



Programming your GPU with OpenMP

Tim Mattson

tgmattso@gmail.com

The Human Learning Group

This content was created with Tom Deakin and Simon McIntosh-Smith of the University of Bristol



This is my favorite picture of my wife ... surfing at Cascade Head in Oregon

Plan for the OpenMP sessions

Note: How much time people need with the exercises never works out as I expect, which is fine. Everything is driven by the needs of the students ... not some concept I might have of a schedule.

Monday, PM

4:00	Introduction: Parallel programming and the OpenMP Common Core
4:30	Working with threads (Including synchronization): the SPMD Pattern
5:30	Worksharing and data sharing: The Loop Parallelism Pattern
~6:30	Dinner

Next Day

Tuesday, All Day

8:30	Task-level parallelism in OpenMP: The Divide and Conquer Pattern
10:00	Break
10:30	Beyond the common core: More Worksharing and synchronization ... plus threadprivate
12:30	Lunch
1:30	Wrapping up the CPU and transitioning to GPU-programming
2:30	The loop construct ... GPU programming made “simple”
3:30	Break
4:00	Explicit Data Movement and basic principles of GPU optimization
5:30	Detailed control of the GPU ... and comparisons to other GPU programming models
6:30	Dinner

Preliminaries: Systems for exercises, Polaris

- Start an interactive job on one node

```
qsub -I -l select=1 -l walltime=00:30:00 -l filesystems=home:grand:eagle -A ATPESC2025 -q ATPESC
```

- Use the Nvidia programming environment

```
module swap PrgEnv-nvhpc PrgEnv-gnu      ← change back to Nvidia programming environment  
cc -mp=gpu heat_map_target.c.  
OMP_TARGET_OFFLOAD=MANDATORY ./a.out.    ← might be needed for tiny programs
```

- It might impact performance to match to the specific GPU architecture ...

```
cc -mp=gpu -gpu=cc80 program.c  
cc -mp=gpu -gpu=sm_80 program .c
```

- For short jobs you may need to force it to run on the GPU

```
OMP_TARGET_OFFLOAD=MANDATORY ./a.out.
```

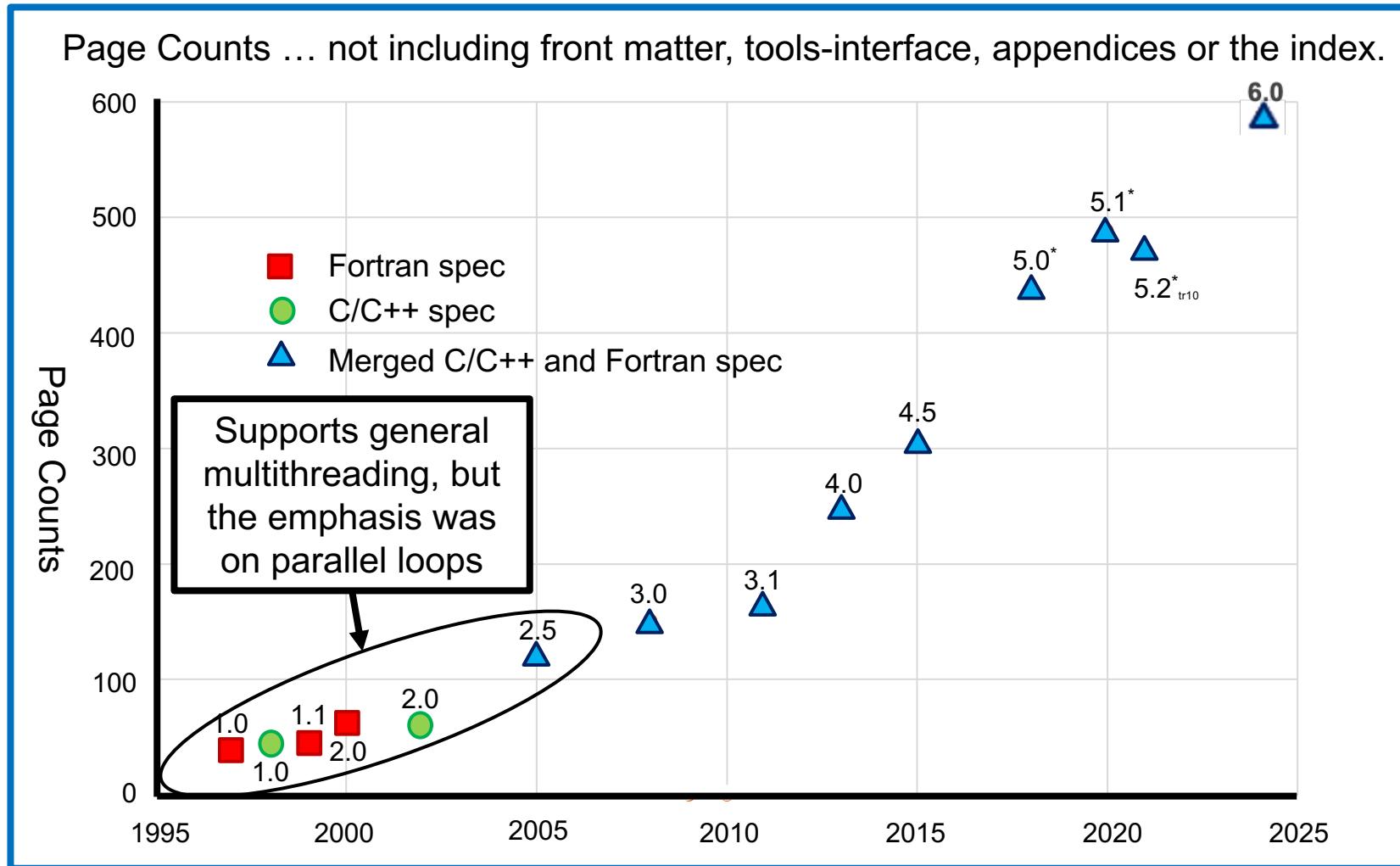
- For the GPU, you can profile an execution using the nvprof profile in nsys:

```
nsys nvprof ./a.out
```

- This will generate all sorts of data about the job. What we care most about is the summary of memory movement at the end of the profile report.

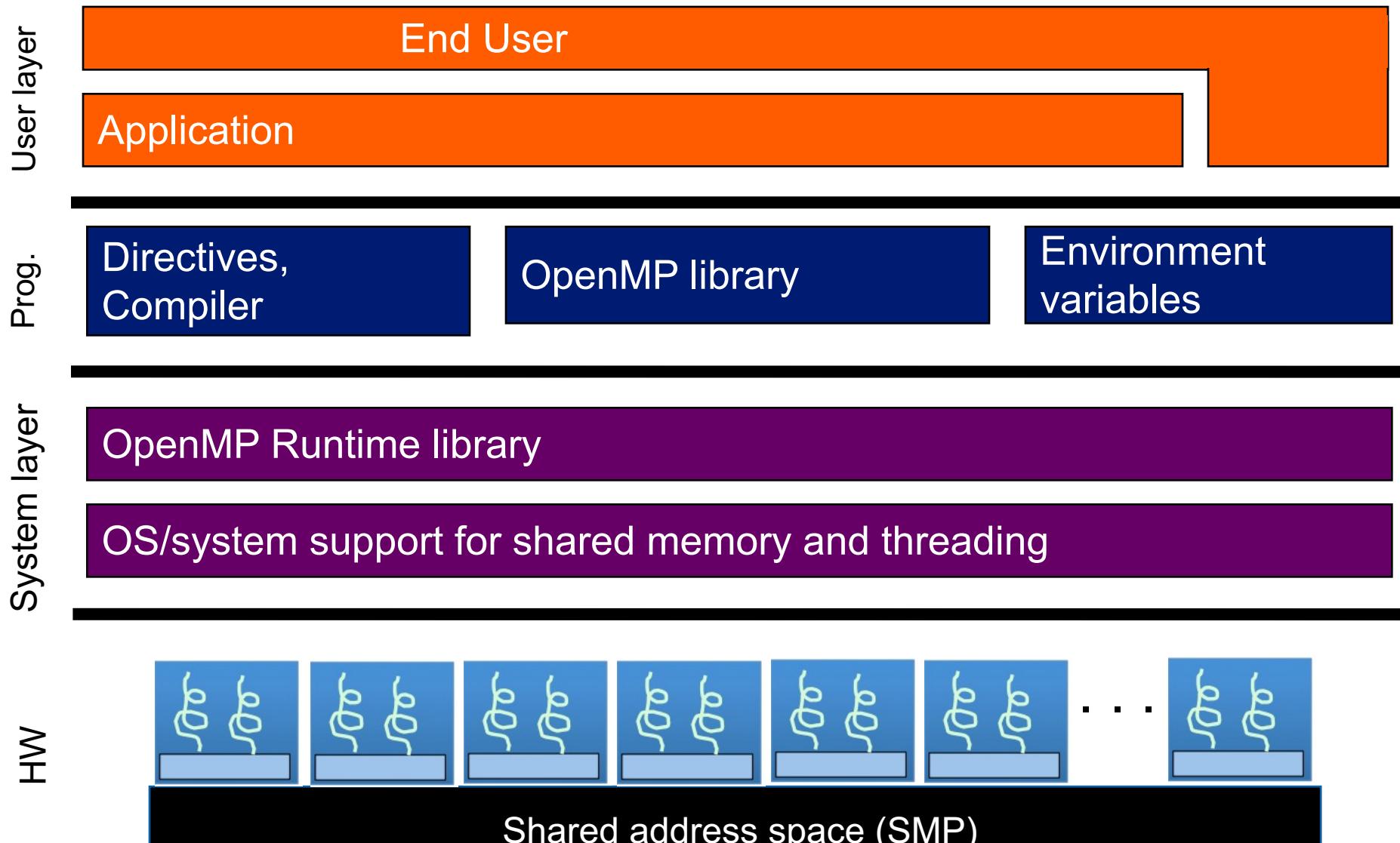
The Growth of Complexity in OpenMP

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

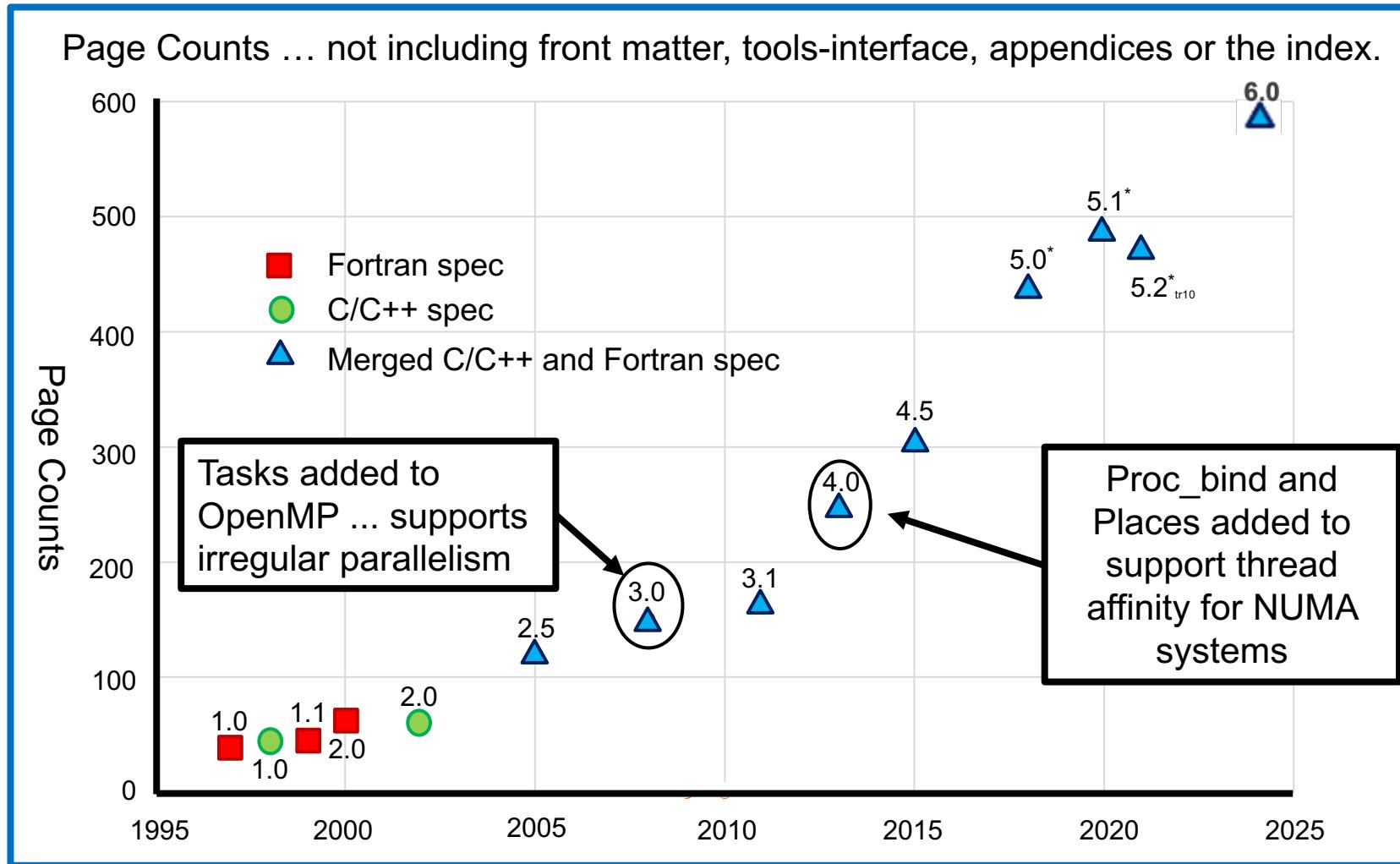
OpenMP Basic Definitions: Basic Solution Stack



For the OpenMP Common Core, we focus on Symmetric Multiprocessor Case
i.e., lots of threads with “equal cost access” to memory

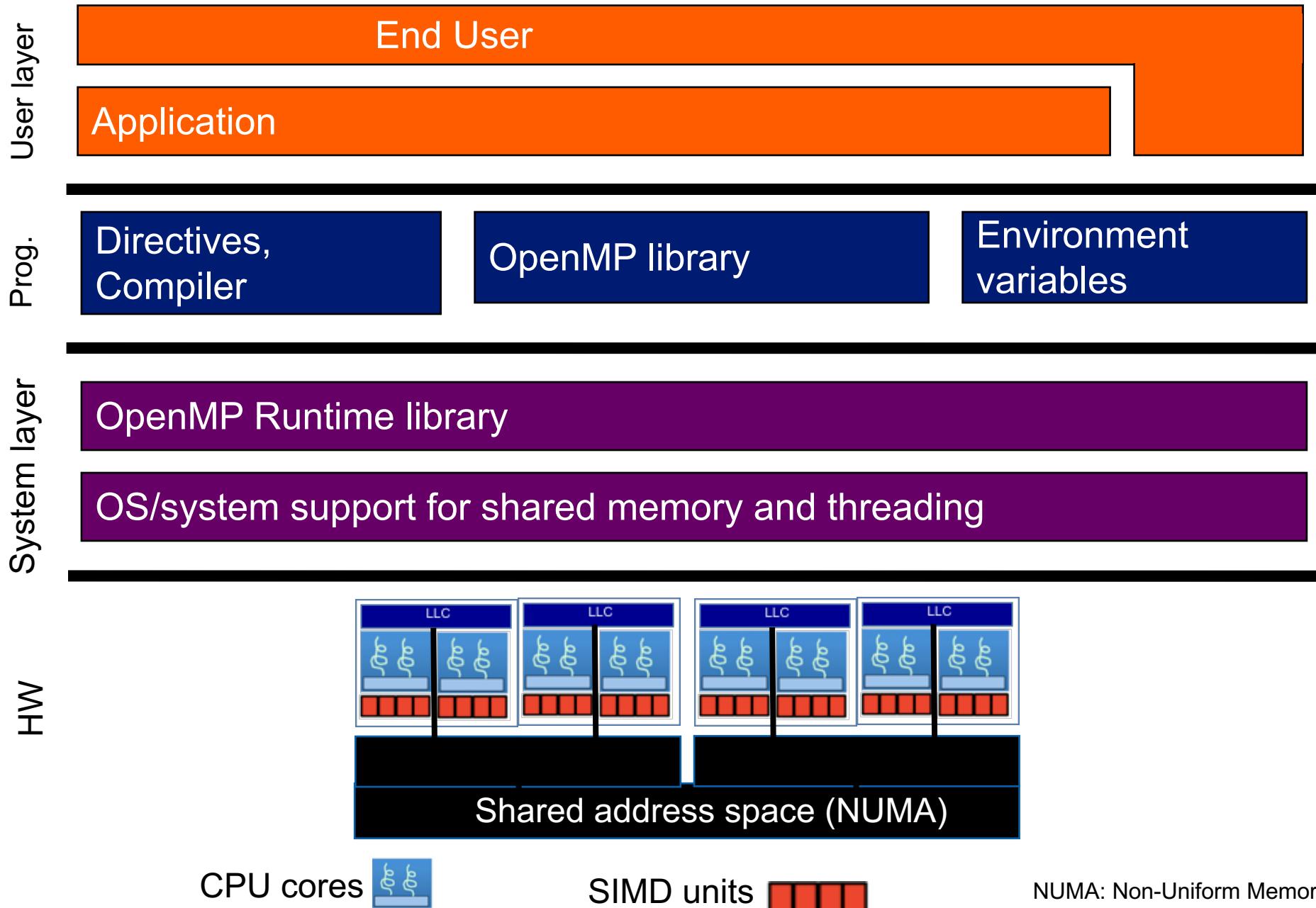
The Growth of Complexity in OpenMP

Our goal in 1997 ... A simple interface for application programmers



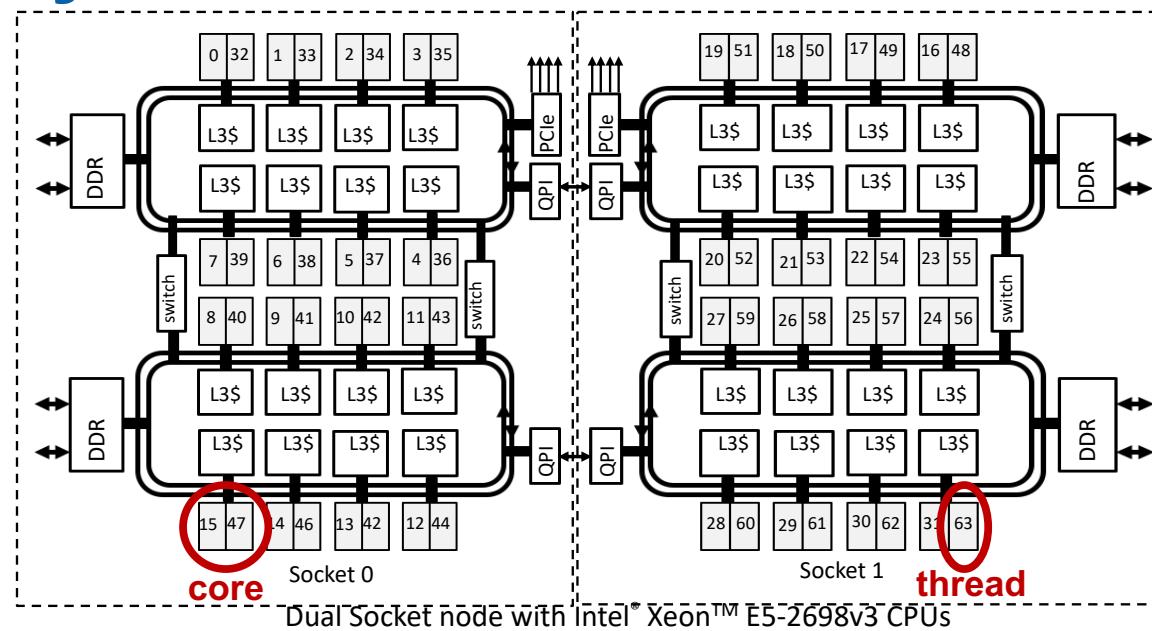
The OpenMP specification is so long and complex that few (if any) humans understand the full document

OpenMP Basic Definitions: Solution stack



OpenMP for non-uniform memory architectures (in one slide)

- A modern CPU is complex. The OS manages threads to emphasize low latency for numerous concurrent threads ... not HPC
- OpenMP includes the ability for full control of NUMA systems ... it can get complicated.
- Keep it simple:
 - Utilize **first touch** page assignment: Initialize data the same way (e.g. with the same "parallel for schedule" clause) as you will compute with it.

