Building High-Performance Tools with Python

Andreas Klöckner

Computer Science · University of Illinois at Urbana-Champaign

May 23, 2016

Thanks

- Tim Warburton (Rice)
- Lucas Wilcox (NPS)
- Jan Hesthaven (Brown)
- Leslie Greengard (NYU)
- PyOpenCL, PyCUDA contributors
- AMD, Nvidia
- National Science Foundation
- Office of Naval Research

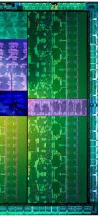
Outline

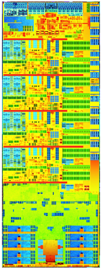
Outline

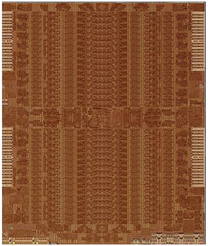
1. Introduction
2. What's wrong with computers, then?
3. PyOpenCL
4. Key Algorithm: Scan
5. Loop Generation
6. Conclusions

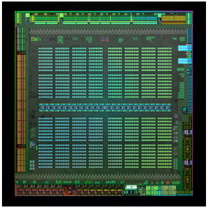
Life is Amazing

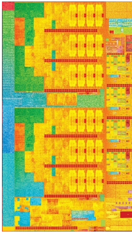
Hardware is advancing at a breakneck pace


K110
12)


Intel Haswell
(2013)


AMD Fiji (2014)


Nvidia GM204
(2014)


Intel Broa
(2015

But. . .

Be honest: Have you noticed your code getting any faster lately?

But. . .

Be honest: Have you noticed your code getting any faster lately?

E.g. Sandy Bridge → Broadwell

Two possible ways forward:

- Be content with what you can get, don’t worry too much about performance.
- Try harder.

Two possible ways forward:

- ~~Be content with what you can get, don't worry too much about performance.~~
- Try harder.

Two possible ways forward:

- ~~Be content with what you can get, don't worry too much about performance.~~
- **Try harder.**

High Performance: What?

What is... High Performance Computing?

The *science* of making code actually fast.

High Performance: What?

What is. . . High Performance Computing?

The *science* of making code ~~actually fast.~~

High Performance: What?

What is. . . High Performance Computing?

The *science* of making code ~~actually fast.~~
achieve the **best** performance possible on a given machine.

High Performance: What?

What is. . . High Performance Computing?

The *science* of making code ~~actually fast.~~
achieve the **best** performance possible on a given machine.

- **NO:** I made my code 300,000x faster.
- **YES:** My code achieves 37% of the achievable floating point capability of my machine.

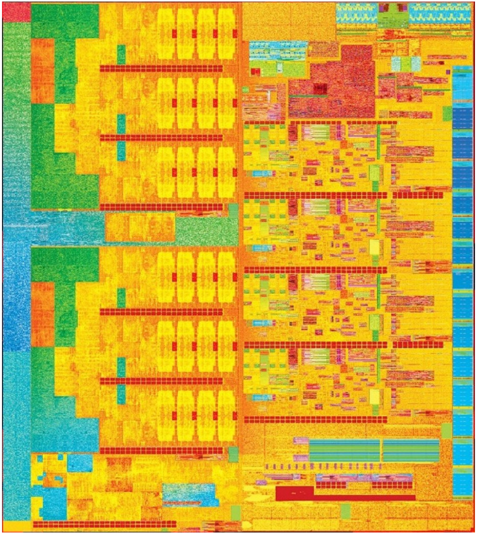
High Performance: What?

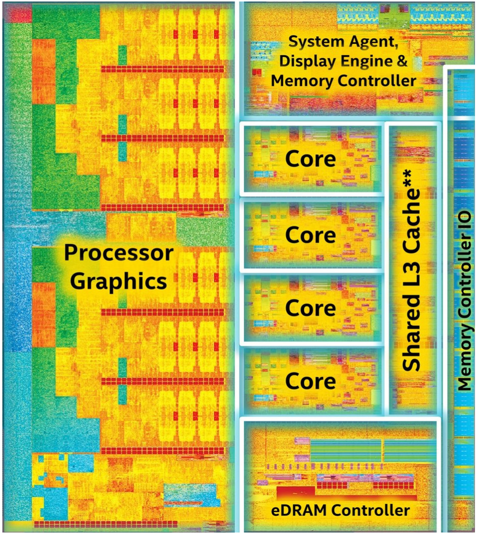
What is. . . High Performance Computing?

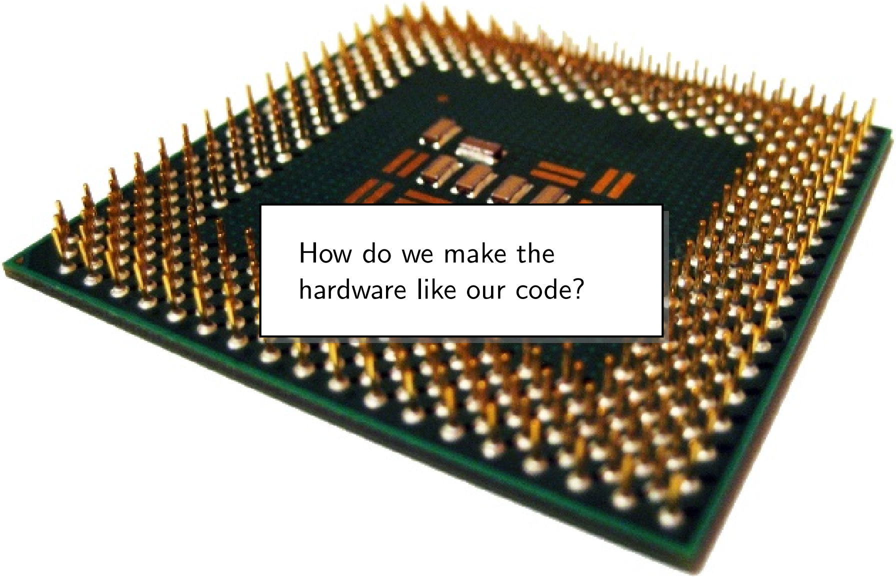
The *science* of making code ~~actually fast.~~
achieve the **best** performance possible on a given machine.

- **NO:** I made my code 300,000x faster.
- **YES:** My code achieves 37% of the achievable floating point capability of my machine.

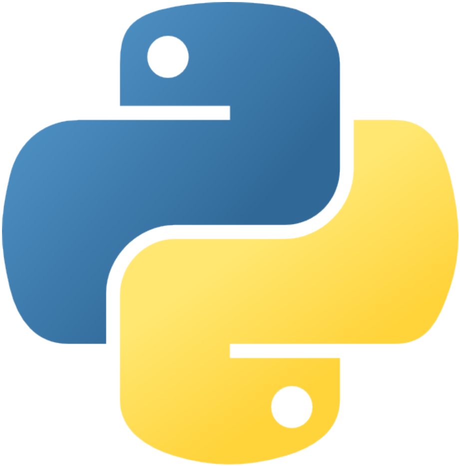
Performance: Measure → Understand → Improve → Measure → Understand → Improve → · · ·




System Agent, Display Engine & Memory Controller
Processor Graphics
Core
Core
Core
Core
Shared L3 Cache**
Memory Controller IO
eDRAM Controller



How do we make the hardware like our code?



But. . .

But. . .

Scripting languages are

But. . .
Scripting languages are **S**

But. . .
Scripting languages are **SL**

But. . .

Scripting languages are **SLO**

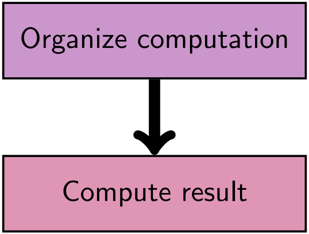
But. . .

Scripting languages are **SLOW**

But. . .
Scripting languages are **SLOW**

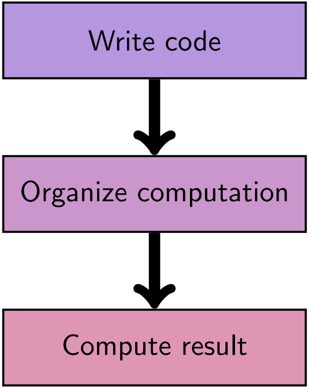
So using them for high performance makes no sense, right?

Compute result



Organize computation

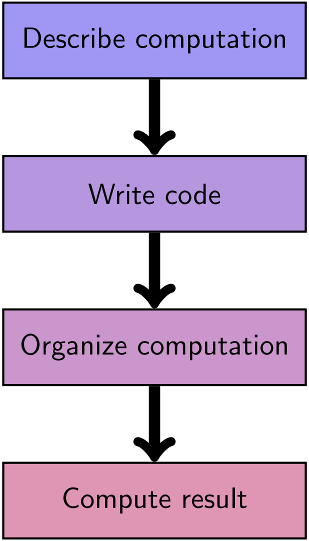
Compute result



Write code

Organize computation

Compute result



Describe computation

Write code

Organize computation

Compute result

Describe computation

↓

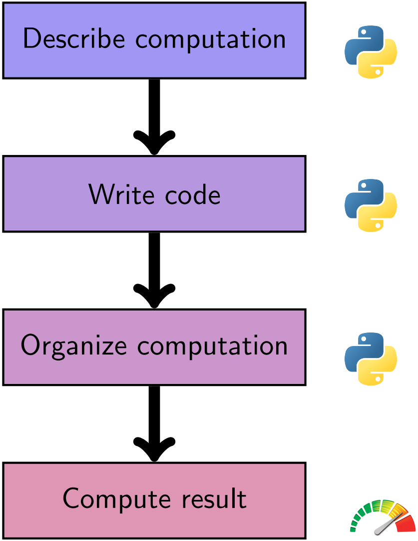
Write code

↓

Organize computation

↓

Compute result



Describe computation

Write code

Organize computation

Compute result

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 (!)

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 standa

Defines:

- Host-side programming
- Device-side programmin

Main advantage: OpenCL's abstract model of the machine is sensible and likely to stick around. . .

. . . on GPUs *and* CPUs.

OpenCL/CUDA/ISPC: same idea.

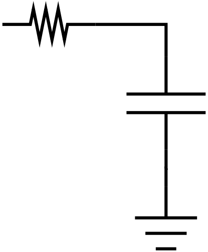
Low-level: sensible

Outline

Moving data

Data is moved through wires.

Wires behave like an RC circuit.



