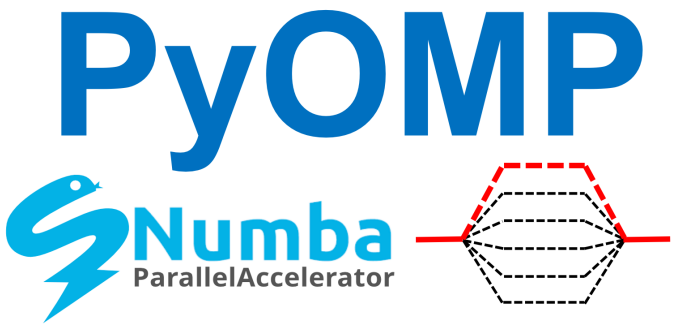


PyOMP: Writing HPC Code with Python

Tim Mattson (University of Bristol and Merly.ai … but mostly retired)

tim@timmattson.com

Acknowledgement: The rest of the PyOMP team ….

Giorgis Georgakoudis (LLNL), Todd Anderson (bodo.ai), and Stuart Archibald (anaconda)

PyOMP: Writing HPC Code with Python

Tim Mattson (University of Bristol and Merly.ai … but mostly retired)

tim@timmattson.com

Acknowledgement: The rest of the PyOMP team ….

Giorgis Georgakoudis (LLNL), Todd Anderson (bodo.ai), and Stuart Archibald (anaconda)

I am best known for my work on OpenMP

I am one of the last members of the original OpenMP team (1996) still active with the language.

```
C$OMP FLUSH
#pragma omp critical
#pragma omp single
C$OMP THREADPRIVATE(/ABC/)
C$OMP ATOMIC
CALL OMP_SET_NUM_THREADS(10)
```

OpenMP: An API for Writing Parallel Applications

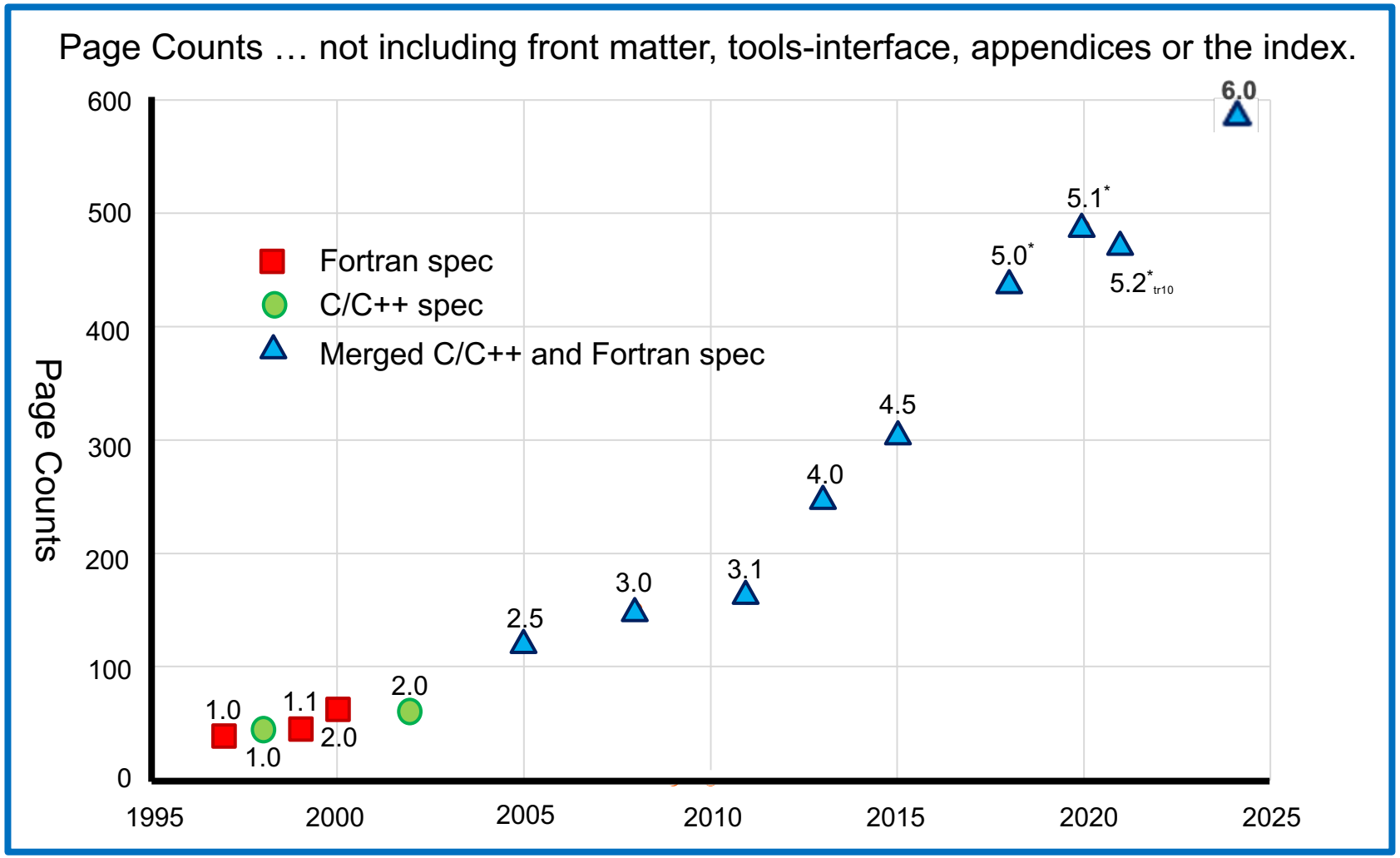
- A set of compiler directives and library routines for parallel application programmers
- Originally … Greatly simplifies writing multithreaded programs in Fortran, C and C++
- Later versions … supports non-uniform memories, vectorization and GPU programming

```
#pragma omp parallel for private(A, B)
C$OMP PARALLEL  REDUCTION (+: A, B)
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
#pragma omp atomic seq_cst
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

The Growth of Complexity in OpenMP

Our goal in 1997 … A simple interface for application programmers



The OpenMP specification is so long and complex that few (if any) humans understand the full document

How much are different parallel programming models used?

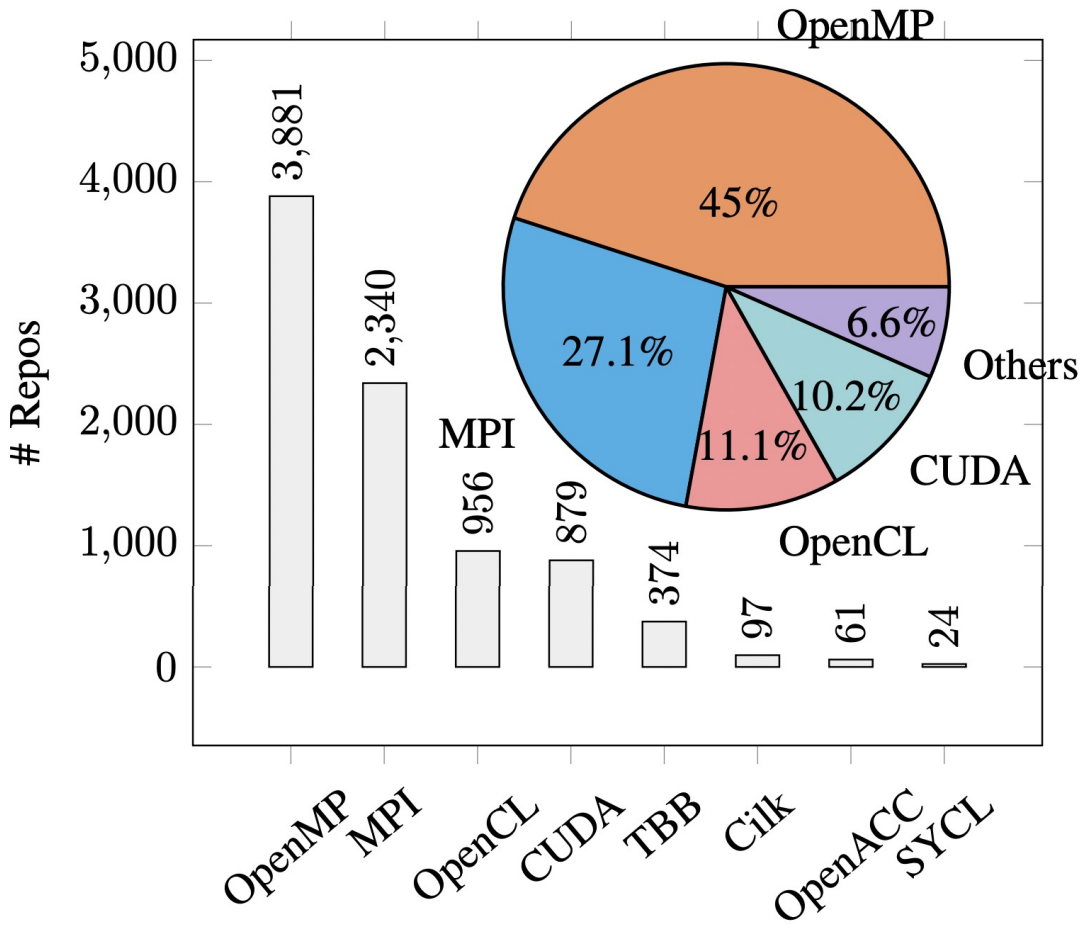
HPCorpus data set for training LLM models for parallel programming

Scan all C, C++ and Fortran codes from github with "last updated" dates between 2012 and mid 2023

	Repos	Size(GB)	Files (#)	Functions (#)
C	144,522	46.23	4,552,736	87,817,591
C++	150,481	26.16	4,735,196	68,233,984
Fortran	3,683	0.68	138,552	359,272

Quantifying OpenMP: Statistical insights into usage and adoption, Tal Kadosh, et al., HPEC'2023, https://arxiv.org/abs/2308.08002

HPCorpus Let us assess usage of HPC programming models … **OpenMP is number 1**!!!



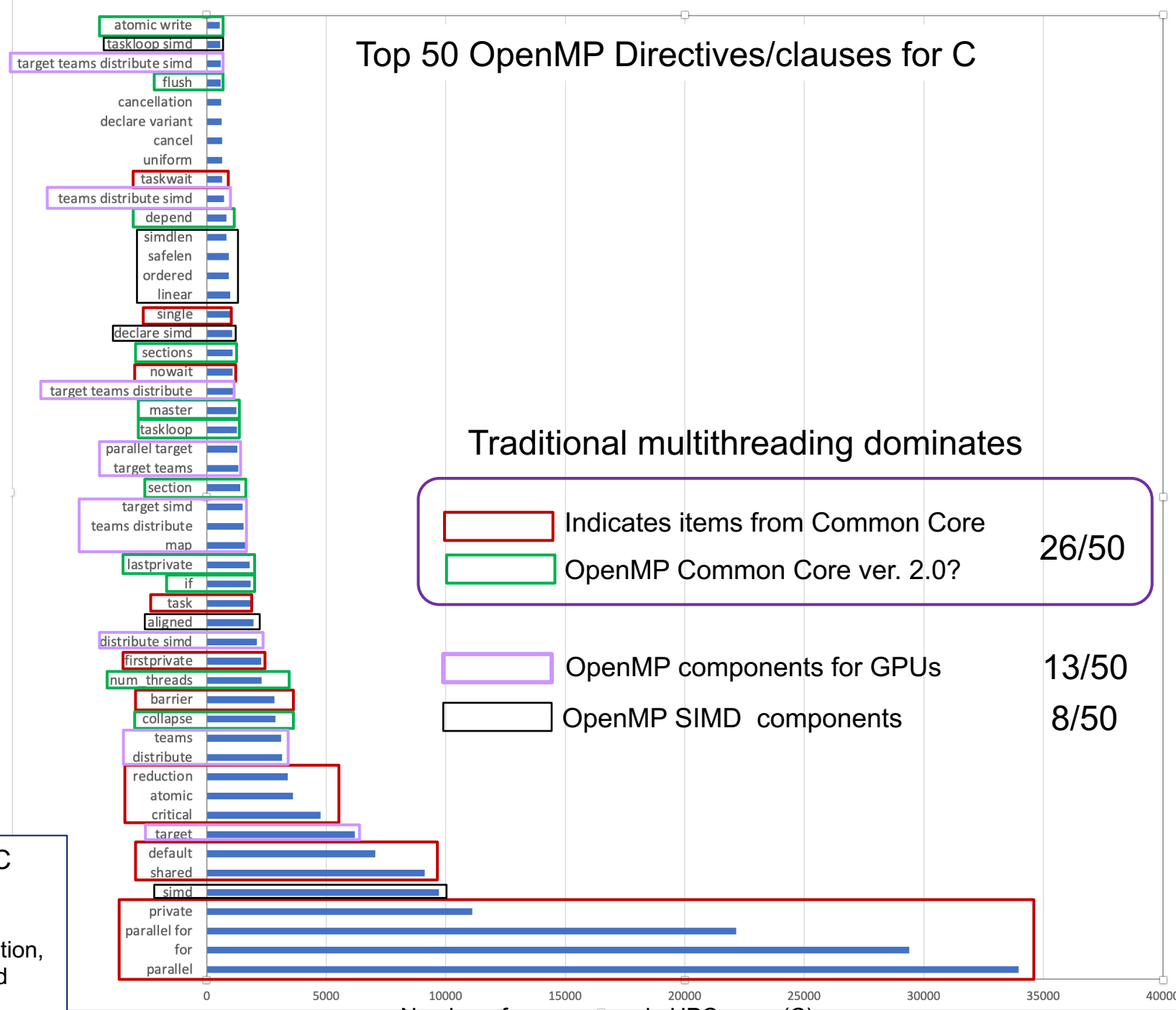
Aggregate numbers over all repositories from 2013 to 2023

Note: since we did not collect files with .cu or .cuf suffixes, we may have undercounted CUDA usage in HPCorpus.

What are people actually using from OpenMP

With the HPCorpus* dataset, we finally have hard-data to analyze what “should” be in the common core.

This data was constructed by summing up counts for different directives and clauses across time from 2013 to the middle of 2023.

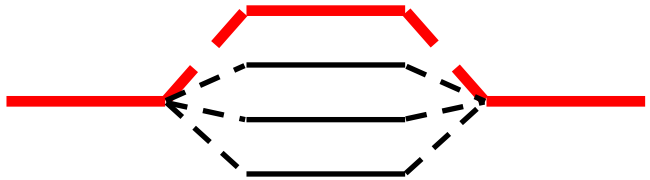


HPCorpus … a data set created by scraping ”all” HPC codes from github written in C, C++ and Fortran.

Quantifying OpenMP: Statistical Insights into Usage and Adoption, Tal Kadosh, Niranjan Hasabnis, Tim Mattson, Yuval Pinter, and Gal Oren, IEEE HPEC 2023

OpenMP CPU Patterns: Map work onto a team of threads

Fork-Join execution: Initial Thread (in red) forks a **team of threads** to execute "in parallel". They join together when done and the initial thread continues execution.



#pragma omp parallel

Parallel Loops

Serial code

```
for(i=0;i<N;i++)
  A[i] = work();
```

Map Loop iterations to threads

Parallel code

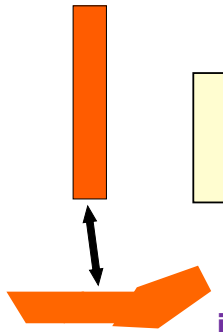
```
#pragma omp for
for(i=0;i<N;i++)
  A[i] = work();
```

Common clauses on the **for** construct

reduction(op:var_list)
private(var_list)

Single Program Multiple Data (SPMD)

Serial code



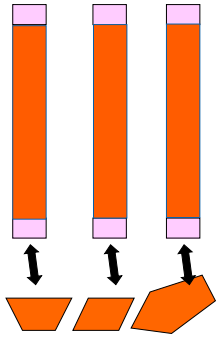
Data

Threads run the same code

Thread ID and count to mange work.

```
int ID = omp_get_thread_num();
Int count  = omp_get_num_threads();
```

Parallel code

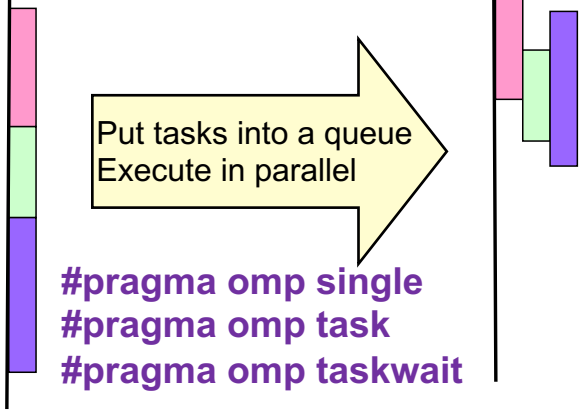


Decomposed Data

Task Parallelism

Serial

Parallel

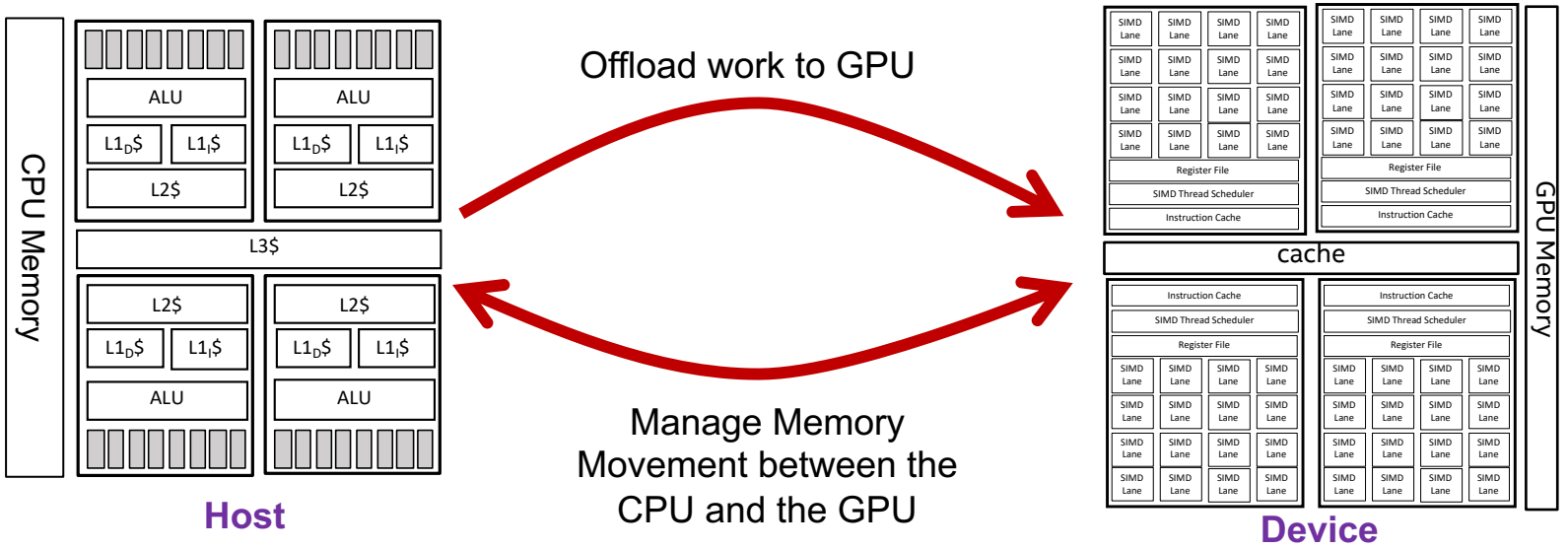


Put tasks into a queue
Execute in parallel

```
#pragma omp single
#pragma omp task
#pragma omp taskwait
```

OpenMP GPU Patterns: Offload work from the CPU to the GPU

A four core CPU and a SIMD width of 8/core

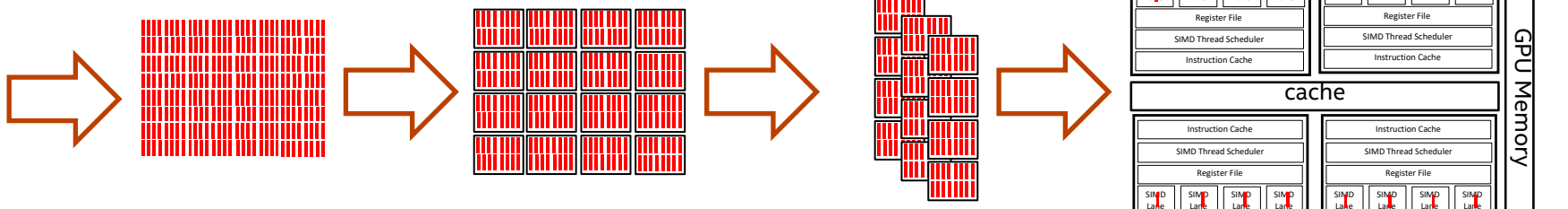


A GPU with 4 compute units and a warp size of 16.

A compute unit is analogous to a CPU core (not the SIMD Lanes … as Nvidia Marketing claims).