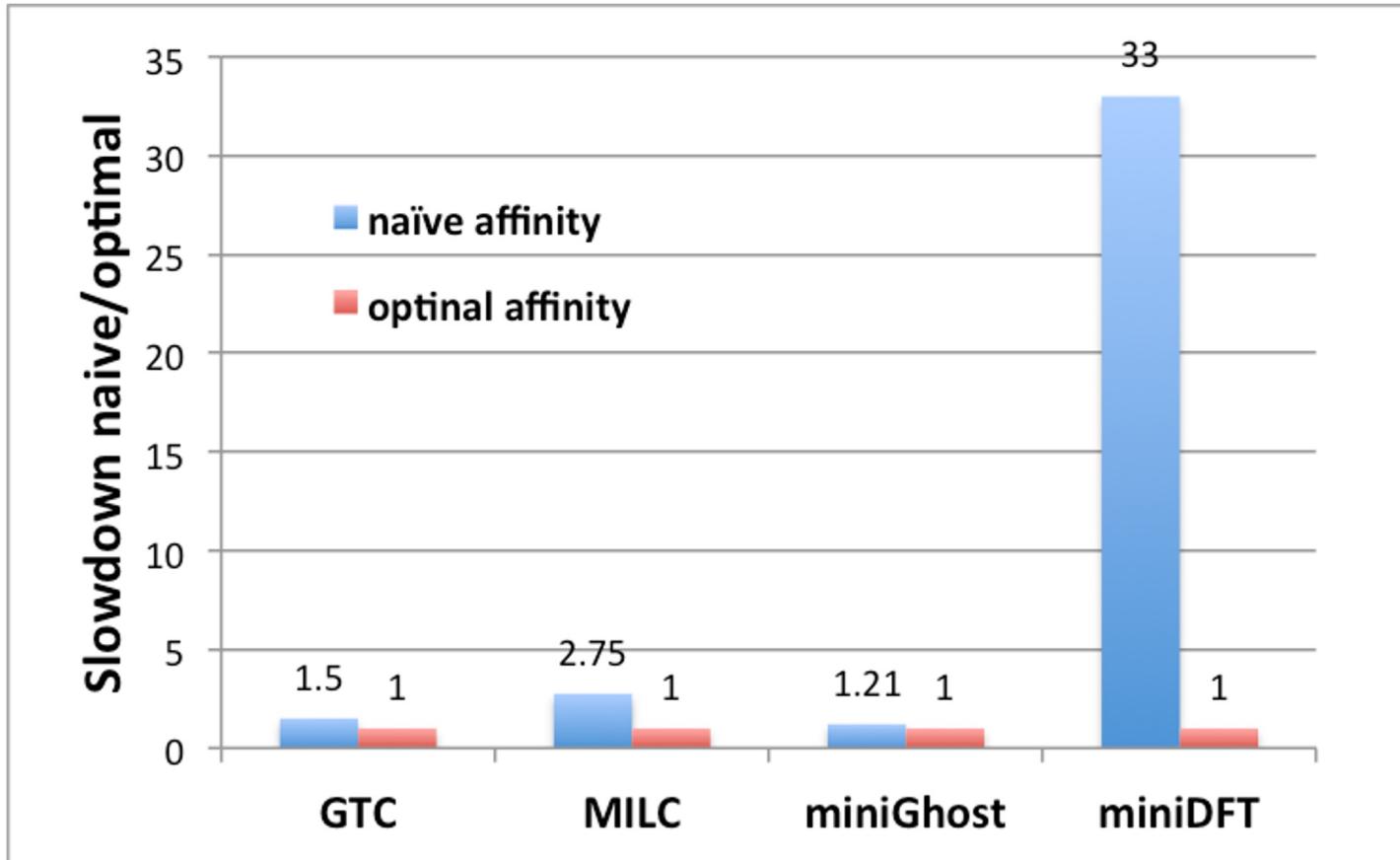


*Primary thread: this is the thread with ID=0 that encountered the parallel construct and created the team of threads

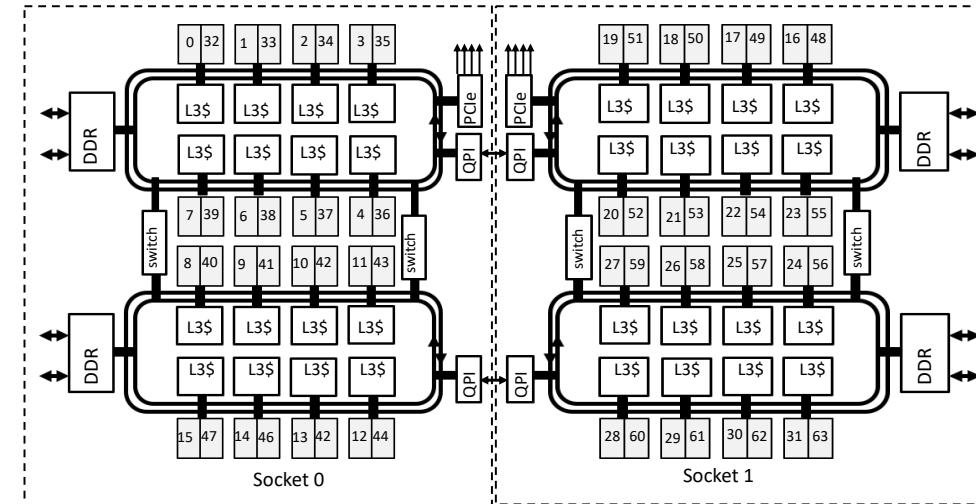
- Examples:
 - export OMP_PLACES=threads
 - export OMP_PLACES=cores
- Examples:
 - export OMP_PROC_BIND=spread
 - export OMP_PROC_BIND=close

Getting the affinity right can have serious impacts on performance

Application Benchmark Performance for a number of benchmarks at NERSC



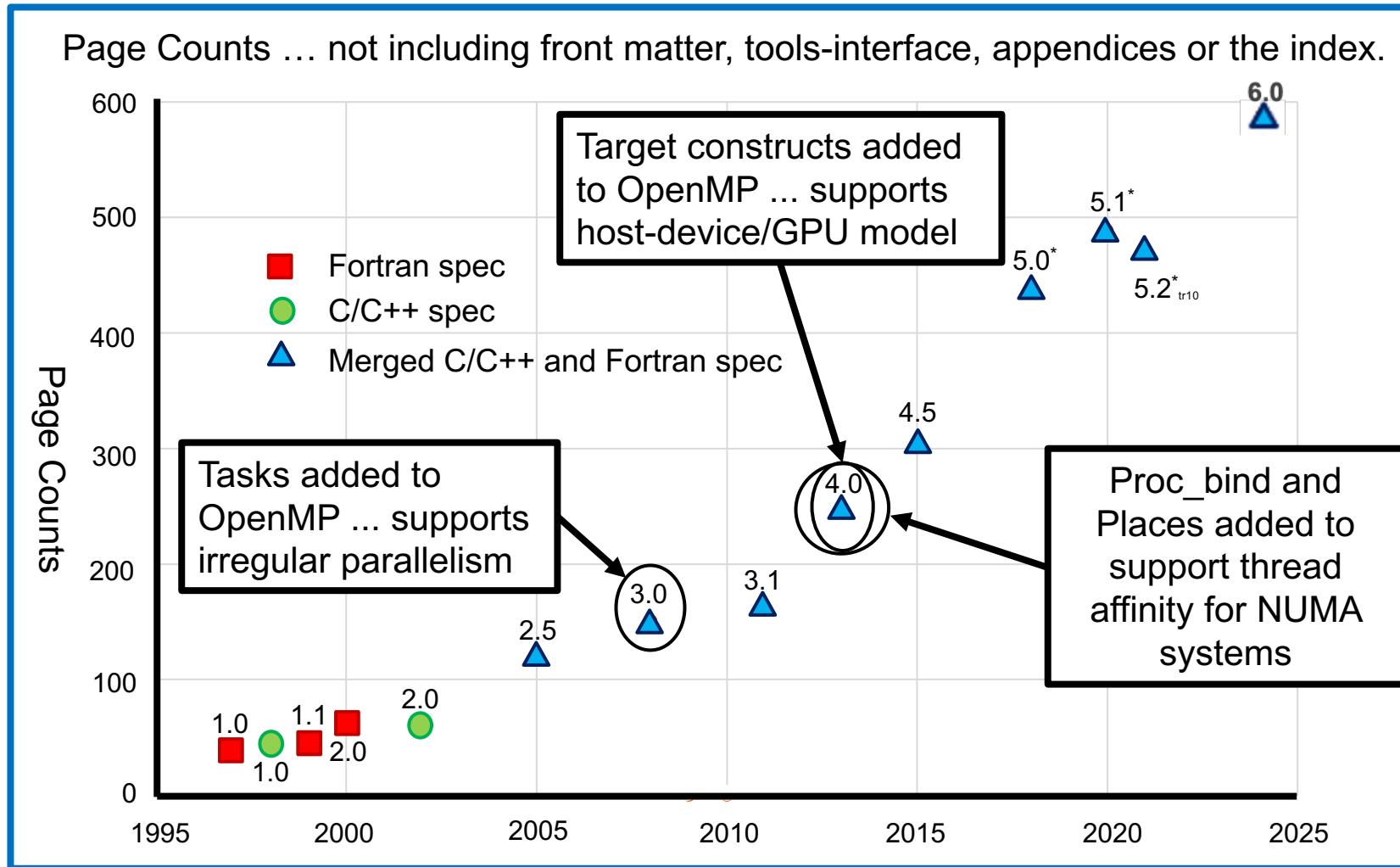
Lower is better



Results running on the Cori system at
NERSC which has dual Socket nodes with
Intel® Xeon™ E5-2698v3 CPUs

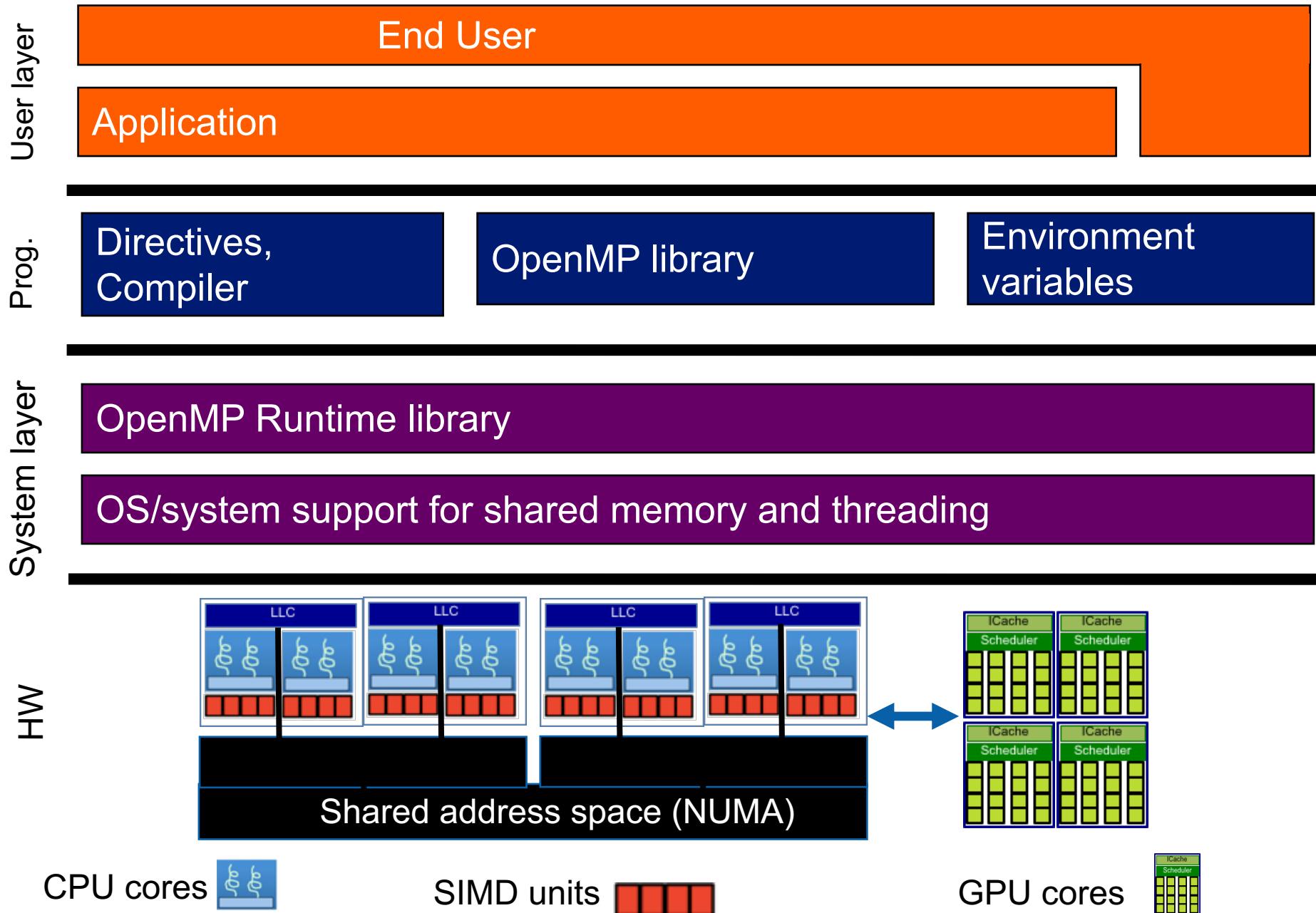
The Growth of Complexity in OpenMP

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

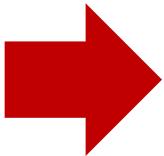
OpenMP Basic Definitions: Solution stack



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. Write kernel code for the scalar work-items

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

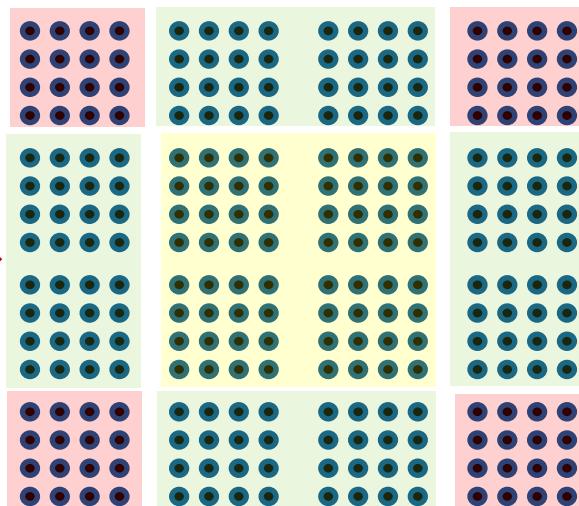
int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a, sizeof(float) * N);
    // ... allocate other arrays (b and c)
    // and fill with data

    // define threadBlocks and the Grid
    dim3 dimBlock(4,4);
    dim3 dimGrid(4,4);