Trade-off:

- Longer response time (“latency”)
- Higher current (more power)

Physics says: *Communication is slow, power-hungry, or both.*

Programming model

Inspired by [Fatahalian '09]

Programming model

Inspired by [Fatahalian '09]

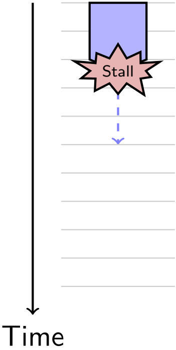
Programming model

Inspired by [Fatahalian '09]

Programming model

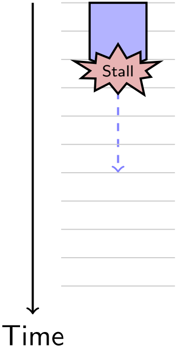
Inspired by [Fatahalian '09]

Programming model



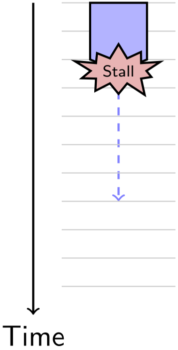
Inspired by [Fatahalian '09]

Programming model



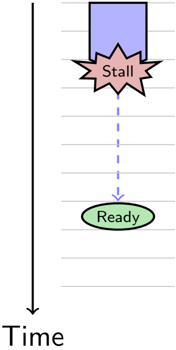
Inspired by [Fatahalian '09]

Programming model



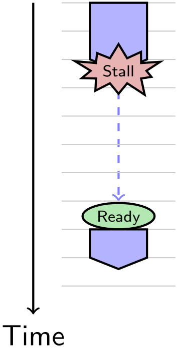
Inspired by [Fatahalian '09]

Programming model



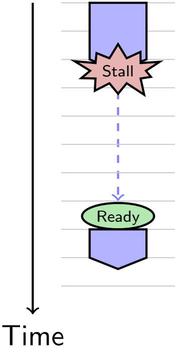
Inspired by [Fatahalian '09]

Programming model



Inspired by [Fatahalian '09]

Programming model

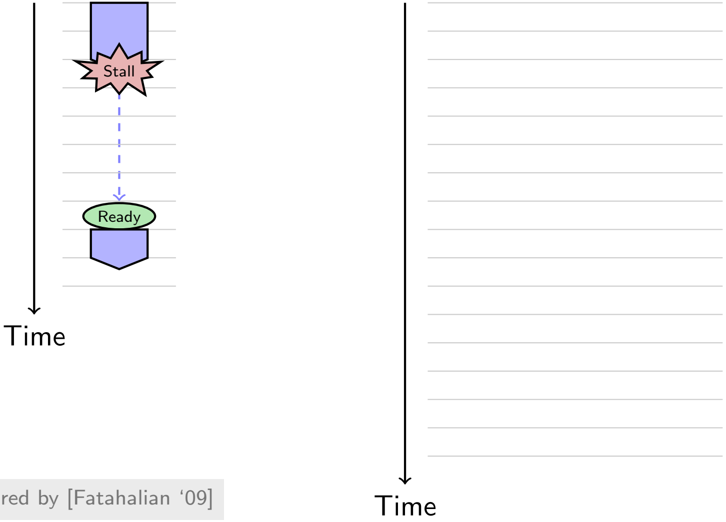


Not all computations require strict ordering.

Especially not numerical ones.

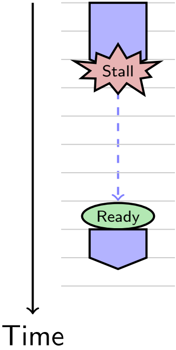
Inspired by [Fatahalian '09]

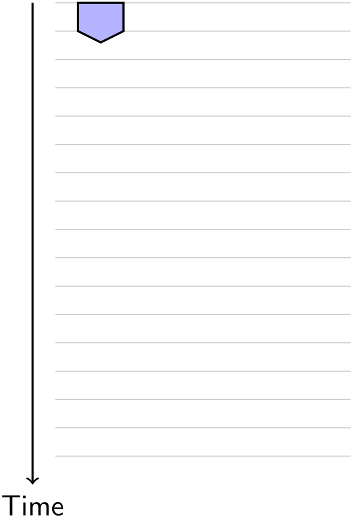
Programming model



Inspired by [Fatahalian '09]

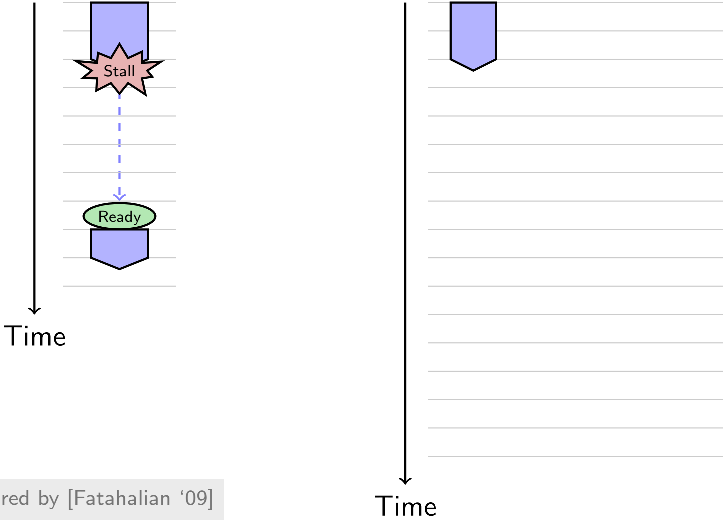
Programming model





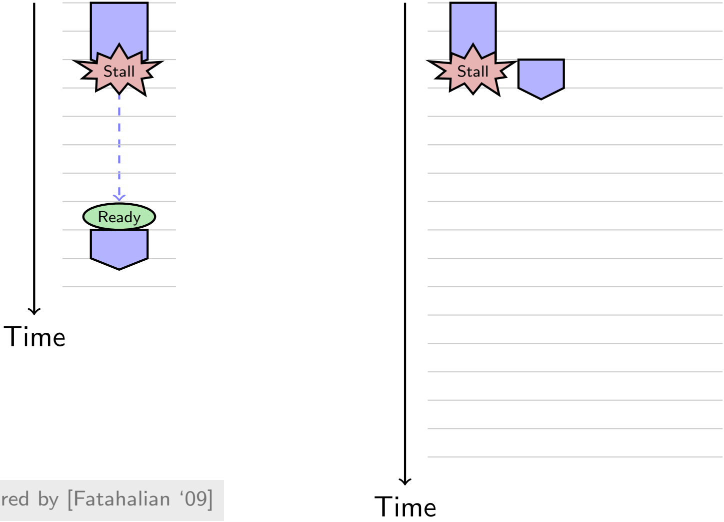
Inspired by [Fatahalian '09]

Programming model



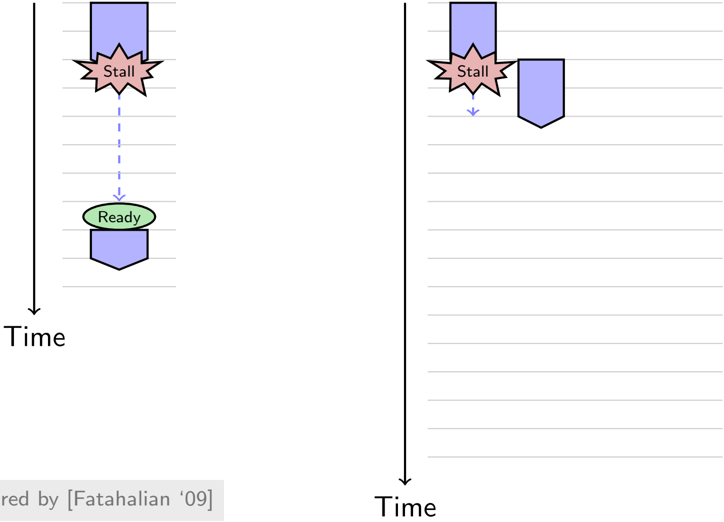
Inspired by [Fatahalian '09]

Programming model



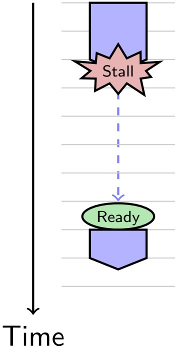
Inspired by [Fatahalian '09]

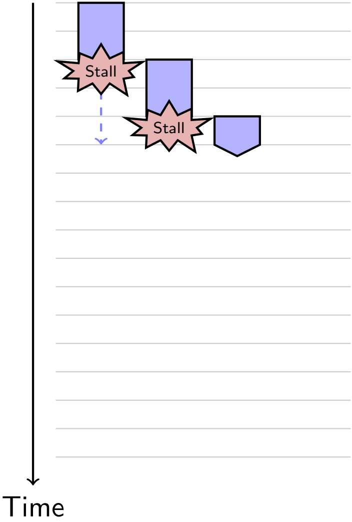
Programming model



Inspired by [Fatahalian '09]

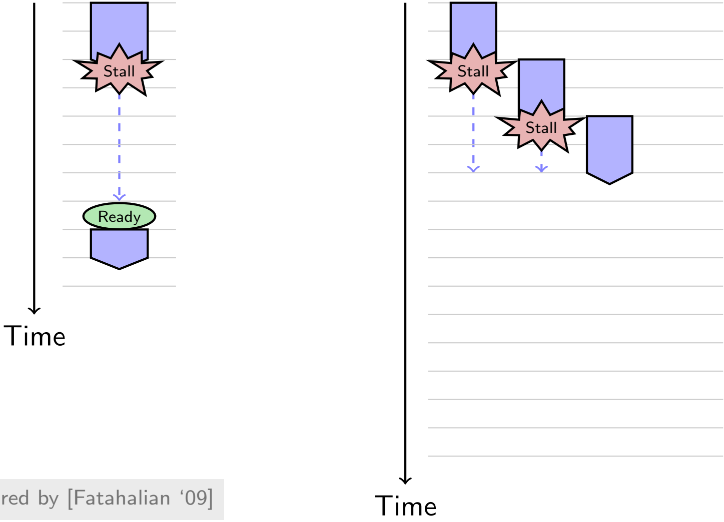
Programming model





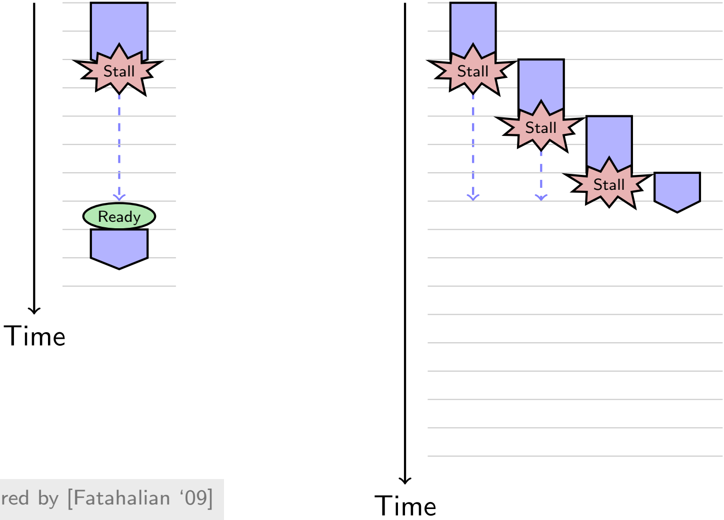
Inspired by [Fatahalian '09]