```
#pragma omp target teams loop collapse(2)
for (i=0; i<N; i++)
   for(j=0; j<N; j++)
      for(k=0; k<N; k++)
         C[i][j] += A[B[i][k] * B[k][j];
```

Parallel loops define an index-space. Individual loop iterations define units of work (work-items)



work-items and data mapped onto index space

Work-items organized into work-groups

Work-groups enqueued for scheduling.

Work-items from a work-group execute toether .

I've been active in HPC since 1980.
That makes me an official HPC old-timer

This is my first parallel computer which I used as a Post Doc at Caltech in 1985.
We wrote our code using Fortran'77 and a message passing library.

The **Caltech Cosmic Cube** developed by Charles Seitz and Geoffrey Fox in1981

- 64 Intel 8086/8087 processors
- 128kB of memory per processor
- 6-dimensional hypercube network

The cosmic cube, Charles Seitz, Comms of the ACM, Vol 28, number 1 January 1985, p. 22

What HPC old-timers think of Python?

(from the paper, There's plenty of room at the top. Leiserson et. al. Science vol. 368, June 2020).

They used matrix multiplication to explore the connection between software and performance

```
for I in range(4096):
   for j in range(4096):
      for k in range (4096):
            C[i][j] += A[i][k]*B[k][j]
```

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60 GB RAM

What HPC old-timers think of Python

(from the paper, There's plenty of room at the top. Leiserson et. al. Science vol. 368, June 2020).

They used matrix multiplication to explore the connection between software and performance

```
for I in range(4096):
   for j in range(4096):
      for k in range (4096):
            C[i][j] += A[i][k]*B[k][j]
```

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

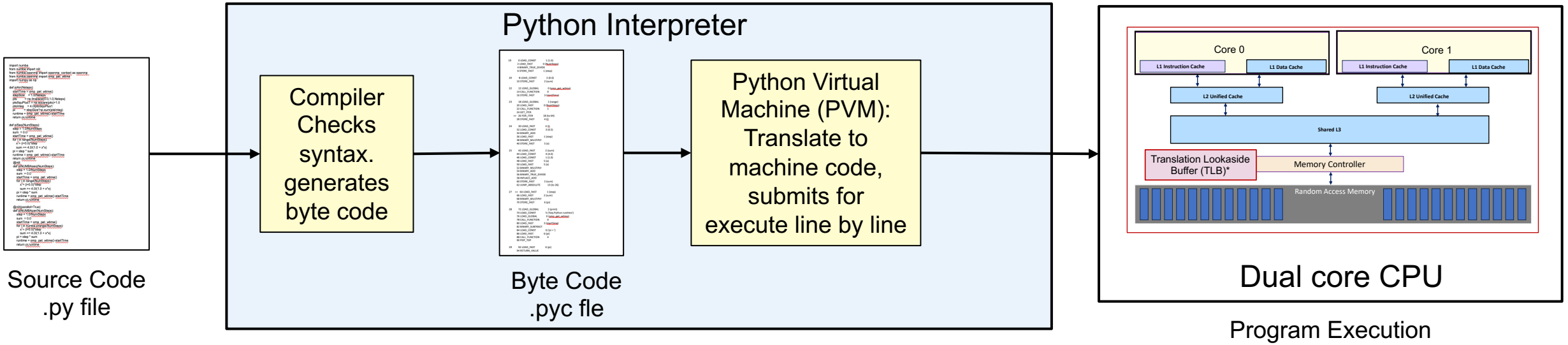
Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

Python performance is a joke. No serious HPC programmer would **EVER** use Python

Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60 GB RAM

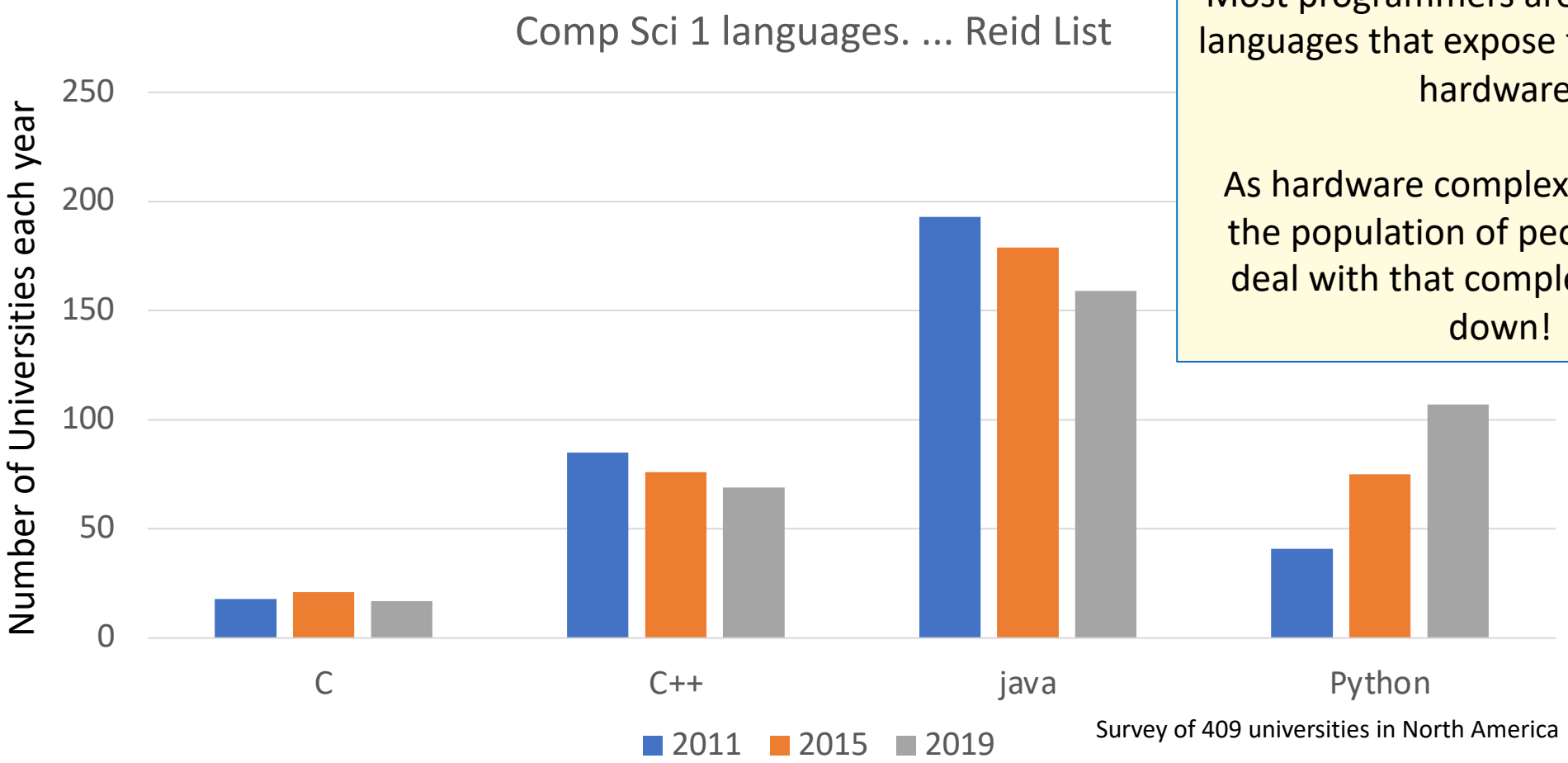
Why is Python so slow?

- Python is interpreted … dynamically compiled



- What if I want my Python program to run in parallel. Does that work?
- Not really. Python has a **Global Interpreter lock** (**GIL**). This is a mutex (mutual exclusion lock) to allow only one thread at a time can make forward progress.

Primary Language used in first year, Computer Science Courses



Survey of 409 universities in North America

Most programmers are NOT learning languages that expose features of the hardware.

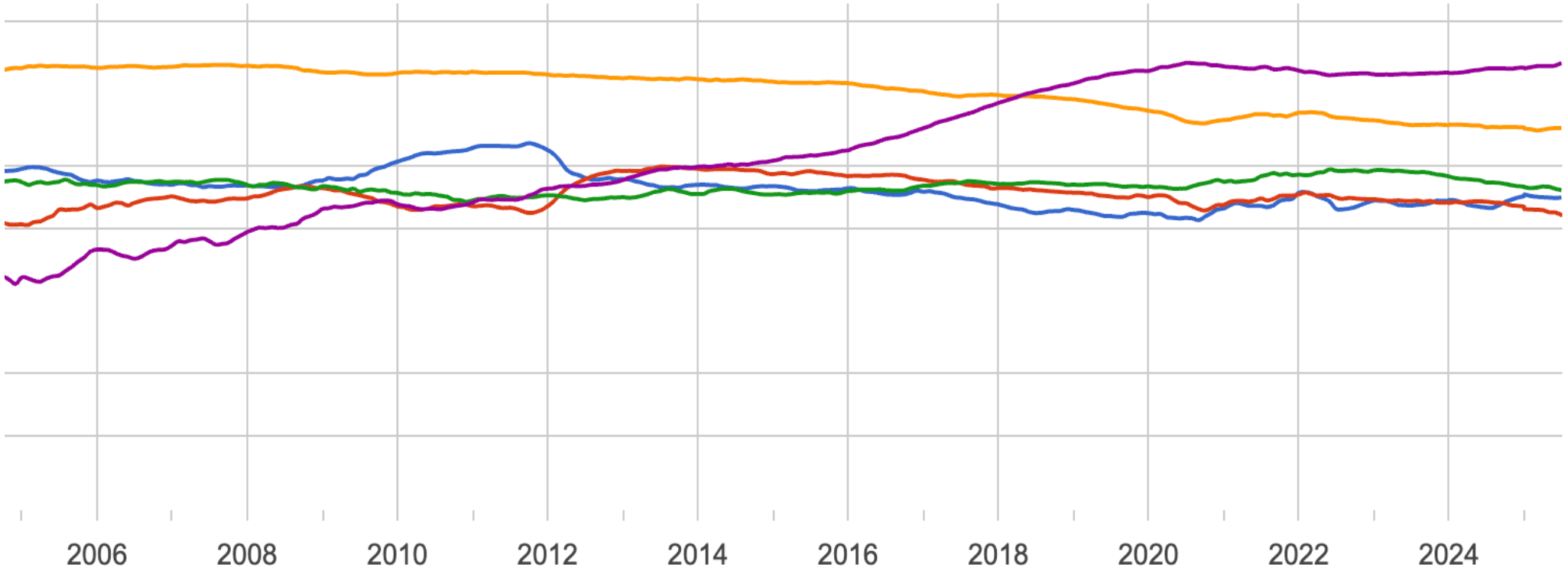
As hardware complexity increases, the population of people who can deal with that complexity is going down!

The Reid List tracks a large sample of North American Universities and the languages they use in teaching.

The Reid List was started by Richard Reid in the 1990s. He has retired but others are carrying on the tradition. The above data comes from Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning, Robert M. Siegfried, Katherine G. Herbert-Berger, Kees Leune, Jason P. Siegfried, The 16th International Conference on Computer Science & Education (ICCSE 2021) August 18-20, 2021.

Python is number One!

Popularity of Programming Languages (PyPI)



Top 5 Languages

Language	Share
Python	31.47 %
Java	15.22%
JavaScript	7.65 %
C/C++	7.05%
C#	5.81%

Programmers have spoken … Python rules. Old-timers (like me) need to stop being such arrogant snobs and help make Python a first class HPC language

https://pypl.github.io/PYPL.html.

Vertical axis is log(PyPl score)

… So perhaps best way to bring parallel computing to the masses would be to combine OpenMP and Python?

Multithreaded parallel Python through OpenMP support in Numba
Todd Anderson, Timothy G. Mattson, SciPy 2021. http://conference.scipy.org/proceedings/scipy2021/tim_mattson.html

PyOMP: Multithreaded Parallel Programming in Python
Timothy G. Mattson, Todd A. Anderson, Giorgis Georgakoudis, Computing in Science and Engineering, IEEE, November/December 2021

PyOMP: Programming GPUs with OpenMP and Python
Giorgis Georgakouis, Todd A. Anderson, Stuart Archibald, Bronis de Supinski, and Timothy G. Mattson. High Performance Python for Science at Scale workshop at SC24, 2024

Design Requirements

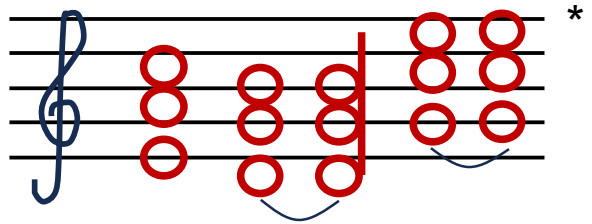
To be an effective parallel computing solution for the Python community, PyOMP must satisfy three requirements

1. It must be Pythonic.
   - It must match the way Python programmers working on scientific computing problems use Python.
   - It cannot change Python syntax

2. It must deliver performance that is on par with what you'd get from C and OpenMP

3. It must be ubiquitous … available and easy to install on any commonly used platform

Pythonic OpenMP in three-part harmony



- Incorporated into the numba JIT compiler. The code is JIT'ed into LLVM and therefore avoids the Global Interpreter Lock (GIL) and supports parallel computing with multiple threads.

- Numpy is the standard module used in scientific computing with Python. Hence, PyOMP is optimized to with numpy arrays.

- OpenMP managed through a context manager (that is, a with statement).

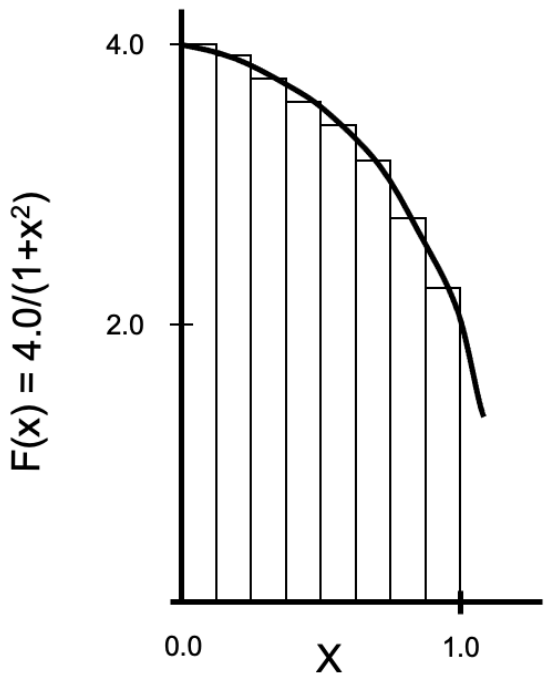
JIT: Just In Time Compilation

*Opening chord progression from the opera *Einstein on the Beach* (Knee Play 1) by Philip Glass

PyOMP by example …

We will understand PyOMP by considering the three fundamental design patterns of OpenMP (**Loop parallelism**, **SPMD**, and **divide and conquer**) applied to the following problem

Numerical Integration (the *hello world* program of parallel computing)



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Each rectangle: width Δx, height $F(x_i)$ at i^{th} interval midpoint.

```
def piFunc(NumSteps):
    step=1.0/NumSteps
    sum = 0.0
    x = 0.5
    for i in range(NumSteps):
        x+=step
        sum += 4.0/(1.0+x*x)
    pi=step*sum
    return pi
```

Loop Parallelism code

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    sum = 0.0

    with openmp ("parallel for private(x) reduction(+:sum)"):
        for i in range(NumSteps):
            x = (i+0.5)*step
            sum += 4.0/(1.0 + x*x)

    pi = step*sum
    return pi

pi = piFunc(100000000)
```

OpenMP constructs managed through an *openmp* context manager.

Pass the OpenMP directive into the OpenMP context manager as a string

Python's implicit data management mapped onto OpenMP. Default rules:

- Variables referenced outside the OpenMP construct are shared
- Variables that only appear inside a construct are private
- Python for technical applications typically based on Numpy arrays, so PyOMP focusses on numpy arrays as well.

OpenMP data environment clauses are supported in PyOMP

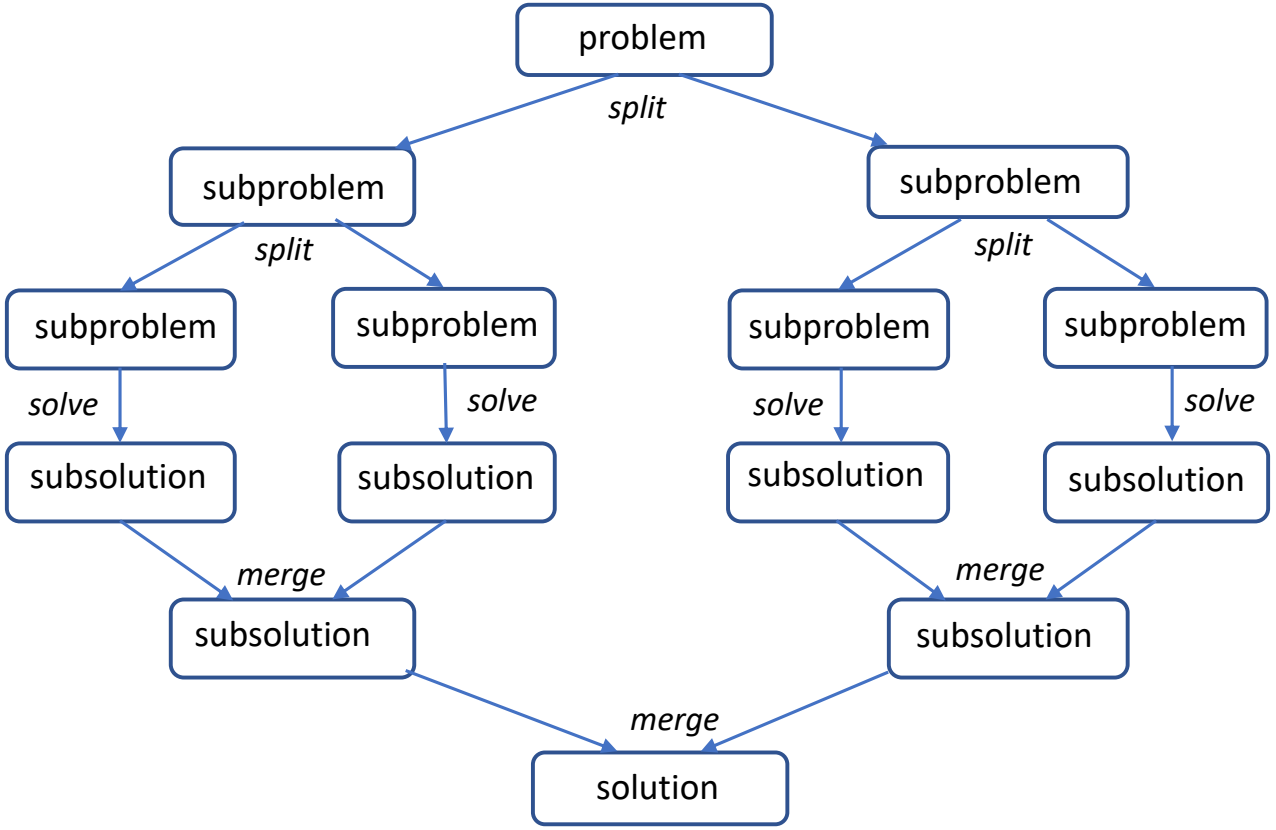
Single Program Multiple Data (SPMD)

```
from numba import njit
import numpy as np
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_thread_num, omp_get_num_threads
MaxTHREADS = 32
@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    partialSums = np.zeros(MaxTHREADS)
    with openmp("parallel shared(partialSums,numThrds) private(threadID,i,x,localSum)"):
        threadID = omp_get_thread_num()
        with openmp("single"):
          numThrds = omp_get_num_threads()
        localSum = 0.0
        for i in range(threadID, NumSteps, numThrds):
            x = (i+0.5)*step
            localSum = localSum + 4.0/(1.0 + x*x)
        partialSums[threadID] = localSum
    return step*np.sum(partialSums)

pi = piFunc(100000000)
```

Deal out loop iterations as if a deck of cards (a cyclic distribution) … each threads starts with the Iteration = ID, incremented by the number of threads, until the whole "deck" is dealt out.