    // Launch kernel on Grid
    matAdd <<< dimGrid, dimBlock >>> (a, b, c, N);
}
```

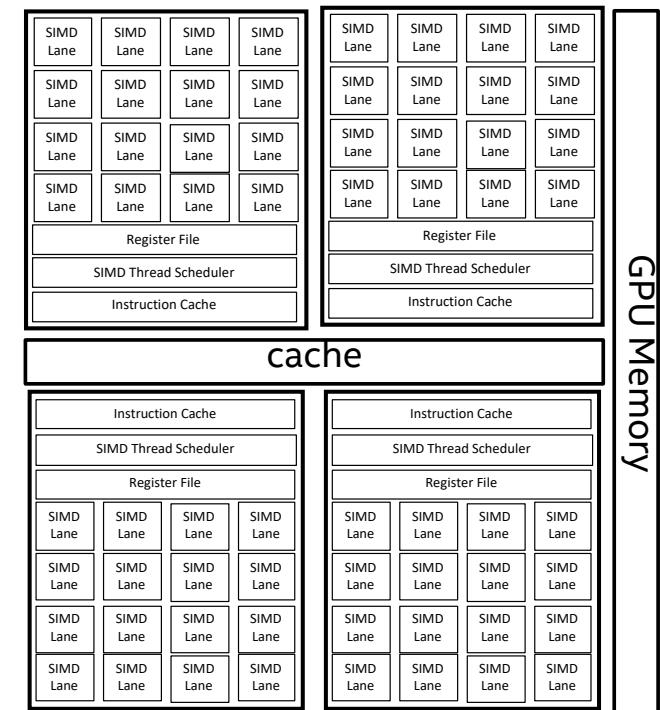
This is CUDA code

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



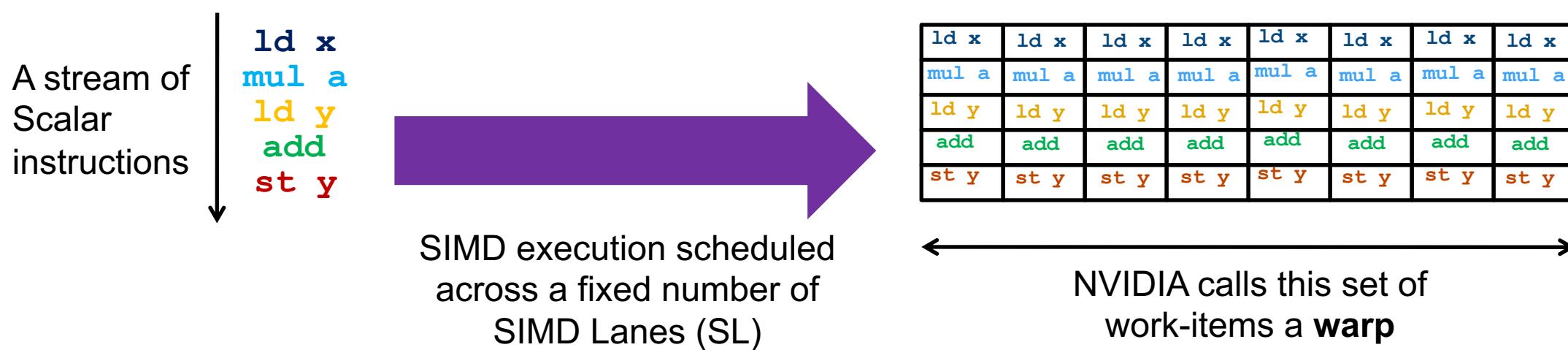
3. Map data structures onto the same index space

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

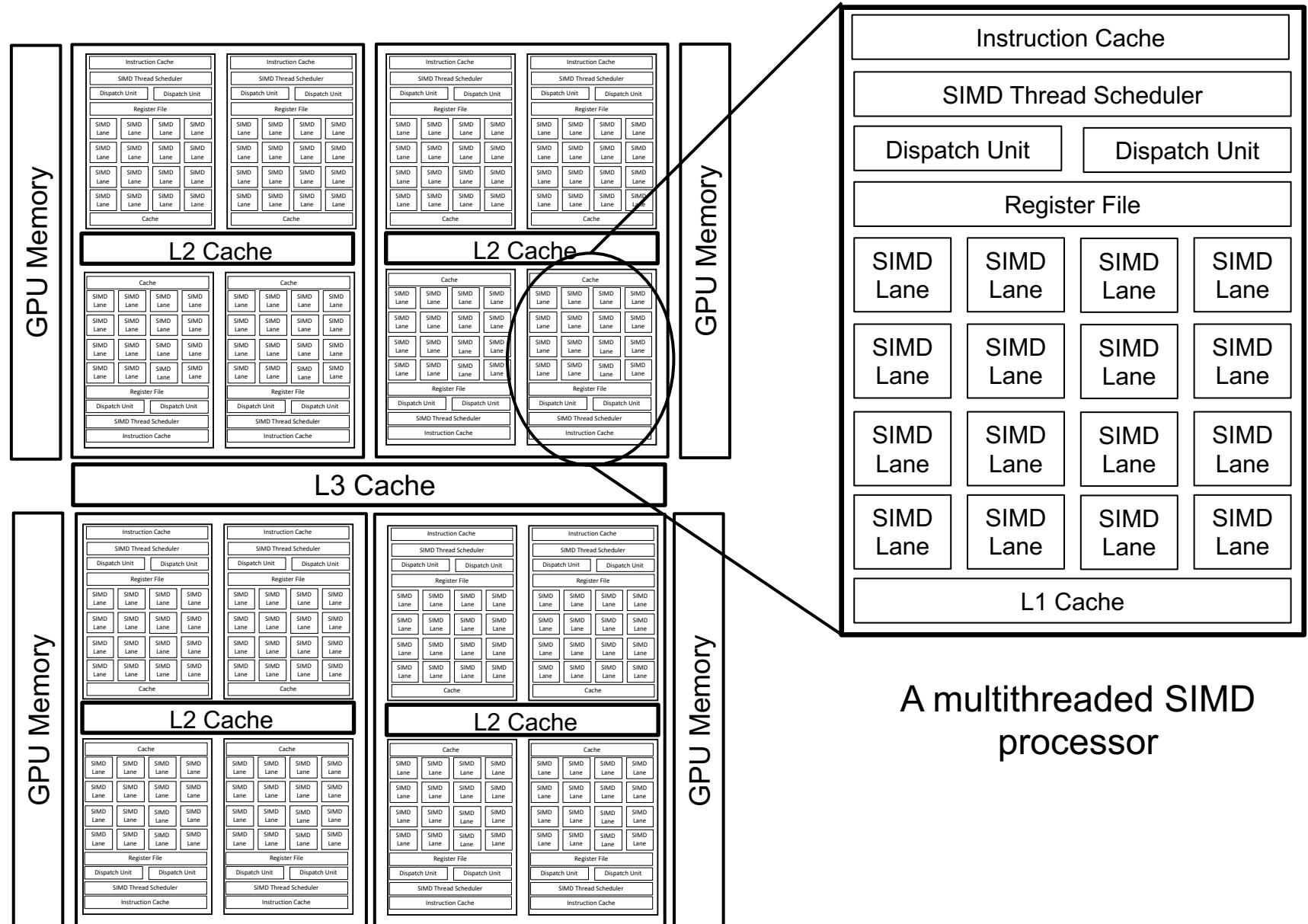


SIMT: One instruction stream maps onto many SIMD lanes

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



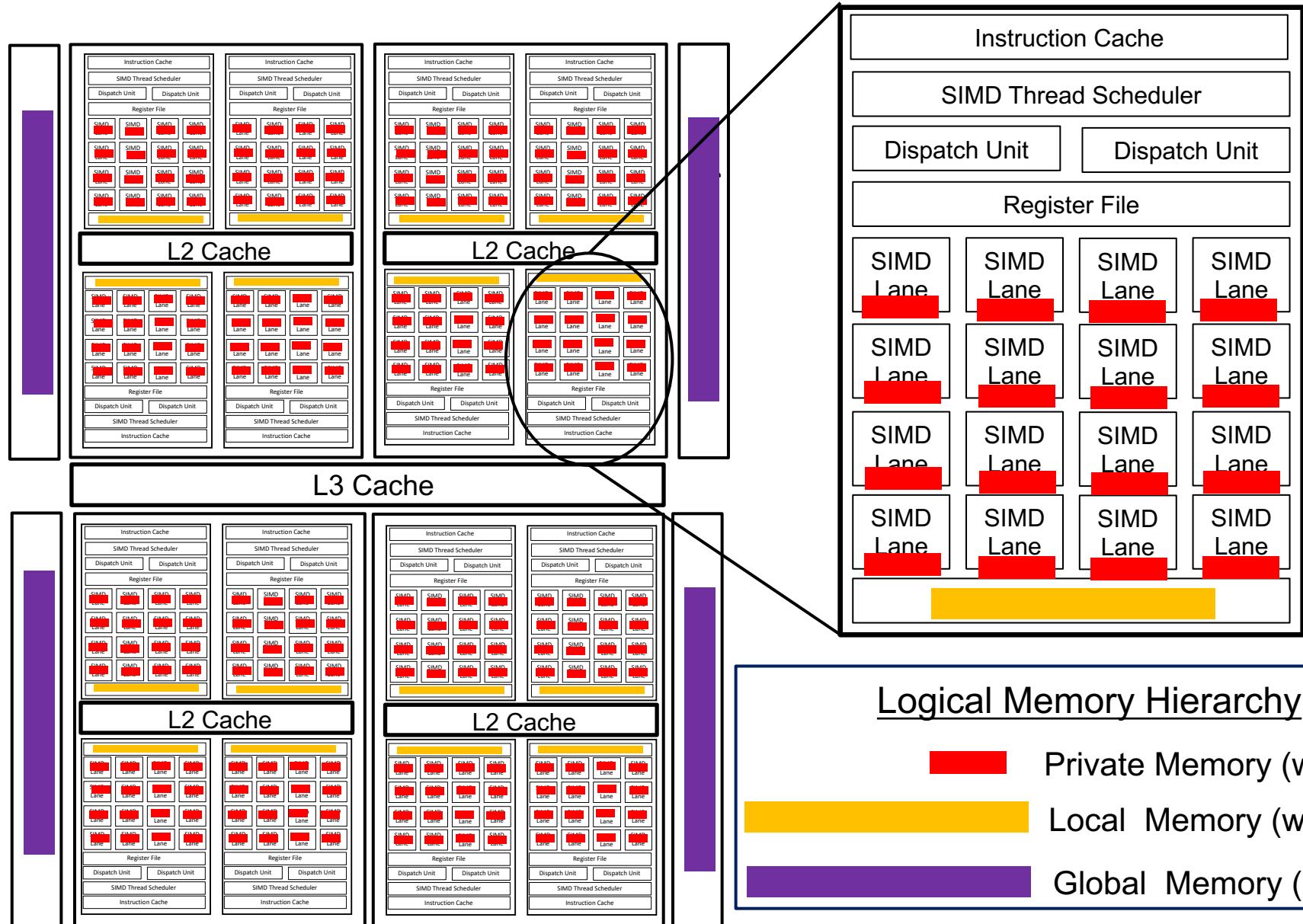
A Generic GPU (following Hennessy and Patterson)



GPU terminology is Broken (sorry about that)

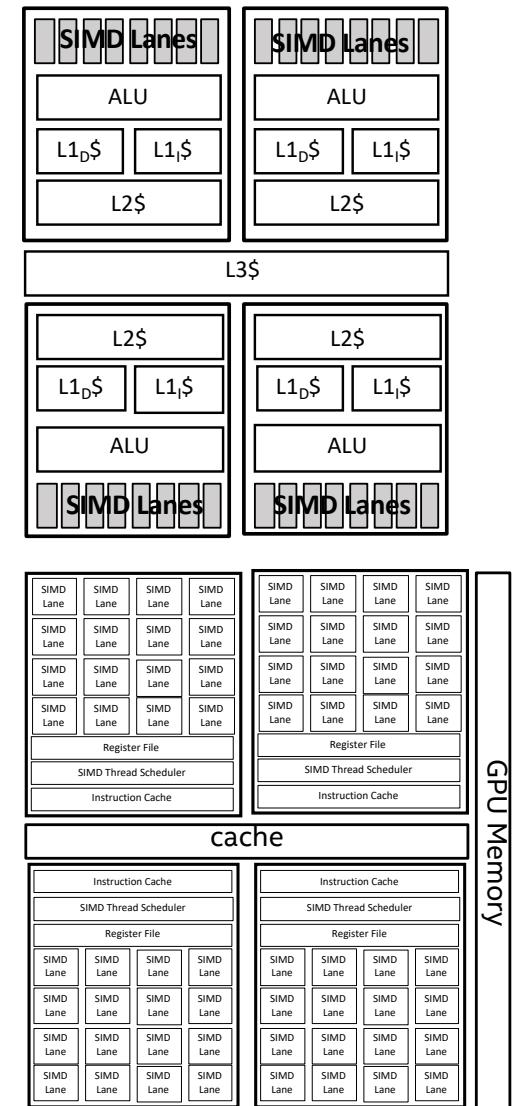
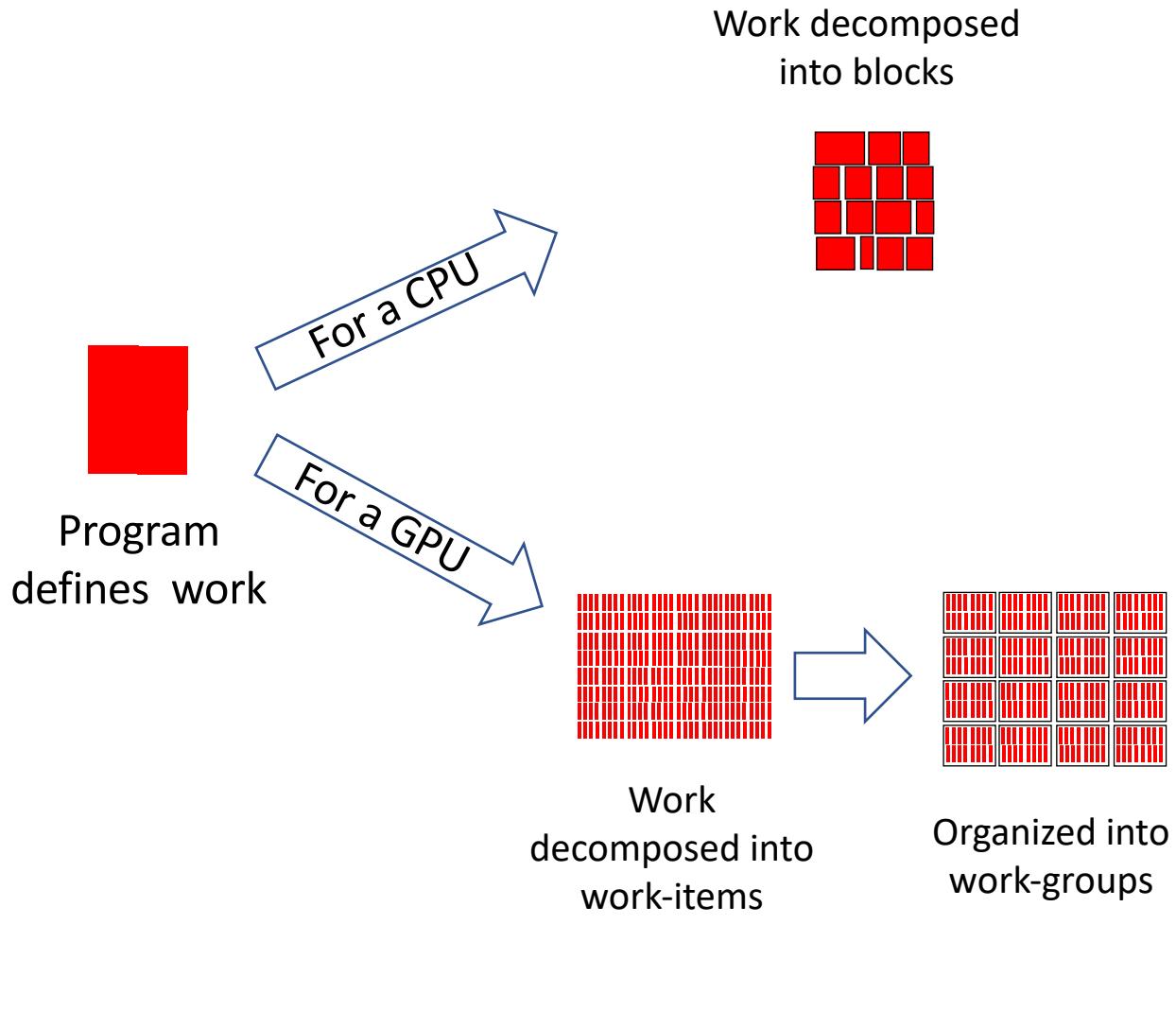
Hennessy and Patterson	CUDA	OpenCL
Multithreaded SIMD Processor	Streaming multiprocessor	Compute Unit
SIMD Thread Scheduler	Warp Scheduler	Work-group scheduler
SIMD Lane	CUDA Core	Processing Element
GPU Memory	Global Memory	Global Memory
Private Memory	Local Memory	Private Memory
Local Memory	Shared Memory	Local Memory
Vectorizable Loop	Grid	NDRange
Sequence of SIMD Lane operations	CUDA Thread	work-item
A thread of SIMD instructions	Warp	sub-group

A Generic GPU (following Hennessy and Patterson)



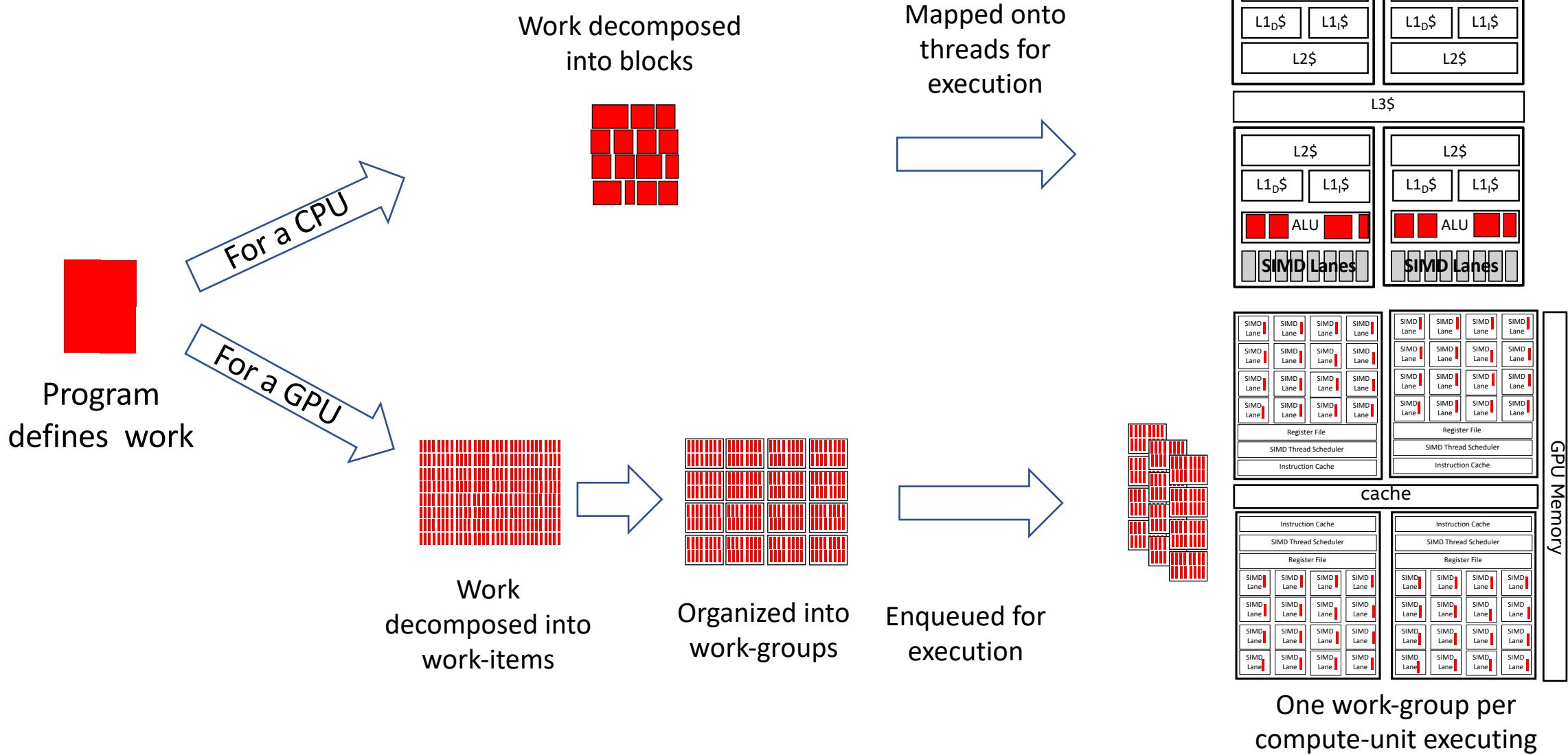
**Let's compare/contrast concurrency on a
CPU and a GPU**

Executing a program on CPUs and GPUs



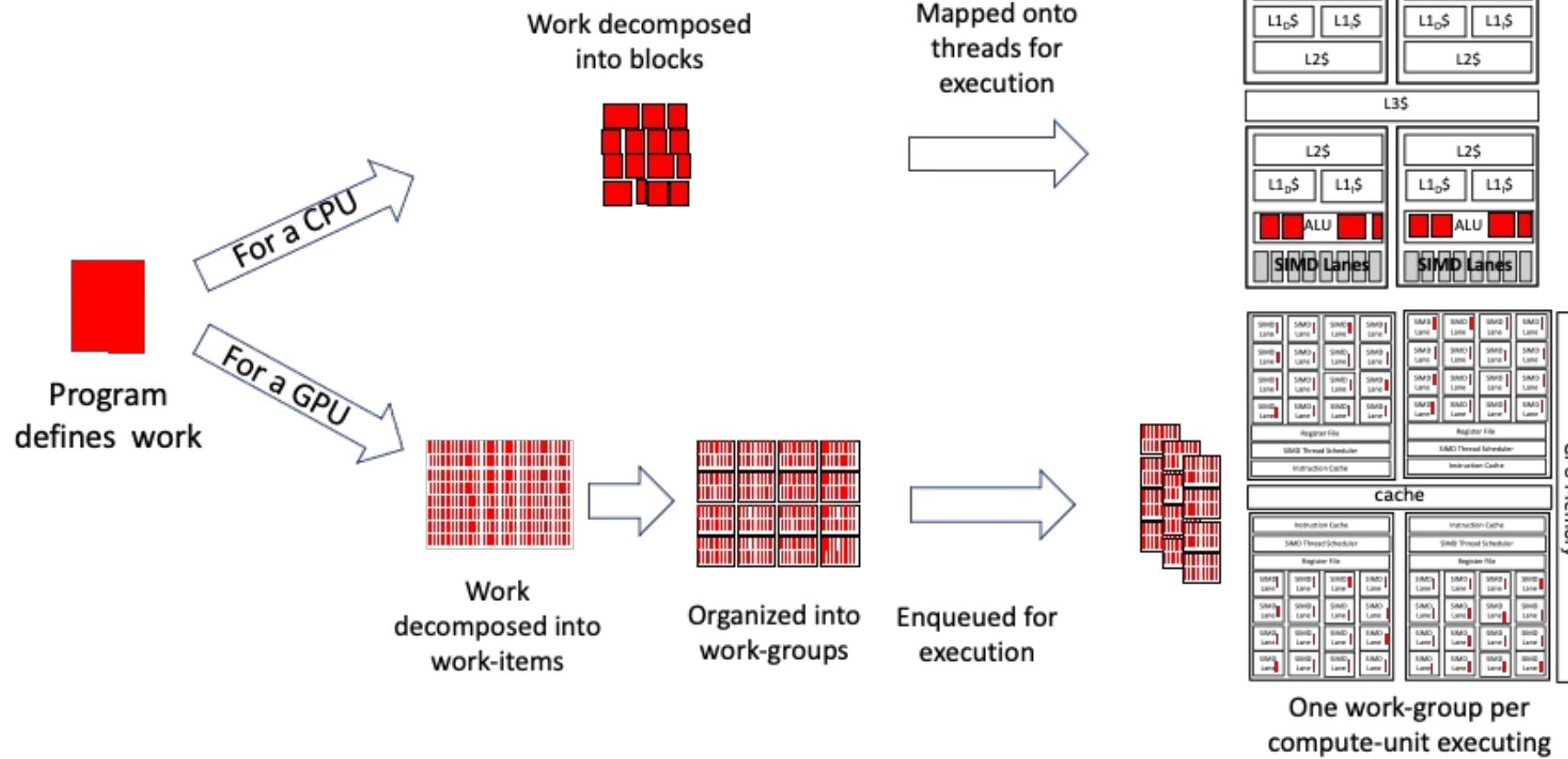
One work-group per
compute-unit executing

Executing a program on CPUs and GPUs



CPU/GPU execution models

Executing a program on CPUs and GPUs



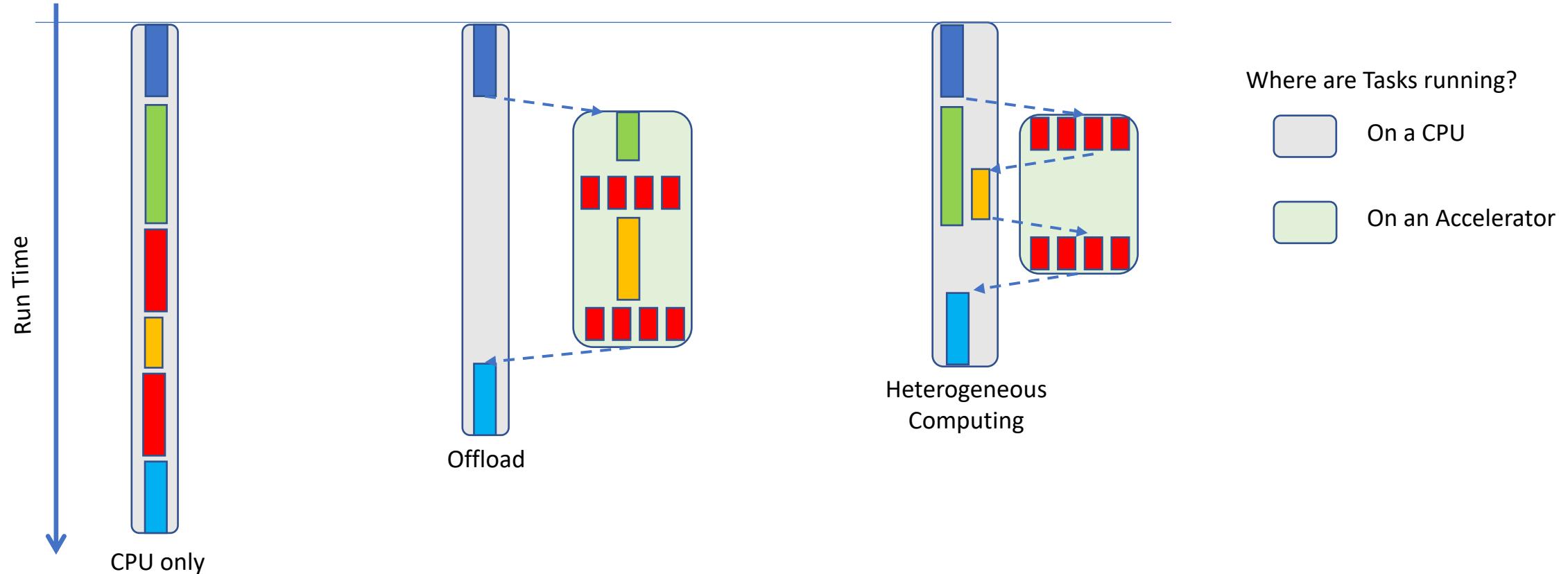
For a CPU, the threads are all active and able to make forward progress.