Programming model



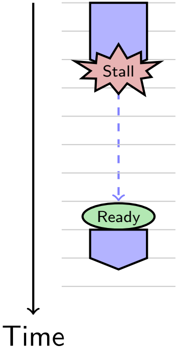
Inspired by [Fatahalian '09]

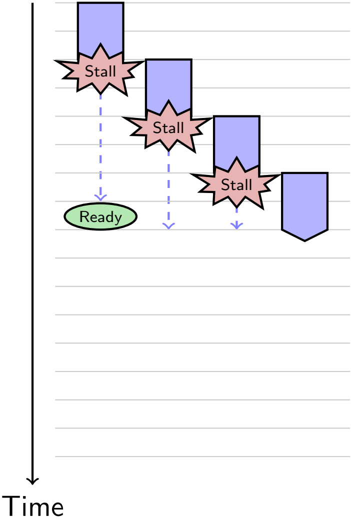
Programming model



Inspired by [Fatahalian '09]

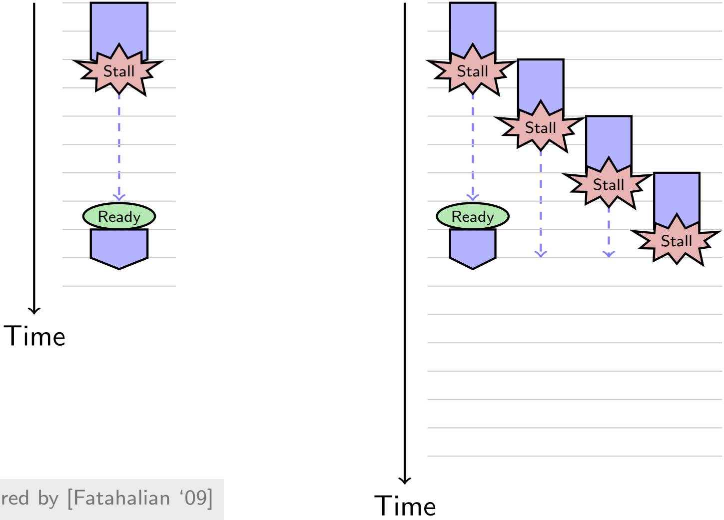
Programming model





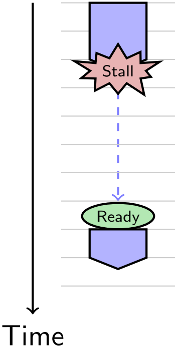
Inspired by [Fatahalian '09]

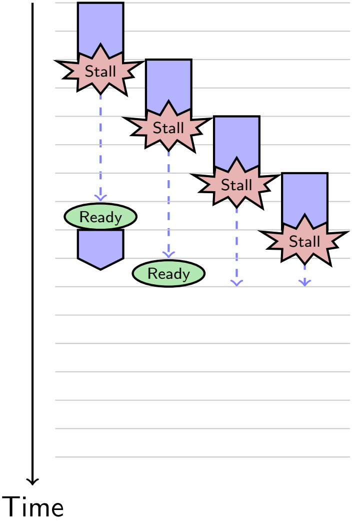
Programming model



Inspired by [Fatahalian '09]

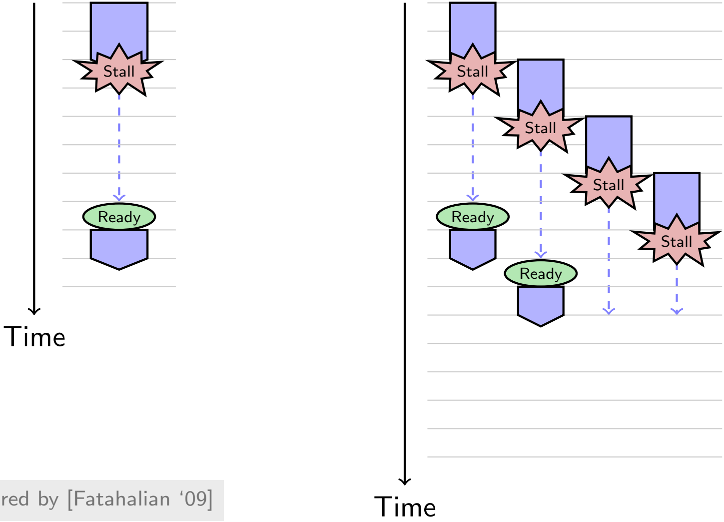
Programming model





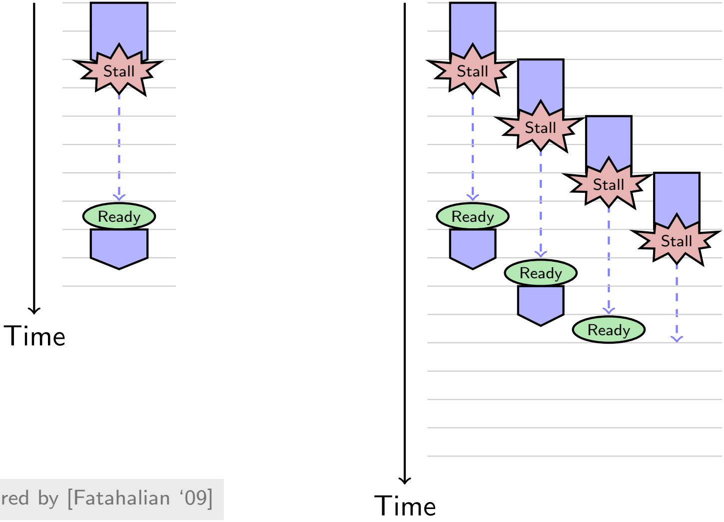
Inspired by [Fatahalian '09]

Programming model



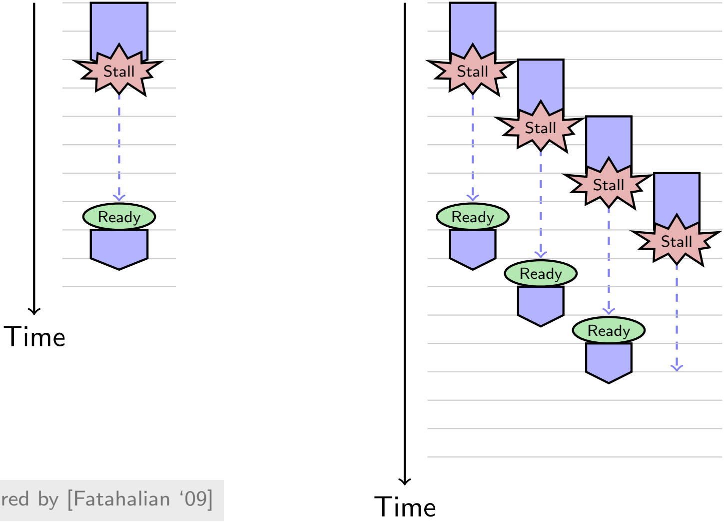
Inspired by [Fatahalian '09]

Programming model



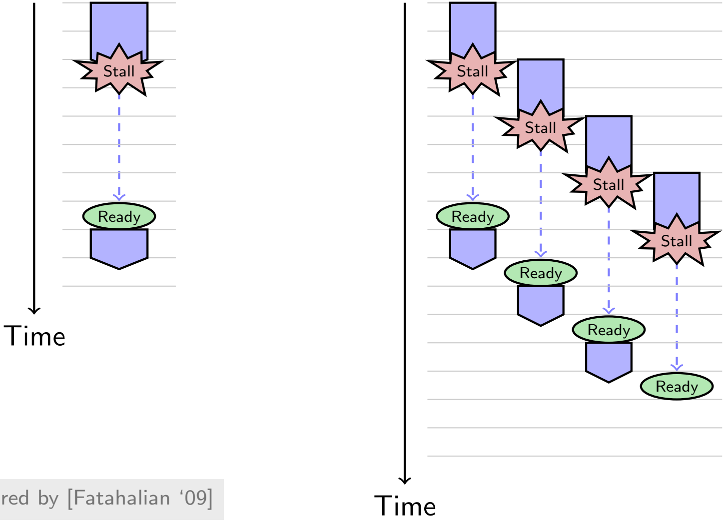
Inspired by [Fatahalian '09]

Programming model



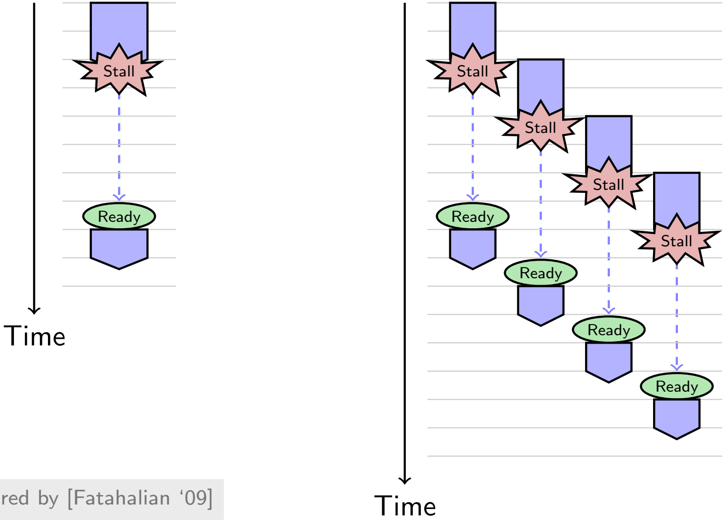
Inspired by [Fatahalian '09]

Programming model



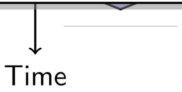
Inspired by [Fatahalian '09]

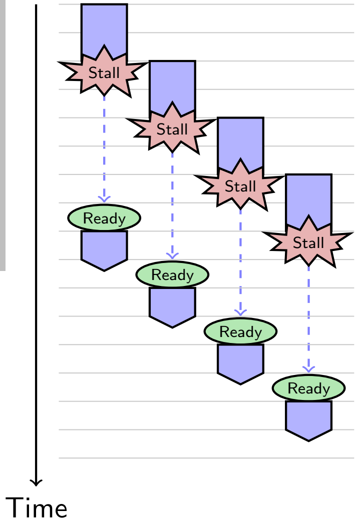
Programming model



Inspired by [Fatahalian '09]

- *Concurrency* hides latency
- Key: No ordering req'mts
    - I.e. no dependencies
- *Issue:* Start-at-the top, end-at-the bottom code
- *Not* parallel programming
    - *But:* parallel execution relies on the same property





Inspired by [Fatahalian '09]

Now add parallelism

Compelled to add concurrency to programming model.

