

SC22

OpenMP[®]

Programming Your GPU with OpenMP



Tom Deakin
University of Bristol
tom.deakin@bristol.ac.uk



Simon McIntosh-Smith
University of Bristol
simonm@cs.bris.ac.uk



Tim Mattson
Intel Corp.
timothy.g.mattson@intel.com

All the tutorial materials are available online



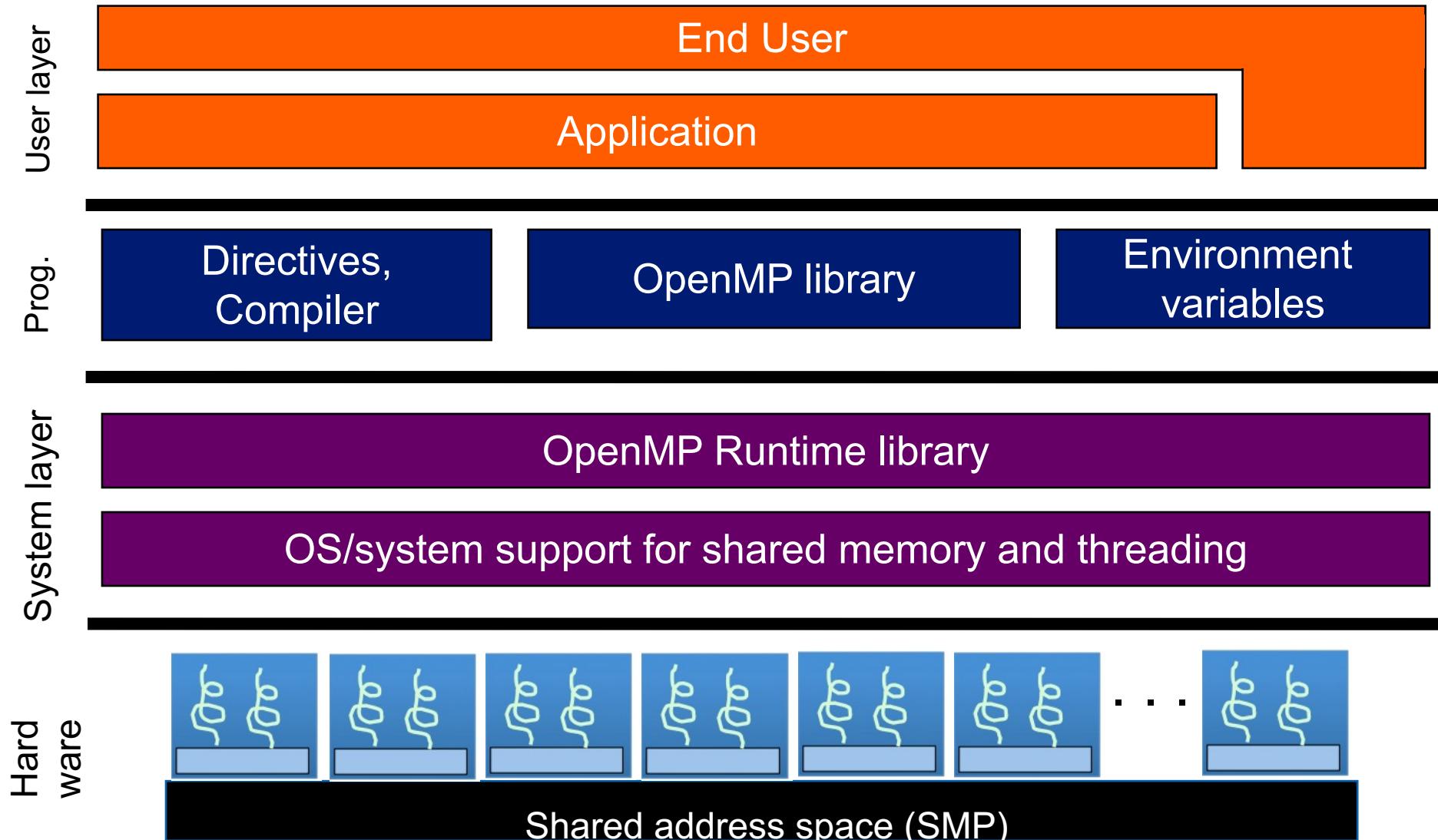
<https://github.com/uob-hpc/openmp-tutorial>

Welcome to the Programming your GPU with OpenMP tutorial!