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

**Programming heterogeneous devices
means splitting up code to get the most
from the available hardware**

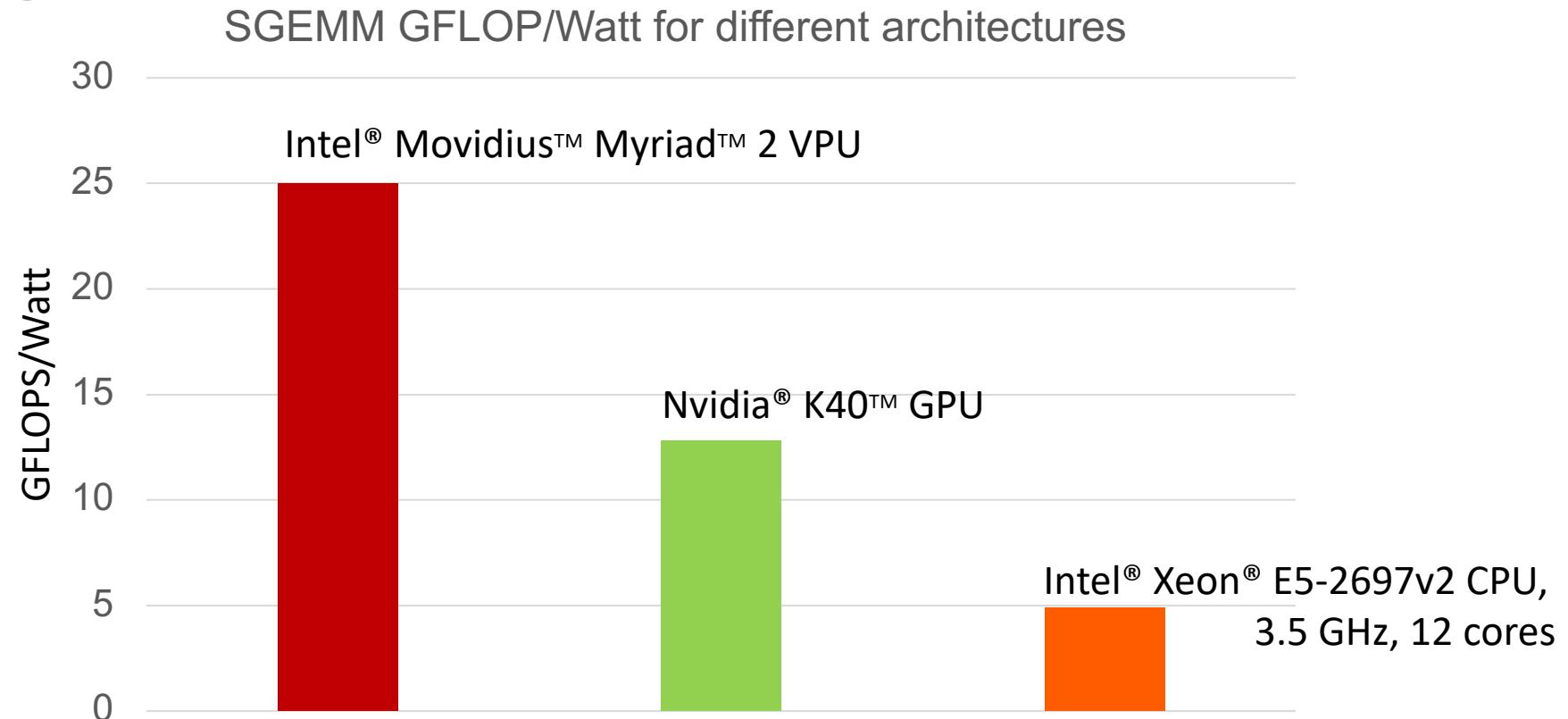
No single processor is best at everything

- The idea that you should move everything to the GPU makes no sense
- **Heterogeneous Computing:** Run sub-problems in parallel on the hardware best suited to them.



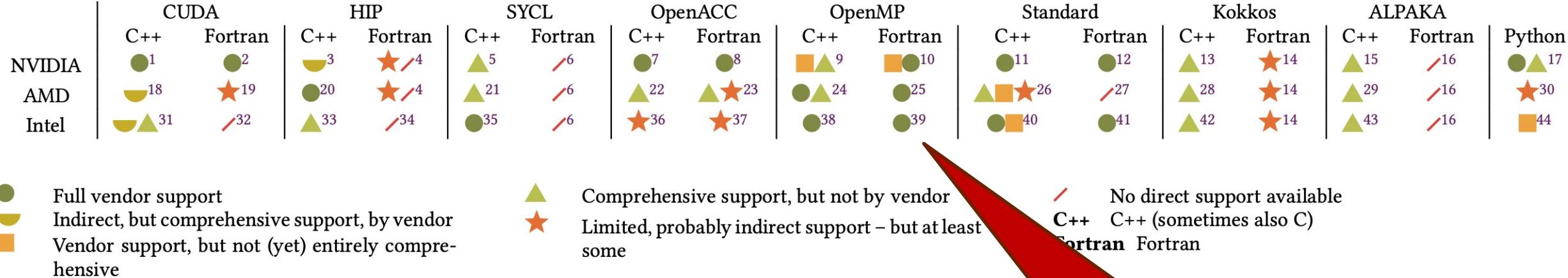
If you care about power, the world is heterogeneous?

Specialized processors doing operations suited to their architecture are more efficient than general purpose processors.



Hence, future systems will be increasingly heterogeneous ... GPUs, CPUs, FPGAs, and a wide range of accelerators

Why is OpenMP so important?



<https://x-dev.pages.jsc.fz-juelich.de/models/>

Table from <https://doi.org/10.1145/3624062.3624178>

Many cores, Many models: GPU programming model vs Vendor Compatibility Overview, Andreas Herten,
SC23 workshop proceedings

OpenMP is the **only** model
with **full** support
from **all** vendors
for C/C++ and Fortran

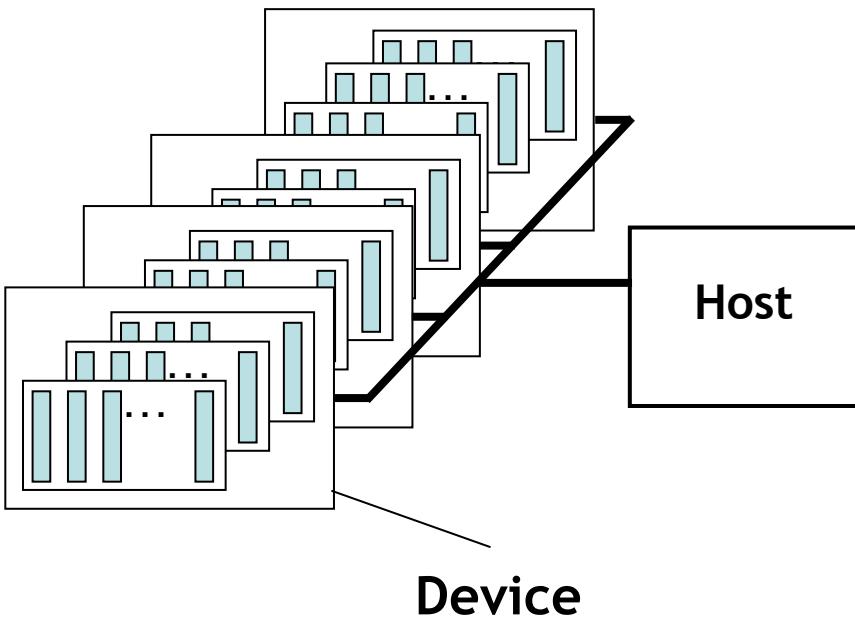
... and OpenMP supports
Python as well (PyOMP)

<https://github.com/Python-for-HPC/PyOMP.git>

**Let's dig into the details of writing GPU
code with OpenMP**

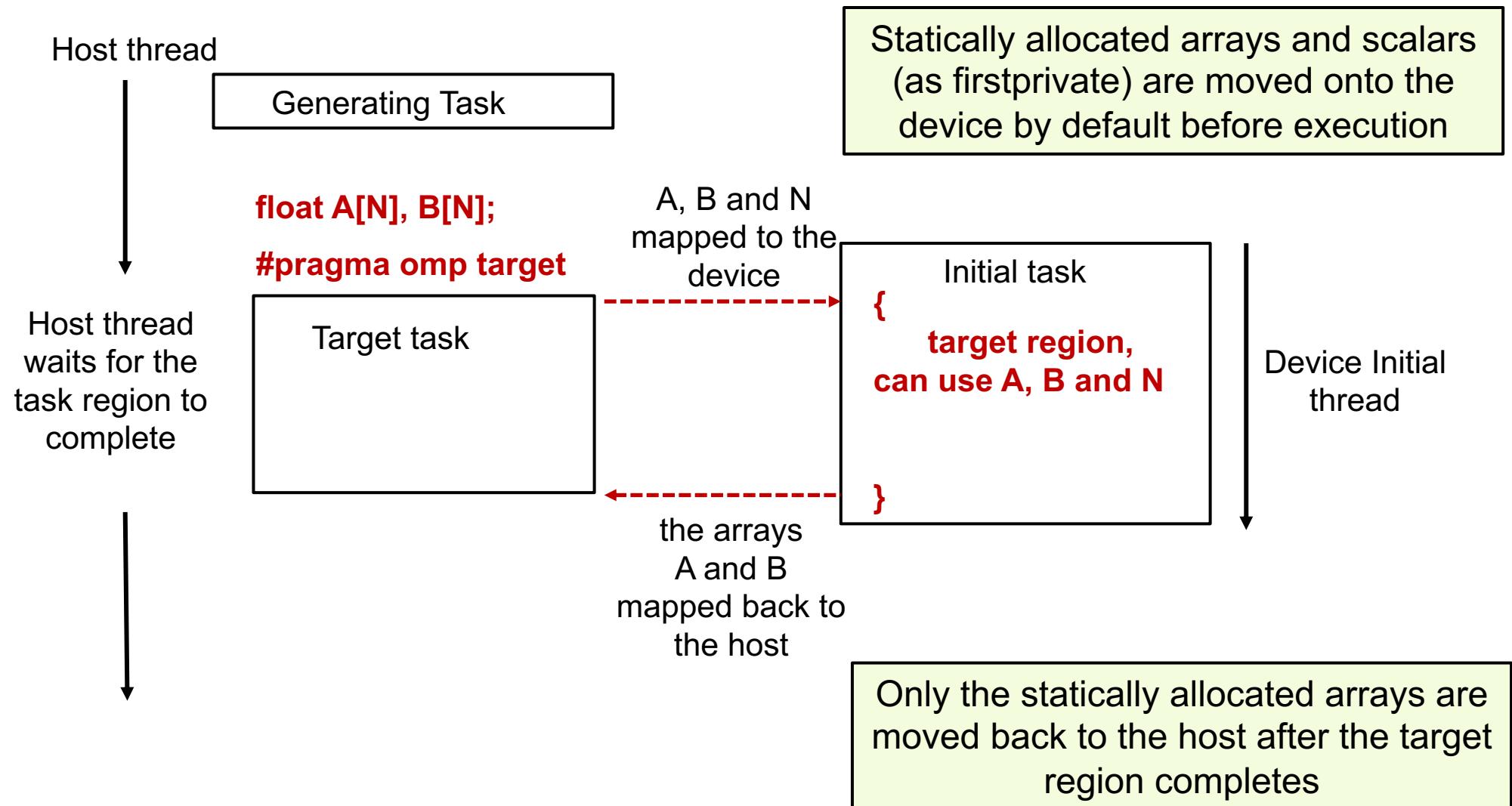
The OpenMP device programming model

- OpenMP uses a host/device model
 - The **host** is where the initial thread of the program begins execution
 - Zero or more **devices** are connected to the host
 - **Device-memory** address space is distinct from **host-memory** address space