> Might as well *use* it for parallel execution.

Seen: Need concurrency within a *core*.

Add:

- Multiple cores
- Vector Parallelism *within* a core

Now add parallelism

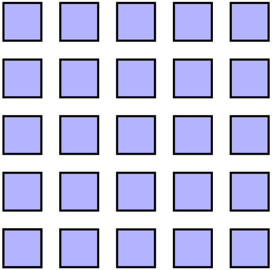
Compelled to add concurrency to programming model.

> Might as well *use* it for parallel execution.

Seen: Need concurrency within a *core*.

Add:

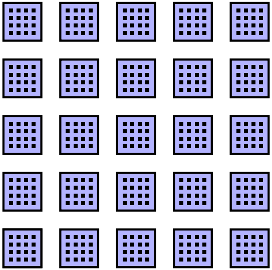
- Multiple cores
- Vector Parallelism *within* a core



Now add parallelism

Compelled to add concurrency to programming model.

> Might as well *use* it for parallel execution.



Seen: Need concurrency within a *core*.

Add:

- Multiple cores
- Vector Parallelism *within* a core

Now add parallelism

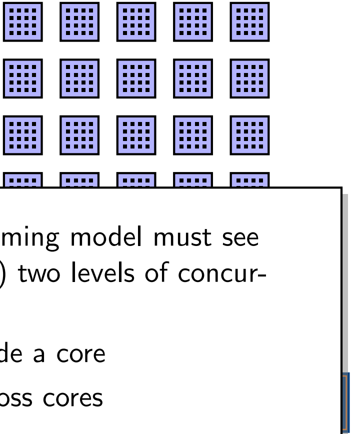
Compelled to add concurrency to programming model.

Might as well *use* it for parallel execution.

Seen: Need concurrency within a *core*.

Add:

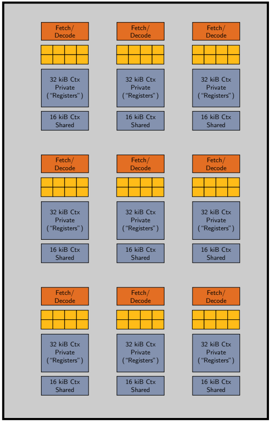
- Multiple cores
- Vector Parallelism *within* a



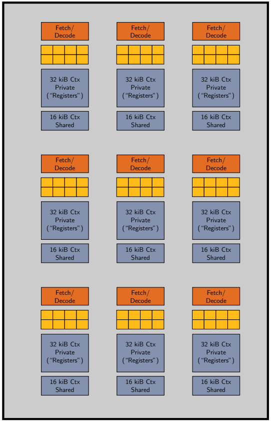
Programming model must see (at least) two levels of concurrency:

- Inside a core
- Across cores

Connection: Hardware ↔ Programming Model



Connection: Hardware ↔ Programming Model

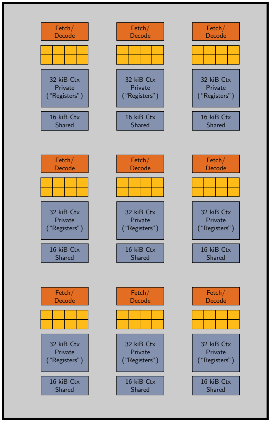


Connection: Hardware ↔ Programming Model

Who cares how many cores?

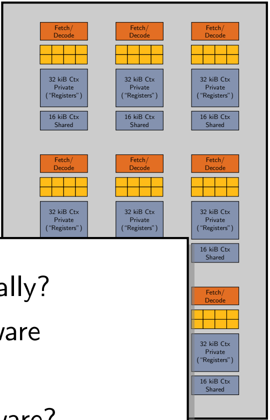
Idea:

- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core



Connection: Hardware ↔ Programming Model

Who cares how many cores?



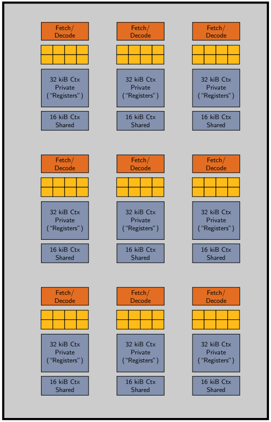
Consider: Which is easy to do automatically?

- Parallel program → sequential hardware

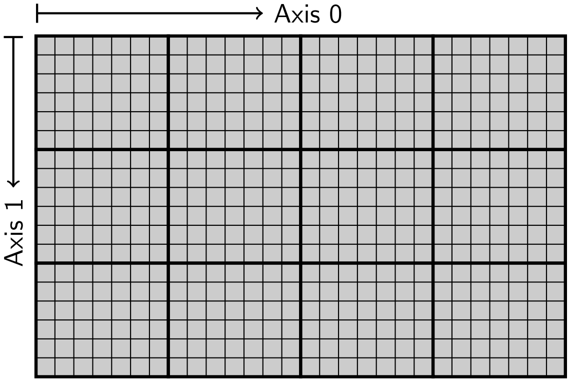
or

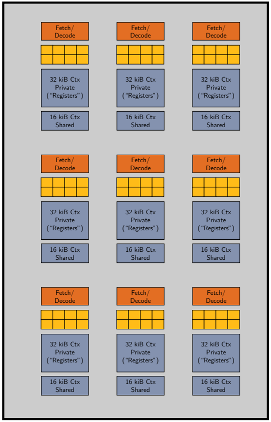
- Sequential program → parallel hardware?

Connection: Hardware ↔ Programming Model

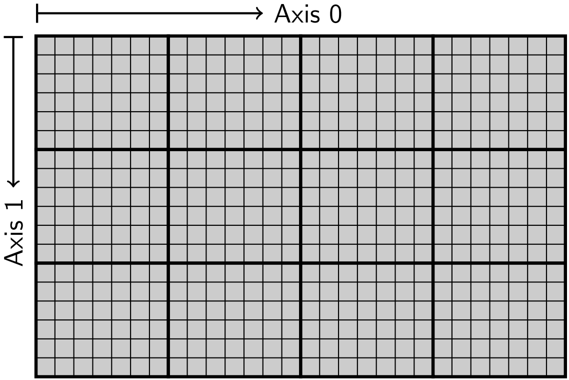


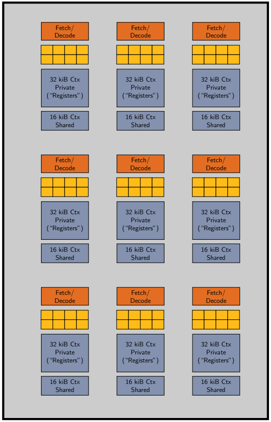
Connection: Hardware ↔ Programming Model





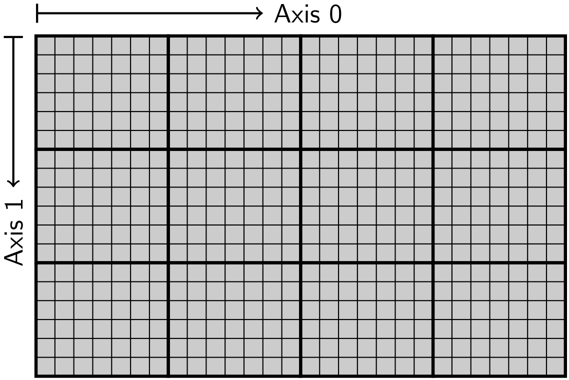
Connection: Hardware ↔ Programming Model