Divide and Conquer

- Split the problem into smaller sub-problems; continue until the sub-problems can be solved directly



- 3 Options for parallelism:
  - Do work as you split into sub-problems
  - Do work only at the leaves
  - Do work as you recombine

Divide and conquer (with explicit tasks)

```
from numba import njit
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_num_threads, omp_set_num_threads
MIN_BLK = 1024*256
@njit
def piComp(Nstart, Nfinish, step):
    iblk = Nfinish-Nstart
    if(iblk<MIN_BLK):
        sum = 0.0
        for i in range(Nstart,Nfinish):
            x= (i+0.5)*step
            sum += 4.0/(1.0 + x*x)
    else:
        sum1 = 0.0
        sum2 = 0.0
        with openmp ("task shared(sum1)"):
            sum1 = piComp(Nstart, Nfinish-iblk/2,step)
        with openmp ("task shared(sum2)"):
            sum2 = piComp(Nfinish-iblk/2,Nfinish,step)
        with openmp ("taskwait"):
            sum = sum1 + sum2
    return sum
```

Solve

Split

Merge

```
@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    sum = 0.0
    startTime = omp_get_wtime()
    with openmp ("parallel"):
        with openmp ("single"):
            sum = piComp(0,NumSteps,step)

    pi = step*sum
    return step*sum

pi = piFunc(100000000)
```

Fork threads and launch the computation

Loop Parallelism code naturally maps onto the GPU

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    sum = 0.0

    with openmp ("target teams loop private(x) reduction(+:sum)"):
        for i in range(NumSteps):
            x = (i+0.5)*step
            sum += 4.0/(1.0 + x*x)

    pi = step*sum
    return pi

pi = piFunc(100000000)
```

OpenMP constructs managed through the *with* context manager.

Map the loop onto a 1D index space … the loop body defines the kernel function

Design Requirements

To be an effective parallel computing solution for the Python community, PyOMP must satisfy three requirements

1. It must be Pythonic.
   - It must match the way Python programmers working on scientific computing problems use Python.
   - It cannot change Python syntax

2. It must deliver performance that is on par with what you'd get from C and OpenMP

3. It must be ubiquitous … available and easy to install on any commonly used platform

We get a "check minus" on programmability since many Python programmers avoid loops and express algorithms solely through numpy array expressions.

PyOMP needs to support the OpenMP workshare construct and implement it with fusion and elision of temporary arrays (something we know how to do based on work on Parallel Accelerator)

What about performance? Multiple threads running slow, dynamically compiled python code is still slow

But with statically compiled JIT'ed code ... PyOMP programs are fast.

JIT: Just In Time compilation

Numerical Integration results in seconds … lower is better

Threads	PyOMP			C/OpenMP		
	Loop	SPMD	Task	Loop	SPMD	Task
1	0.447	0.450	0.453	0.444	0.448	0.445
2	0.252	0.255	0.245	0.245	0.242	0.222
4	0.160	0.164	0.146	0.149	0.149	0.131
8	0.0890	0.0890	0.0898	0.0827	0.0826	0.0720
16	0.0520	0.0503	0.0517	0.0451	0.0451	0.0431

10^8 steps

Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

For the C programs we used Intel® icc compiler version 19.1.3.304 as icc -qnextgen -O3 –fiopenmp

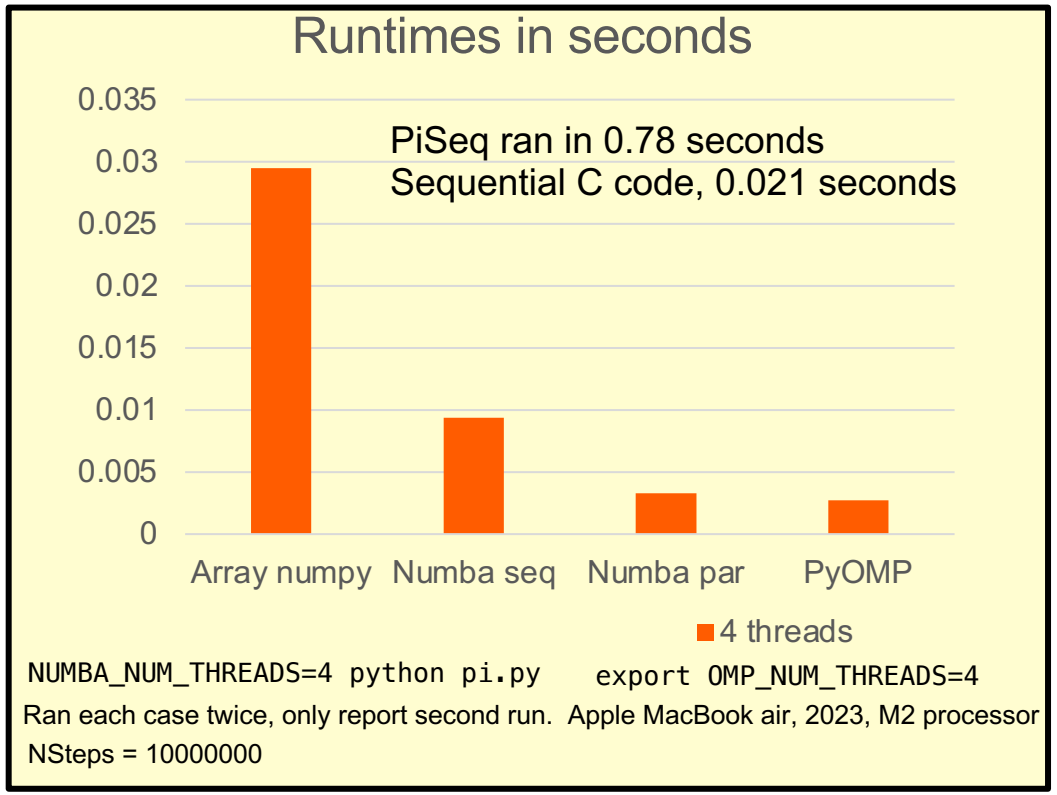
Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

Various Pi programs in Python

```
import numba
from numba import njit
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_wtime
import numpy as np

def piArr(Nsteps):
    startTime = omp_get_wtime()
    stepSize    = 1.0/Nsteps
    pts         = np.linspace(0.0,1.0,Nsteps)
    ptsSquPlus1 = np.square(pts)+1.0
    ptsInteg    = 4.0/ptsSquPlus1
    pi          = stepSize*np.sum(ptsInteg)
    runtime = omp_get_wtime()-startTime
    return pi,runtime

def piSeq(NumSteps):
    step = 1.0/NumSteps
    sum  = 0.0
    startTime = omp_get_wtime()
    for i in range(NumSteps):
        x = (i+0.5)*step
        sum += 4.0/(1.0 + x*x)
    pi = step * sum
    runtime = omp_get_wtime()-startTime
    return pi,runtime
```

```
@njit
def piNUMBAseq(NumSteps):
    step = 1.0/NumSteps
    sum  = 0.0
    startTime = omp_get_wtime()
    for i in range(NumSteps):
        x = (i+0.5)*step
        sum += 4.0/(1.0 + x*x)
    pi = step * sum
    runtime = omp_get_wtime()-startTime
    return pi,runtime

@njit(parallel=True)
def piNUMBApar(NumSteps):
    step = 1.0/NumSteps
    sum  = 0.0
    startTime = omp_get_wtime()
    for i in numba.prange(NumSteps):
        x = (i+0.5)*step
        sum += 4.0/(1.0 + x*x)
    pi = step * sum
    runtime = omp_get_wtime()-startTime
    return pi,runtime
```

```
@njit
def piOMP(NumSteps):
    step = 1.0/NumSteps
    sum  = 0.0
    startTime = omp_get_wtime()
    with openmp("parallel for reduction(+:sum)"):
      for i in range(NumSteps):
        x = (i+0.5)*step
        sum += 4.0/(1.0 + x*x)
    pi = step * sum
    runtime = omp_get_wtime()-startTime
    return pi,runtime
```

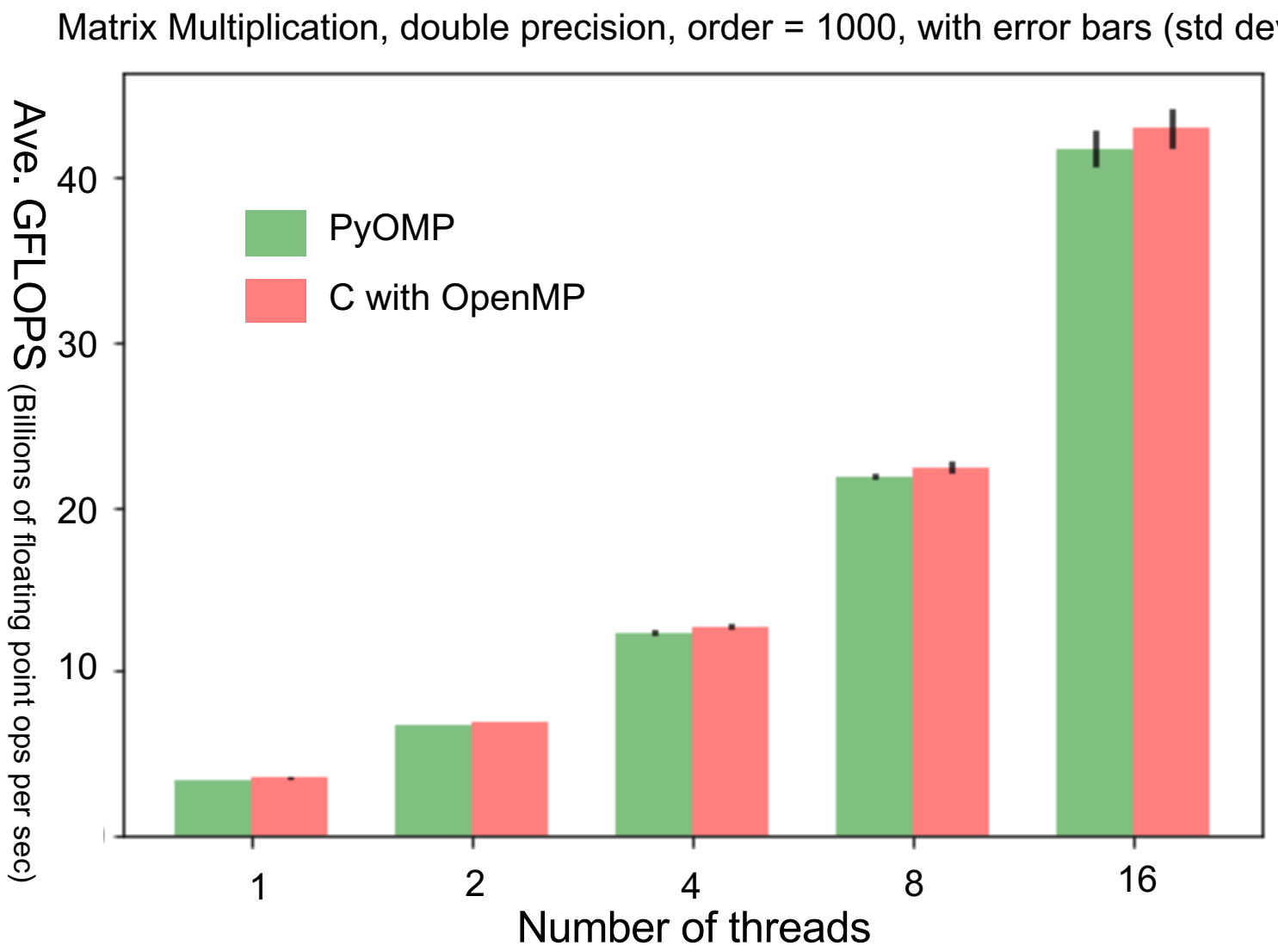


PyOMP DGEMM (Mat-Mul with double precision numbers)

```
from numba import njit
import numpy as np
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_wtime

@njit(fastmath=True)
def dgemm(iterations,order):

    # allocate and initialize arrays
    A = np.zeros((order,order))
    B = np.zeros((order,order))
    C = np.zeros((order,order))

    # Assign values to A and B such that
    # the product matrix has a known value.
    for i in range(order):
        A[:,i] = float(i)
        B[:,i] = float(i)
```

```
tInit = omp_get_wtime()
with openmp("parallel for private(j,k)"):
      for i in range(order):
          for k in range(order):
              for j in range(order):
                  C[i][j] += A[i][k] * B[k][j]

dgemmTime = omp_get_wtime() - tInit

# Check result
checksum = 0.0;
for i in range(order):
    for j in range(order):
        checksum += C[i][j];
ref_checksum = order*order*order
ref_checksum *= 0.25*(order-1.0)*(order-1.0)
eps=1.e-8
if abs((checksum - ref_checksum)/ref_checksum) < eps:
    print('Solution validates')
    nflops = 2.0*order*order*order
    print('Rate (MF/s): ',1.e-6*nflops/dgemmTime)
```

DGEMM PyOMP vs C-OpenMP



250 runs for order 1000 matrices

PyOMP times **DO NOT** include the one-time JIT cost of ~2 seconds.

Intel® Xeon® E5-2699 v3 CPU, 18 cores, 2.30 GHz, threads mapped to a single CPU, one thread/per core, first 16 physical cores.
Intel® icc compiler ver 19.1.3.304 (icc –std=c11 –pthread –O3 xHOST –qopenmp)

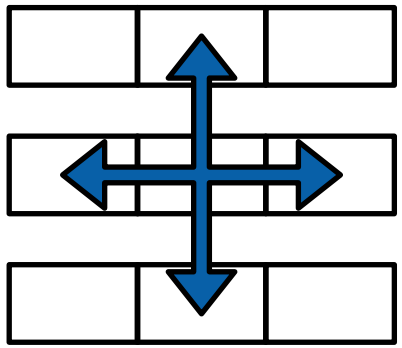
5-point stencil: Heat diffusion problem

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$

```
# Loop over time steps
for _ in range(nsteps):
    # solve over spatial domain for step t
    solve(n, alpha, dx, dt, u, u_tmp)

    # Array swap to get ready for next step
    u, u_tmp = u_tmp, u
```

$$\frac{\partial u}{\partial t} \approx \frac{u(t+1,x,y) - u(t,x,y)}{dt}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t,x+1,y) - 2u(t,x,y) + u(t,x-1,y)}{dx^2}$$



5-point stencil: solve kernel

25,000x25,000 grid for 10 time steps
* Xeon Platinum 8480+: 67.6 secs

```
@njit
def solve(n, alpha, dx, dt, u, u_tmp):
    # Finite difference constant multiplier
    r = alpha * dt / (dx ** 2)
    r2 = 1 - 4 * r
    # Loop over the nxn grid
        for i in range(n):
            for j in range(n):
                # Update the 5-point stencil.
                # Using boundary conditions on the edges of the domain.
                # Boundaries are zero because the MMS solution is zero there.
                u_tmp[j, i] = (r2 * u[j, i] +
                               (u[j, i+1] if i < n-1 else 0.0) +
                               (u[j, i-1] if i > 0   else 0.0) +
                               (u[j+1, i] if j < n-1 else 0.0) +
                               (u[j-1, i] if j > 0 else 0.0))
```

Solution: parallel stencil (heat)

25,000x25,00 grid for 10 time steps
- Xeon Platinum 8480+: 67.6 secs
- Nvidia V100: 22.6 secs

```
@njit
def solve(n, alpha, dx, dt, u, u_tmp):
    """Compute the next timestep, given the current timestep"""

    # Finite difference constant multiplier
    r = alpha * dt / (dx ** 2)
    r2 = 1 - 4 * r
    with openmp ("target loop collapse(2) map(tofrom: u, u_tmp)"):
        # Loop over the nxn grid
        for i in range(n):
            for j in range(n):
                u_tmp[j, i] = (r2 * u[j, i] +
                               (u[j, i+1] if i < n-1 else 0.0) +
                               (u[j, i-1] if i > 0   else 0.0) +
                               (u[j+1, i] if j < n-1 else 0.0) +
                               (u[j-1, i] if j > 0 else 0.0))
```

Data Movement dominates…

Solution: parallel stencil (heat)

25,000x25,00 grid for 10 time steps
- Xeon Platinum 8480+: 67.6 secs
- Nvidia V100: 22.6 secs

```
@njit
def solve(n, alpha, dx, dt, u, u_tmp):
    """Compute the next timestep, given the current timestep"""

    # Finite difference constant multiplier
    r = alpha * dt / (dx ** 2)
    r2 = 1 - 4 * r
    with openmp ("target loop collapse(2) map(tofrom: u, u_tmp)"):
        # Loop over the nxn grid
        for i in range(n):
            for j in range(n):
                u_tmp[j, i] = (r2 * u[j, i] +
                               (u[j, i+1] if i < n-1 else 0.0) +
                               (u[j, i-1] if i > 0   else 0.0) +
                               (u[j+1, i] if j < n-1 else 0.0) +
                               (u[j-1, i] if j > 0 else 0.0))
```

There can be many time steps …

For each step, $(2*N^2)$*sizeof(TYPE) bytes move between the host and the device

- We need to keep data resident on the device *between* target regions
- We need a way to manage the device data environment across iterations.

Solution: Explicitly manage the device data environment

```
with openmp ("target enter data map(to: u, u_tmp)"):
    pass

for _ in range(nsteps):

    solve(n, alpha, dx, dt, u, u_tmp);