```
#include <omp.h>
#include <stdio.h>
int main()
{
    printf("There are %d devices\n",
           omp_get_num_devices());
}
```

Running code on the GPU: The target construct and default data movement



Default Data Sharing: example

```
int main(void) {  
    int N = 1024;  
    double A[N], B[N];  
  
    #pragma omp target  
    {  
  
        for (int ii = 0; ii < N; ++ii) {  
  
            A[ii] = A[ii] + B[ii];  
  
        }  
    } // end of target region  
}
```

1. Variables created in host memory.

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

3. **ii** is **private** on the device as it's declared within the target region

4. Execution on the device.

5. stack arrays **A** and **B** are copied *from* device memory back to the host. Host resumes execution.

Now let's run code in parallel on the device

```
int main(void) {  
    int N = 1024;  
    double A[N], B[N];  
  
    #pragma omp target  
    {  
        #pragma omp loop  
        for (int ii = 0; ii < N; ++ii) {  
            A[ii] = A[ii] + B[ii];  
  
        }  
    } // end of target region  
}
```

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

Exercise: Parallel vector addition on a GPU

- Start with the provided vadd.c program. Parallelize it for a CPU and time it for large N.
 - vadd.c Adds together two arrays, element by element: $\text{for}(i=0;i<N;i++) c[i]=a[i]+b[i];$
- Parallelize the vadd program for a GPU and time it for large N.
- How does it compare to the CPU version?

- double omp_get_wtime();
- #pragma omp parallel
- #pragma omp for
- #pragma omp target
- #pragma omp loop

For tiny little programs, OpenMP may opt to run the code on the host. You can force the OpenMP runtime to use the GPU by setting the OMP_TARGET_OFFLOAD environment variable
> OMP_TARGET_OFFLOAD=MANDATORY ./a.out

Get interactive access to a node:

```
qsub -I -l select=1 -l walltime=00:30:00 -l filesystems=home:eagle -A ATPESC2025 -q ATPESC
```

Compiler with cc ... which is a wrapper around the Nvidia compilers (cc, CC or ftn)
cc -mp=gpu vadd.c

Solution: Simple vector add in OpenMP on GPU

```
int main()
{
    float a[N], b[N], c[N], res[N];
    int err=0;

    // fill the arrays
    #pragma omp parallel for
    for (int i=0; i<N; i++) {
        a[i] = (float)i;
        b[i] = 2.0*(float)i;
        c[i] = 0.0;
        res[i] = i + 2*i;
    }

    // add two vectors
    #pragma omp target
    #pragma omp loop
    for (int i=0; i<N; i++) {
        c[i] = a[i] + b[i];
    }

    // test results
    #pragma omp parallel for reduction(+:err)
    for(int i=0;i<N;i++) {
        float val = c[i] - res[i];
        val = val*val;
        if(val>TOL) err++;
    }
    printf("vectors added with %d errors\n", err);
    return 0;
}
```

CUDA Toolkit: nsys

Simple profiling: nsys nvprof ./exe <params>

```
> nsys nvprof ./flow.omp4. flow-params
```

```
Problem dimensions 4000x4000 for 1 iterations.  
==188532== NVPROF is profiling process 188532, command: ./flow.omp4 flow.params  
Number of ranks: 1  
Number of threads: 1
```

```
Iteration 1  
Timestep: 1.816932845523e-04  
Total mass: 2.561400875000e+06  
Total energy: 5.442884982081e+06  
Simulation time: 0.0001s  
Wallclock: 0.0325s
```

```
Expected energy 3.231871108096e+07, result was 3.231871108096e+07.
```

```
Expected density 2.561400875000e+06, result was 2.561400875000e+06.
```

```
PASSED validation.
```

```
Wallclock 0.0325s, Elapsed Simulation Time 0.0001s
```

```
==188532== Profiling application: ./flow.omp4 flow.params
```

```
==188532== Profiling result:
```

Time(%)	Time	Calls	Avg	Min	Max	Name	
55.51%	205.74ms	53	3.8818ms	896ns	12.821ms	[CUDA memcpy HtoD]	Time to copy data onto GPU
28.69%	106.32ms	14	7.5942ms	576ns	55.648ms	[CUDA memcpy DtoH]	Time to copy data back from GPU
5.31%	19.682ms	2	9.8411ms	3.8686ms	15.814ms	set_problem_2d\$ck_L240_28	
1.52%	5.6321ms	2	2.8160ms	2.8121ms	2.8199ms	set_timestep\$ck_L92_5	
1.05%	3.9072ms	32	122.10us	1.2160us	217.21us	allocate_data\$ck_L30_1	
0.80%	2.9801ms	1	2.9801ms	2.9801ms	2.9801ms	artificial_viscosity\$ck_L198_16	
0.73%	2.7061ms	1	2.7061ms	2.7061ms	2.7061ms	pressure_acceleration\$ck_L128_9	

Exercise: Parallel vector addition on a GPU

- Run your vector add program using nsys and see if the profiling output matches your expectations for vadd.

- double omp_get_wtime();
- #pragma omp parallel
- #pragma omp for
- #pragma omp parallel for
- #pragma omp task
- #pragma omp taskwait
- #pragma single
- #pragma omp target
- #pragma omp loop

For tiny little programs, OpenMP may opt to run the code on the host. You can force the OpenMP runtime to use the GPU by setting the OMP_TARGET_OFFLOAD environment variable

```
> OMP_TARGET_OFFLOAD=MANDATORY ./a.out
```

Get interactive access to a node:

```
qsub -I -l select=1 -l walltime=00:30:00 -l filesystems=home:eagle -A ATPESC2025 -q ATPESC
```

Compiler with cc ... which is a wrapper around the Nvidia compilers (nvc)

```
cc -mp=gpu program.c  
nsys nvprof ./a.out
```

**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**

Explicit data movement

- Previously, we described the rules for *implicit* data movement.
- We can *explicitly* control the movement of data using the **map** clause.
- **Data allocated on the heap needs to be explicitly copied to/from the device:**

```
int main(void) {
    int ii=0, N = 1024;
    int* A = (int *)malloc(sizeof(int)*N);

#pragma omp target
{
    // N, ii and A all exist here
    // The data that A points to (*A , A[ii]) DOES NOT exist here!
}
```

Moving data with the map clause

```
int main(void) {  
    int N = 1024;  
    int* A = malloc(sizeof(int)*N);  
  
    #pragma omp target map(A[0:N])  
    {  
        // N, ii and A all exist here  
        // The data that A points to DOES exist here!  
    }  
}
```

Default mapping
map(tofrom: A[0:N])

Copy at start and end of
target region.

OpenMP array notation

- For mapping data arrays/pointers you must use array section notation:
 - In C, notation is **pointer[lower-bound : length]**
 - **map(to: a[0:N])**
 - Starting from the element at $a[0]$, copy N elements to the target data region
 - **Be careful!**
 - It's common to confuse this with the Fortran notation: (begin : end).
 - Without the map, OpenMP defines that the pointer itself (**a**) is mapped as a zero-length array section.
 - Zero length arrays: $a[:0]$

Controlling data movement

```
int i, a[N], b[N], c[N];  
#pragma omp target map(to:a,b) map(tofrom:c)
```

Data movement
defined from the
host perspective.

- The various forms of 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)**.

Briefly, attached pointers

- Pointers appearing with array sections in map clauses are called a *base pointer*
 - E.g., in `map(tofrom: A[0:N])`, A is a base pointer
- The base pointer is mapped `firstprivate`, and is an *attached* pointer
- Attached pointers *cannot* be modified in the target region
- The OpenMP runtime keeps a lookup table of mapped memory addresses to translate between the data on the host and the mapped data on the device
 - The translation happens when variables are mapped (target, target data, etc)

Exercise: Parallel vector addition on a GPU

- Start from vadd_heap.c
 - Vadd_heap.c Adds together two arrays, element by element:
$$\text{for}(i=0;i<N;i++) c[i]=a[i]+b[i];$$
- Parallelize for a GPU
 - double omp_get_wtime();
 - #pragma omp parallel
 - #pragma omp for
 - #pragma omp parallel for
 - #pragma omp task
 - #pragma omp taskwait
 - #pragma single
 - #pragma omp target
 - #pragma omp loop
 - Plus the clauses
 - private(), firstprivate(), reduction(+:var)
 - map(to:vptr[Lower:Count]) map(from:vptr[Lower:Count]) map(tofrom:vptr[Lower:Count])

Default is tofrom: map(vptr[Lower:Count])



Solution: vector add with dynamic memory on GPU

```
int main()
{
    float *a    = malloc(sizeof(float) * N);
    float *b    = malloc(sizeof(float) * N);
    float *c    = malloc(sizeof(float) * N);
    float *res = malloc(sizeof(float) * N);
    int err=0;

    // fill the arrays <<<code not shown>>>

    // add two vectors
#pragma omp target map(to: a[0:N],b[0:N]) map (tofrom: c[0:N])
#pragma omp loop
for (int i=0; i<N; i++){
    c[i] = a[i] + b[i];
}

    // test results <<<code not shown>>>

#pragma omp parallel for reduction(+:err)
printf("vectors added with %d errors\n", err);
return 0;
}
```

Commonly used clauses on target and loop constructs

- The basic construct* is:

`#pragma omp target [clause[,]clause]...]`

`#pragma omp loop [clause[,]clause]...]`

for-loops

- The most commonly used clauses are:

- **map(to | from | tofrom list)** ← default is tofrom

- **private(list)** **firstprivate(list)** **lastprivate(list)** **shared(list)**

- behave as data environment clauses in the rest of OpenMP, but note values are only created or copied into the region, not back out “at the end”.

- **reduction(reduction-identifier : list)**

- behaves as in the rest of OpenMP

- **collapse(n)**

- Combines loops before the distribute directive splits up the iterations between teams

Loop and reductions

```
#include <omp.h>
#include <stdio.h>
static long num steps = 100000000;
int main() {
double sum = 0.0;
double step = 1.0 / ( double ) num steps ;
#pragma omp target map (tofrom:sum)
#pragma omp loop reduction (+:sum)
for (int i=0; i<numsteps; i++) {
    double x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);
}
double pi = step * sum;
printf(" pi with %ld steps is %lf\n", num steps, pi);
```

We will talk about explicit mapping of variables between the host and a device latter. This uses the **map()** clause.

When using the loop directive, you need to explicitly define this mapping for the reduction variable.

This will all make sense when we cover the **map()** clause later on.

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:

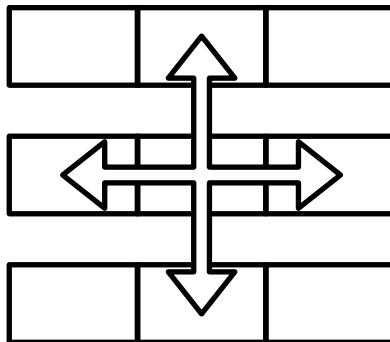
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

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

- 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: 5-point stencil code

```

const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {           Loop over time steps

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {           Loop over NxN spatial domain
            u_tmp[i+j*n] = r2 * u[i+j*n] +  

                r * ((i < n-1) ? u[i+1+j*n] : 0.0) +  

                r * ((i > 0)   ? u[i-1+j*n] : 0.0) +  

                r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +  

                r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
        }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
}

```

Heat program (heat.c) ...

```
// Loop over time steps
for (int t = 0; t < nsteps; ++t) {

    // solve over spatial domain for step t
    solve(n, alpha, dx, dt, u, u_tmp);

    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
}
```

- Takes two optional command line arguments: <ncells> <nsteps>
 - E.g. ./heat 1000 10
 - 1000x1000 cells, 10 timesteps (the default problem size).
- If no command line arguments are provided, it uses a default:
 - These two commands both run the default problem size of 1000x1000 cells, 10 timesteps.
 - ./heat
 - ./heat 1000 10
- A sensible bigger problem is 8000 x 8000 cells and 10 timesteps.

Heat program (heat.c) ...

```
// Loop over time steps
for (int t = 0; t < nsteps; ++t) {

    // solve over spatial domain for step t
    solve(n, alpha, dx, dt, u, u_tmp);

    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
}
```

Note: Swapping pointer on the host before entering the target region on the next iteration works on a GPU.

When you map pointers between the host and the device, OpenMP remembers the address.

Swapped addresses on the hosts swaps addresses on the device

Exercise: parallel stencil (heat)

- Take the provided heat stencil code (heat.c)
 1. Add OpenMP directives to parallelize the loops on the **GPU**
 2. Add OpenMP directives to parallelize the loops on the **CPU**
- Most of the runtime occurs in the solve() routine. Focus on that function. The rest of the code is there to just support the work inside solve.

- double omp_get_wtime();
- #pragma omp parallel
- #pragma omp for
- #pragma omp parallel for
- #pragma omp task
- #pragma omp taskwait
- #pragma single
- #pragma omp target
- #pragma omp loop
- Plus the clauses

- private(), firstprivate(), reduction(+:var), collapse(n)
- map(to:vptr[Lower:Count]) map(from:vptr[Lower:Count]) map(tofrom:vptr[Lower:Count])

If you have time, profile
your GPU code using nsys

Default is tofrom: map(vptr[Lower:Count])



Heat diffusion problem: 5-point stencil code

```

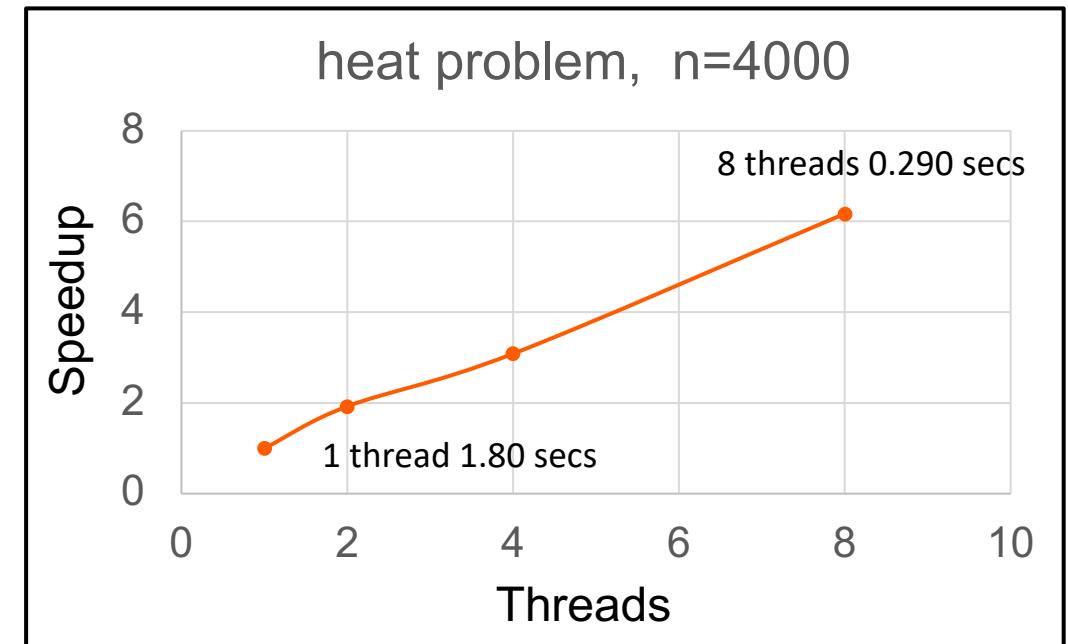
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {

    #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            u_tmp[i+j*n] = r2 * u[i+j*n] + 
                r * ((i < n-1) ? u[i+1+j*n] : 0.0) +
                r * ((i > 0)   ? u[i-1+j*n] : 0.0) +
                r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
                r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
        }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
}

```



Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores. Nvidia HPC Toolkit compiler
nvc –fast –fopenmp heat.c

Heat diffusion problem: 5-point stencil code

```

const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {
    #pragma omp target map(tofrom: u[0:n*n], u_tmp[0:n*n])
    #pragma omp loop
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            u_tmp[i+j*n] = r2 * u[i+j*n] +
                r * ((i < n-1) ? u[i+1+j*n] : 0.0) +
                r * ((i > 0) ? u[i-1+j*n] : 0.0) +
                r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
                r * ((j > 0) ? u[i+(j-1)*n] : 0.0);
        }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
}

```

When you map pointers between the host and the device, OpenMP remembers the address.

Swapped addresses on the hosts swaps addresses on the device

GPU Solver time = 1.40 secs

This isn't much better than the runtime for a single CPU (1.8 secs) and worse than 8 cores on a CPU (0.29 secs).

Why is the performance so bad?

NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
 Nvidia HPC Toolkit compiler
 nvc -fast -mp=gpu -gpu=cc75 heat.c

Heat diffusion problem: 5-point stencil code

```

const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {
    #pragma omp target map(tofrom: u[0:n*n], u_tmp[0:n*n])
    #pragma omp loop
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            u_tmp[i+j*n] = r2 * u[i+j*n] +
                r * ((i < n-1) ? u[i+1+j*n] : 0.0) +
                r * ((i > 0) ? u[i-1+j*n] : 0.0) +
                r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
                r * ((j > 0) ? u[i+(j-1)*n] : 0.0);
        }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
}

```

At the end of each iteration, **copy**
 $(2*N^2)*\text{sizeof}(\text{TYPE})$ bytes
from the device

With a runtime of 1.4 secs (worse than the CPU time) we see that Data Movement dominates performance.

At the beginning of each iteration, **copy**
 $(2*N^2)*\text{sizeof}(\text{TYPE})$ bytes **to** the device

We need to create a **data region** on the GPU that is distinct from the target region.

That way, we can keep the data on the device between target constructs

How do we control how data is mapped onto a device separately from running kernels ... so data is well defined and persistent between kernel invocations?

Finer control over data movement

- Recall that data is mapped to/from device at start/end of target region
 - `#pragma omp target map(tofrom: A[0:N])`
{
 ...
}
- Inefficient to move data around all the time
- Want to keep data resident on the device *between* target regions
- Will explain how to interact with the device data environment

Target data directive

- The **target data** construct creates a target data region
 - ... use **map** clauses for explicit data management

Data is mapped onto the device at the beginning of the construct

```
#pragma omp target data map(to:A[0:N], B[0:M]) map(from: C[0:P])  
{
```

```
#pragma omp target  
{do lots of stuff with A, B and C}
```

{do something on the host}

```
#pragma omp target  
{do lots of stuff with A, B, and C}
```

one or more **target regions** work within the **target data region**

Data is mapped back to the host at the end of the target data region

Target update directive

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

```
#pragma omp target data map(to: A[0:N],B[0:M]) map(from: C[0:P])
{
    #pragma omp target
        {do lots of stuff with A, B and C on the device}

    #pragma omp target update from(A[0:N])

    host_do_something_with(A)

    #pragma omp target update to(A[0:N])

    #pragma omp target
        {do lots more stuff with A, B, and C on the device}
}
```

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

Target update details

- **#pragma omp target update clause[[[,]clause]...]**
- Creates a target task to handle data movement between the host and a device.
- Clause: a motion-clause:
 - to(list)
 - from(list)

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:
#pragma omp target enter data map(...)
#pragma omp 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.

Target enter/exit data example

```
void init_array(int *A, int N) {  
    for (int i = 0; i < N; ++i)  
        A[i] = i;  
    #pragma omp target enter data map(to: A[0:N])  
}  
  
int main(void) {  
  
    int N = 1024;  
    int *A = malloc(sizeof(int) * N);  
    init_array(A, N);  
  
    #pragma omp target  
    #pragma loop  
    for (int i = 0; i < N; ++i)  
        A[i] = A[i] * A[i];  
  
    #pragma omp target exit data map(from: A[0:N])  
}
```

Target enter/exit data details

- **#pragma omp target enter data clause[[[,]clause]...]**
- Creates a target task to handle data movement between the host and a device.
- clause is one of the following:
 - if(scalar-expression)
 - device(integer-expression)
 - map (map-type: list)

Exercise

- Modify your parallel heat code from the last exercise.
- Use the ‘target data’ family of constructs to control the device data environment.
- Minimize data movement with map clauses to minimize data movement.
- Question ... will the pointer swap on the host still work?
 - `#pragma omp target`
 - `#pragma omp target enter data`
 - `#pragma omp target exit data`
 - `#pragma omp target update`
 - `map(to:list) map(from:list) map(tofrom:list)`
 - `#pragma omp teams distribute parallel for simd`

Solution: Pointer swapping in action

```
#pragma omp target enter data map(to: u[0:n*n], u_tmp[0:n*n])
```

Copy data to device
before iteration loop

```
for (int t = 0; t < nsteps; ++t) {
```

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

Update solve() routine to remove map clauses:
~~#pragma omp target map(u_tmp[0:n*n], u[0:n*n])~~

```
// Pointer swap
```

```
tmp = u;
```

```
u = u_tmp;
```

```
u_tmp = tmp;
```

```
}
```

Pointer-swap on the host works. Why?

The pointers (u and u_tmp) are “on the stack” scalars the value of which is a pointer to memory. They are copied onto the device at the target construct.

The association between host and device addresses is fixed with the start of a target data region. Hence, as you swap the pointers, the references to the addresses in device memory are swapped i.e. pointer-swapping on the host works.