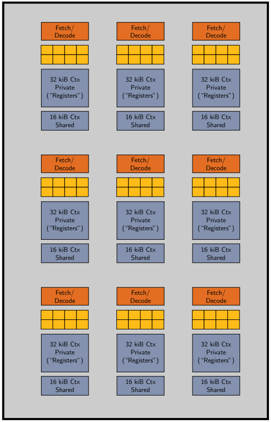


Hardware

Connection: Hardware ↔ Programming Model

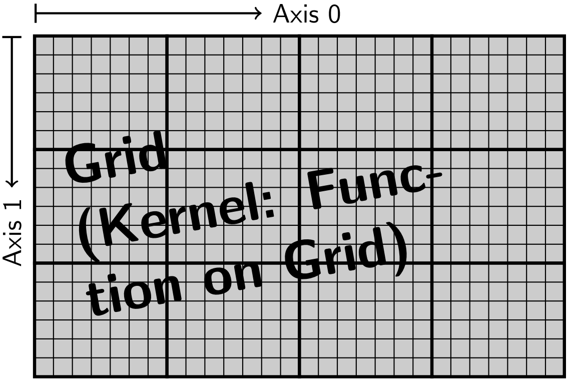


Software representation

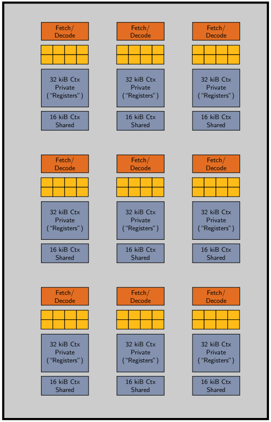


Hardware

Connection: Hardware ↔ Programming Model

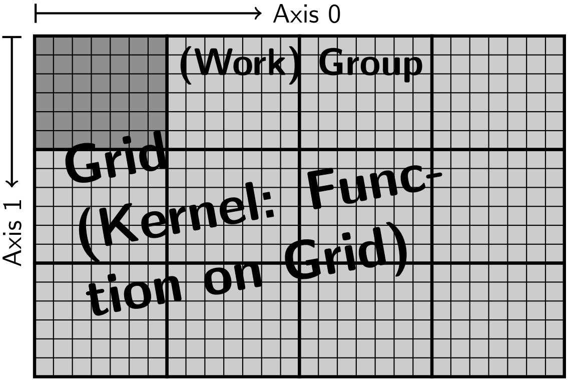


Software representation

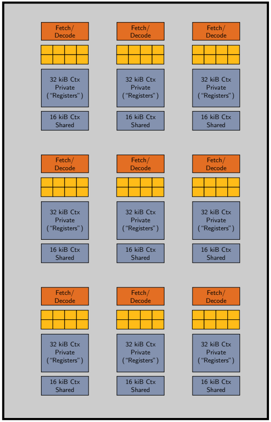


Hardware

Connection: Hardware ↔ Programming Model

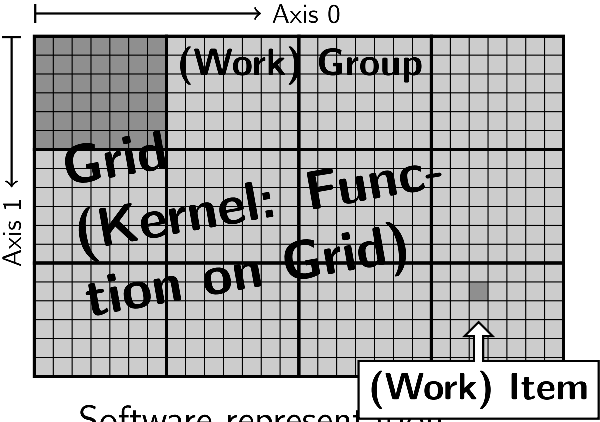


Software representation

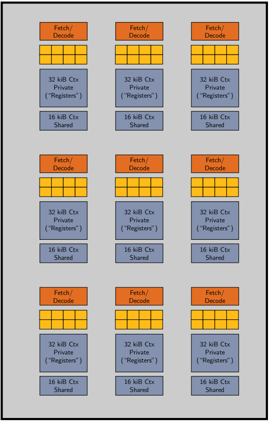


Hardware

Connection: Hardware ↔ Programming Model

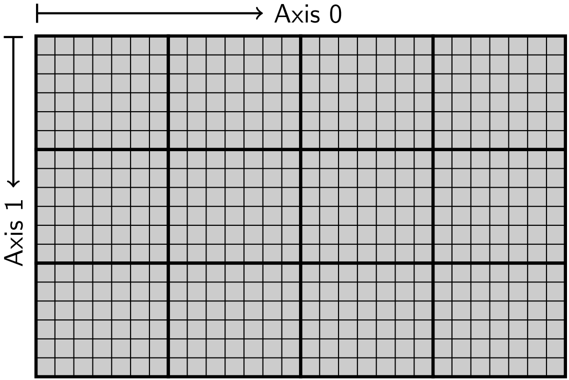


Software representation

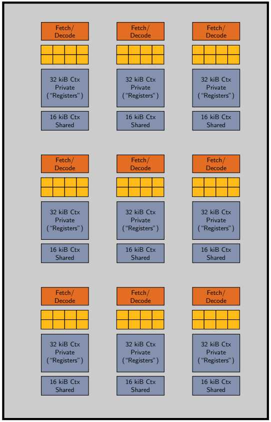


Hardware

Connection: Hardware ↔ Programming Model

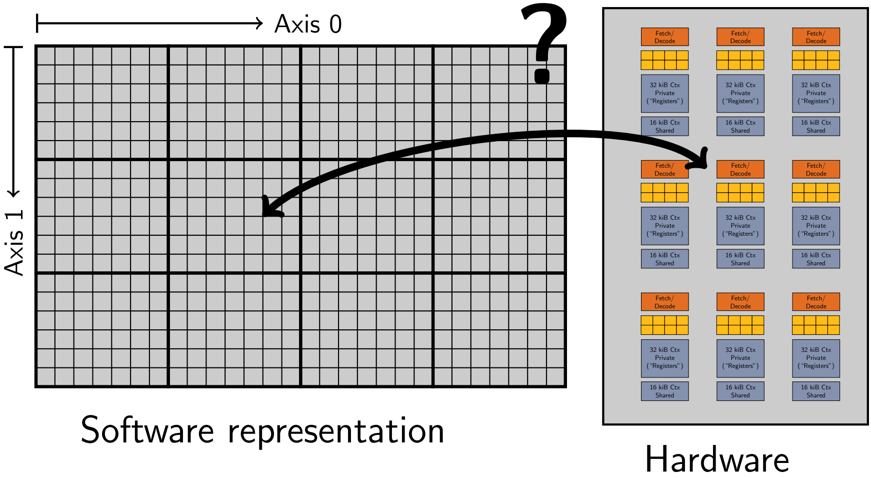


Software representation

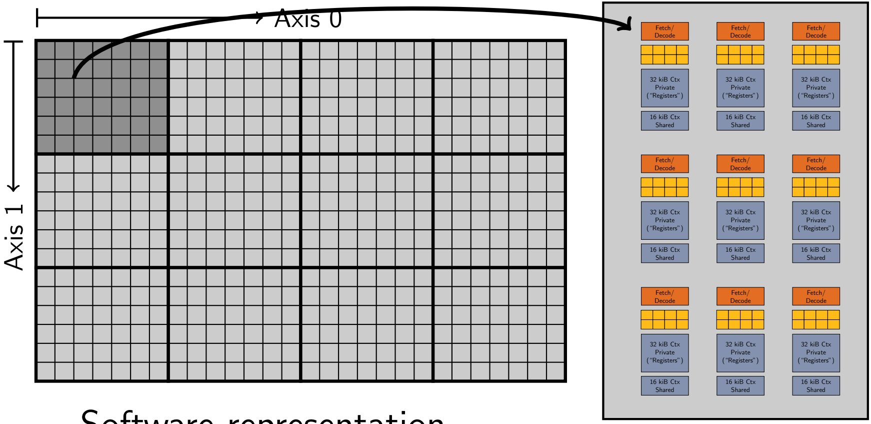


Hardware

Connection: Hardware ↔ Programming Model



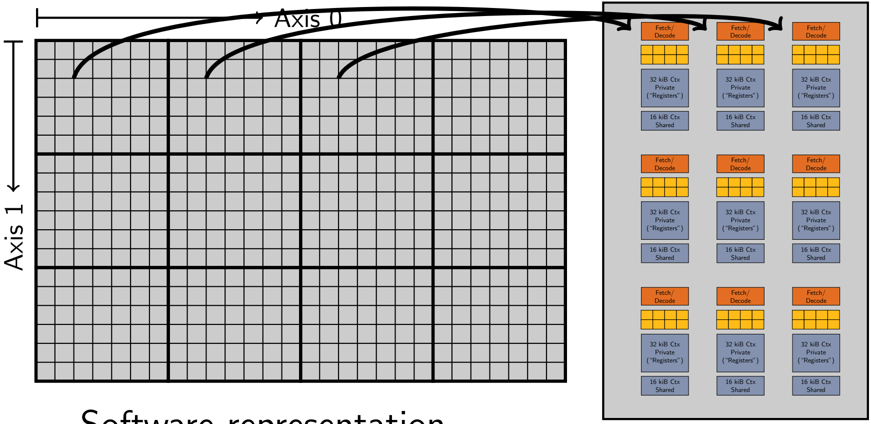
Connection: Hardware ↔ Programming Model



Software representation

Hardware

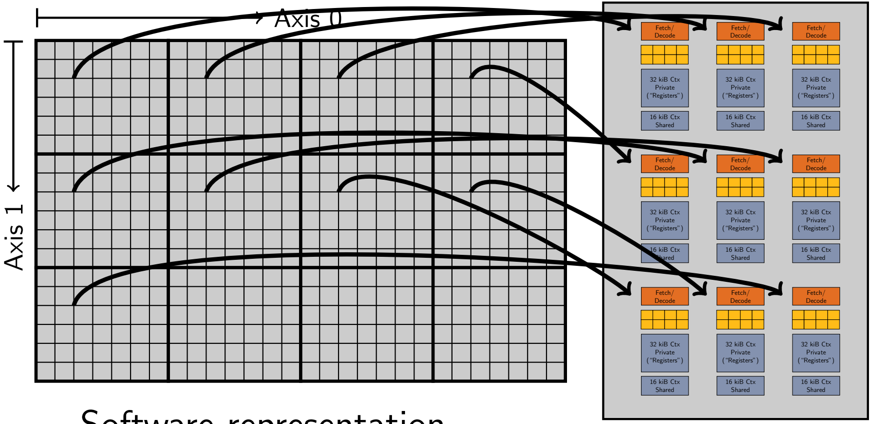
Connection: Hardware ↔ Programming Model



Software representation

Hardware

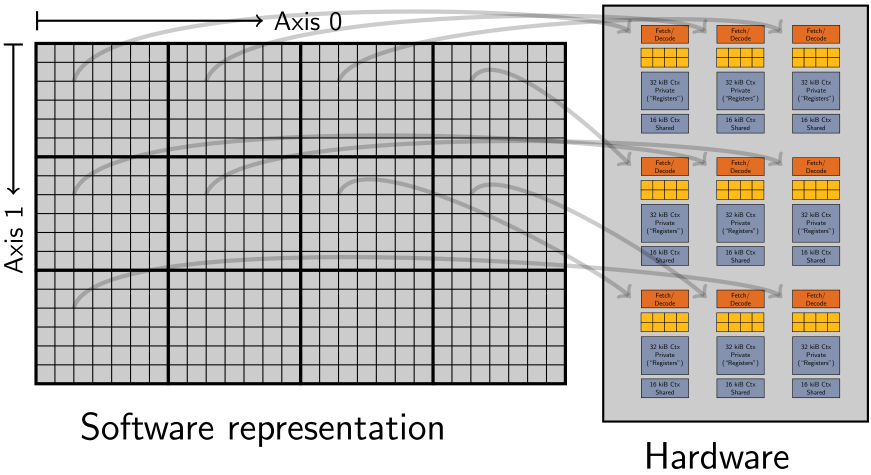
Connection: Hardware ↔ Programming Model