    # Array swap to get ready for next step
    u, u_tmp = u_tmp, u

with openmp ("target exit data map(from: u)"):
    pass
```

Copy data to device before iteration loop

Change solve() routine to remove map clauses:
`with openmp ("target loop collapse(2)”)`

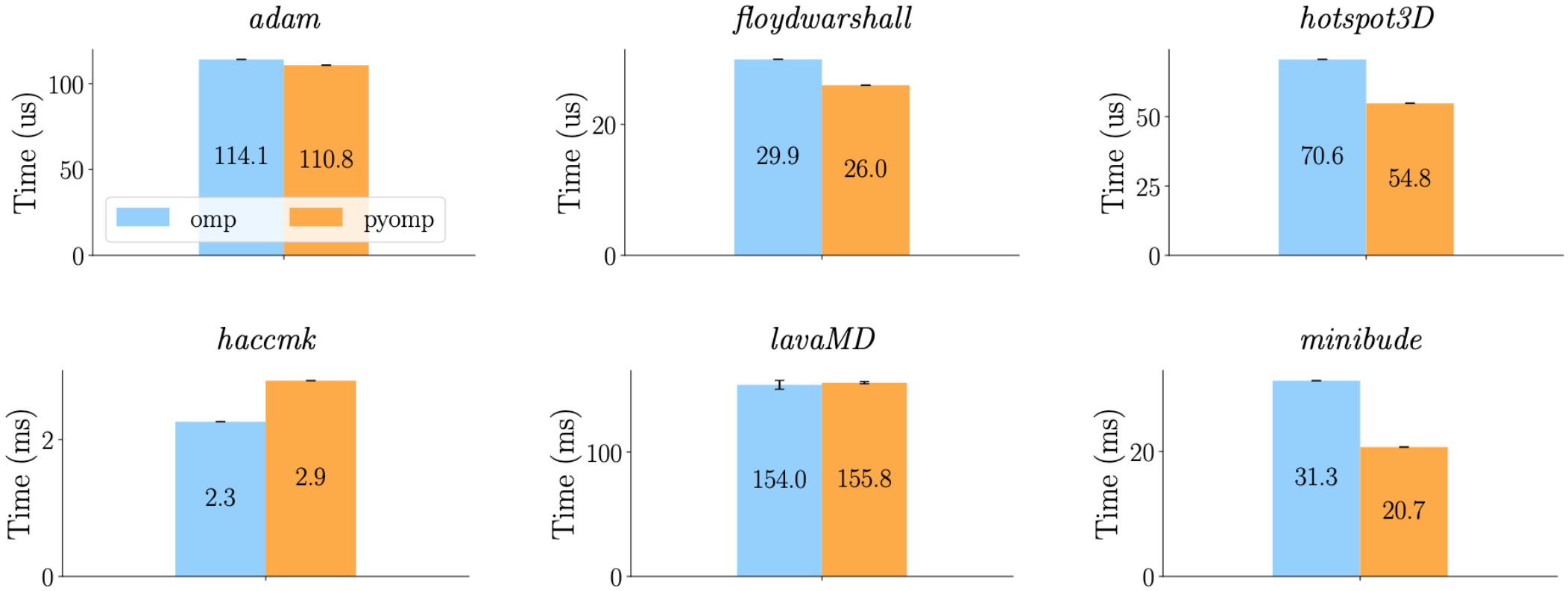
Copy data from device after iteration loop

25,000x25,00 grid for 10 time steps

- Xeon Platinum 8480+ default data movement: 67.6 secs
- Nvidia V100 default data movement: 22.6 secs
- Nvidia V100 target enter/exit: 1.2 secs

PyOMP HECBench GPU results

Program	Description	Input
adam	Adaptive moment estimation stochastic optimization for machine learning	10000 200 100
floydwarshall	Floyd-Warshall path finding algorithm	1024 100 16
haccmk	HACC cosmology code micro-kernel	1000
hotspot3D	Thermal modeling for 3D integrated circuits stencil computation	512 8 5000 power_512x8 temp_512x8
lavaMD	Molecular dynamics	-boxes1d 30
miniBUDE	Ligand-protein docking	–deck data/bm1 –wgsize 256 –iterations 100



Details in an IWOMP'25 paper submission

AMD EPYC 7763 CPU with an NVIDIA A100 GPU with 80 GB or memory.
Python 3.9.18, Numba 0.57, llvm-lite 0.40, CUDA 12.2 with driver version 525.105.17

Design Requirements

To be an effective parallel computing solution for the Python community, PyOMP must satisfy three requirements

1. It must be Pythonic.
   - It must match the way Python programmers working on scientific computing problems use Python.
   - It cannot change Python syntax

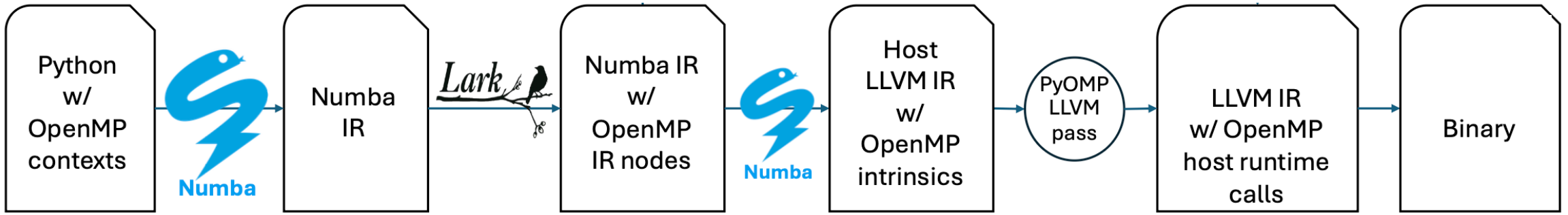
2. It must deliver performance that is on par with what you'd get from C and OpenMP

3. It must be ubiquitous … available and easy to install on any commonly used platform

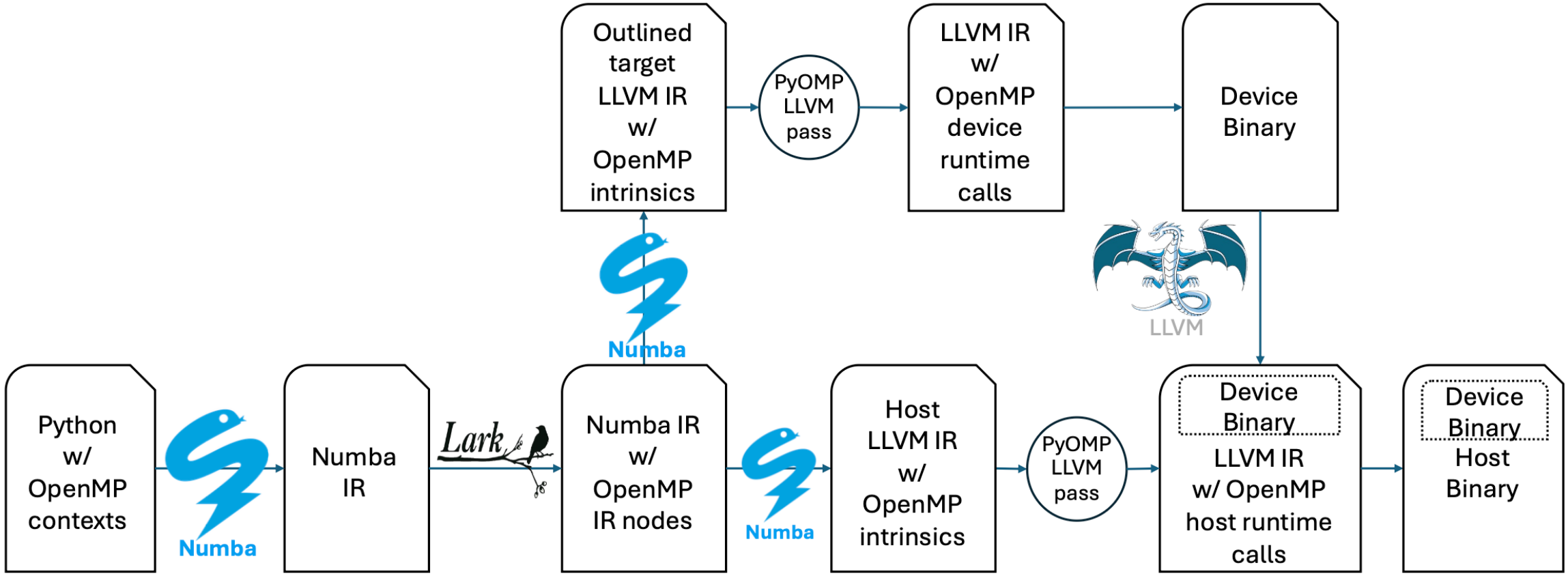
How did we implement PyOMP?

We build on established tools following standard practice in the Python Community

PyOMP implementation: CPU



PyOMP implementation: CPU + GPU



PyOMP: a Numba extension for upgradeability and maintainability

- Depends on Numba as a compiler toolkit
  - Similar to numba-cuda, numba-hip
- Uses Numba's LLVM dependencies
  - llvmlite: provides python bindings for the LLVM API (Currently supports LLVM 14.x – We may need to patch PyOMP when Numba moves to LLVM 18/19)
- Tested with Numba 0.57.x, 0.58.x
  - Architectures: linux-64 (x86_64), osx-arm64 (mac), linux-arm64, linux-ppc64le

PyOMP piggybacks on the off-the-shelf Numba ecosystem.

We don't need to do any extra work to adapt as new versions of Numba are released

PyOMP is easy to install and use

- Conda one-line installation

```
conda install -c python-for-hpc -c conda-forge pyomp
```

- PyPi package is underway

```
pip install pyomp
```

- Fast ways to try
  - Binder: https://mybinder.org/v2/gh/Python-for-HPC/binder/HEAD
  - Docker: `docker pull ghcr.io/python-for-hpc/pyomp:latest`

Open Source code on github: https://github.com/Python-for-HPC/PyOMP

Design Requirements

To be an effective parallel computing solution for the Python community, PyOMP must satisfy three requirements

1. It must be Pythonic.
   - It must match the way Python programmers working on scientific computing problems use Python.
   - It cannot change Python syntax

2. It must deliver performance that is on par with what you'd get from C and OpenMP

3. It must be ubiquitous … available and easy to install on any commonly used platform

We get a "check minus" on ubiquity since we only work with Nvidia GPUs.

Nvidia works closely with the Numba community so it was straightforward for us to support their GPUs. There is no reason we can't support AMD GPUs, but it will take a small bit or work to make it happen.

In anticipation of future work on AMD GPUs we are refactoring the PyOMP software to make adding other GPUs much easier. Stay tuned

Related work: Other OpenMP API bindings

- Pythran
  - Transpiles python to C++
  - OpenMP using # comments
- PyKokkos
  - Transpiles python to C++
  - OpenMP through Kokkos abstractions
  - Limited support: parallel_for, parallel_reduce, parallel_scan
- OMP4Py
  - Pure python implementation (Threads with GIL disabled)
  - No GPU support (OpenMP version 3.0)
  - Similar interface to PyOMP
  - Slow

Why PyOMP is so important …
and why all of you should start using it

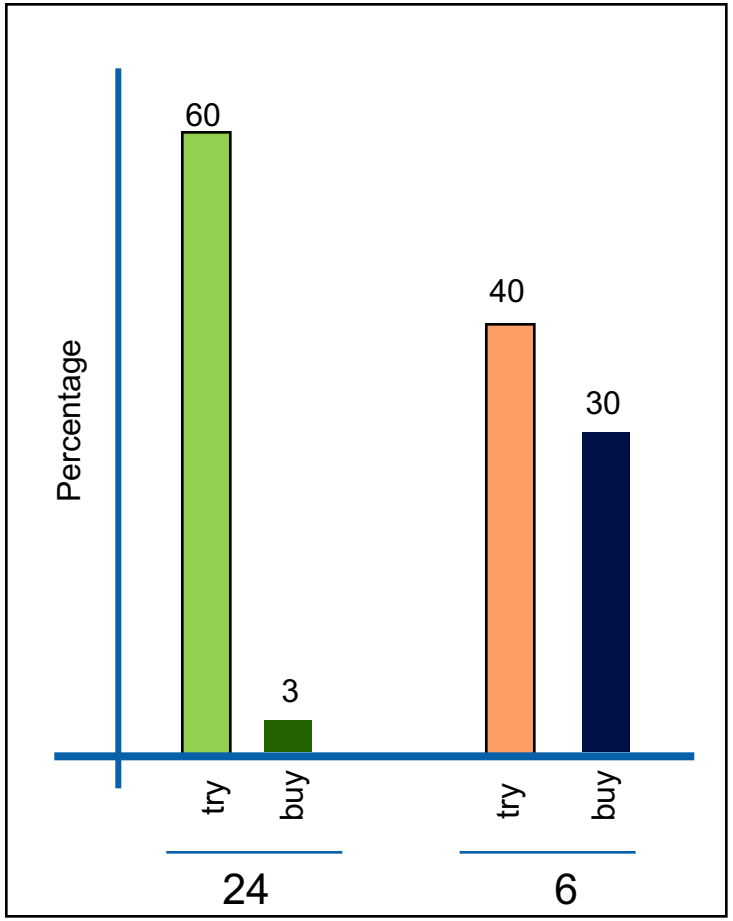
In the early days of parallel computing, we were obsessed with finding the "right" parallel programming environment

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL
CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda
GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
HPF
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm
Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM
pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

Parallel program environments in the 90's

Language obsessions: More isn't always better

- The Draeger Grocery Store experiment and consumer choice:
  - Two Jam-displays with coupons for a discount on purchase.
    - 24 different Jam's
    - 6 different Jam's
  - How many stopped by to try samples at the display?
  - Of those who "tried", how many bought jam?



The findings from this study show that an extensive array of options can at first seem highly appealing to consumers, yet can reduce their subsequent motivation to purchase the product.

Iyengar, Sheena S., & Lepper, Mark (2000). When choice is demotivating: Can one desire too much of a good thing? *Journal of Personality and Social Psychology*, 76, 995-1006.

In the early days of parallel computing, we were obsessed with finding the "right" parallel programming environment

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
[illegible]
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
HPF
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
[illegible]
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
[illegible]
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
[illegible]
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
CGMSG.
hreads.h++.
readMarks
RAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

With Choice overload in mind … what did we accomplish with all these different options for parallel programming?

Parallel program environments in the 90's

In the early days of parallel computing, we were obsessed with finding the "right" parallel programming environment

ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.

Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
HPF
IMPACT
ISIS.
JAVAR
JADE

Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI

Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++

Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
eads
nd.
F.
ergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

Furthermore, engineering is a zero-sum game … time spent chasing the next great programming model is time NOT spent making the models we have actually work

Parallel program environments in the 90's

The end of the crisis

- In the early 90's, the HPC community was fed up with message passing chaos. Driven largely by application developers, we created MPI (version 1.0 released in 1994).

- In the late 90's, the HPC community working in the Accelerated Strategic Computing Initiative (ASCI) used their influence over which HPC systems were purchased to "force" vendor's hands to support a standard for programming shared memory systems. The result was OpenMP (version 1.0 released in 1997).

Portable parallel programming is important for the people who create HPC applications. It took their direct involvement and dedication to create open standards and end parallel programming chaos.

The major parallel Programming systems in 2024 …

well at least we have our act together in two cases. ☹

- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.
  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP** … : GPU programming (use CUDA if you don't mind locking yourself to a single vendor … it is a really nice programming model)

Parallel programming with Python is terribly fragmented

dispy
Delegate
forkmap
forkfun
Jobibppmap
POSH
pp
pprocess
processing
PyCSP
PyMP
Ray
remoteD
torcp
VecPy
batchlib
Celery
Charm4py
PyCUDA
Ramba

Dask
Deap
disco
dispy
DistributedPYthon
exec_proxy
execnet
iPython
job_stream jug
mpi4py
NetWorkSpaces
PaPy
papyrus
PyCOMPSs
PyLinda
pyMPI
pypar
multiprocessing
PyOpenCL

pyPastSet
pypvm
pynpvm
Pyro
Ray
Rthread
ScientificPython.BSP
Scientific.DistrubedComputing.MasterSlave
Scientific.MPI
SCOOP
seppo
PySpark
Star-P
superrpy
torcpy
StarCluster
dpctl
arkouda
PyOMP
dpnp

Python programmers are locked into the same dystopic world of HPC in the 90's.

History suggests that this won't get better until the python applications community demands (and dedicates themselves) to a minimal set of open, standard solutions

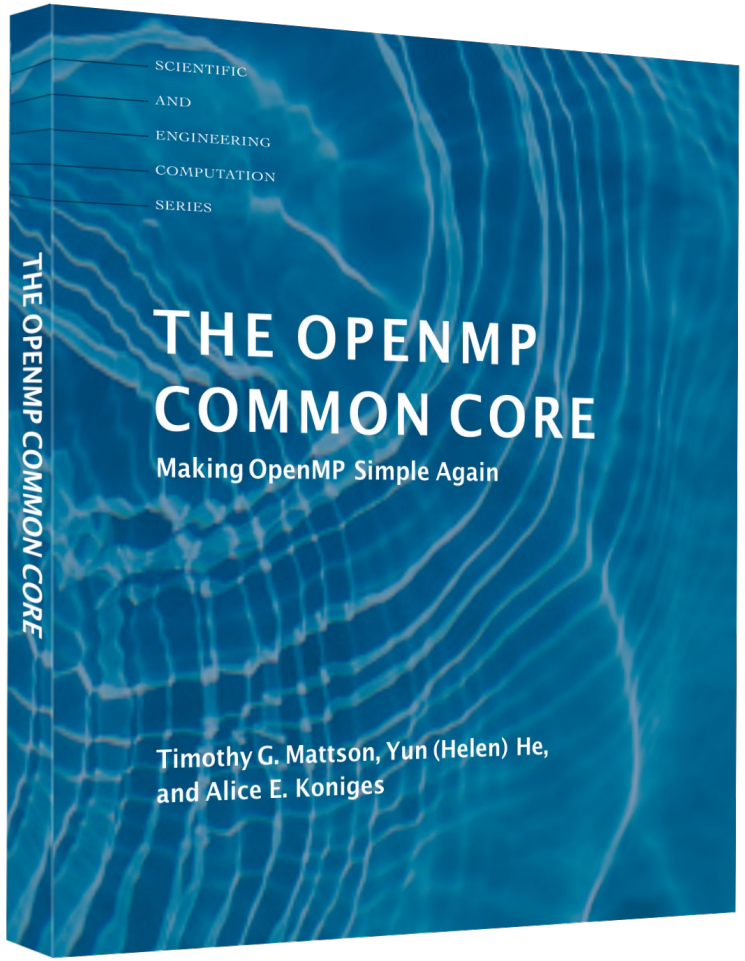
Building on the list at https://wiki.python.org/moin/ParallelProcessing

Conclusion

- Python is the language of choice for most programmers … so let's stop telling them to learn C/C++ or Fortran to do HPC
- PyOMP lets you write OpenMP code in Python. Try it, you'll like it.
- But we need your help …
  - We need a user base. Please use it and tell us about your successes and failures.
  - Help drive convergence around a minimal number of open, portable parallel programming environments in Python. We all win if this happens.
  - We need more people to join the PyOMP team and help us grow the technology. For example, I want the OpenMP workshare construct with fusion and array elision. I need someone to work with us to make that happen.

My Greenlandic skin-on-frame kayak in the middle of Budd Inlet during a negative tide

The OpenMP Common Core



For many years now, we've been teaching the subset of OpenMP that is most commonly used. We call this the

OpenMP Common Core

We even wrote a book about it.

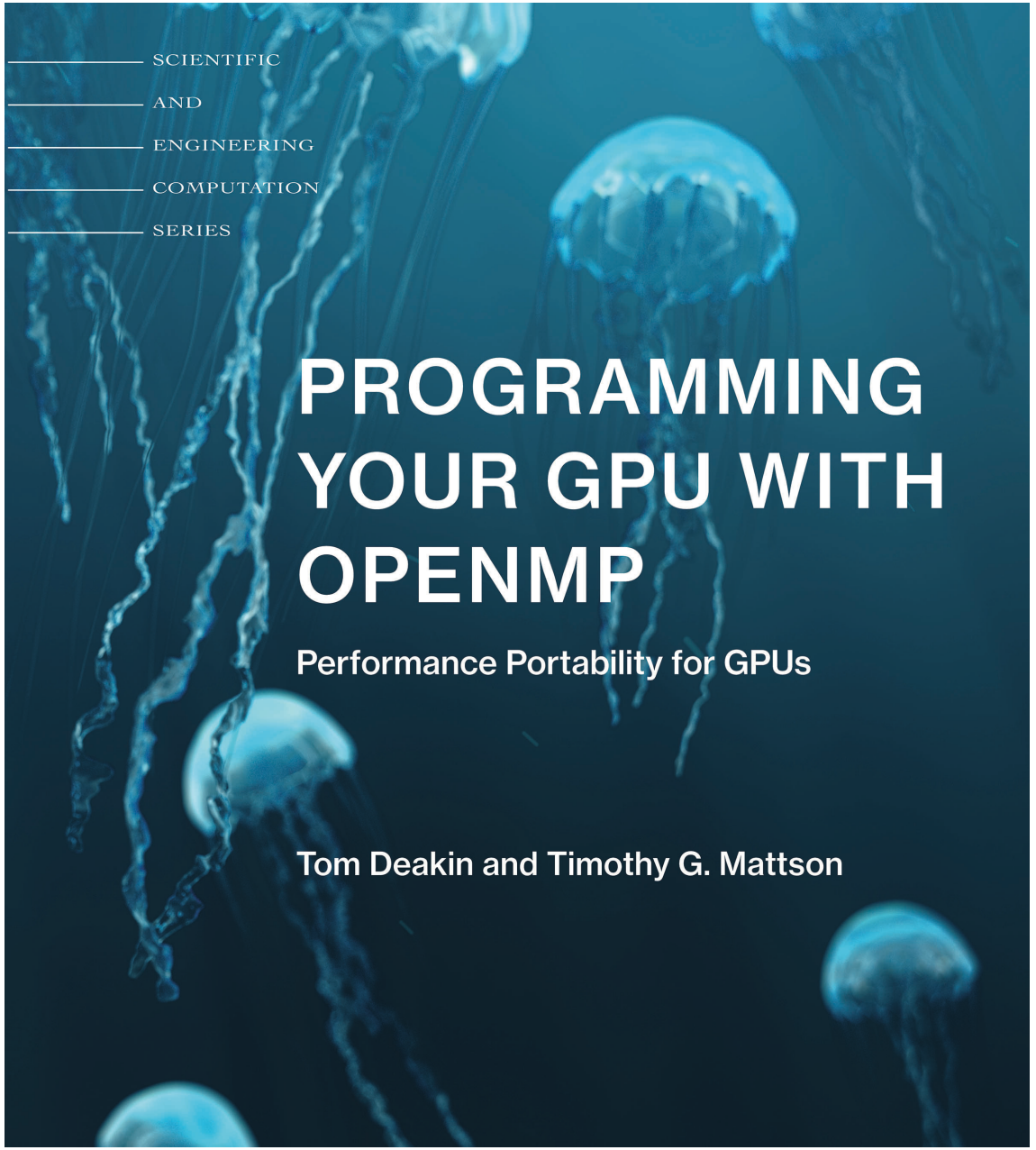
The list of items in the common core were determined by experience/anecdote … we didn't have hard data to drive the analysis.

The OpenMP Common Core
#pragma omp parallel
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()
double omp_get_wtime()
setenv OMP_NUM_THREADS N
#pragma omp barrier #pragma omp critical
#pragma omp for #pragma omp parallel for
reduction(op:list)
schedule (static [,chunk]) schedule(dynamic [,chunk])
shared(list), private(list), firstprivate(list)
default(none)
nowait
#pragma omp single
#pragma omp task #pragma omp taskwait

To learn more about GPU programming with OpenMP

The latest book on OpenMP …

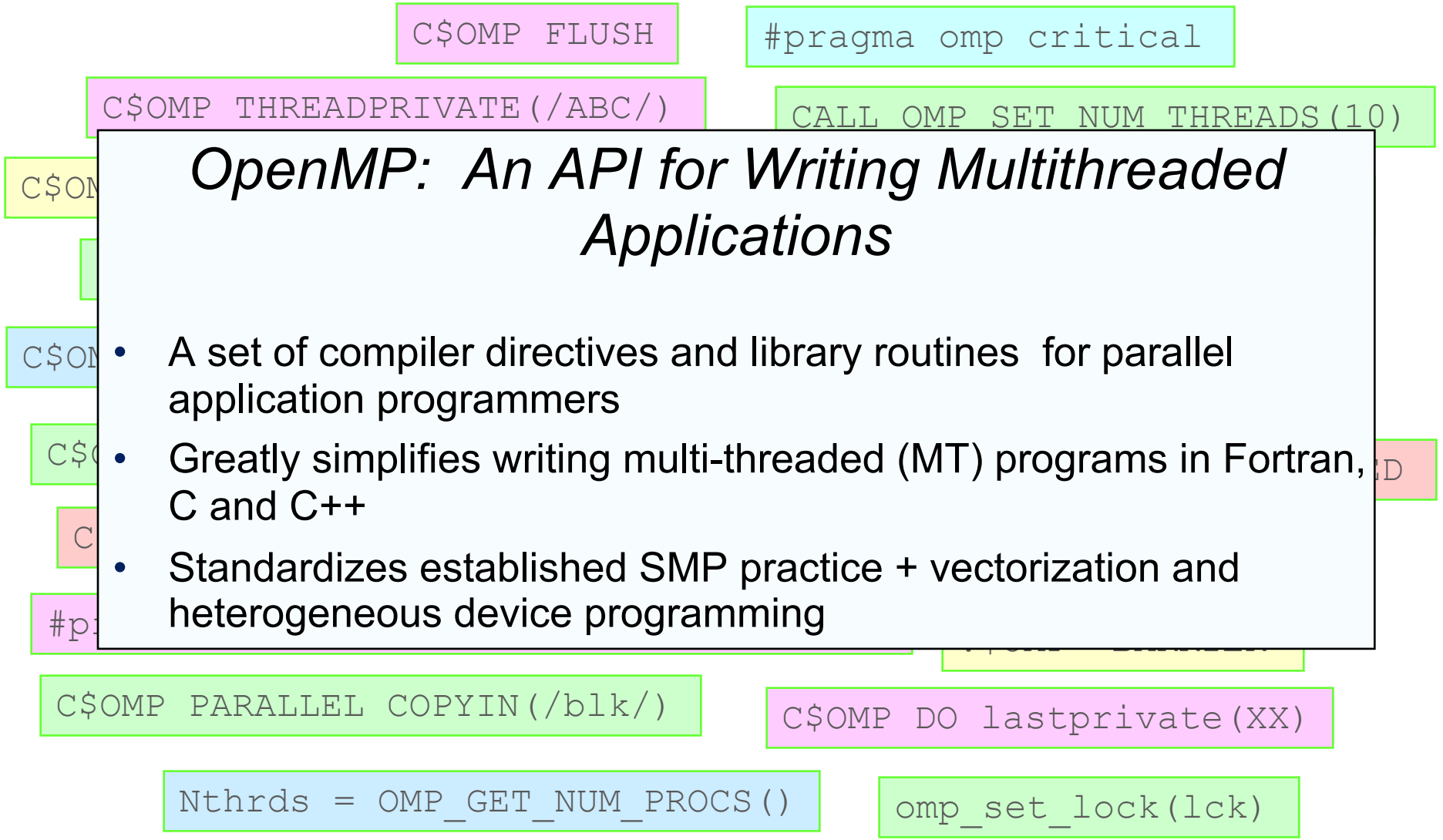
A book about how to use OpenMP to program a GPU (focusses on C and C++ … not Python)



Extra content …

A tutorial introduction to PyOMP (for programmers new to OpenMP)

OpenMP* Overview



OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

PyOMP: OpenMP projected into Python

- A parallel multithreaded “hello world” program with PyOMP

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def hello():
    with openmp("parallel"):
        print("hello")
        print("world")

hello()
print("DONE")
```