- GPUs are becoming increasingly important as most Exascale machines will be relying on them
- Given there are now at least 3 mainstream GPU vendors, we need a **portable** way to program them
- OpenMP offload support for GPUs has been maturing nicely in recent years, so this is a good time to learn how to use it
- This will be a **hands-on tutorial**, with a mix of pre-recorded short lectures, interspersed with live exercises



OpenMP basic definitions: the solution stack

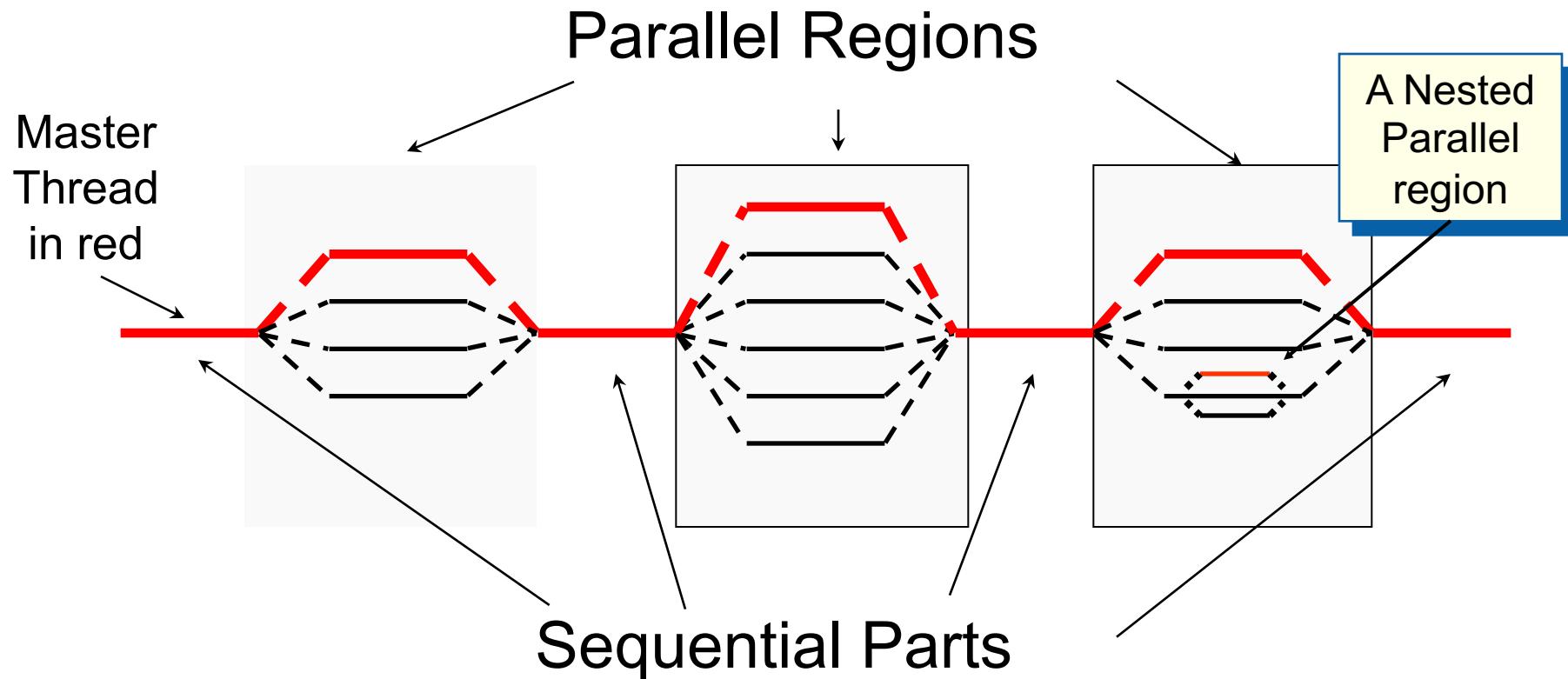


When OpenMP was originally launched, the focus was on **Symmetric Multiprocessing**
.... i.e. lots of threads with “equal cost access” to memory