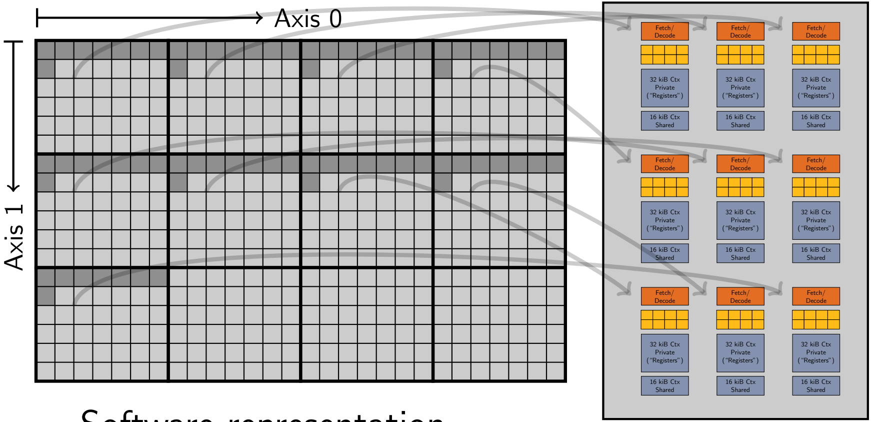
Software representation

Hardware

Connection: Hardware ↔ Programming Model



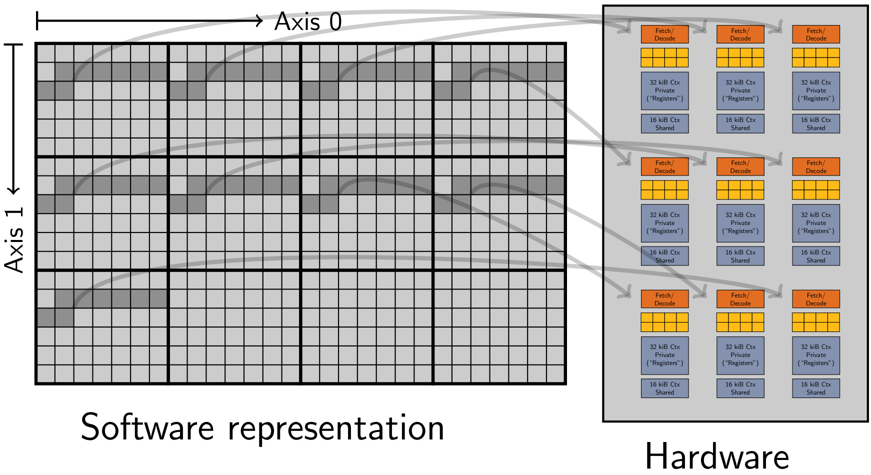
Connection: Hardware ↔ Programming Model



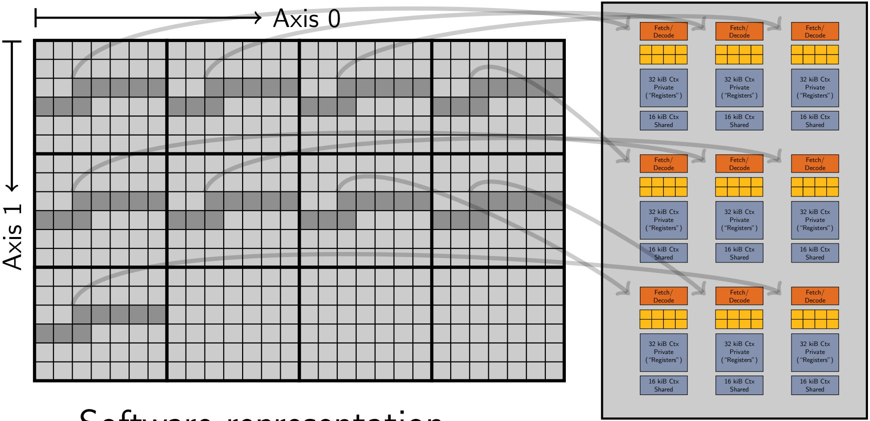
Software representation

Hardware

Connection: Hardware ↔ Programming Model



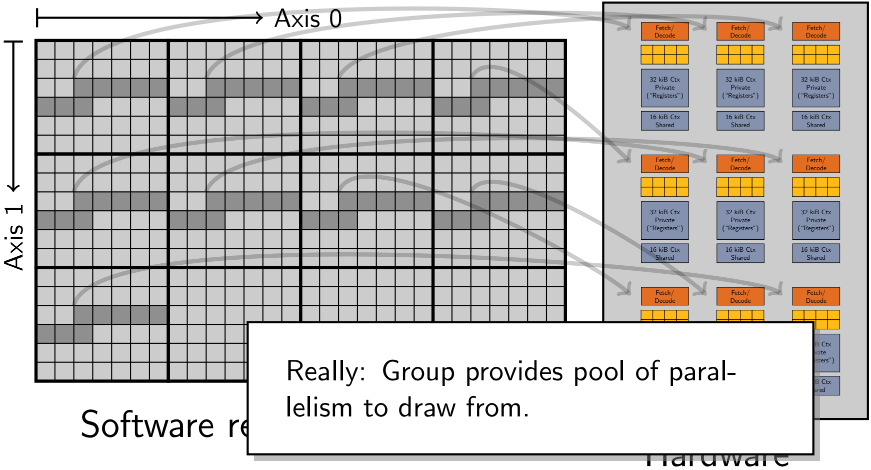
Connection: Hardware ↔ Programming Model



Software representation

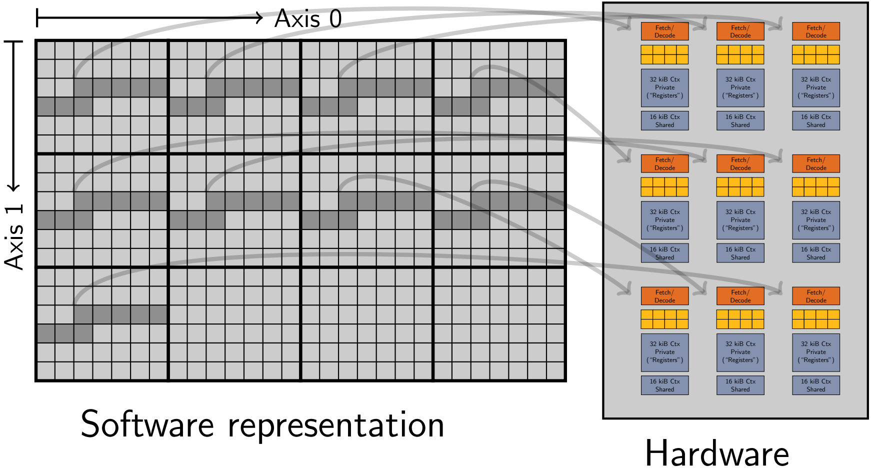
Hardware

Connection: Hardware ↔ Programming Model

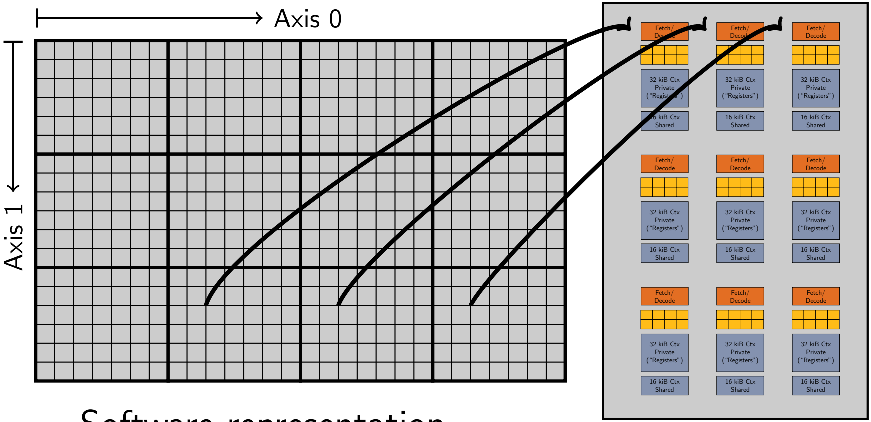


Really: Group provides pool of parallelism to draw from.

Connection: Hardware ↔ Programming Model



Connection: Hardware ↔ Programming Model

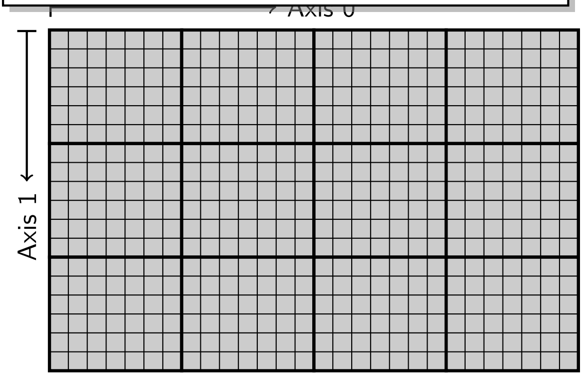


Software representation

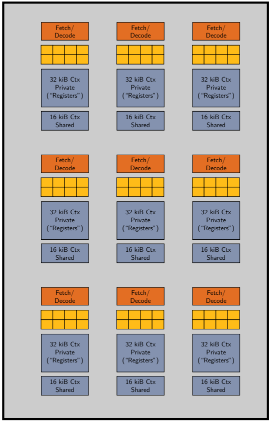
Hardware

Co g Model

Grids can be 1,2,3-dimensional.



Software representation



Hardware

Outline

1 Introduction

2 What's wrong with computers, then?

3 PyOpenCL
  - About PyOpenCL

4 Key Algorithm: Scan

5 Loop Generation

6 Conclusions

DEMO TIME

Outline

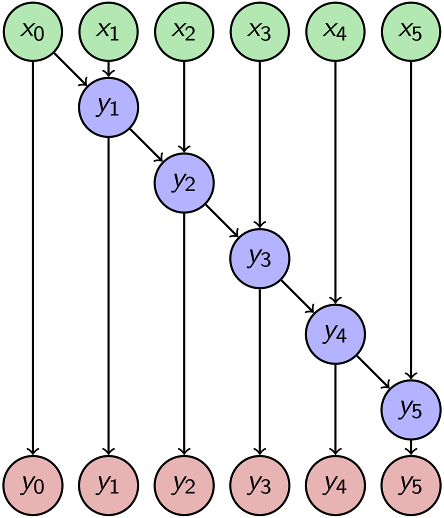
PyOpenCL, PyCUDA: Vital Information

- `http://mathema.tician.de/software/pyopencl` (or `/pycuda`)
- Downloads:
  - Direct: PyOpenCL 210k, PyCUDA 250k
  - Binaries: Win, Debian, Arch, Fedora, Gentoo, . . .
- MIT License
- Compiler Cache, Auto cleanup, Error checking
- Require: numpy, Python 2.4+ (Win/OS X/Linux)
- Community: mailing list, wiki, add-on packages (PyFFT, scikits.cuda, Sailfish, PyWENO, Copperhead. . .)