PyOMP: OpenMP projected into Python

- A parallel multithreaded “hello world” program with PyOMP

Numba Just In Time (JIT) compiler compiles the Python code into LLVM.

Compiled code cached for later use.

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def hello():
    with openmp("parallel"):
        print("hello")
        print("world")

hello()
print("DONE")
```

“parallel” creates a team of threads

The code inside the with context manager is packaged into a function and executed by each thread

OpenMP managed through the *with* context manager.

- Numba Just In Time (JIT) compiler compiles the Python code into LLVM thereby bypassing the GIL. Hence, the threads execute in parallel.
- The string in the with openmp context manager is identical to the constructs in OpenMP. If you know OpenMP for C/C++/Fortran, then you know it for Python

PyOMP: OpenMP projected into Python

- A parallel multithreaded “hello world” program with PyOMP

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def hello():
    with openmp("parallel"):
        print("hello")
        print("world")

hello()
print("DONE")
```

When I run this program, here is the output.

```
hello
world
hello
hello
hello
world
hello
world
hello
world
hello
world
hello
world
world
world
hello
world
DONE
```

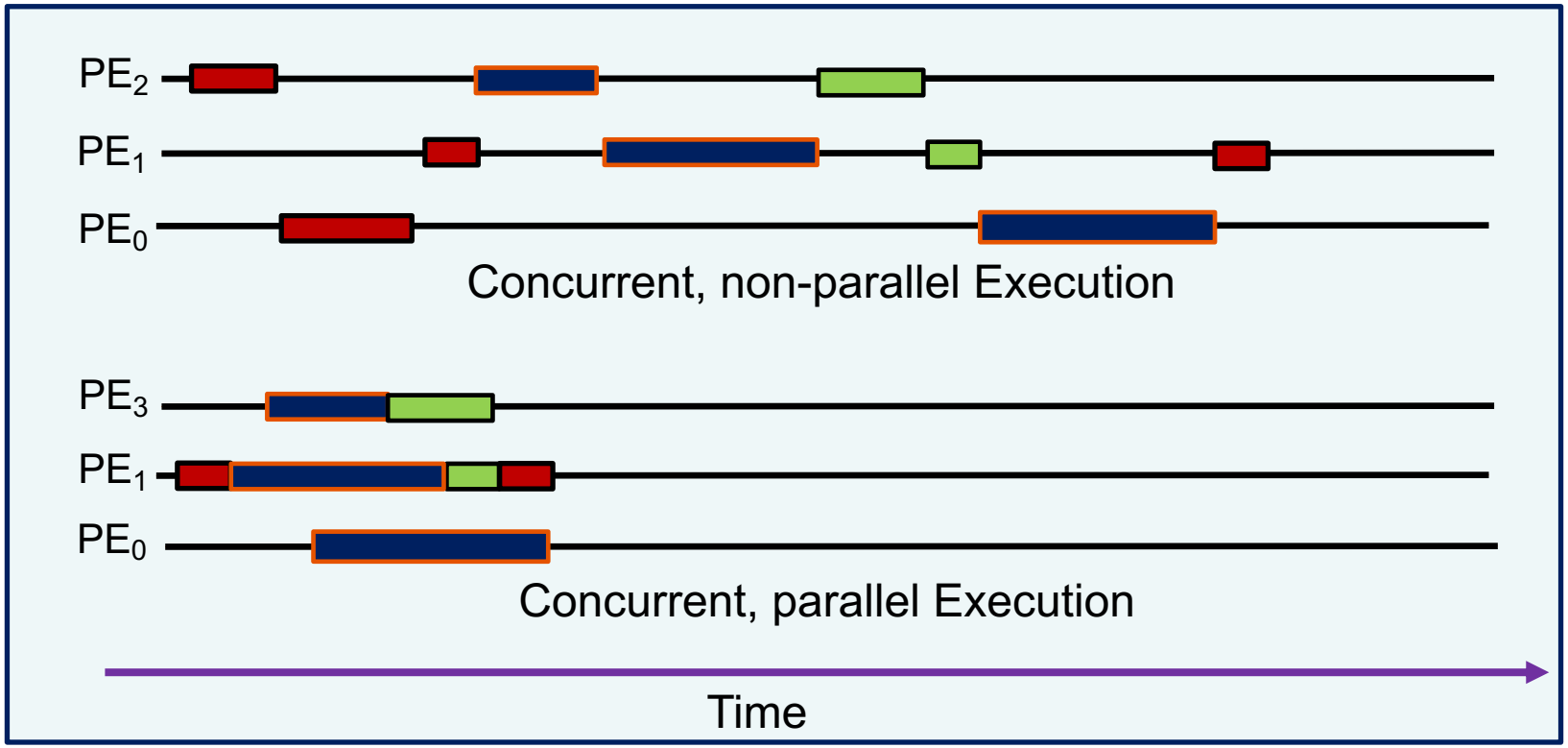
The interleaved print output is different each time I run the program

Why is the output from our hello world program so weird?

To answer that question, we must digress briefly and settle on a few key definitions

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***logically*** making **forward progress** at the same time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** making **forward progress** at the same time.



PE = Processing Element

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

PyOMP: OpenMP projected into Python

- A parallel multithreaded “hello world” program with PyOMP

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def hello():
    with openmp("parallel"):
        print("hello")
        print("world")

hello()
print("DONE")
```

When I run this program, here is the output.

```
hello
world
hello
hello
hello
world
hello
world
hello
world
hello
world
hello
world
world
world
hello
world
DONE
```

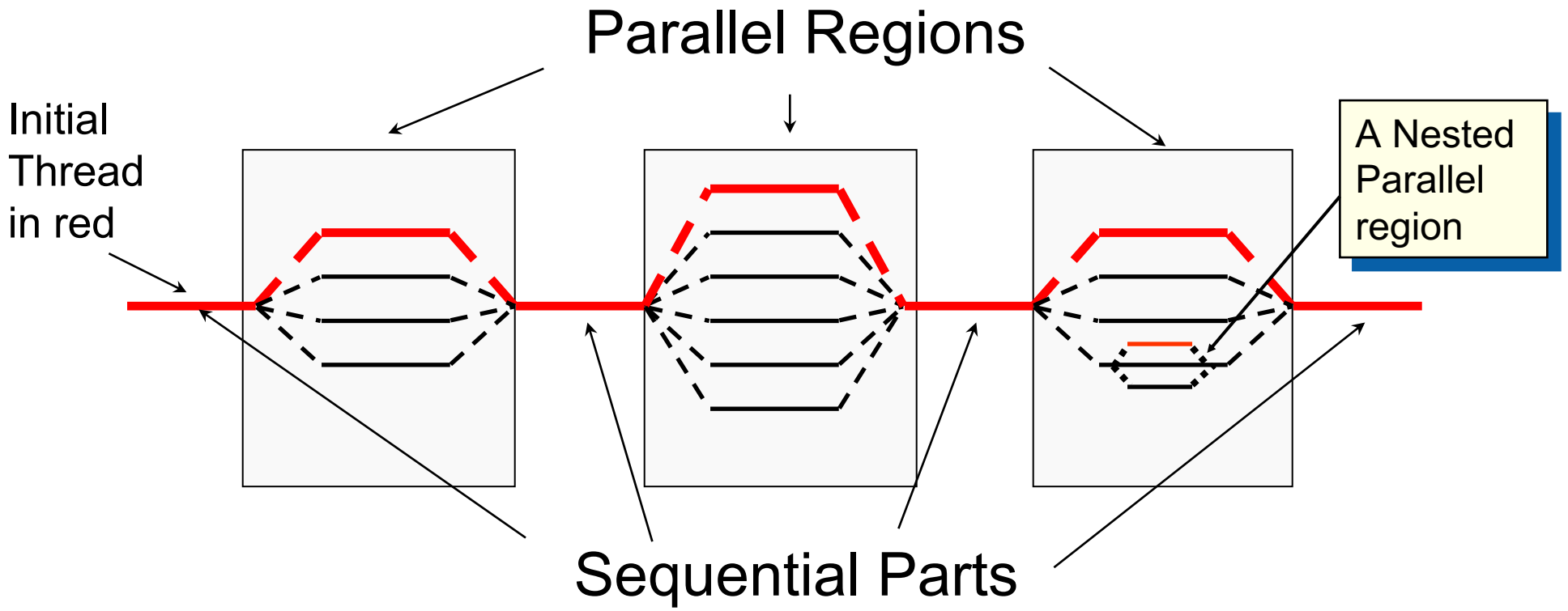
The challenge for programmers writing multithreaded code is to make sure every semantically allowed way statements can interleave results in correct code.

Lets dive into the details of multithreading and how they are most commonly used in an application

OpenMP Execution Model

Fork-Join Parallelism:

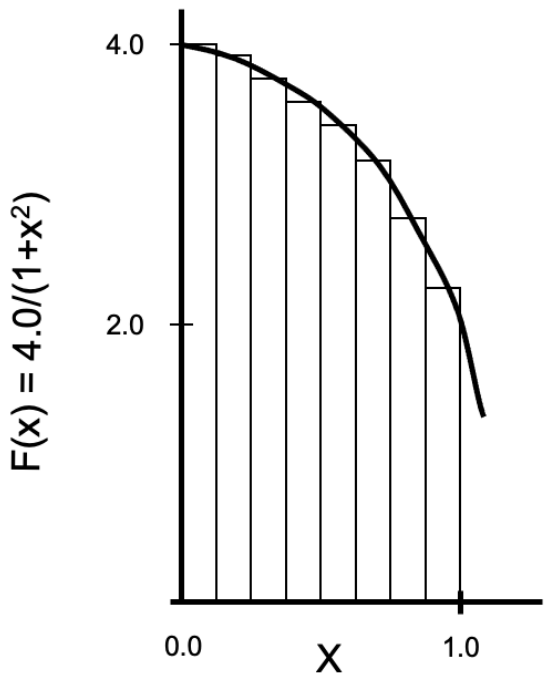
- Initial thread **forks** a team of threads as needed.
- They execute in a shared address space … All reads read/write a common set of the variables.
- When the team is finished, the threads **join** together and the initial thread continues
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



Understanding OpenMP

We will explain the key elements of OpenMP as we explore the three fundamental design patterns of OpenMP (**Loop parallelism**, **SPMD**, and **divide and conquer**) applied to the following problem

Numerical Integration (the *hello world* program of parallel computing)



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Each rectangle: width Δx, height $F(x_i)$ at i^{th} interval midpoint.

```
def piFunc(NumSteps):
    step=1.0/NumSteps
    pisum = 0.0
    x = 0.5
    for i in range(NumSteps):
        x+=step
        pisum += 4.0/(1.0+x*x)
    pi=step*pisum
    return pi
```

The Loop-level parallelism design pattern

- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.

The Loop-level parallelism design pattern

- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.

```
def piFunc(NumSteps):
    step=1.0/NumSteps
    pisum = 0.0
    x = 0.5
    for i in range(NumSteps):
        x+=step
        pisum += 4.0/(1.0+x*x)
    pi=step*pisum
    return pi
```

The Loop-level parallelism design pattern

- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.

```
def piFunc(NumSteps):
    step=1.0/NumSteps
    pisum = 0.0
    x = 0.5
    for i in range(NumSteps):
        x+=step
        pisum += 4.0/(1.0+x*x)
    pi=step*pisum
    return pi
```

A loop carried dependency

The Loop-level parallelism design pattern

- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.

```
def piFunc(NumSteps):
    step=1.0/NumSteps
    pisum = 0.0
    x = 0.5
    for i in range(NumSteps):
        x+=step
        pisum += 4.0/(1.0+x*x)
    pi=step*pisum
    return pi
```

A loop carried dependency → Recast to compute from i

```
def piFunc(NumSteps):
    step=1.0/NumSteps
    pisum = 0.0

    for i in range(NumSteps):
        x=(i+0.5)*step
        pisum += 4.0/(1.0+x*x)
    pi=step*pisum
    return pi
```

The Loop-level parallelism design pattern

- Parallelism defined in terms of parallel loops … that is, loops where iterations can safely execute when divided between a collection of threads.
- Key elements:
  - identify compute intensive loops in a program
  - Expose concurrency by removing or managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.

```
def piFunc(NumSteps):
    step=1.0/NumSteps
    pisum = 0.0
    x = 0.5
    for i in range(NumSteps):
        x+=step
        pisum += 4.0/(1.0+x*x)
    pi=step*pisum
    return pi
```

A loop carried dependency → Recast to compute from i →

```
def piFunc(NumSteps):
    step=1.0/NumSteps
    pisum = 0.0

    for i in range(NumSteps):
        x=(i+0.5)*step
        pisum += 4.0/(1.0+x*x)
    pi=step*pisum
    return pi
```

This dependency is more complicated. It's called a reduction

Loop Parallelism code

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    pisum = 0.0

    with openmp ("parallel for private(x) reduction(+:pisum)"):
        for i in range(NumSteps):
            x = (i+0.5)*step
            pisum += 4.0/(1.0 + x*x)

    pi = step*pisum
    return pi

pi = piFunc(100000000)
```

OpenMP managed through the *with* context manager.

Numba Just In Time (JIT) compiler compiles the Python code into LLVM thereby bypassing the GIL. Compiled code cached for later use.

Pass the OpenMP directive into the OpenMP context manager as a string

- **parallel**: creates a team of threads
- **for**: maps loop iterations onto threads.
- **private(x)**: each threads gets its own x
- Loop control index of a parallel for (**i**) is private to each thread.
- **reduction(+:sum)**: combine sum from each thread using +

GIL: Global Interpreter Lock

Reduction

- OpenMP reduction clause added to a **parallel for**:

  reduction (op : list)

- Inside the **parallel for**:
  - Each thread gets a private copy of each variable in **list** … initialized depending on the "op"
    (e.g., 0 for "+").
  - Updates to the reduction variable from each thread happens to its private copy.
  - The private copies from each thread are combined into a single value … and then combined with the original global value … all using the **op** from the reduction clause.

- The variables in the "list" must be shared in the enclosing parallel region.

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    pisum = 0.0

    with openmp ("parallel for private(x) reduction(+:pisum)"):
        for i in range(NumSteps):
            x = (i+0.5)*step
            pisum += 4.0/(1.0 + x*x)

    pi = step*pisum
    return pi

pi = piFunc(100000000)
```

We don't discuss the details here, but you can also add a **reduction** clause to a **parallel** or a **for** construct.

Numerical Integration results in seconds … lower is better

Threads	PyOMP		C	
	Loop		Loop	
1	0.447		0.444	
2	0.252		0.245	
4	0.160		0.149	
8	0.0890		0.0827	
16	0.0520		0.0451	

10^8 steps

Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

For the C programs we used Intel® icc compiler version 19.1.3.304 as icc -qnextgen -O3 –fopenmp

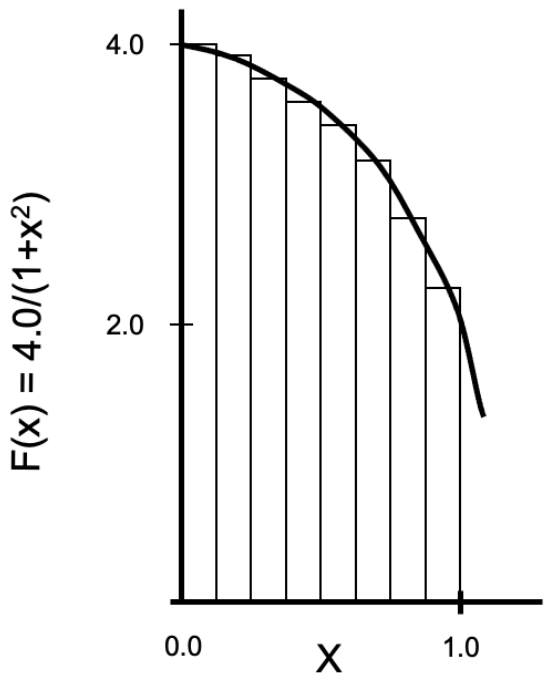
Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

Parallel Loop are great … but sometimes you want more control over individual threads

Understanding OpenMP

We will explain the key elements of OpenMP as we explore the three fundamental design patterns of OpenMP (**Loop parallelism**, **SPMD**, and **divide and conquer**) applied to the following problem

Numerical Integration (the *hello world* program of parallel computing)



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

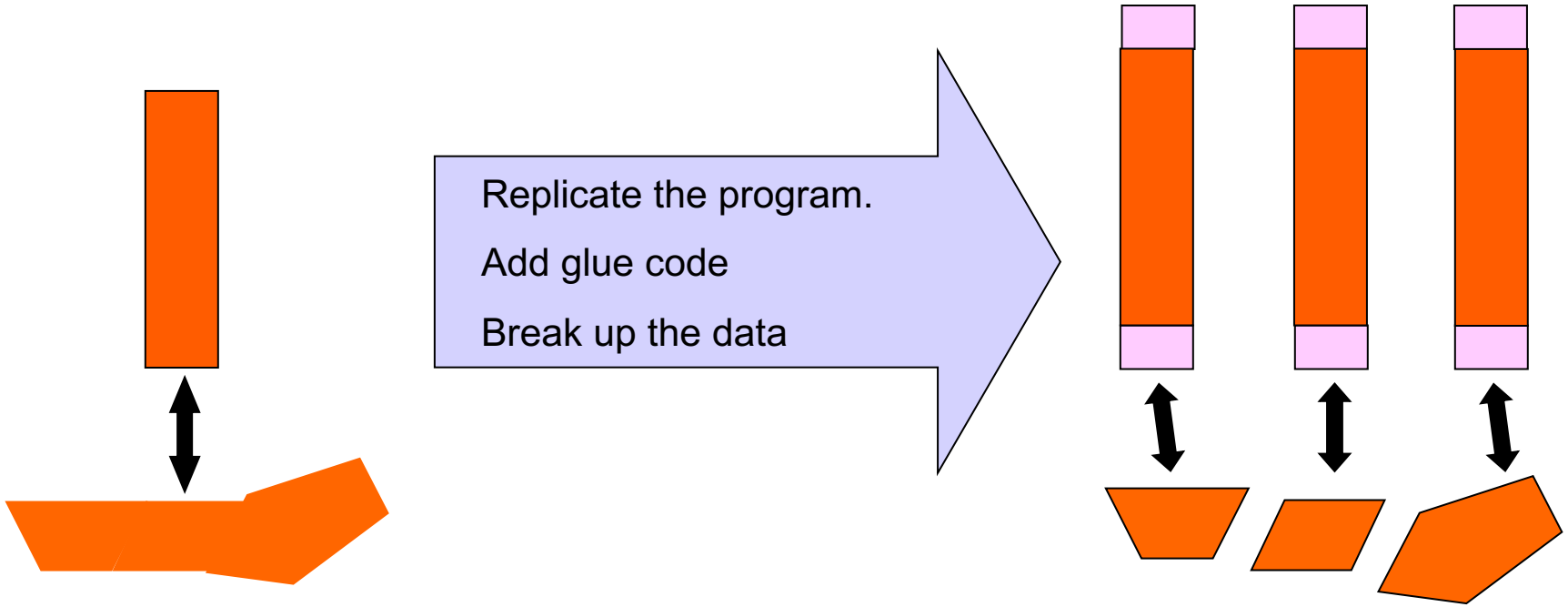
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Each rectangle: width Δx, height $F(x_i)$ at i^{th} interval midpoint.