```
#pragma omp target exit data map(from: u[0:n*n])
```

Copy data from device
after iteration loop

Data movement summary

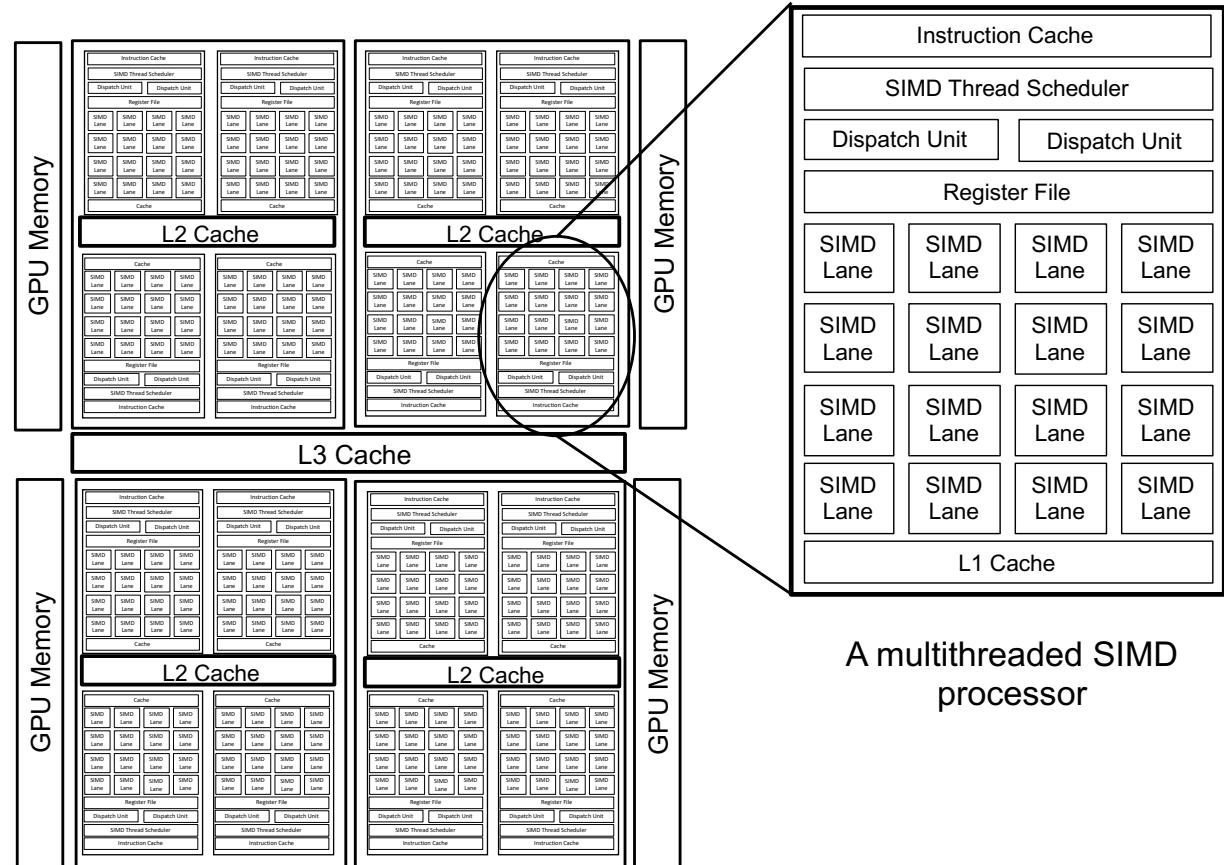
- Data transfers between host/device occur at:
 - Beginning and end of **target** region
 - Beginning and end of **target data** region
 - At the **target enter data** construct
 - At the **target exit data** construct
 - At the **target update** construct
- Can use **target data** and **target enter/exit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

Getting the data movement between host memory and device memory is key.

What are the other major issues to consider when optimizing performance?

Occupancy: Keep all the GPU resources busy

- In our “GPU cartoon” we have 16 multithreaded SIMD processors each with 16 SIMD lanes For a total of $16^2=256$ processing elements.
- You want all resources busy at all times. You do that by keeping excess work for the multithreaded SIMD processors ... if they are other busy on some high latency operation, you want a new work-group is ready to be scheduled for execution.
- Occupancy having enough work-groups to keep the GPU busy. To support high occupancy, you need many more work-items than SIMD-lanes.



```
#pragma omp parallel for  
for(int i=0;i<N;i++)  
    for(int j=0;j<N;j++)  
        for(int k=0;k<N;k++)  
            *(C+(i*N+j)) += *(A+(i* N +k)) * *(B+(k* N +j));
```

Parallelize i-loop
parallelism O(N)

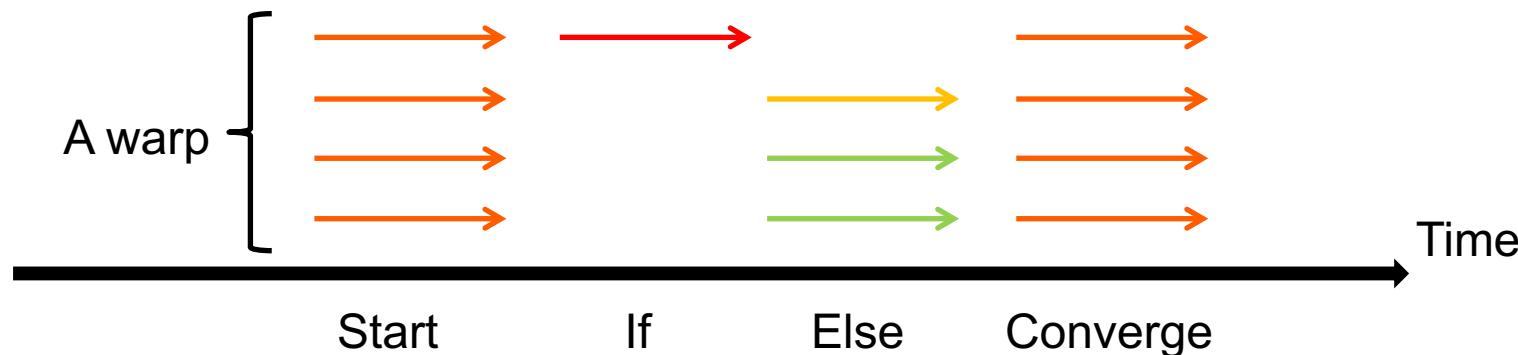


```
#pragma omp parallel for collapse(2)  
for(int i=0;i<N;i++)  
    for(int j=0;j<N;j++)  
        for(int k=0;k<N;k++)  
            *(C+(i*N+j)) += *(A+(i* N +k)) * *(B+(k* N +j));
```

Parallelize combined i/j-loops
parallelism O(N²)

Converged Execution: Single Instruction Multiple Data

- Individual work-items of a warp start together at the same program address
- Each work-item has its own instruction address counter and register state
 - Each work-item is free to branch and execute independently
 - Supports the Single Program Multiple Data (SPMD) pattern.
- Branch behavior
 - Each branch will be executed serially
 - Work-items not following the current branch will be disabled



Converged Execution: Branching

- GPUs tend not to support speculative execution, which means that branch instructions have high latency
- This latency can be hidden by switching to alternative work-items/work-groups, but avoiding branches where possible is still a good idea to improve performance
- When different work-items executing within the same SIMD ALU array take different paths through conditional control flow, we have ***divergent branches*** (vs. ***uniform branches***)
- Divergent branches are bad news: some work-items will stall while waiting for the others to complete
- We can use predication, selection and masking to convert conditional control flow into straight line code and significantly improve the performance of code that has lots of conditional branches

Branching

Conditional execution

```
// Only evaluate expression  
// if condition is met  
if (a > b)  
{  
    acc += (a - b*c);  
}
```

Selection and masking

```
// Always evaluate expression  
// and mask result  
temp = (a - b*c);  
mask = (a > b ? 1.f : 0.f);  
acc += (mask * temp);
```

Coalesced memory accesses

- **Coalesced memory accesses** are key for high performance code, especially on GPUs
- In principle, it's very simple, but frequently requires transposing or transforming data on the host before sending it to the GPU
- Sometimes this is an issue of Array of Structures vs. Structure of Arrays (AoS vs. SoA)

Memory layout is critical to performance

- Structure of Arrays vs. Array of Structures

- Array of Structures (AoS) more natural to code:

```
struct Point{ float x, y, z, a; };
```

```
Point *Points;
```



- Structure of Arrays (SoA) suits memory coalescence in vector units

```
struct { float *x, *y, *z, *a; } Points;
```



Adjacent work-items/vector-lanes like to access adjacent memory locations

Coalescence

- **Coalesce** - to combine into one
- Coalesced memory accesses are key for high bandwidth
- Simply, it means, if thread i accesses memory location n then thread $i+1$ accesses memory location $n+1$
- In practice, it's not quite as strict...

```
for (int id = 0; id < size; id++)
{
    // ideal
    float val1 = memA[id];

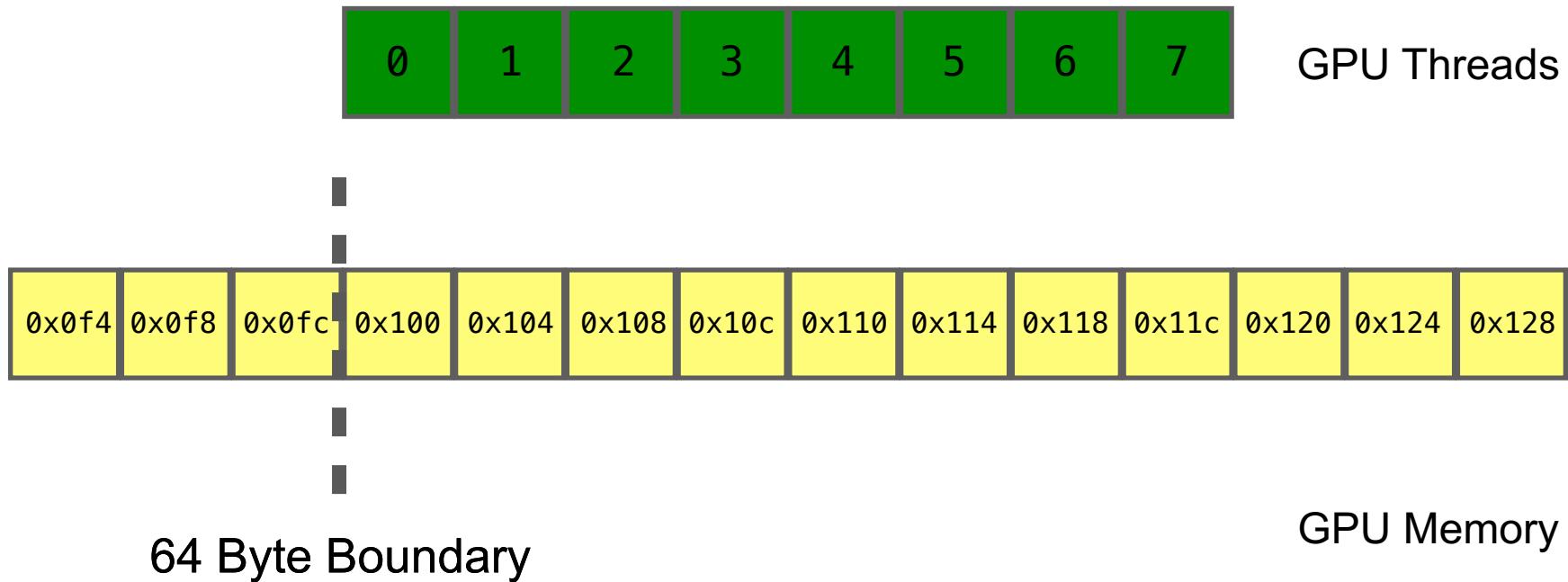
    // still pretty good
    const int c = 3;
    float val2 = memA[id + c];

    // stride size is not so good
    float val3 = memA[c*id];

    // terrible
    const int loc =
        some_strange_func(id);

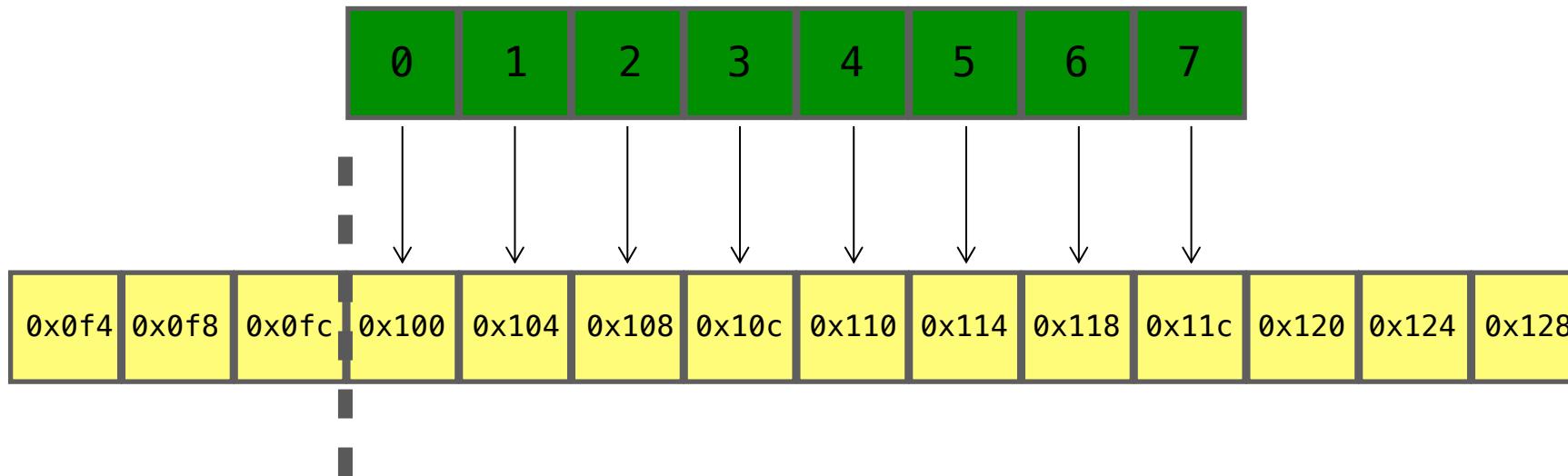
    float val4 = memA[loc];
}
```

Memory access patterns



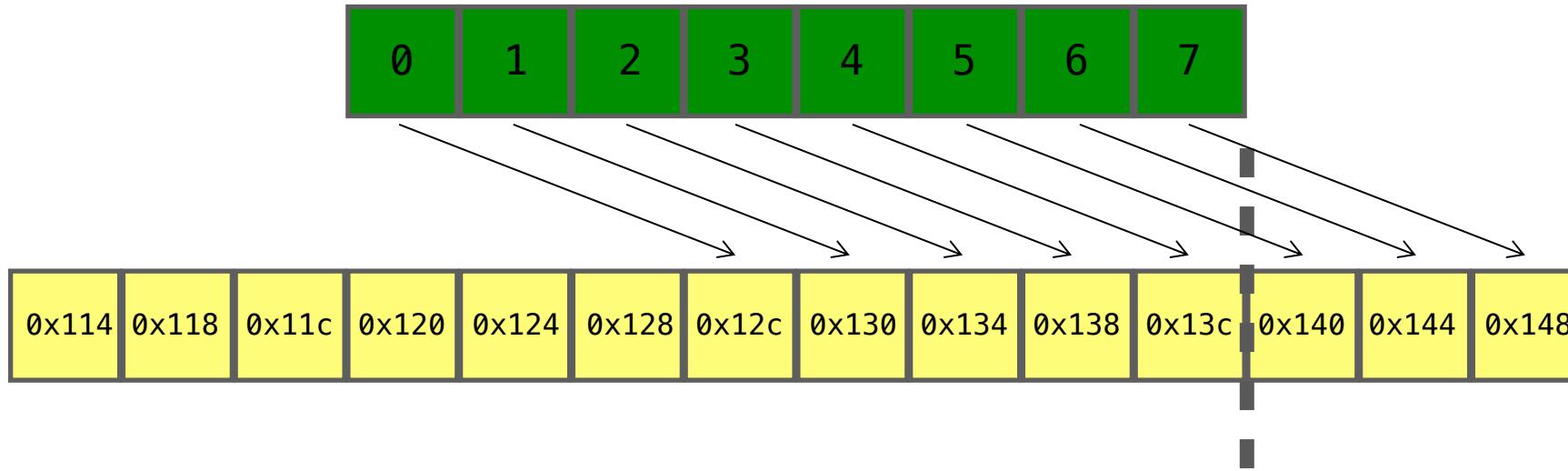
Memory access patterns

```
float val1 = memA[id];
```



Memory access patterns

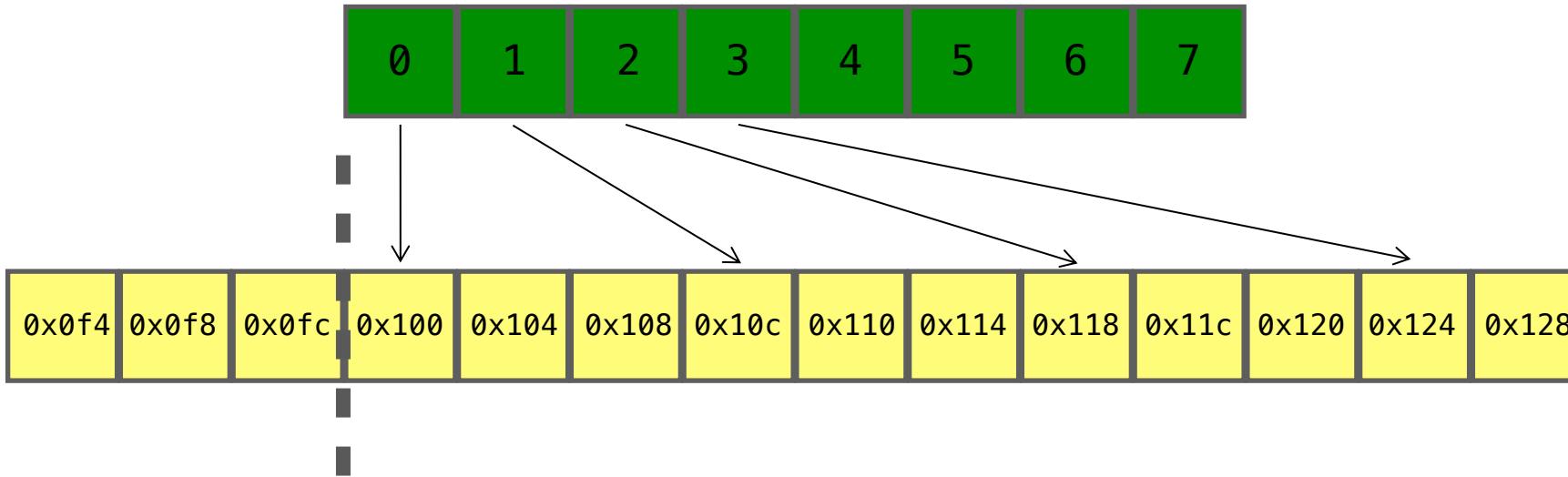
```
const int c = 3;  
float val2 = memA[id + c];
```



64 Byte Boundary

Memory access patterns