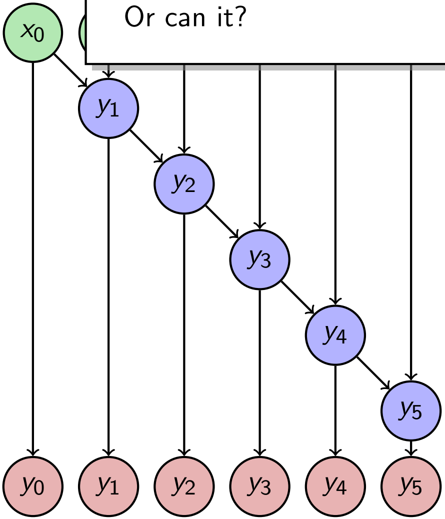
Outline

1. Introduction
2. What's wrong with computers, then?
3. PyOpenCL
4. Key Algorithm: Scan
5. Loop Generation
6. Conclusions

Scan: Graph

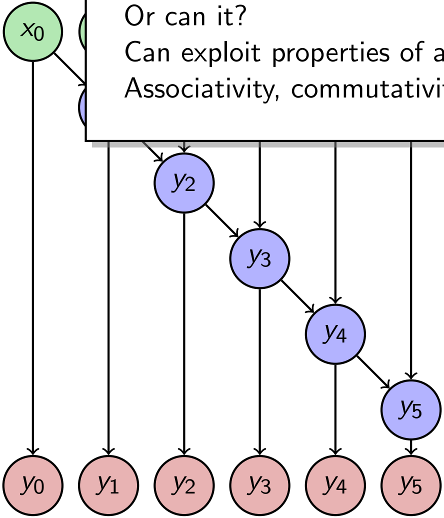


Scan: Graph



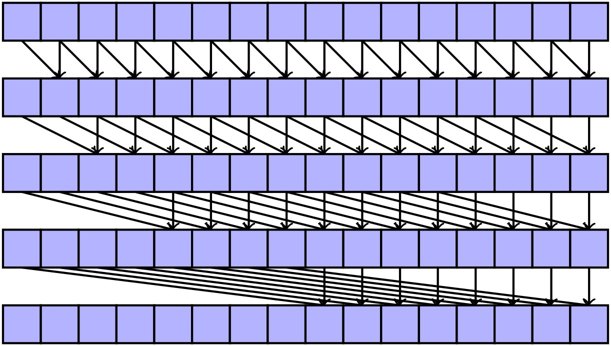
This can't possibly be parallelized.
Or can it?

Scan: Graph



This can't possibly be parallelized.
Or can it?
Can exploit properties of addition:
Associativity, commutativity.

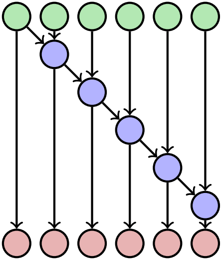
Scan: Implementation



DEMO TIME

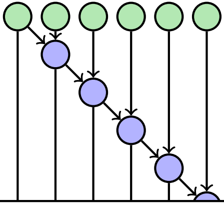
Scan: Features

- “Map” processing on input: $f(x_i)$
    - Also: stencils $f(x_{i-1}, x_i)$
- “Map” processing on output
    - Output stencils
    - Inclusive/Exclusive scan
- Segmented scan
- Works on compound types
- Efficient!



Scan: Features

- "Map" processing on input: $f(x_i)$
  - Also: stencils $f(x_{i-1}, x_i)$
- "Map" processing on output
  - Output stencils
  - Inclusive/Exclusive scan
- Segmented scan
- Works on compound
- Efficient!



Scan: a **fundamental** parallel primitive.

Anything involving index changes/renumbering!
(e.g. sort, filter, . . .)

Scan: More Algorithms

- `copy_if`
- `remove_if`
- `partition`
- `unique`
- `sort` (plain and key-value)
- `build_list_of_lists`

All in `pyopencl`, all built on scan.

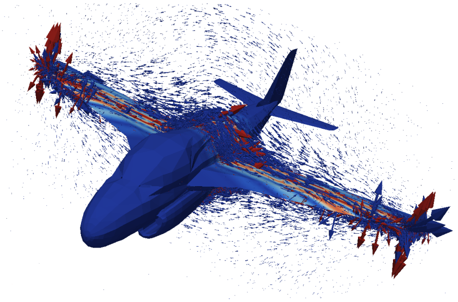
Outline

1. Introduction
2. What's wrong with computers, then?
3. PyOpenCL
4. Key Algorithm: Scan
5. Loop Generation
   - Loo.py
6. Conclusions

Outline

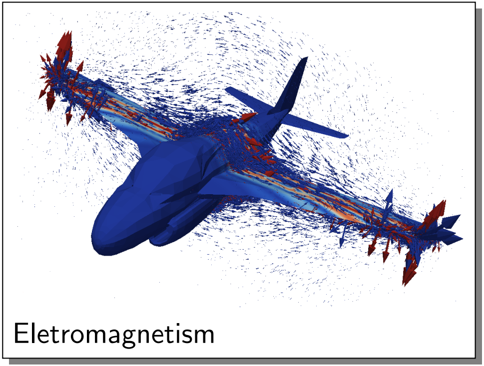
1. Introduction
2. What's wrong with computers, then?
3. PyOpenCL
4. Key Algorithm: Scan
5. Loop Generation
    - Loo.py
6. Conclusions

GPU DG Showcase

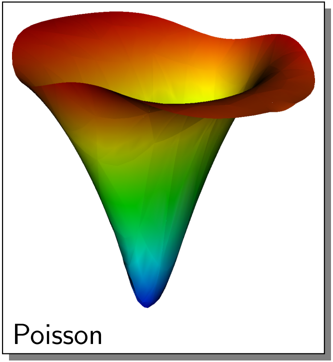


Eletromagnetism

GPU DG Showcase

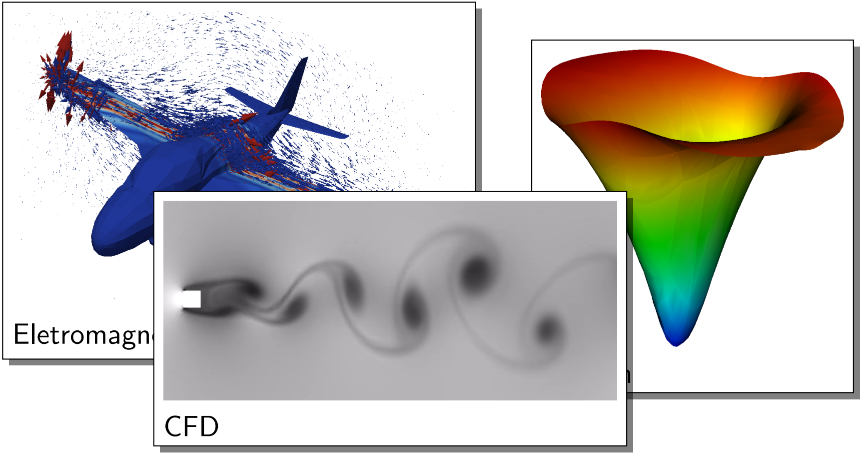


Eletromagnetism



Poisson

GPU DG Showcase

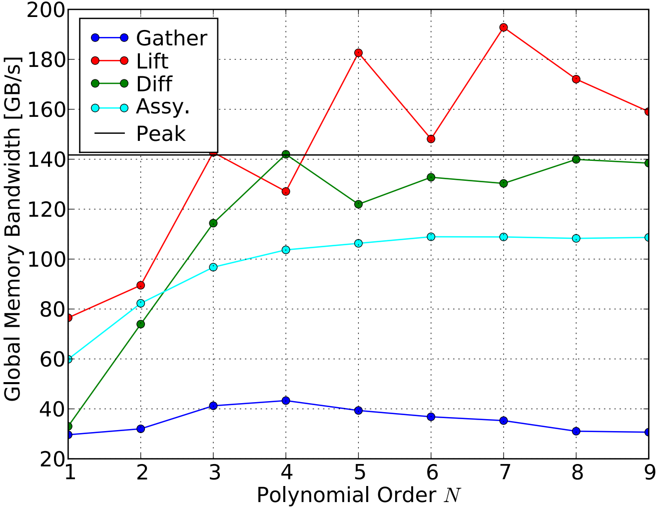


Eletromagn

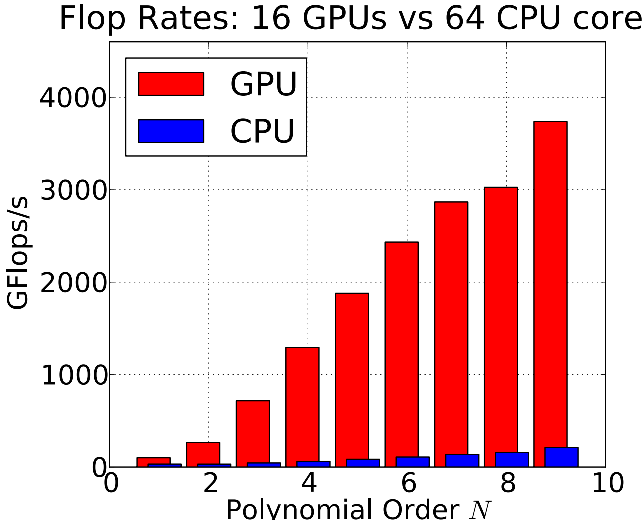
CFD

n

Memory Bandwidth on a GTX 280



Multiple GPUs via MPI: 16 GPUs vs. 64 CPUs



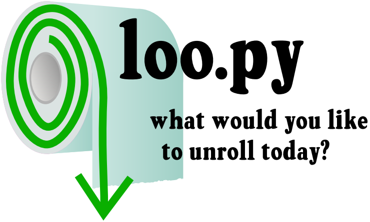
Setting the Stage

Idea:

- Start with math-y statement of the operation
- "Push a few buttons" (transformations) to optimize for the target device
- Strongly separate these two parts

Philosophy:

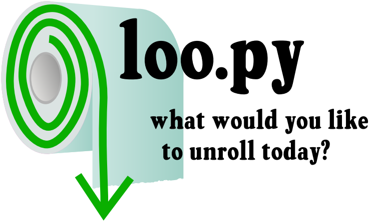
- Avoid "intelligence"
- User can assume partial responsibility for correctness
- Embedding in Python provides generation/transform flexibility



Setting the Stage

Idea:

- Start with math-y statement of the operation
- "Push a few buttons" (transformations) to optimize for the target device
- Strongly separate these two parts



Philosophy:

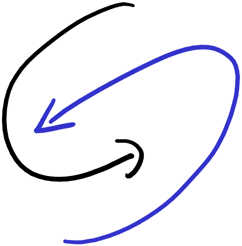
- Avoid "intelligence"
- User can assu
  responsibility
- Embedding i
  generation/tr

Loopy is infrastructure.