```
def piFunc(NumSteps):
    step=1.0/NumSteps
    pisum = 0.0
    x = 0.5
    for i in range(NumSteps):
        x+=step
        pisum += 4.0/(1.0+x*x)
    pi=step*pisum
    return pi
```

SPMD (Single Program Multiple Data) design pattern

- Run the same program on P processing elements where P can be arbitrarily large.
- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.



This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

Single Program Multiple Data (SPMD)

```
from numba import njit
import numpy as np
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_thread_num, omp_get_num_threads
MaxTHREADS = 32
@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    partialSums = np.zeros(MaxTHREADS)
    with openmp("parallel shared(partialSums,numThrds) private(threadID,i,x,localSum)"):
        threadID = omp_get_thread_num()
        with openmp("single"):
            numThrds = omp_get_num_threads()
        localSum = 0.0
        for i in range(threadID, NumSteps, numThrds):
            x = (i+0.5)*step
            localSum = localSum + 4.0/(1.0 + x*x)
        partialSums[threadID] = localSum
    return step*np.sum(partialSums)

pi = piFunc(100000000)
```

- **omp_get_num_threads()**: get N=number of threads
- **omp_get_thread_num()**: thread rank = 0…(N-1)
- **single**: One thread does the work, others wait
- **private(x)**: each threads gets its own x
- **shared(x)**: all threads see the same x

Deal out loop iterations as if a deck of cards (a cyclic distribution) … each threads starts with the Iteration = ID, incremented by the number of threads, until the whole "deck" is dealt out.

The data environment seen by OpenMP threads

- The data environment is the collection of variables visible to the threads in a team.

- Variables can be **shared** or **private**.
  - **Shared variable**: A variable that is visible (i.e. can be read or written) to all threads in a team.
  - **Private variable**: A variable that is only visible to an individual thread.

- All the code associated with an OpenMP directive (such as **parallel** or **for**), including the code in functions called inside that code, is called a **region**. A directive plus code in the immediate block associated with it, is called a **construct**

- Rules for defining a variable as shared or private:
  - A variable is **shared** if it is used before or after an OpenMP construct, otherwise it is **private**.
  - Variables can be made shared or private through clauses included with a directive.

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    pisum = 0.0
    with openmp ("parallel for reduction(+:pisum)"):
        for i in range(NumSteps):
            x = (i+0.5)*step
            pisum += 4.0/(1.0 + x*x)

    pi = step*pisum
    return pi

pi = piFunc(100000000)
```

x first used inside the OpenMP construct … it is private.

Numerical Integration results in seconds … lower is better

Threads	PyOMP		C	
	Loop	SPMD	Loop	SPMD
1	0.447	0.450	0.444	0.448
2	0.252	0.255	0.245	0.242
4	0.160	0.164	0.149	0.149
8	0.0890	0.0890	0.0827	0.0826
16	0.0520	0.0503	0.0451	0.0451

10^8 steps

Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

For the C programs we used Intel® icc compiler version 19.1.3.304 as icc -qnextgen -O3 –fiopenmp

Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

How do we handle problems without such regular structure or with complex load balancing problems?

We do this in OpenMP with explicit tasks

Explicit tasks in PyOMP

- A task is a sequence of statements and an associated data environment. Lots of flexibility in how those tasks are created, so handles irregular parallelism, recursive parallelism, and many other control structures.

- A common pattern … one thread creates explicit tasks and puts them in a queue. All the threads work together to execute them. The single construct causes one thread to execute statements while the other threads wait at a barrier at the end of the single. It's perfect for task level parallelism.

```
from numba import njit
from numba.openmp import openmp_context as openmp

@njit
def irregularPar():
    with openmp("parallel"):
       with openmp("single"):
            StateVal = 1
            while (StateVal > 0):
                with openmp("task firstprivate(StateVal)"):
                    BigComp(StateVal)
                StateVal = ExitYet()
    return

irregularPar()
```

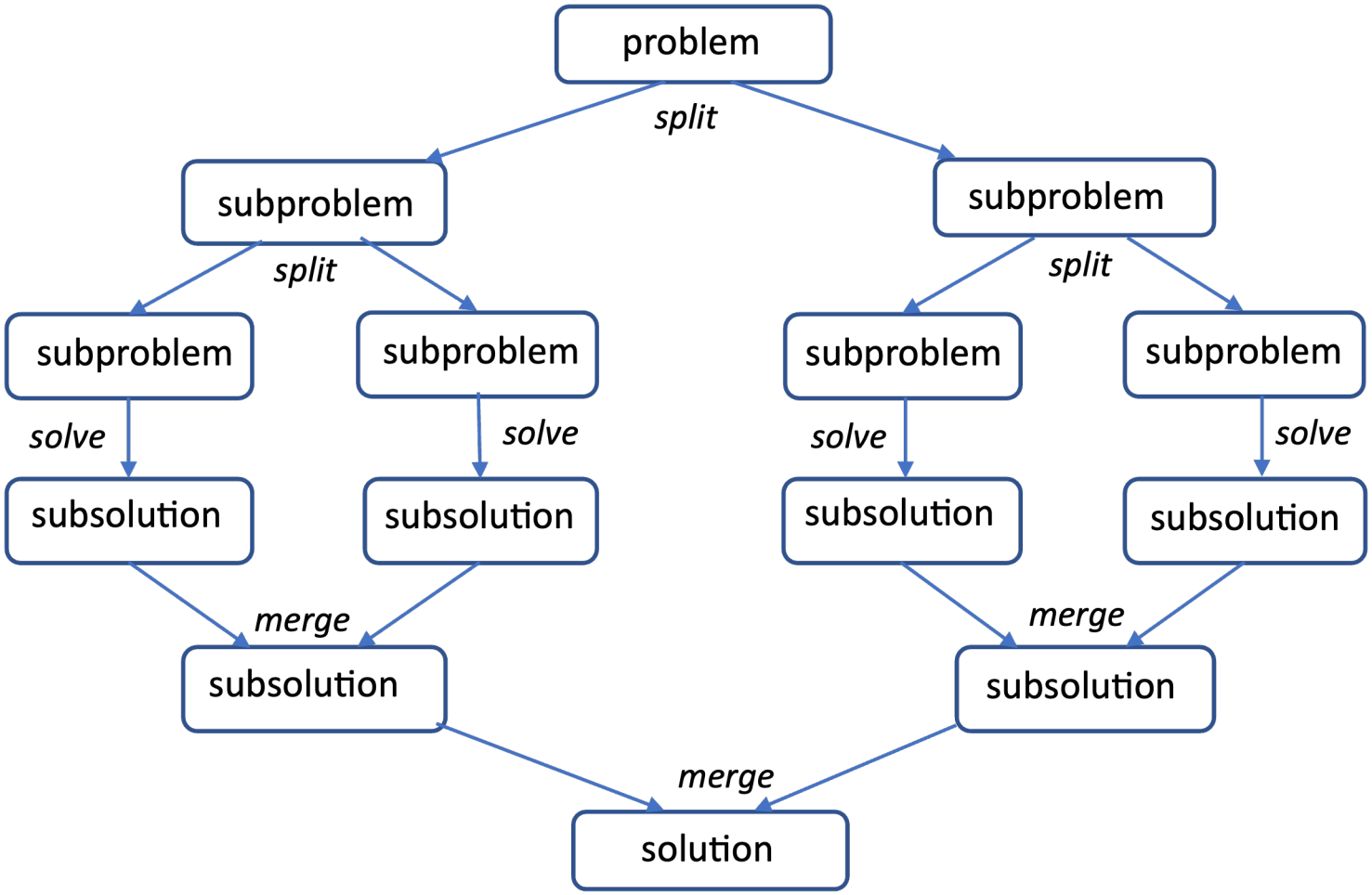
Single: one thread does the work while the other threads wait (and execute tasks) at the barrier implied at the end of single

An explicit task … captures value of the variable StateVal and calls BigComp.

Returns a negative value at some point (function not shown)

Divide and conquer design pattern

- Split the problem into smaller sub-problems; continue until the sub-problems can be solved directly



3 Options for parallelism:

- Do work as you split into sub-problems
- Do work at the leaves
- Do work as you recombine

Divide and conquer (with explicit tasks)

```
from numba import njit
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_num_threads, omp_set_num_threads
MIN_BLK = 1024*256
@njit
def piComp(Nstart, Nfinish, step):
    iblk = Nfinish-Nstart
    if(iblk<MIN_BLK):
        pisum = 0.0
        for i in range(Nstart,Nfinish):
            x= (i+0.5)*step
            pisum += 4.0/(1.0 + x*x)
    else:
        sum1 = 0.0
        sum2 = 0.0
        with openmp ("task shared(sum1)"):
            sum1 = piComp(Nstart, Nfinish-iblk/2,step)
        with openmp ("task shared(sum2)"):
            sum2 = piComp(Nfinish-iblk/2,Nfinish,step)
        with openmp ("taskwait"):
            pisum = sum1 + sum2
    return pisum
```

Solve

Split

Merge

```
@njit
def piFunc(NumSteps):
    step = 1.0/NumSteps
    sum = 0.0
    startTime = omp_get_wtime()
    with openmp ("parallel"):
        with openmp ("single"):
            pisum = piComp(0,NumSteps,step)

    pi = step*pisum
    return pi

pi = piFunc(100000000)
```

Fork threads and launch the computation

- **single**: One thread does the work, others wait
- **task**: code block enqueued for execution
- **taskwait**: wait until task in the code block finish

Numerical Integration results in seconds … lower is better

Threads	PyOMP			C		
	Loop	SPMD	Task	Loop	SPMD	Task
1	0.447	0.450	0.453	0.444	0.448	0.445
2	0.252	0.255	0.245	0.245	0.242	0.222
4	0.160	0.164	0.146	0.149	0.149	0.131
8	0.0890	0.0890	0.0898	0.0827	0.0826	0.0720
16	0.0520	0.0503	0.0517	0.0451	0.0451	0.0431

10^8 steps

Intel® Xeon® E5-2699 v3 CPU with 18 cores running at 2.30 GHz.

For the C programs we used Intel® icc compiler version 19.1.3.304 as icc -qnextgen -O3 –fiopenmp

Ran each case 5 times and kept the minimum time. **JIT time is not included** for PyOMP (it was about 1.5 seconds)

There is more …. But this is enough to get you started with CPU programming in PyOMP

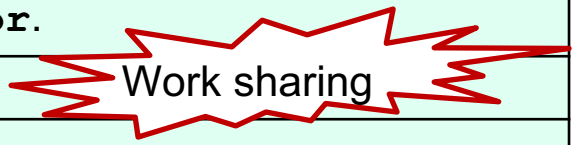
So let's wrap up our discussion of CPU programming

PyOMP subset of OpenMP for CPU programming

`with openmp("parallel"):`	Create a team of threads. Execute a parallel region
`with openmp("for"):`	Use inside a parallel region. Split up a loop across the team.
`with openmp("parallel for"):`	A combined construct. Same a **`parallel`** followed by a **`for`**.
`with openmp ("single"):`	One thread does the work. Others wait for it to finish
`with openmp("task"):`	Create an explicit task for work within the construct.
`with openmp("taskwait"):`	Wait for all tasks in the current task to complete.
`with openmp("barrier"):`	All threads arrive at a barrier before any proceed.
`with openmp("critical"):`	Mutual exclusion. One thread at a time executes code
`schedule(static [,chunk])`	Map blocks of loop iterations across the team. Use with **`for`**.
`reduction(op:list)`	Combine values with op across the team. Used with **`for`**
`private(list)`	Make a local copy of variables for each thread. Use with **`parallel`**, **`for`** or **`task`**.
`firstprivate(list)`	**`private`**, but initialize with original value. Use with **`parallel`**, **`for`** or **`task`**
`shared(list)`	Variables shared between threads. Use with **`parallel`**, **`for`** or **`task`**.
`default(none)`	Force definition of variables as **`private`** or **`shared`**.
`omp_get_num_threads()`	Return the number of threads in a team
`omp_get_thread_num()`	Return an ID from 0 to the number of threads minus one
`omp_set_num_threads(int)`	Set the number of threads to request for parallel regions
`omp_get_wtime()`	Return a snapshot of the wall clock time.
`OMP_NUM_THREADS=N`	Environment variable to set the default number of threads

PyOMP subset of OpenMP for CPU programming

`with openmp("parallel"):`	Create a team of threads. Execute a parallel region
`with openmp("for"):`	Use inside a parallel region. Split up a loop across the team.
`with openmp("parallel for"):`	A combined construct. Same a **`parallel`** followed by a **`for`**.
`with openmp ("single"):`	One thread does the work. Others wait for it to finish
`with openmp("task"):`	Create an explicit task for work within the construct.
`with openmp("taskwait"):`	Wait for all tasks in the current task to complete.
`with openmp("barrier"):`	All threads arrive at a barrier before any proceed.
`with openmp("critical"):`	Mutual exclusion. One thread at a time executes code
`schedule(static [,chunk])`	Map blocks of loop iterations across the team. Use with **`for`**.
`reduction(op:list)`	Combine values with op across the team. Used with **`for`**
`private(list)`	Make a local copy of variables for each thread. Use with **`parallel`**, **`for`** or **`task`**.
`firstprivate(list)`	**`private`**, but initialize with original value. Use with **`parallel`**, **`for`** or **`task`**
`shared(list)`	Variables shared between threads. Use with **`parallel`**, **`for`** or **`task`**
`default(none)`	Force definition of variables as **`private`** or **`shared`**.
`omp_get_num_threads()`	Return the number of threads in a team
`omp_get_thread_num()`	Return an ID from 0 to the number of threads minus one
`omp_set_num_threads(int)`	Set the number of threads to request for parallel regions
`omp_get_wtime()`	Return a snapshot of the wall clock time.
`OMP_NUM_THREADS=N`	Environment variable to set the default number of threads



The view of Python from an HPC perspective

```
for I in range(4096):
    for j in range(4096):
        for k in range (4096):
            C[i][j] += A[i][k]*B[k][j]
```

We know better … the IKJ order is more cache friendly

And we picked a smaller problem

```
for I in range(1000):
    for k in range(1000):
        for j in range (1000):
            C[i][j] += A[i][k]*B[k][j]
```

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

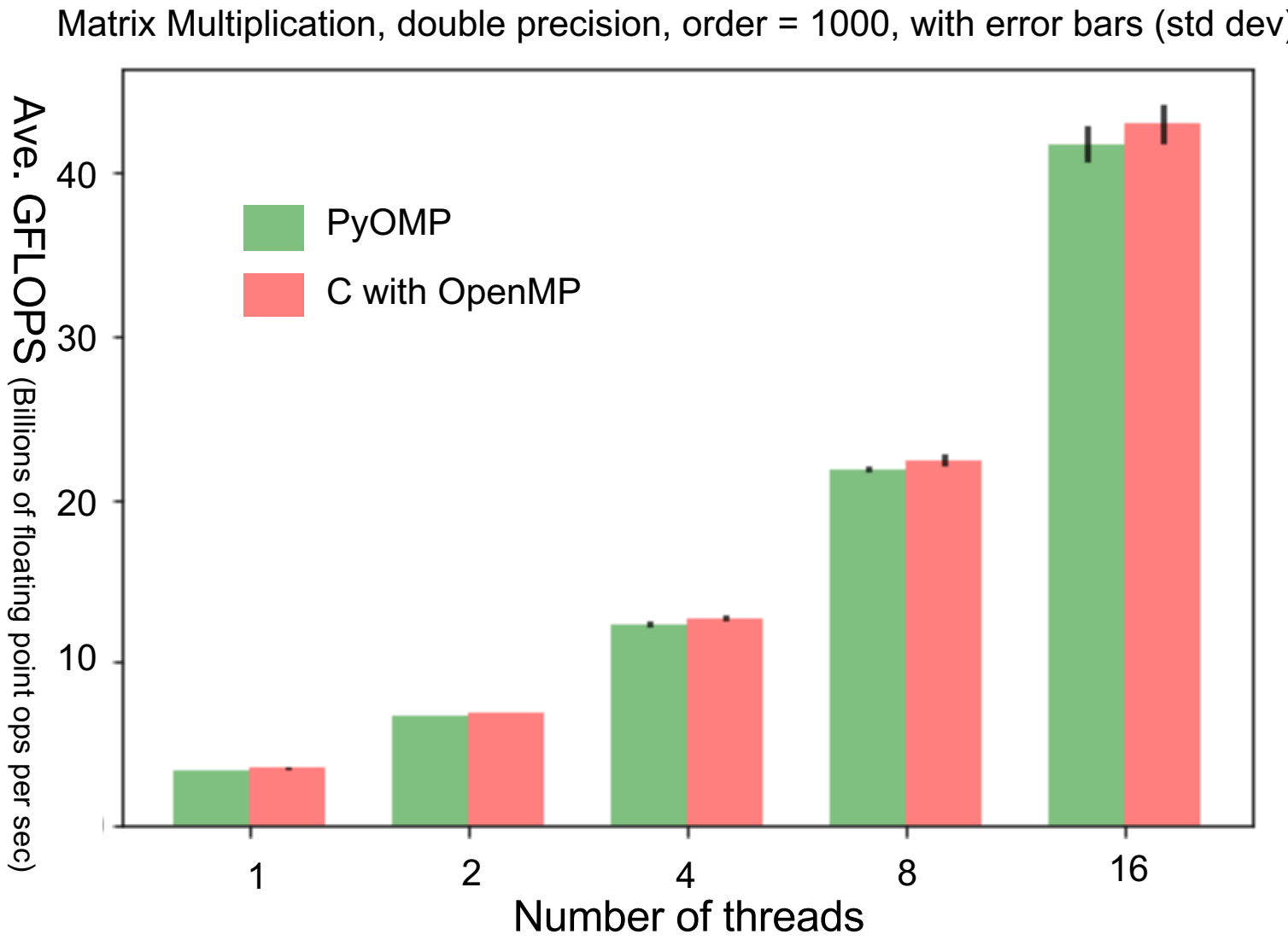
Amazon AWS c4.8xlarge spot instance, Intel® Xeon® E5-2666 v3 CPU, 2.9 Ghz, 18 core, 60 GB RAM

PyOMP DGEMM (Mat-Mul with double precision numbers)

```
from numba import njit
import numpy as np
from numba.openmp import openmp_context as openmp
from numba.openmp import omp_get_wtime

@njit(fastmath=True)
def dgemm(iterations,order):

    # allocate and initialize arrays
    A = np.zeros((order,order))
    B = np.zeros((order,order))
    C = np.zeros((order,order))

    # Assign values to A and B such that
    # the product matrix has a known value.
    for i in range(order):
        A[:,i] = float(i)
        B[:,i] = float(i)
```

```
tInit = omp_get_wtime()
with openmp("parallel for private(j,k)"):
      for i in range(order):
          for k in range(order):
              for j in range(order):
                  C[i][j] += A[i][k] * B[k][j]

dgemmTime = omp_get_wtime() - tInit

# Check result
checksum = 0.0;
for i in range(order):
    for j in range(order):
        checksum += C[i][j]
ref_checksum = order*order*order
ref_checksum *= 0.25*(order-1.0)*(order-1.0)
eps=1.e-8
if abs((checksum - ref_checksum)/ref_checksum) < eps:
    print('Solution validates')
    nflops = 2.0*order*order*order
    print('Rate (MF/s): ',1.e-6*nflops/dgemmTime)
```

DGEMM PyOMP vs C-OpenMP



250 runs for order 1000 matrices

PyOMP times **DO NOT** include the one-time JIT cost of ~2 seconds.