```
float val3 = memA[3*id];
```



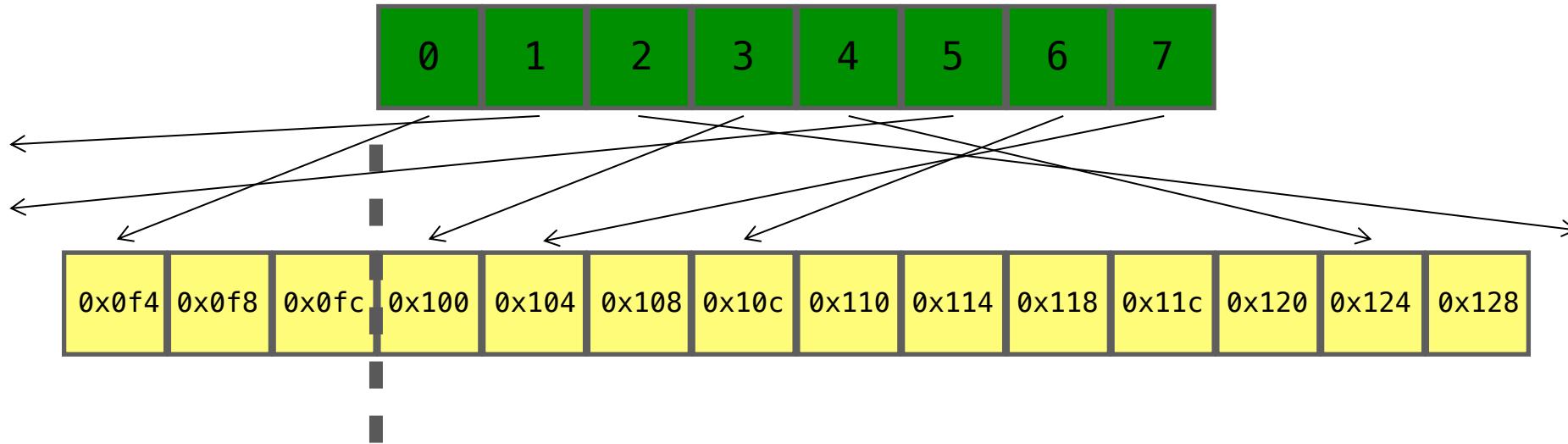
64 Byte Boundary

Strided access results in multiple
memory transactions (and
kills throughput)

Memory access patterns

```
const int loc =  
    some_strange_func(id);
```

```
float val4 = memA[loc];
```



64 Byte Boundary

Exercise

- Optimize the stencil ‘solve’ kernel.
- Start with your code with optimized memory movement from the last exercise.
- Experiment with the optimizations we’ve discussed.
- Focus on the memory access pattern.
- Try different input sizes to see the effect of the optimizations.
- Keep an eye on the solve time as reported by the application.

Solution (only the solve function): collapse + swap loop order

```
// Compute the next timestep, given the current timestep
void solve(const int n, const double alpha, const double dx, const double dt, const double * restrict u,
double * restrict u_tmp) {
    // Finite difference constant multiplier
    const double r = alpha * dt / (dx * dx);
    const double r2 = 1.0 - 4.0*r;

    // Loop over the nxn grid
    #pragma omp target
    #pragma omp loop collapse(2)
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            // 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[i+j*n] = r2 * u[i+j*n] +
                           r * ((i < n-1) ? u[i+1+j*n] : 0.0) +
                           r * ((i > 0) ? u[i-1+j*n] : 0.0) +
                           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
                           r * ((j > 0) ? u[i+(j-1)*n] : 0.0);
        }
    }
}
```

Create more work ... to better fill the processing elements of the GPU

Swap the i and j loops so that the $i+j \cdot n$ memory accesses are contiguous

Heat diffusion problem: 5-point stencil code

Parallel CPU and GPU results, n=4000

```
const double r = alpha * dt / (dx * dx);  
const double r2 = 1.0 - 4.0*r;  
// malloc and initialize u_tmp and u (code not shown)
```

```
for (int t = 0; t < nsteps; ++t) {
```

```
#pragma omp parallel for
```

```
for (int i = 0; i < n; ++i) {
```

```
    for (int j = 0; j < n; ++j) {
```

```
        u_tmp[i+j*n] = r2 * u[i+j*n] +  
            r * ((i < n-1) ? u[i+1+j*n] : 0.0) +  
            r * ((i > 0) ? u[i-1+j*n] : 0.0) +  
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +  
            r * ((j > 0) ? u[i+(j-1)*n] : 0.0);
```

```
}
```

```
}
```

```
// Pointer swap for next step
```

```
tmp = u;
```

```
u = u_tmp;
```

```
u_tmp = tmp;
```

```
}
```

This is the ij loop order.
Swap these loops to get
the ji order.

GPU

ij without timing
enter and exit data

0.056830

Num threads	ij loop order	ji loop order
1	1.512849	0.262260
2	0.776229	0.132453
4	0.400822	0.064220
8	0.227317	0.046586

Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.

Nvidia HPC Toolkit compiler nvc-fast-fopenmp heat.c

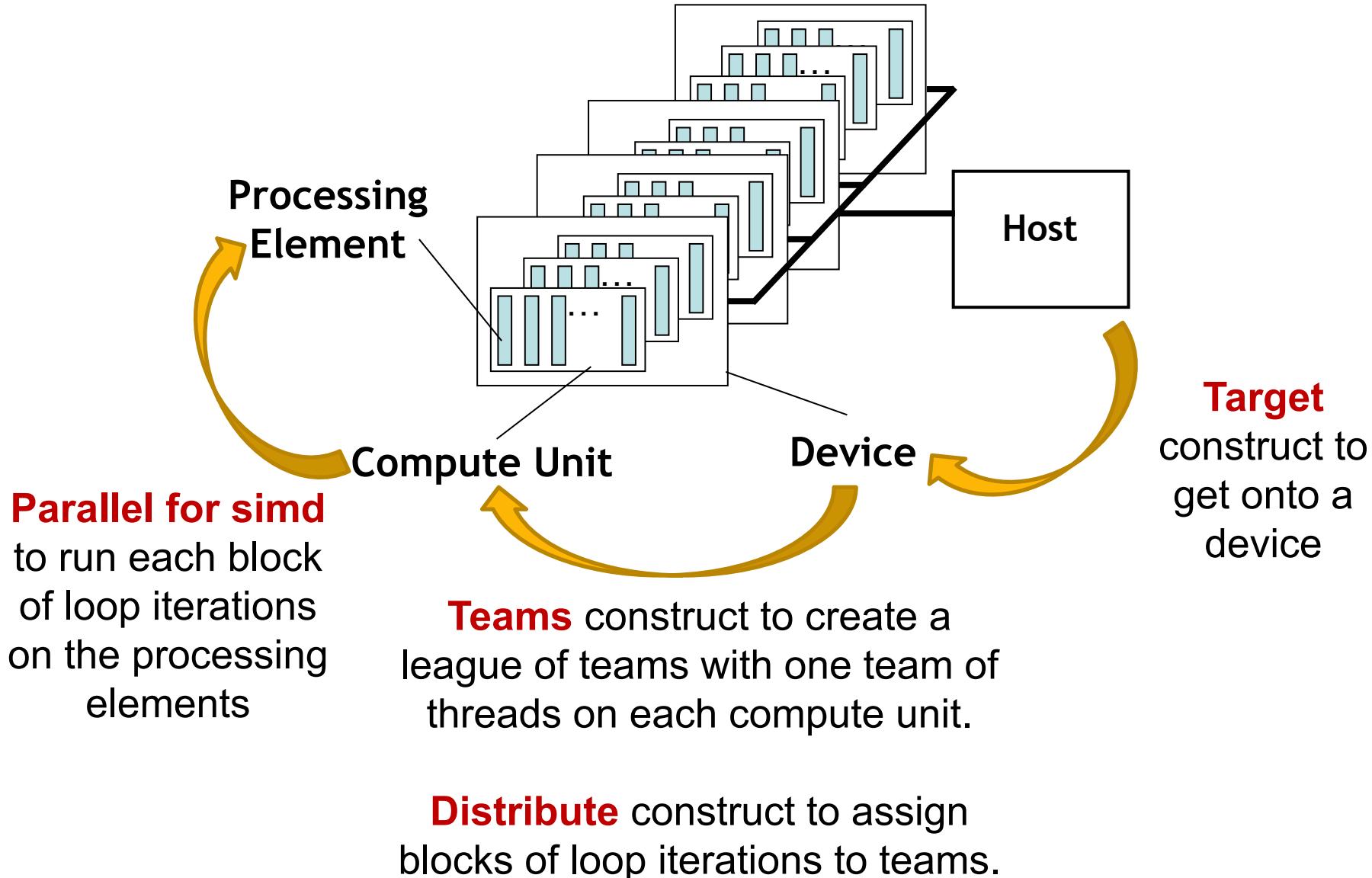
All times in seconds

This GPU code used the
target enter data and
target exit data

NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
Nvidia HPC Toolkit compiler. nvc -fast -mp=gpu heat.c

The loop construct is great, but sometimes you want more control.

Our host/device Platform Model and OpenMP



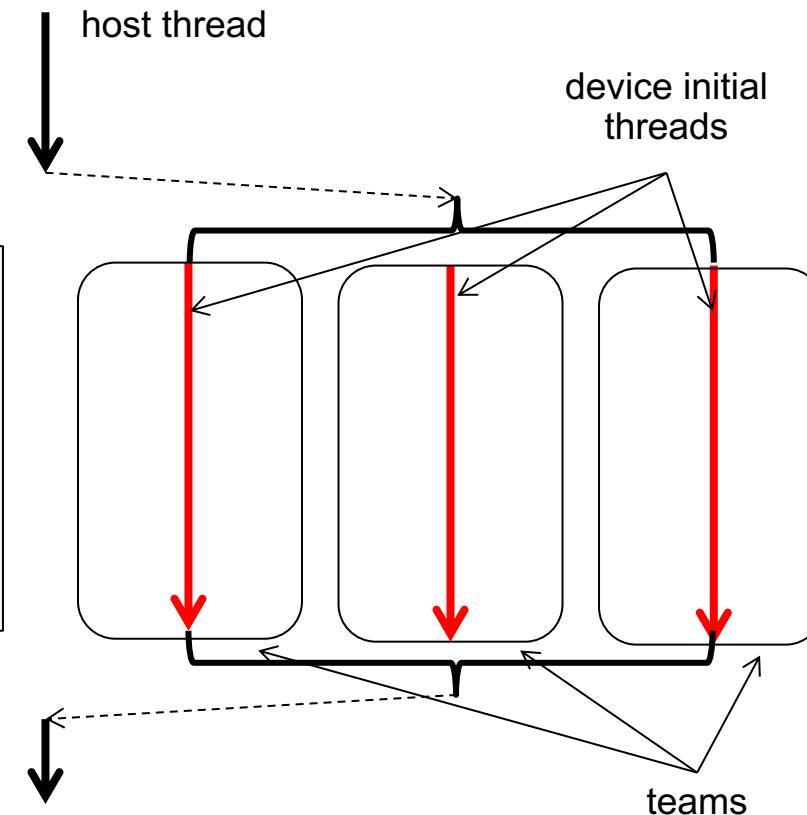
teams and distribute constructs

- The **teams** construct
 - Similar to the **parallel** construct
 - It starts a league of thread teams
 - Each team in the league starts as one initial thread – a team of one
 - Threads in different teams cannot synchronize with each other
 - The construct must be “perfectly” nested in a **target** construct
- The **distribute** construct
 - Similar to the **for** construct
 - Loop iterations are workshared across the initial threads in a league
 - No implicit barrier at the end of the construct
 - **dist_schedule(*kind[, chunk_size]*)**
 - If specified, scheduling kind must be static
 - Chunks are distributed in round-robin fashion in chunks of size **chunk_size**
 - If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

Create a league of teams and distribute a loop among them

- teams construct
- distribute construct

```
#pragma omp target
#pragma omp teams
#pragma omp distribute
for (i=0;i<N;i++)
...
...
```



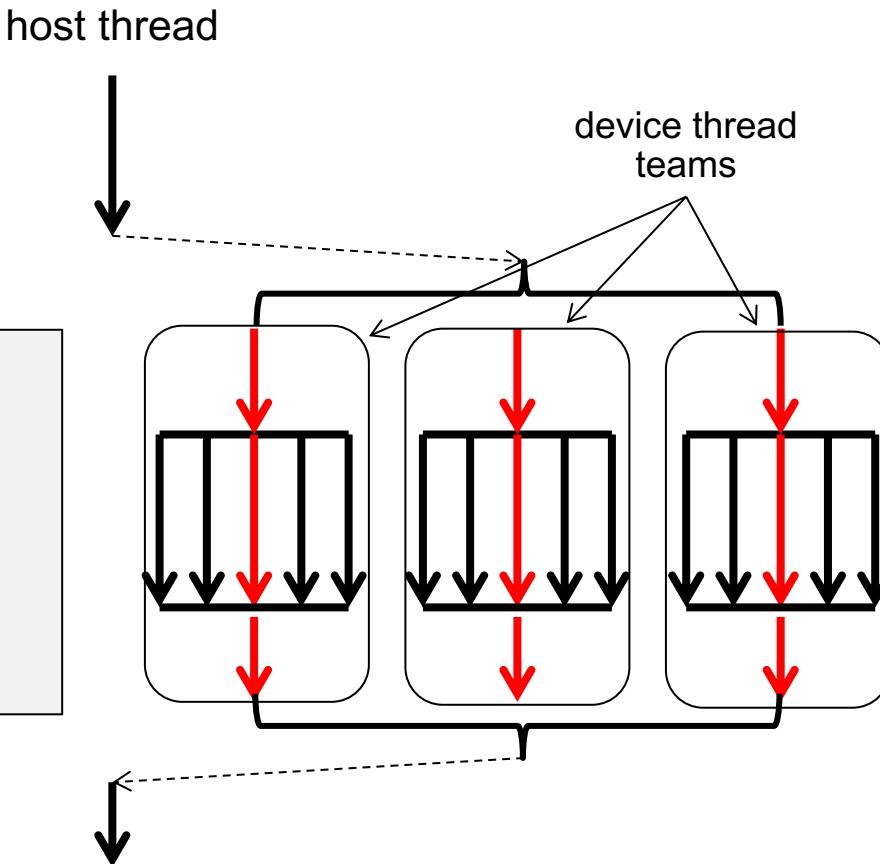
- Transfer execution control to **MULTIPLE** device initial threads
- Workshare loop iterations across the initial threads.

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for simd

```
#pragma omp target
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for simd
for (i=0;i<N;i++)
...

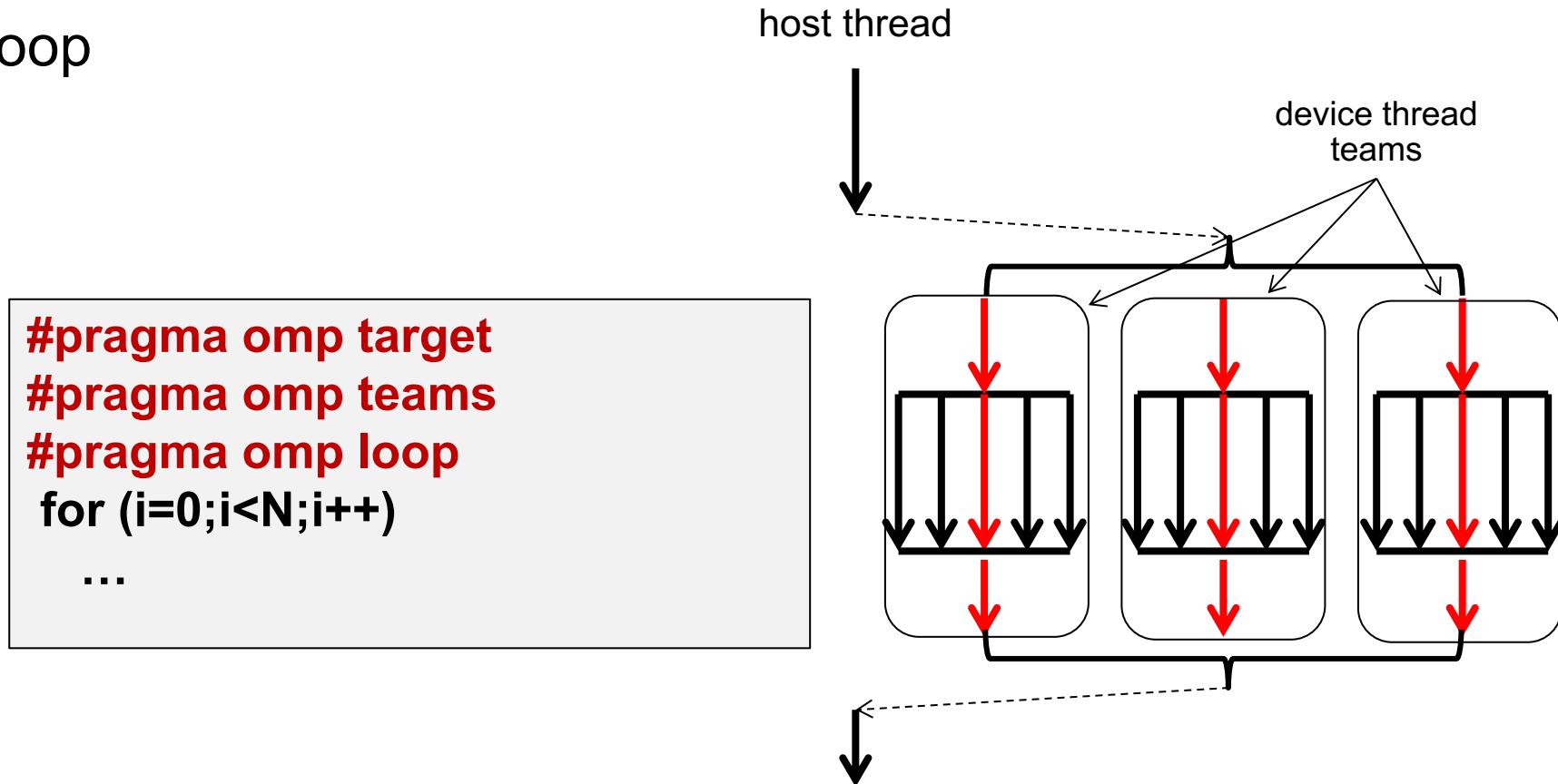
```



- Transfer execution control to **MULTIPLE** device initial threads
 - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
 - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- loop



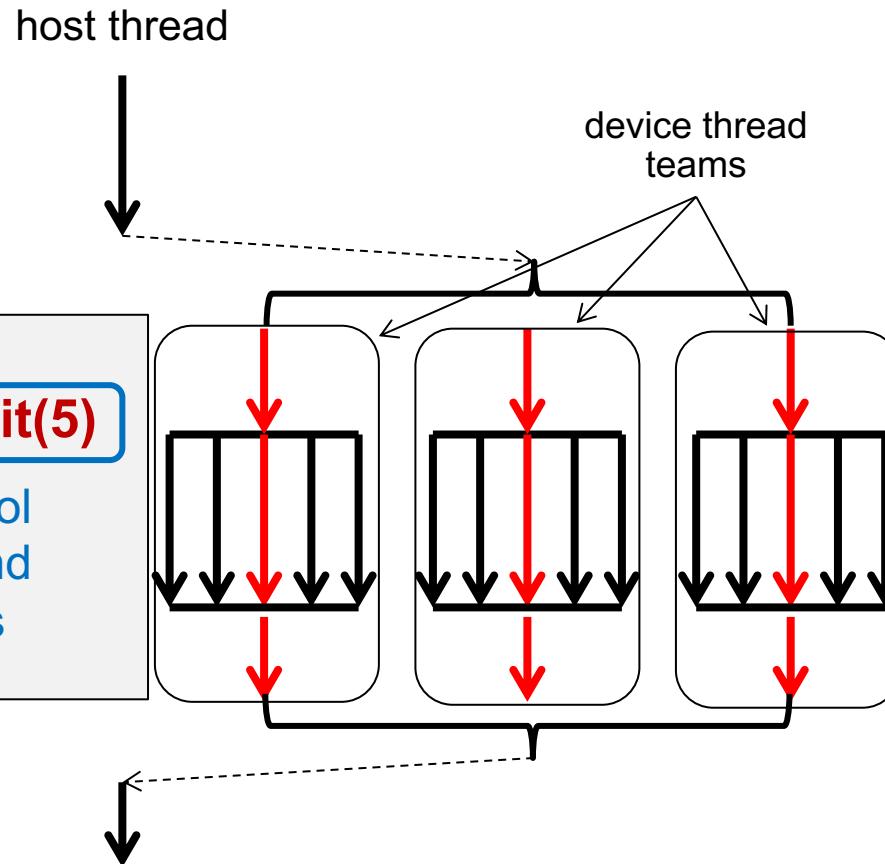
- Transfer execution control to **MULTIPLE** device initial threads
 - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
 - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for simd