Computational software builds *on top* of loopy.

DEMO TIME

Code Transforms in Loopy



- Unroll
- Prefetch
- Precompute
- Tile
- Reorder loops
- Fix constants
- Parallelize
- Affine map loop domains
- Texture-based data access
- SoA ↔ AoS

New Code Transforms in Loopy for 2015

- Kernel **Fusion**
- Computation of **Intermediate Results**
- SIMD **Vectorization**
- Naming of array axes
- Aliasing of temporaries
- Temporary result buffering
- Distributive law
- Arbitrary nesting of **Data Layouts**
- Realization of **ILP**

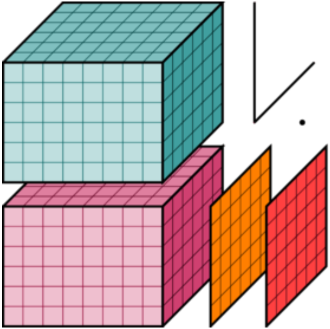
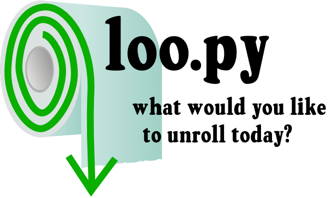


Image credit: Xray/Stephan Hoyer

Loopy: Reachable Performance

		Intel	AMD	Nvidia
`saxpy`	[GBytes/s]	18.6	231.0	232.1
`sgemm`	[GFlops/s]	12.3	492.3	369.4
3D Coulomb pot.	[M Pairs/s]	231	10949	9985
dG FEM volume	[GFlops/s]	77.4	1251	351
dG FEM surface	[GFlops/s]	25.9	527	214

Features



- A-priori bounds checking
- Generate a sequential version of the code
- Automatic Benchmarking
- Automatic Testing
  - . . . against sequential version
  - . . . which is easier to verify
- Data layout transformation
- Fortran program input

```
subroutine dgemm(m,n,l,alpha,a,b,c)
 implicit none
 real*8 temp, a(m,l),b(l,n),c(m,n), alpha
 integer m,n,k,i,j,l

 do j = 1,n
   do k = 1,l
     do i = 1,m
       c(i,j) = c(i,j) + alpha*b(k,j)*a(i,k)
     end do
   end do
 end do
end subroutine

!$loopy begin
! dgemm, = lp.parse_fortran(SOURCE, FILENAME)
! dgemm = lp.split_iname(dgemm, ”i”, 16,
!         outer_tag=”g.0”, inner_tag=”l.1”)
! dgemm = lp.split_iname(dgemm, ”j”, 8,
!         outer_tag=”g.1”, inner_tag=”l.0”)
! dgemm = lp.split_iname(dgemm, ”k”, 32)
! RESULT = [dgemm]
!$loopy end
```

NUMA Differentiation: Fortran view

```
do e = 1, elements

  do k = 1, Nq
    do j = 1, Nq
      do i = 1, Nq
        do n = 1,Nq

!$loopy begin tagged: local_prep
          U  = Q(n, j, k, 1, e)
          V  = Q(n, j, k, 2, e)
          W  = Q(n, j, k, 3, e)
          R  = Q(n, j, k, 5, e)
          T  = Q(n, j, k, 6, e)
          Qa = Q(n, j, k, 7, e)
          Qw = Q(n, j, k, 8, e)

          Jrx  = volumeGeometricFactors(n, j, k, 1, e)
          Jry  = volumeGeometricFactors(n, j, k, 2, e)
          Jrz  = volumeGeometricFactors(n, j, k, 3, e)

          Jinv = volumeGeometricFactors(i, j, k, 10, e)

          P = p_p0*(p_R*T/p_p0) ** p_Gamma
          UdotGradR = (Jrx*U + Jry*V + Jrz*W)/R
!$loopy end tagged: local_prep
```

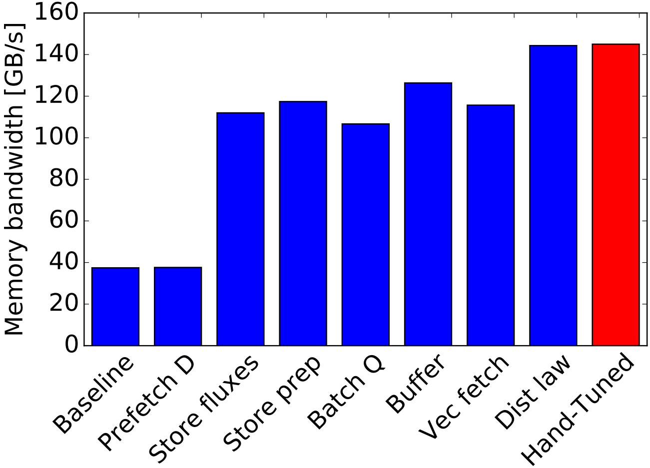
```
          JinvD = Jinv*D(i,n)

!$loopy begin tagged: compute_fluxes
          Uflux  = U*UdotGradR + Jrx*P
          Vflux  = V*UdotGradR + Jry*P
          Wflux  = W*UdotGradR + Jrz*P
          Rflux  = R*UdotGradR
          Tflux  = T*UdotGradR

          Qaflux = Qa*UdotGradR
          Qwflux = Qw*UdotGradR
!$loopy end tagged: compute_fluxes

          rhsQ(i, j, k, 1, e) = rhsQ(i, j, k, 1, e) - JinvD*Uflux
          rhsQ(i, j, k, 2, e) = rhsQ(i, j, k, 2, e) - JinvD*Vflux
          rhsQ(i, j, k, 3, e) = rhsQ(i, j, k, 3, e) - JinvD*Wflux

          rhsQ(i, j, k, 5, e) = rhsQ(i, j, k, 5, e) - JinvD*Rflux
          rhsQ(i, j, k, 6, e) = rhsQ(i, j, k, 6, e) - JinvD*Tflux
          rhsQ(i, j, k, 7, e) = rhsQ(i, j, k, 7, e) - JinvD*Qaflux
          rhsQ(i, j, k, 8, e) = rhsQ(i, j, k, 8, e) - JinvD*Qwflux
        end do
      end do
    end do
  end do
end do
```

NUMA Differentiation: Kernel view

(100 more lines of this. . .)

```
barrier (CLK_LOCAL_MEM_FENCE) /* for temp_storage_0 (insn52 conflicts with Qwflux_s_subst (via 'tmpgrp_flux_store_13')) */;
flux_store_13 [ lid (0) + 8 * lid (1)] = Qw_r_subst_0 * UdotGradS_subst_0_0;
flux_store_12 [8 * lid (1) + lid (0)] = Qw_r_subst_0 * UdotGradR_subst_0_0;
barrier (CLK_LOCAL_MEM_FENCE) /* for flux_store_12 (insn27 depends on Qwflux_r_subst) */;
for (int n_Qwflux = 0; n_Qwflux <= 7; ++n_Qwflux)
{
  rhsQ_buf[1].s3 = rhsQ_buf[1].s3 + flux_store_13 [ lid (0) + 8 * n_Qwflux] * D_fetch[ lid (1) + 8 * n_Qwflux];
  rhsQ_buf[1].s3 = rhsQ_buf[1].s3 + flux_store_12 [8 * lid (1) + n_Qwflux] * D_fetch[ lid (0) + 8 * n_Qwflux];
}
barrier (CLK_LOCAL_MEM_FENCE) /* for temp_storage_0 (insn55 conflicts with Qaflux_s_subst (via 'tmpgrp_flux_store_11')) */;
flux_store_11 [ lid (0) + 8 * lid (1)] = Qa_r_subst_0 * UdotGradS_subst_0_0;
flux_store_10 [8 * lid (1) + lid (0)] = Qa_r_subst_0 * UdotGradR_subst_0_0;
barrier (CLK_LOCAL_MEM_FENCE) /* for flux_store_10 (insn26 depends on Qaflux_r_subst) */;
for (int n_Qaflux = 0; n_Qaflux <= 7; ++n_Qaflux)
{
  rhsQ_buf[1].s2 = rhsQ_buf[1].s2 + flux_store_11 [ lid (0) + 8 * n_Qaflux] * D_fetch[ lid (1) + 8 * n_Qaflux];
  rhsQ_buf[1].s2 = rhsQ_buf[1].s2 + flux_store_10 [8 * lid (1) + n_Qaflux] * D_fetch[ lid (0) + 8 * n_Qaflux];
}
rhsQ[lid (0) + 8 * lid (1) + 64 * k + (2048 * elements / 4) * 0 + 512 * gid(0)] =
         volumeGeometricFactors[lid(0) + 8 * lid(1) + 64 * k + 4608 + 5632 * gid(0)] * -1.0 * rhsQ_buf[0];
rhsQ[lid (0) + 8 * lid (1) + 64 * k + (2048 * elements / 4) * 1 + 512 * gid(0)] =
         volumeGeometricFactors[lid(0) + 8 * lid(1) + 64 * k + 4608 + 5632 * gid(0)] * -1.0 * rhsQ_buf[1];
```

Applying Optimizations Step-By-Step



Device peak: 220 GB/s

Outline

1. Introduction
2. What's wrong with computers, then?
3. PyOpenCL
4. Key Algorithm: Scan
5. Loop Generation
6. Conclusions

Conclusions

- Exciting time to be in HPC
    - Many fast and (fortunately!) somewhat coherent developments
    - Great opportunities!
- GPUs and scripting work surprisingly well together
    - Enable Run-Time Code Generation
- Hopes for loopy:
    - General enough to be broadly useful
    - Possible future addition: distributed memory

`http://www.cs.illinois.edu/~andreask/`

Outline

- PyCUDA

Whetting your appetite

```
import pycuda.driver as cuda
import pycuda.autoinit, pycuda.compiler
import numpy

a = numpy.random.randn(4,4).astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)
```

[This is `examples/demo.py` in the PyCUDA distribution.]

Whetting your appetite

```
mod = pycuda.compiler.SourceModule("""
     __global__ void twice(float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
    """)

func = mod.get_function("twice")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

Whetting your appetite

```python
mod = pycuda.compiler.SourceModule("""
    __global__ void twice(float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
    """)

func = mod.get_function("twice")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

Compute kernel

Whetting your appetite, Part II

Did somebody say “Abstraction is good”?

Whetting your appetite, Part II

```
import numpy
import pycuda.autoinit
import pycuda.gpuarray as gpuarray

a_gpu = gpuarray.to_gpu(
    numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```