… but remember, the JIT'ed code can be cached for future use. It's straightforward to hide the JIT cost.

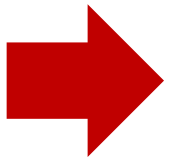
Intel® Xeon® E5-2699 v3 CPU, 18 cores, 2.30 GHz, threads mapped to a single CPU, one thread/per core, first 16 physical cores.
Intel® icc compiler ver 19.1.3.304 (icc –std=c11 –pthread –O3 xHOST –qopenmp)

And we can use PyOMP for GPU programming

The "BIG idea" Behind GPU programming

Traditional Loop based vector addition (vadd)

```
int main() {
   int N = . . . ;
   float *a, *b, *c;

   a* =(float *) malloc(N * sizeof(float));

   // ... allocate other arrays (b and c)
   // and fill with data

   for (int i=0;i<N; i++)
      c[i] = a[i] + b[i];

}
```



Data Parallel vadd with CUDA

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

Assume a GPU with unified shared memory … allocate on host, visible on device too

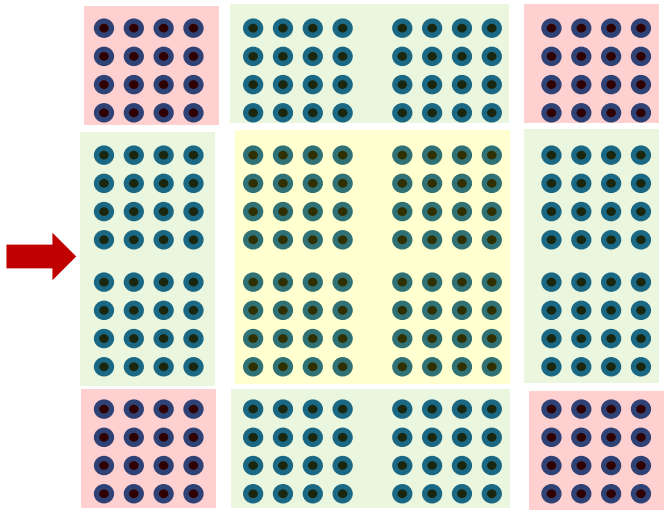
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Turn source code into a scalar work-item

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

This is CUDA code … the sort of code the OpenMP compiler generates on your behalf

2. Map work-items onto an N dim index space.



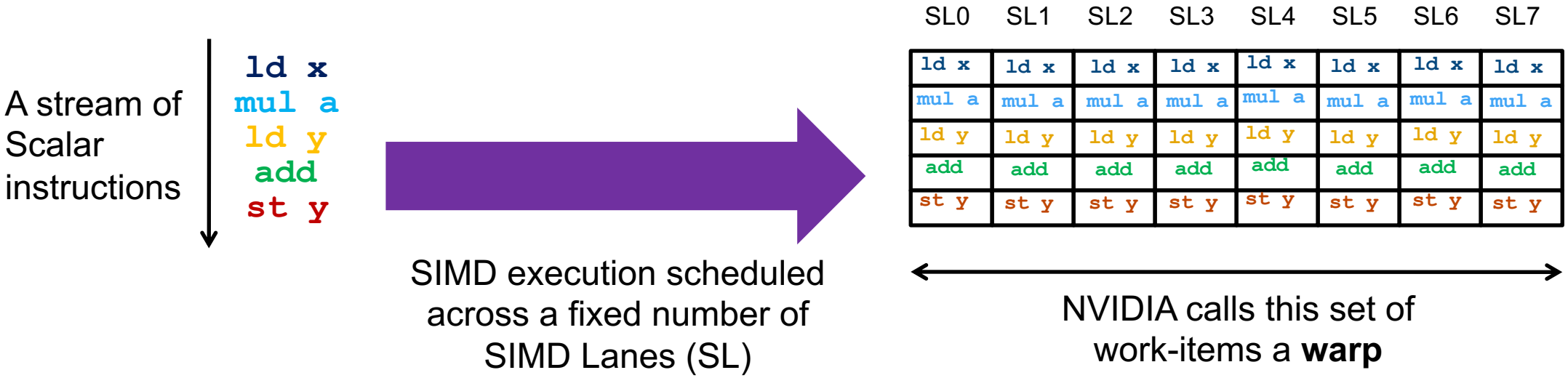
3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model

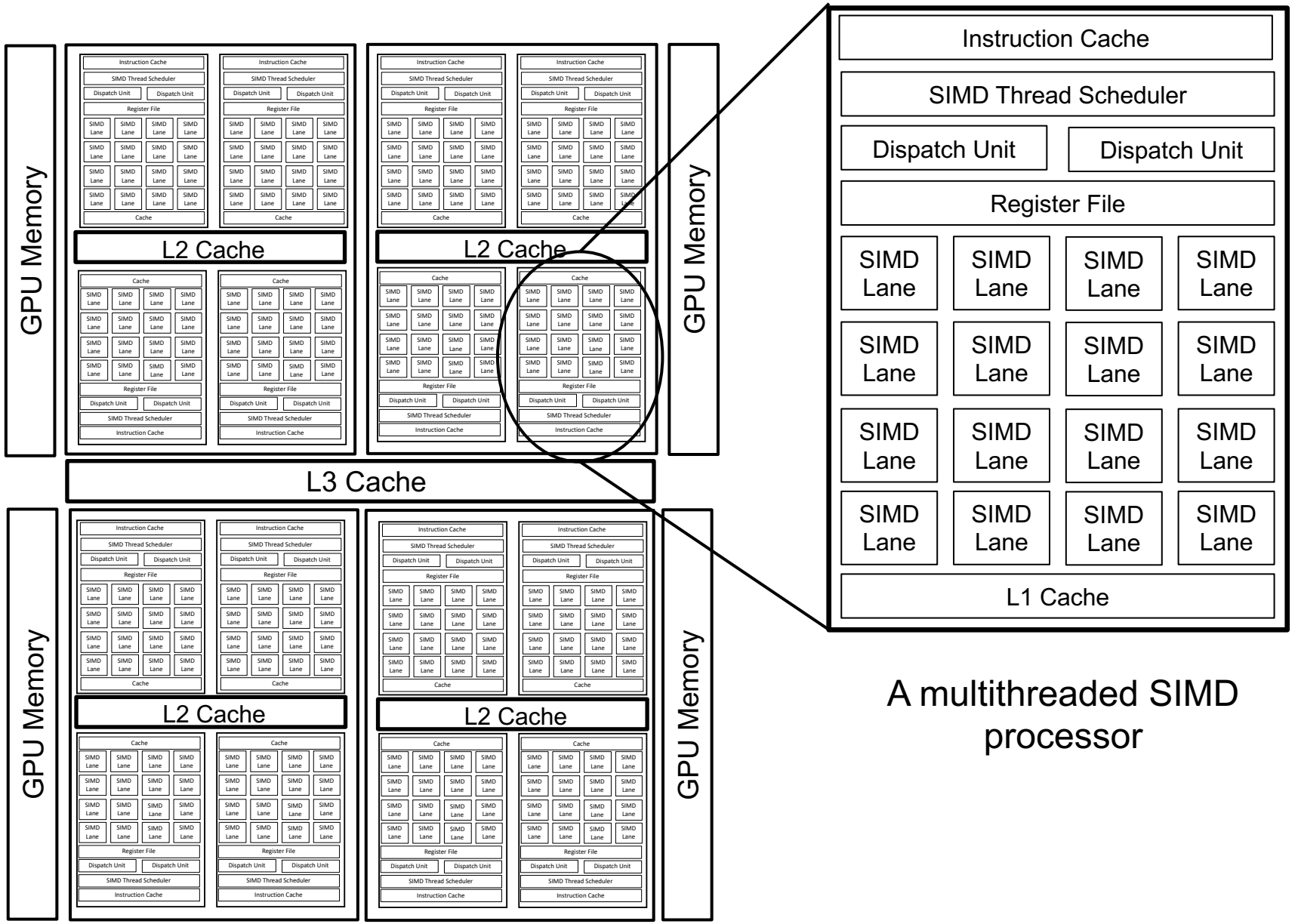
Note: The CUDA code defines a 1D grid. I show a 2D grid on this slide to make kernel execution and its relation to data more clear.

SIMT: One instruction stream maps onto many SIMD lanes

- SIMT model: Individual scalar instruction streams are grouped together for SIMD execution on hardware

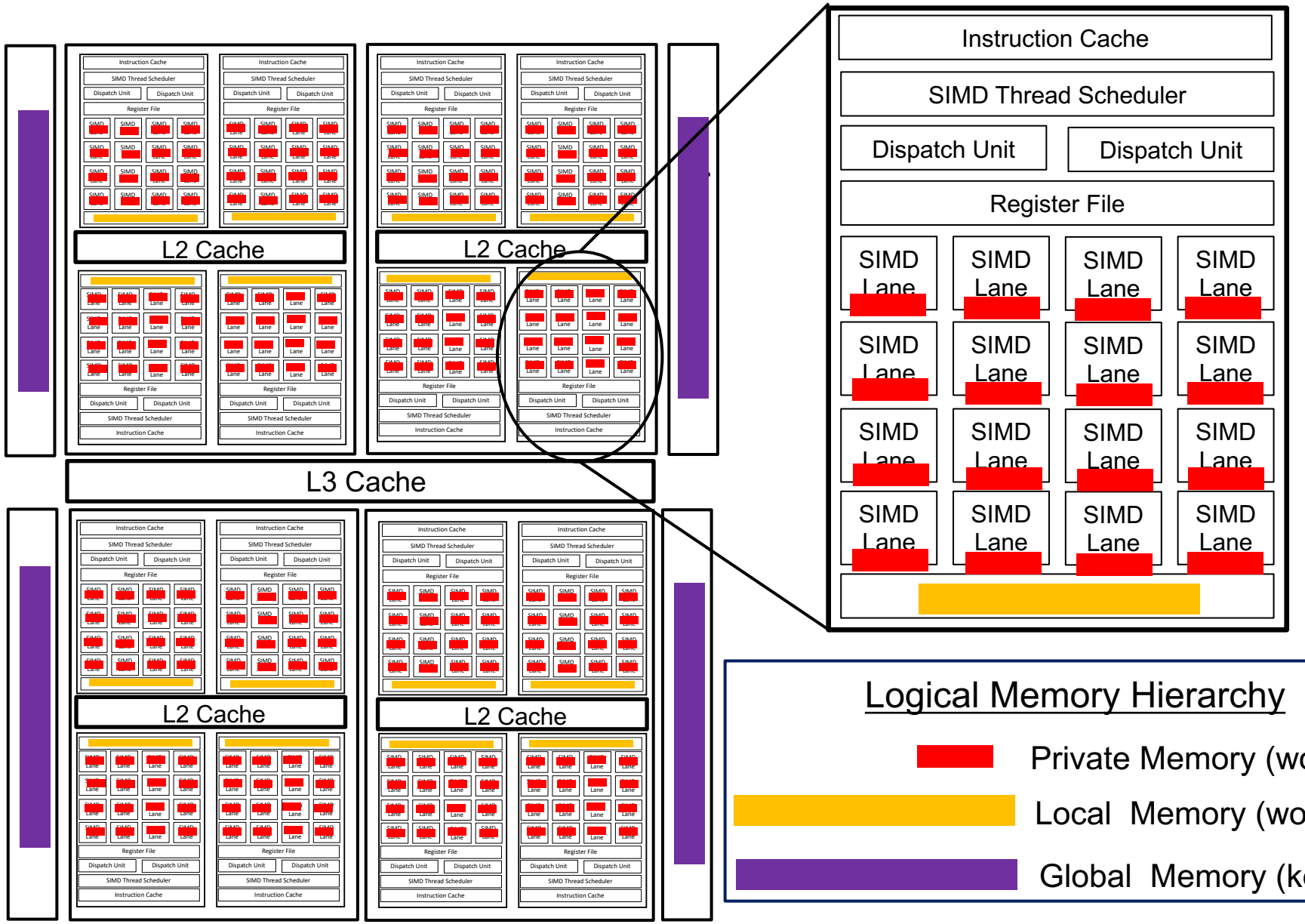


A Generic GPU (following Hennessey and Patterson)

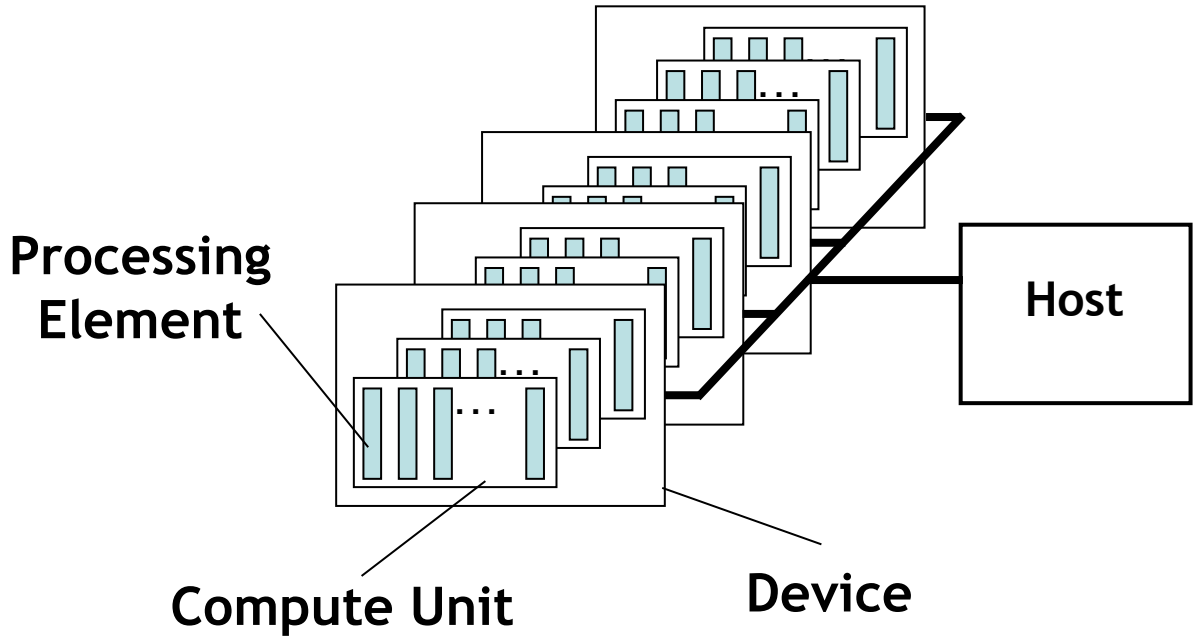


A multithreaded SIMD processor

A Generic GPU (following Hennessey and Patterson)



A Generic Host/Device Platform Model

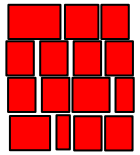


- One ***Host*** and one or more ***Devices***
  - Each Device is composed of one or more Compute Units
  - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

PE: processing element. The finest-grained processing element inside a GPU. Also known as a SiMD-lane or CUDA-core.