```
#pragma omp target
#pragma omp teams num_teams(3) thread_limit(5)
#pragma omp distribute
#pragma omp parallel for simd
for (i=0;i<N;i++)
...
...
```

Explicit control
of number and
size of teams

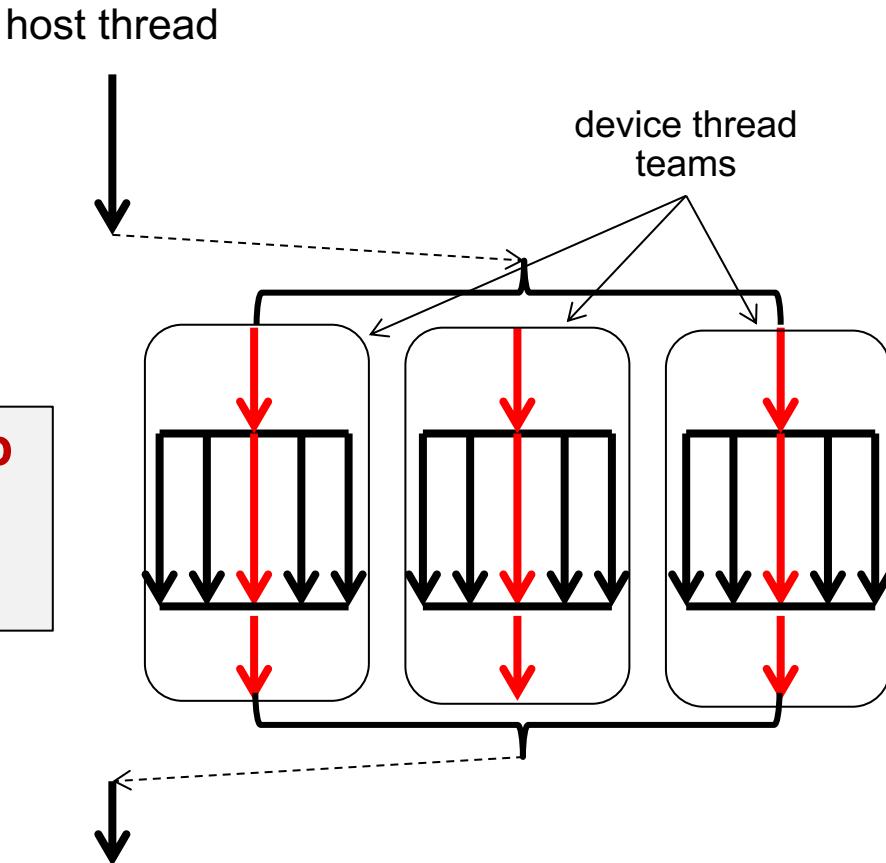


- Transfer execution control to **MULTIPLE** device initial threads
 - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
 - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- Combined construct

```
#pragma omp target teams loop  
for (i=0;i<N;i++)  
...
```



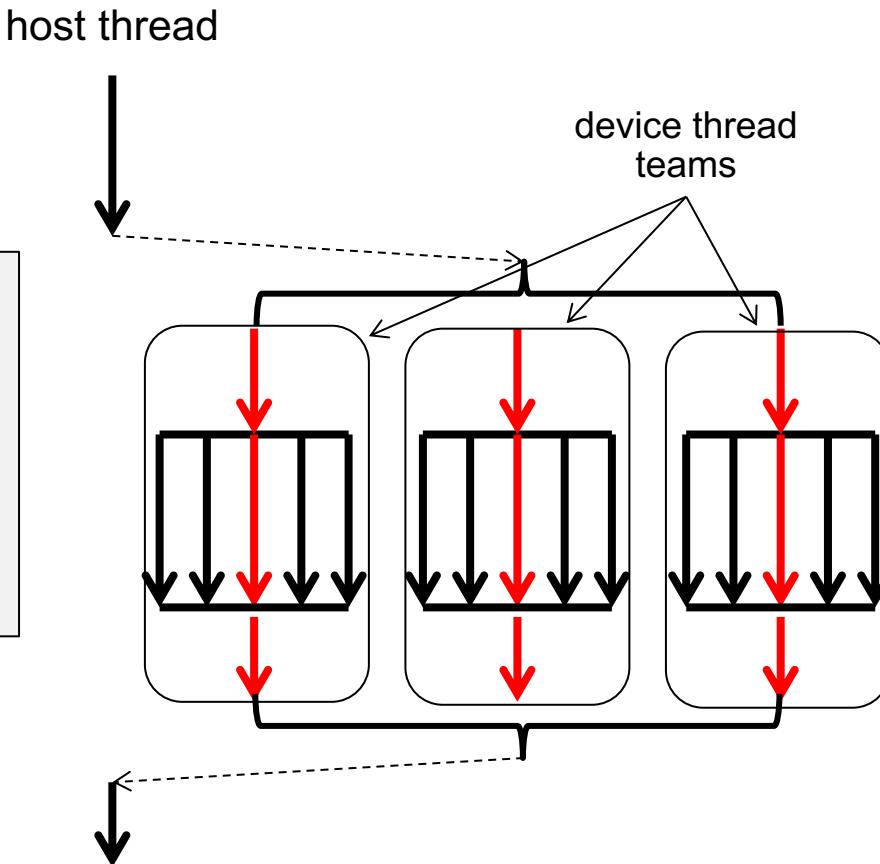
- Transfer execution control to **MULTIPLE** device initial threads
 - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
 - Workshare loop iterations across the threads in a team (parallel for)

Create a league of teams and distribute a loop among them and run each team in parallel with its partition of the loop

- teams distribute
- parallel for simd

Works with nested loops as well

```
#pragma omp target
#pragma omp teams distribute
for (i=0;i<N;i++)
#pragma omp parallel for simd
for (j=0;j<M;j++)
...
...
```



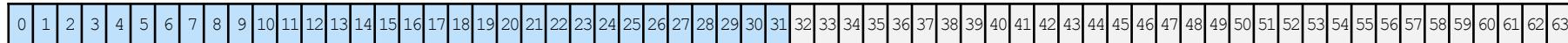
- Transfer execution control to **MULTIPLE** device initial threads
 - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
 - Workshare loop iterations across the threads in a team (parallel for)

Worksharing example

```
#pragma omp target teams distribute parallel for simd \
    num_teams(2) num_threads(4) simdlen(2)
for (i=0; i<64; i++)
    ...
    ...
```

64 iterations assigned to 2 teams;
Each team has 4 threads;
Each thread has 2 SIMD lanes

Distribute iterations across 2 teams



In a team, **workshare** (parallel
for) iterations across 4 threads



In each thread use
SIMD parallelism



Commonly used clauses on teams distribute parallel for simd

- The basic construct* is:

#pragma omp teams distribute parallel for simd [clause[,]clause]...]
for-loops

- The most commonly used clauses are:

- **private(list)** **firstprivate(list)** **lastprivate(list)** **shared(list)**

- behave as data environment clauses in the rest of OpenMP, but note values are only created or copied into the region, not back out “at the end”.

- **reduction(reduction-identifier : list)**

- behaves as in the rest of OpenMP ... but the variable must appear in a map(tofrom) clause on the associated target construct in order to get the value back out at the end (more on this later)

- **collapse(n)**

- Combines loops before the distribute directive splits up the iterations between teams

- **dist_schedule(kind[, chunk_size])**

- only supports kind = static. Otherwise works the same as when applied to a for construct. Note: this applies to the operation of the distribute directive and controls distribution of loop iterations onto teams (NOT the distribution of loop iterations inside a team).

*We often refer to this as the Big Ugly Directive, or **BUD**

There is MUCH more ... beyond what have time to cover

- Do as much as you can with a simple loop construct. It's portable and as compilers improve over time, it will keep up with compiler driven performance improvements.
- But sometimes you need more:
 - Control over number of teams in a league and the size of the teams
 - Explicit scheduling of loop iterations onto the teams
 - Management of data movement across the memory hierarchy: global vs. shared vs. private ...
 - Calling optimized math libraries (such as cuBLAS)
 - Multi-device programming
 - Asynchrony
- Ultimately, you may need to master all those advanced features of GPU programming. But start with loop. Start with how data on the host maps onto the device (i.e. the GPU). Master that level of GPU programming before worrying about the complex stuff.

This is the end ... well almost the end.

**Let's wrap up with a few high-level comments
about the state of GPU programming more
generally**

SIMT Programming models: it's more than just OpenMP

- CUDA:
 - Released ~2006. Made GPGPU programming “mainstream” and continues to drive innovation in SIMT programming.
 - Downside: proprietary to NVIDIA
- OpenCL:
 - Open Standard for SIMT programming created by Apple, Intel, NVIDIA, AMD, and others. 1st release in 2009.
 - Supports CPUs, GPUs, FPGAs, and DSP chips. The leading cross platform SIMT model.
 - Downside: extreme portability means verbose API. Painfully low level especially for the host-program.
- Sycl:
 - C++ abstraction layer implements SIMT model with kernels as lambdas. Closely aligned with OpenCL. 1st release 2014
 - Downside: Cross platform implementations only emerging recently.
- Directive driven programming models:
 - **OpenACC**: they split from an OpenMP working group to create a competing directive driven API emphasizing descriptive (rather than prescriptive) semantics.
 - ~~Downside: NOT an Open Standard. Controlled by NVIDIA.~~ They've made it more open, but it still doesn't add anything you can't do in OpenMP
 - **OpenMP**: Mixes multithreading and SIMT. Semantics are prescriptive which makes it more verbose. A truly Open standard supported by all the key GPU players. And with the loop construct ... its now prescriptive (hence there is no longer any reason for OpenACC to exist)

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

Enqueue the kernel
to execute on the
Grid

CUDA kernel as
function

Unified shared
memory ... allocate
on host, visible on
device too

Vector addition with SYCL

```
// Compute sum of length-N vectors: C = A + B
#include <CL/sycl.hpp>

int main () {
    int N = ... ;
    float *a, *b, *c;
    sycl::queue q;
    *a = (float *)sycl::malloc_shared(N * sizeof(float), q);
    // ... allocate other arrays (b and c), fill with data

    q.parallel_for(sycl::range<1>{N},
                   [=](sycl::id<1> i) {
                       c[i] = a[i] + b[i];
                   });
    q.wait();
}
```

Create a queue
for SYCL
commands

Unified shared
memory ... allocate
on host, visible on
device too

Kernel as a C++
Lambda function
[=] means capture external
variables by value.

Vector addition with OpenACC

- Let's add two vectors together $C = A + B$

Host waits here until the kernel is done. Then the output array c is copied back to the host.

```
void vadd(int n,
          const float *a,
          const float *b,
          float *restrict c)
{
    int i;
#pragma acc parallel loop
    for (i=0; i<n; i++)
        c[i] = a[i] + b[i];
}
int main(){
float *a, *b, *c;  int n = 10000;
// allocate and fill a and b

vadd(n, a, b, c);

}
```

Assure the compiler that c is not aliased with other pointers

Turn the loop into a kernel, move data to a device, and launch the kernel.

Why so many ways to do the same thing?

- The parallel programming model people have failed you ...
 - It's more fun to create something new in your own closed-community that work across vendors to create a portable API
- The hardware vendors have failed you ...
 - Don't you love my “walled garden”? It's so nice here, programmers, just don't even think of going to some other platform since your code is not portable.
- The standards community has failed you ...
 - Standards are great, but they move too slow. OpenACC stabbed OpenMP in the back and I'm pissed, but their comments at the time were spot-on (OpenMP was moving so slow ... they just couldn't wait).
- The applications community failed themselves ...
 - If you don't commit to a standard and use “the next cool thing” you end up with the diversity of overlapping options we have today. Think about what happened with OpenMP and MPI.

Summary

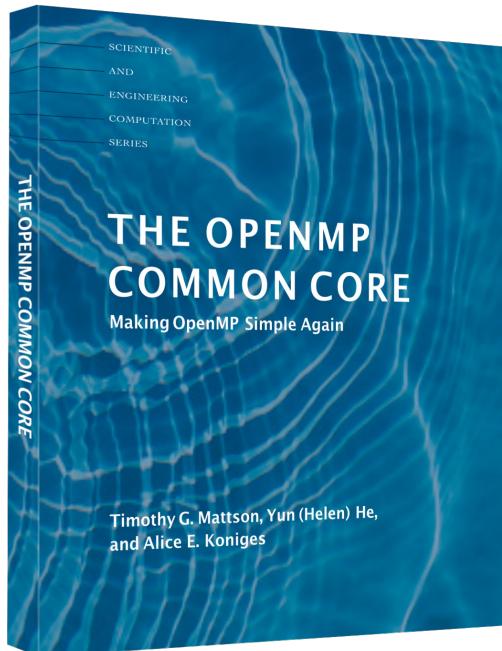
- Application developers ... if you respect yourself, you will only write code using a cross-platform, vendor neutral programming model.
- I am not sure how to pull it off ... but you need to band together and fight back against vendors trying to tie you to their platforms. I don't know how you can make this work, but its up to you.
- GPU programming is fun. But the need to optimize power efficiency will push us to specialized hardware. How to support specialized hardware while honoring portable parallel programming is unclear.

To learn more about OpenMP

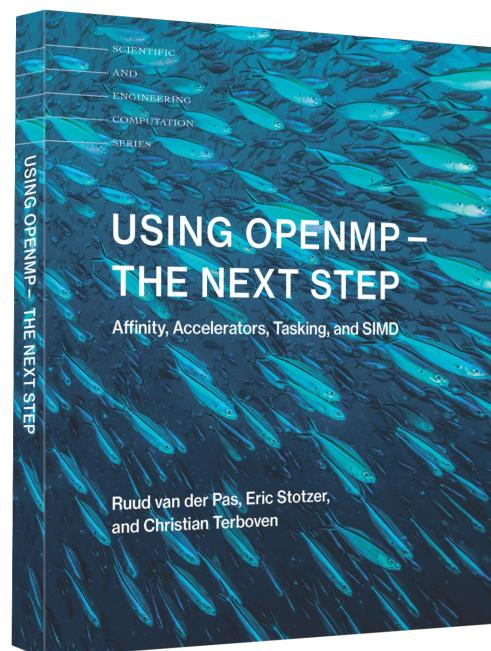
The OpenMP web site has a great deal of material to help you with OpenMP www.openmp.org

Reading the spec is painful ... but each spec has a collection of examples. Study the examples, don't try to read the specs

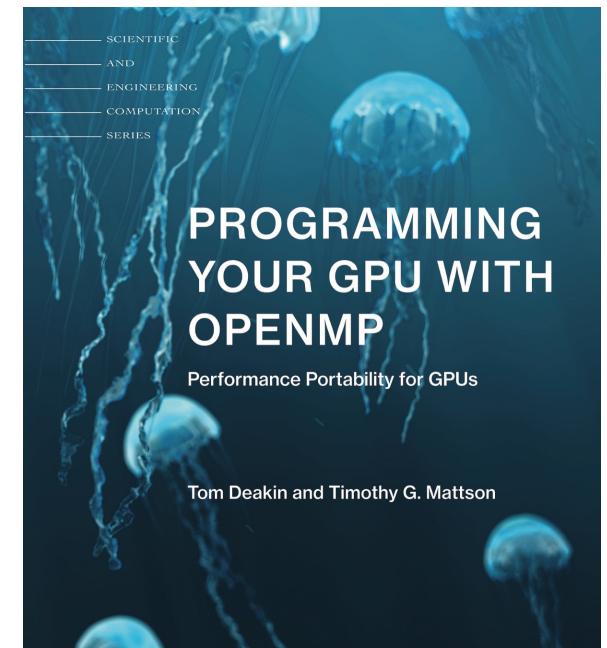
Since the specs are written ONLY for implementors ... programmers need the OpenMP Books to master OpenMP.



Start here ... learn the basics and build a foundation for the future



Learn advanced features in OpenMP including tasking and GPU programming (up to version 4.5)



Learn all the details of GPU programming with OpenMP (up to version 5.2)