Executing a program on CPUs and GPUs

Work decomposed into blocks



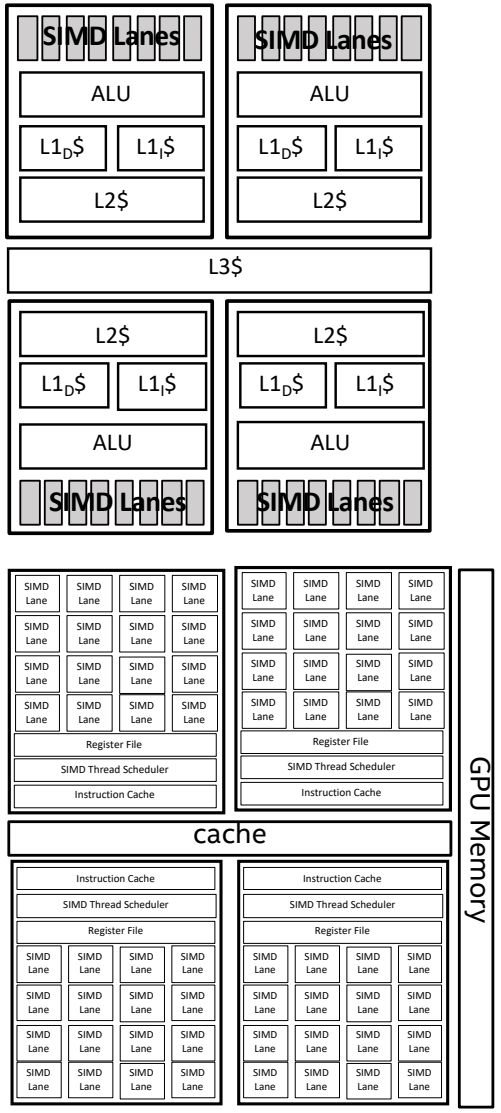
For a CPU

For a GPU

Program defines work

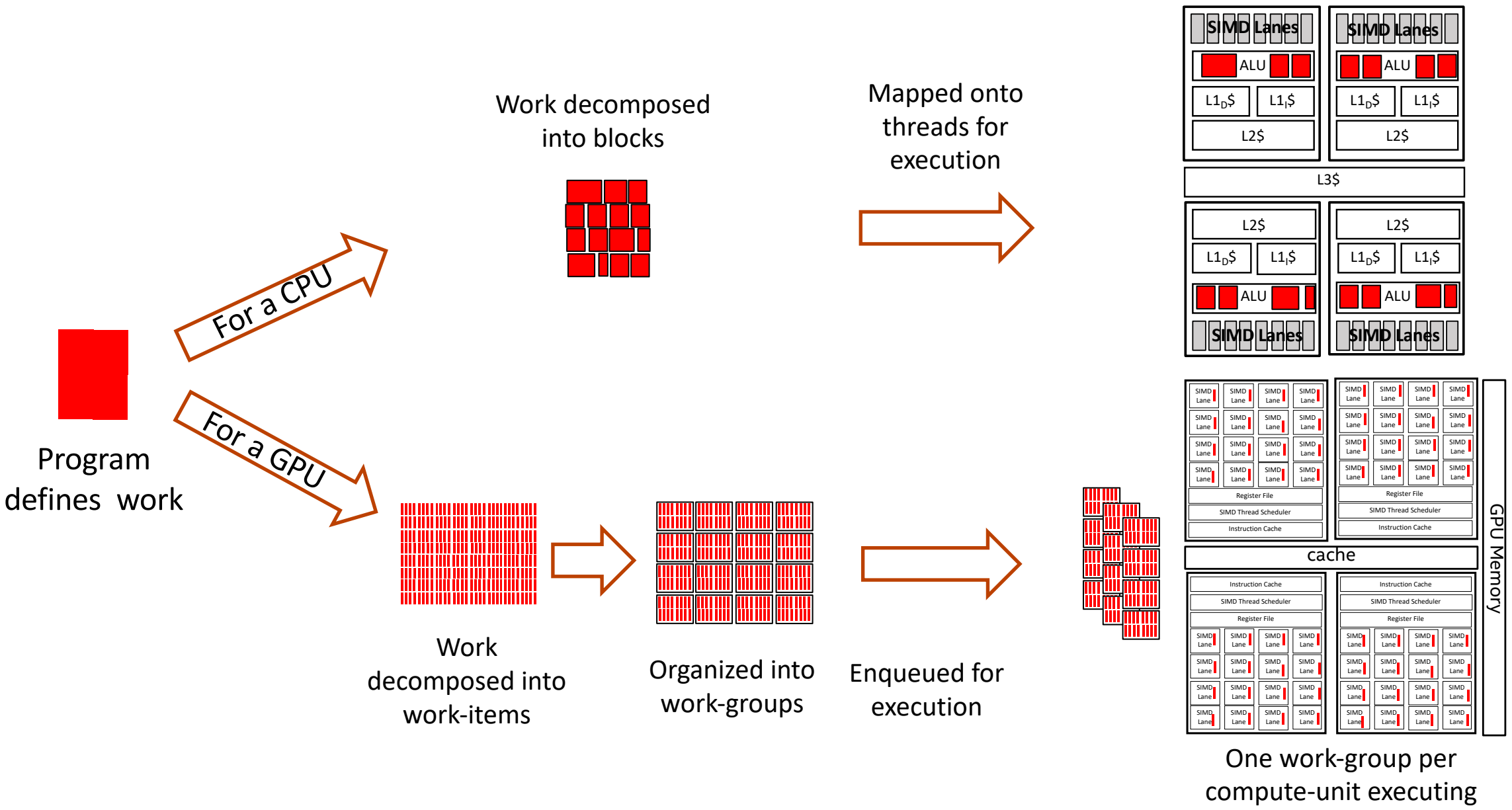
Work decomposed into work-items

Organized into work-groups

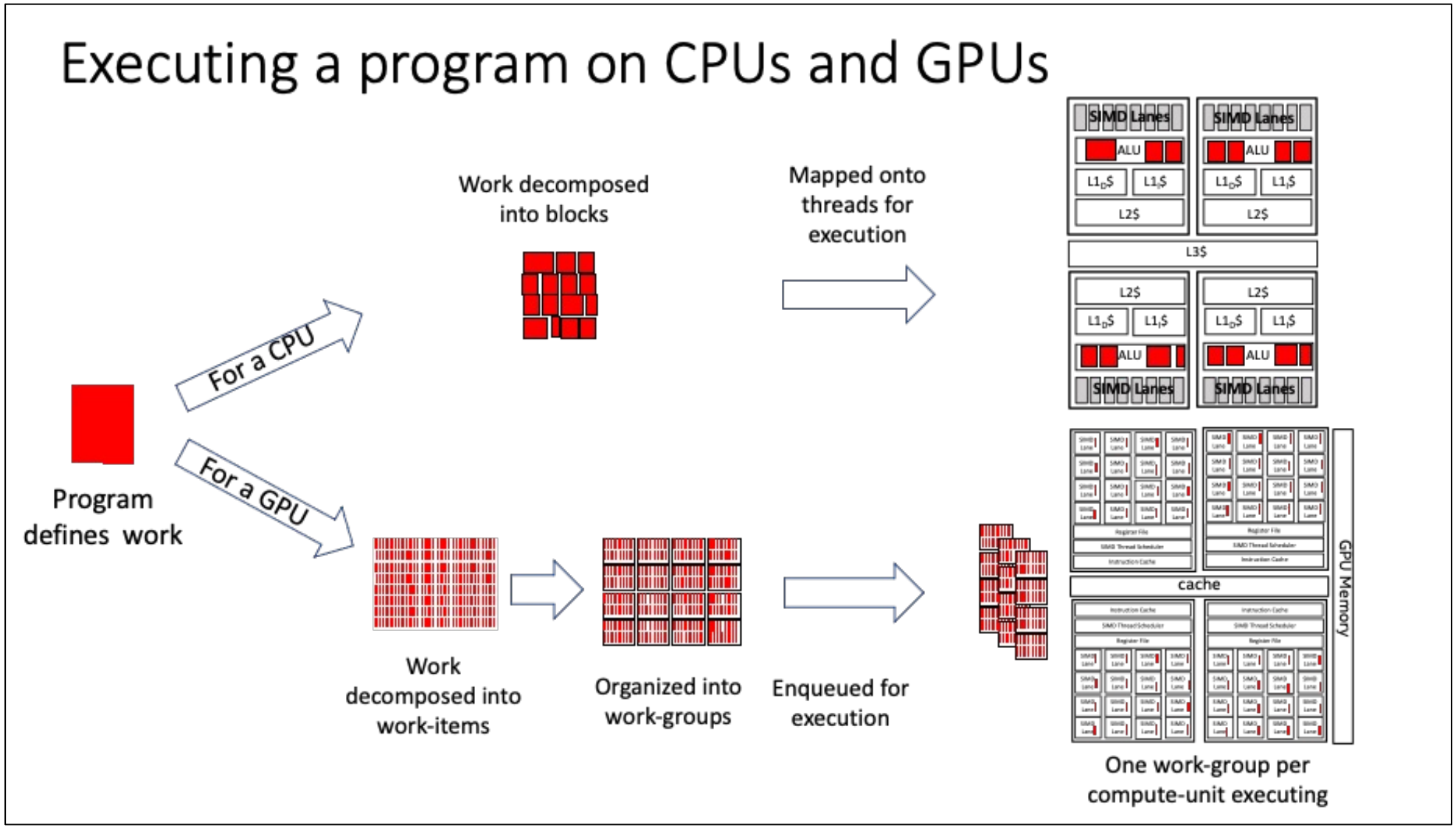


One work-group per compute-unit executing

Executing a program on CPUs and GPUs



CPU/GPU execution models



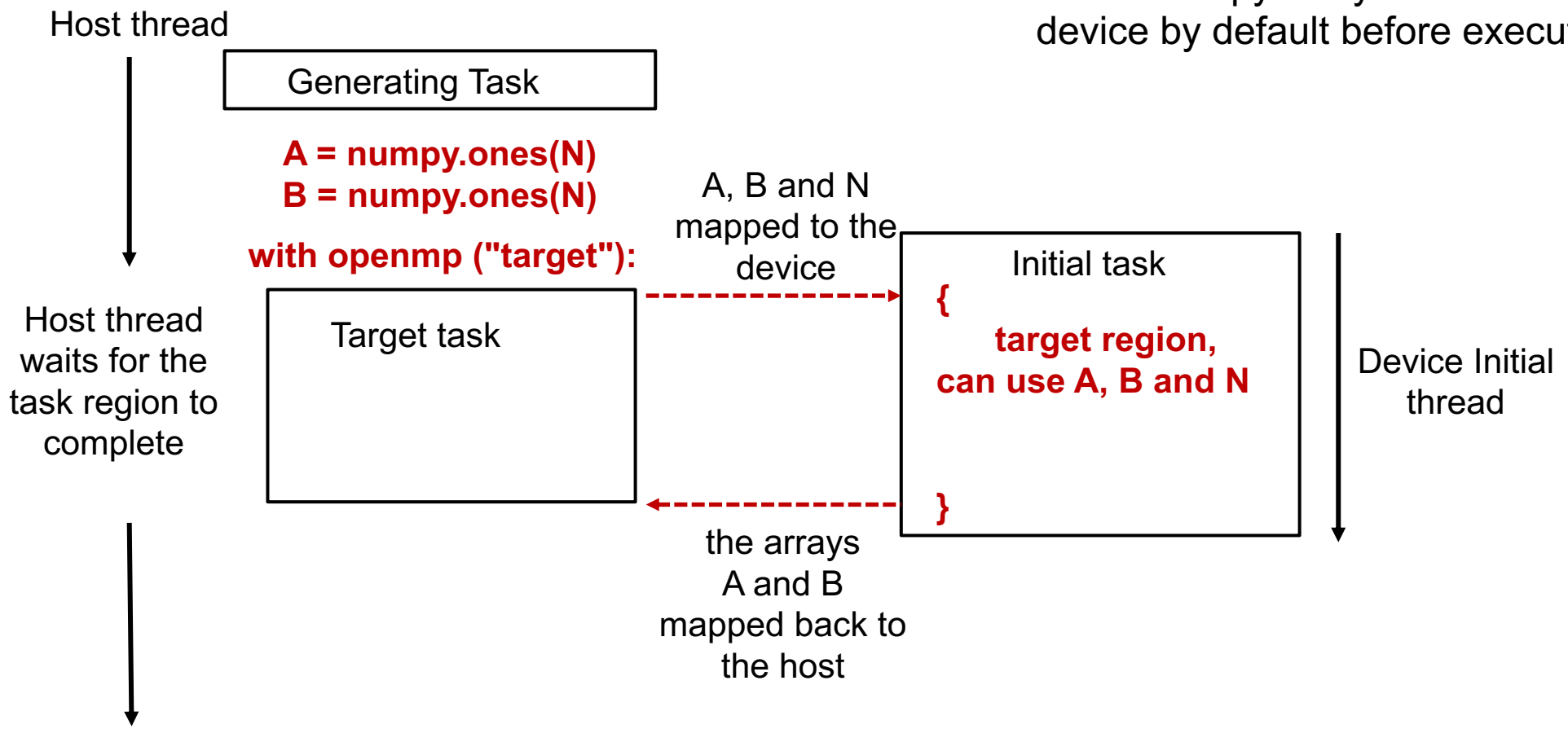
For a CPU, the threads are all active and able to make forward progress.

For a GPU, any given work-group might be in the queue waiting to execute.

How do we map a loop onto the GPU execution model in PyOMP?

Step 1: move code and data onto the GPU:
The target construct and default data movement

Scalars and numpy arrays are moved onto the device by default before execution.



Only the arrays are moved back to the host after the target region completes

Based on figure 6.4 in **Using OpenMP – The Next Step** by van der Pas, Stotzer and Terboven, MIT Press, 2017

Step 2: Map loop iterations onto the GPU's SIMD lanes

```
@njit
def main():
    N = 1024
    A = numpy.ones(N)
    B = numpy.ones(N)

    with openmp ("target "):
        with openmp ("loop"):
            for i in range(N):
                A[i] += B[i]
```

The loop construct tells the compiler:

"this loop will execute correctly if the loop iterations run in any order. You can safely run them concurrently. And the loop-body doesn't contain any OpenMP constructs. So do whatever you can to make the code run fast"

The loop construct is a declarative construct. You tell the compiler what you want done but you DO NOT tell it how to "do it". This is new for OpenMP

Step 2: Map loop iterations onto the GPU's SIMD lanes

```
@njit
def main():
    N = 1024
    A = numpy.ones(N)
    B = numpy.ones(N)

    with openmp ("target "):
        with openmp ("loop"):
            for i in range(N):
                A[i] += B[i]
```

1. Variables created in host memory.

2. Scalar **N** and arrays **A** and **B** are copied *to* device memory. Execution transferred to device.

3. For-loop index variables (such as **i**) are **private** in openmp regions

4. Loop iterations define the index space, work-items, and work-groups.

5. After the OpenMP construct, arrays **A** and **B** are copied *from* device memory back to the host. Host resumes execution.

Difference from OpenMP/C: PyOMP only has NumPy arrays, which carry size information. So, PyOMP arrays sent in full by default ... as it is with C static-arrays.

Loop Parallelism code naturally maps onto the CPU

```
from numba import njit
import numpy as np
from numba.openmp import openmp_context as openmp

@njit(fastmath=True)
def dgemm(iterations,N):

    # allocate and initialize numpy arrays
    # A, B and C of size N by N.   <<< code not shown>>>

    with openmp("parallel for private(j,k)"):
        for i in range(N):
            for k in range(N):
                for j in range(N):
                    C[i][j] += A[i][k] * B[k][j]
```

OpenMP constructs managed through the *with* context manager.

Create a team of threads. Map loop iterations onto them

- **parallel**: creates a team of threads
- **for**: maps loop iterations onto threads.
- **private(j,k)**: each threads gets its own j and k variables
- Loop control index of a parallel for (**i**) is private to each thread.

Loop Parallelism code naturally maps onto the CPU

```
from numba import njit
import numpy as np
from numba.openmp import openmp_context as openmp

@njit(fastmath=True)
def dgemm(iterations,N):

    # allocate and initialize numpy arrays
    # A, B and C of size N by N.   <<< code not shown>>>

    with openmp("target teams loop collapse(2) private(j)"):
        for i in range(N):
            for k in range(N):
                for j in range(N):
                    C[i][j] += A[i][k] * B[k][j]
```

OpenMP constructs managed through the *with* context manager.

Map the loop onto a 2D index space … the loop body defines the kernel function

- **target**: map execution from the host onto the device
- **teams loop**: Map kernel instances onto PEs inside the compute units
- **collapse(2)**: combine following two loops into a single iteration space.
- **private(j)**: each threads gets its own j variable
- Indices of parallelized loops (**i,k**) are private to each thread.

PE: processing element. The finest-grained processing element inside a GPU. Also known as a SiMD lane or CUDA-core.

Implicit data movement covers a small subset of the cases you need in a real program.

To be more general … we need to manage data movement explicitly

Implicit data movement

- Previously, we described the rules for *implicit* data movement … **N**, **A** and **B** moved to the GPU on entry to the target construct. **A** and **B** moved to the CPU on exit from the target construct.

- Notice that in this case, **B** is not changed on the GPU … moving it is a waste of resources

```
@njit
def main():
    N = 1024
    A = numpy.ones(N)
    B = numpy.ones(N)

    with openmp ("target"):
        for i in range(N):
            A[i] += B[i]
```

Controlling data movement with the map clause

```
@njit
def main():
    N = 1024
    A = numpy.ones(N)
    B = numpy.ones(N)

    with openmp ("target map(tofrom: A) map(to: B)"):
        for i in range(N):
            A[i] += B[i]
```

map(tofrom: A) Map data at the start and end of **target** region.

map(to: B) map data at the start of **target** region but NOT at the end.

We use the term “map” since depending on the detailed memory architecture of the CPU and the GPU, data may be in a shared address space so copying may not be needed.

PyOMP array notation

- When mapping data arrays, if you only give the array name then PyOMP transfers the entire array (using the NumPy array metadata to determine the size)

- To transfer less than the full array, the array section syntax can be used
  - **array_name[begin:end]**
  - This follows Python/NumPy slicing syntax where **begin** is inclusive but **end** is exclusive.

    A[N:M]. In set notation implies elements [N:M)
  - Multi-dimensional arrays work as expected when transferred in full. Currently PyOmp doesn't support array-section syntax for multi-dimensional arrays.

C Difference: In C, arrays are usually dynamically allocated and referenced through a pointer. You must use array-section syntax to move data. In C, array-syntax is "(initial-offset: number-of-items)". Fortran uses "begin:end" syntax (as Python does), but the ending index is inclusive (i.e., [begin:end]).

Controlling data movement: the map clause

- **map(to:list)**: On entering the region, variables in the list are initialized on the device using the original values from the host (host to device copy).
- **map(from:list)**: At the end of the target region, the values from variables in the list are copied into the original variables on the host (device to host copy). On entering the region, the initial value of the variables on the device is not initialized.
- **map(tofrom:list)**: the effect of both a map-to and a map-from (host to device copy at start of region, device to host copy at end).
- **map(alloc:list)**: On entering the region, data is allocated and uninitialized on the device.
- **map(list)**: equivalent to **map(tofrom:list)**.

Note: Data movement is defined from the perspective of the host.

```
@njit
def main():
    a = numpy.ones(N)
    b = numpy.ones(N)
    c = numpy.empty(N)
    with openmp ("target teams loop map(to: a,b) map(tofrom: c)"):
        for i in range(N):
                c[i] = a[i] + b[i]
```

When applied to an array, the mapping mode applies only to the array's data. Array metadata is always transferred as **to** and no operations which would change the metadata (e.g., resize) are permitted.

Going beyond simple vector addition …

Using OpenMP for GPU application programming … the heat diffusion problem

5-point stencil: the heat program

- The heat equation models changes in temperature over time.

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$

- We'll solve this numerically on a computer using an explicit **finite difference** discretisation.
- $u = u(t, x, y)$ is a function of space and time.
- Partial differentials are approximated using diamond difference formulae:

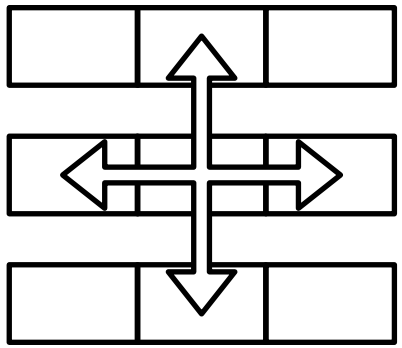
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t, x+1, y) - 2u(t, x, y) + u(t, x-1, y)}{dx^2}$$

  - Forward finite difference in time, central finite difference in space.

5-point stencil: the heat program

- Given an initial value of u, and any boundary conditions, we can calculate the value of u at time t+1 given the value at time t.
- Each update requires values from the north, south, east and west neighbours only:



- Computation is essentially a weighted average of each cell and its neighbouring cells.
- If on a boundary, look up a boundary condition instead.

Heat diffusion problem …

```
# Loop over time steps
for _ in range(nsteps):
    # solve over spatial domain for step t
    solve(n, alpha, dx, dt, u, u_tmp)

    # Array swap to get ready for next step
    u, u_tmp = u_tmp, u
```

Array-swap on the host works. Why?

u and u_tmp are references to structs that hold NumPy metadata and a data pointer.

The OpenMP runtime creates a device struct at the target enter data construct and maintains a fixed association between host and device struct references.

Hence, as you swap the array variables, the references to the struct addresses in device memory are swapped.

5-point stencil: solve kernel

```
@njit
def solve(n, alpha, dx, dt, u, u_tmp):
    # Finite difference constant multiplier
    r = alpha * dt / (dx ** 2)
    r2 = 1 - 4 * r
    # Loop over the nxn grid
        for i in range(n):
            for j in range(n):
                # Update the 5-point stencil.
                # Using boundary conditions on the edges of the domain.
                # Boundaries are zero because the MMS solution is zero there.
                u_tmp[j, i] = (r2 * u[j, i] +
                               (u[j, i+1] if i < n-1 else 0.0) +
                               (u[j, i-1] if i > 0   else 0.0) +
                               (u[j+1, i] if j < n-1 else 0.0) +
                               (u[j-1, i] if j > 0 else 0.0))
```

25,000x25,000 grid for 10 time steps
* Xeon Platinum 8480+: 67.6 secs

Solution: parallel stencil (heat)

25,000x25,00 grid for 10 time steps
- Xeon Platinum 8480+: 67.6 secs
- Nvidia V100: 22.6 secs

```
@njit
def solve(n, alpha, dx, dt, u, u_tmp):
    """Compute the next timestep, given the current timestep"""

    # Finite difference constant multiplier
    r = alpha * dt / (dx ** 2)
    r2 = 1 - 4 * r
    with openmp ("target loop collapse(2) map(tofrom: u, u_tmp)"):
        # Loop over the nxn grid
        for i in range(n):
            for j in range(n):
                u_tmp[j, i] = (r2 * u[j, i] +
                               (u[j, i+1] if i < n-1 else 0.0) +
                               (u[j, i-1] if i > 0   else 0.0) +
                               (u[j+1, i] if j < n-1 else 0.0) +
                               (u[j-1, i] if j > 0 else 0.0))
```

Data Movement dominates…

25,000x25,00 grid for 10 time steps
- Xeon Platinum 8480+: 67.6 secs
- Nvidia V100: 22.6 secs

```
# Loop over time steps
for _ in range(nsteps):
    # solve over spatial domain for step t
    solve(n, alpha, dx, dt, u, u_tmp)

    # Array swap to get ready for next step
    u, u_tmp = u_tmp, u
```

Typically, many time steps!

solve() function uses this context:
with openmp ("target loop collapse(2) map(tofrom: u, u_tmp)"):

For each iteration, copy from device $(2*N^2)*sizeof(TYPE)$ bytes

- We need to keep data resident on the device *between* target regions
- We need a way to manage the device data environment across iterations.

Target enter/exit data constructs

- The **target data** construct requires a *structured* block of code.
  - Often inconvenient in real codes.

- Can achieve similar behavior with two standalone directives:
  with openmp ("target enter data map(…"):
  with openmp ("target exit data map(…"):

- The **target enter data** maps variables to the device data environment.
- The **target exit data** unmaps variables from the device data environment.
- Future **target** regions inherit the existing data environment.

Solution: Reference swapping in action

```
with openmp ("target enter data map(to: u, u_tmp)"):
    pass

for _ in range(nsteps):

    solve(n, alpha, dx, dt, u, u_tmp);

    # Array swap to get ready for next step
    u, u_tmp = u_tmp, u

with openmp ("target exit data map(from: u)"):
    pass
```

Copy data to device before iteration loop

Change solve() routine to remove map clauses:
`with openmp ("target loop collapse(2)”)`

Copy data from device after iteration loop

25,000x25,00 grid for 10 time steps

- Xeon Platinum 8480+ default data movement: 67.6 secs
- Nvidia V100 default data movement: 22.6 secs
- Nvidia V100 target enter/exit: 1.2 secs

Target update directive

- You can update data between target regions with the **target update** directive.

```
with openmp ("target data map(to: A, B) map(from: C)"):

    with openmp ("target"):
        {do lots of stuff with A, B and C}

    with openmp ("target update from(A)"):
        {do something on the host}

    with openmp ("target update to(A)"):
        pass

    with openmp ("target"):
        {do lots of stuff with A, B and C}
```

Set up the data region ahead of time.

map A on the device to A on the host.

map A on the host to A on the device. Note: openmp context body cannot be empty so use “pass”

Note: update directive has the transfer direction as the clause: e.g. update to(…)
Compare to map clause with direction inside: map(to: …)

Data movement summary

- Data transfers between host/device occur at:
  - Beginning and end of **target** region
  - Beginning and end of **target data** region
  - At the **target enter data** construct
  - At the **target exit data** construct
  - At the **target update** construct

- Can use **target data** and **target enter/exit data** to reduce redundant transfers.

- Use the **target update** construct to transfer data on the fly within a **target data** region or between **target enter/exit data** directives.