OpenMP programming model

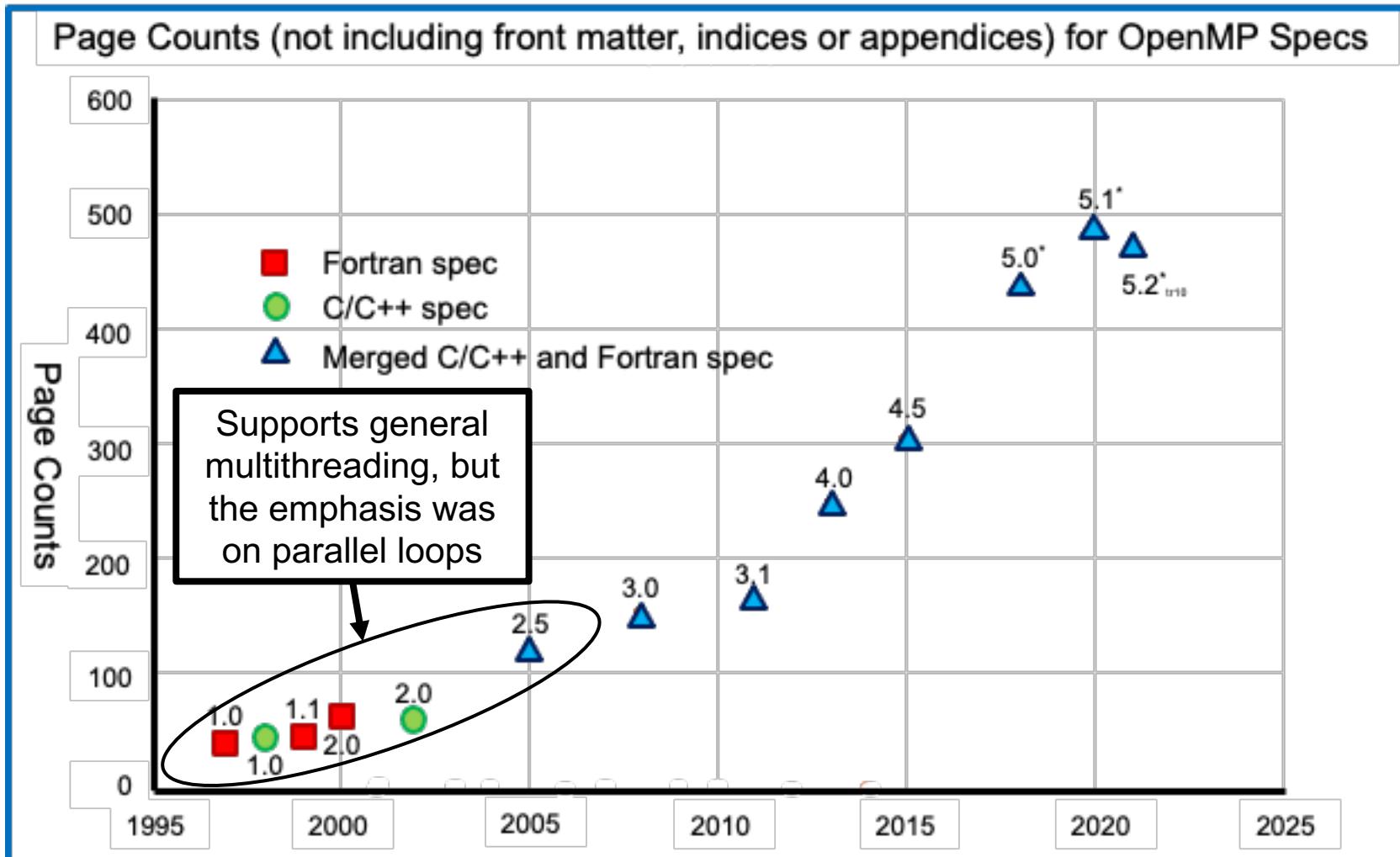
Fork-Join Parallelism:

- ◆ Master thread spawns a team of threads as needed.
- ◆ Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.

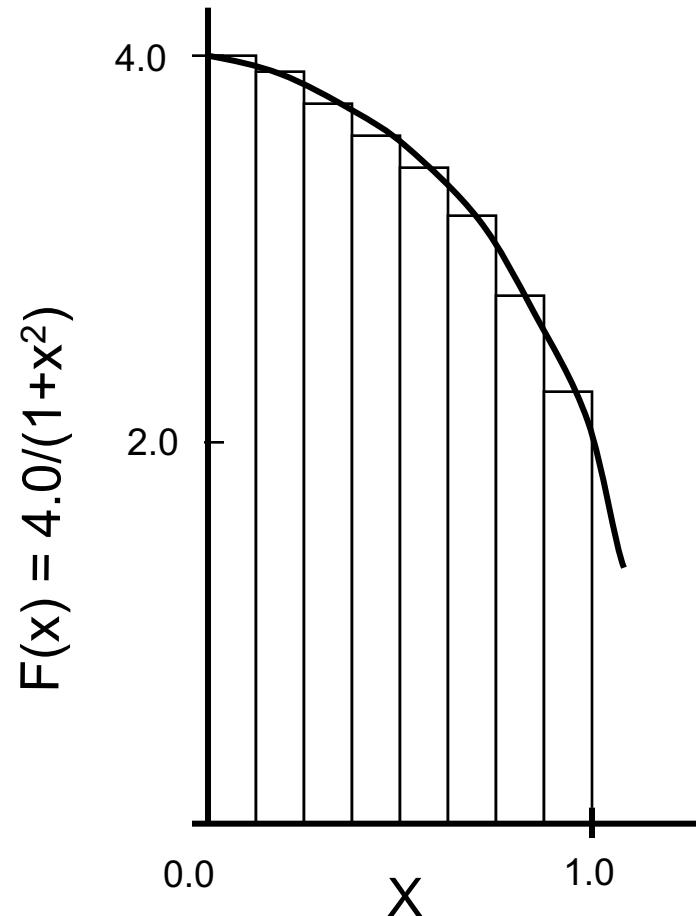


The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.
- The complexity has grown considerably over the years!



Numerical integration: the Pi program



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:

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

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial Pi program

```
static long num_steps = 100000;
double step;
int main ()
{
    int i;    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

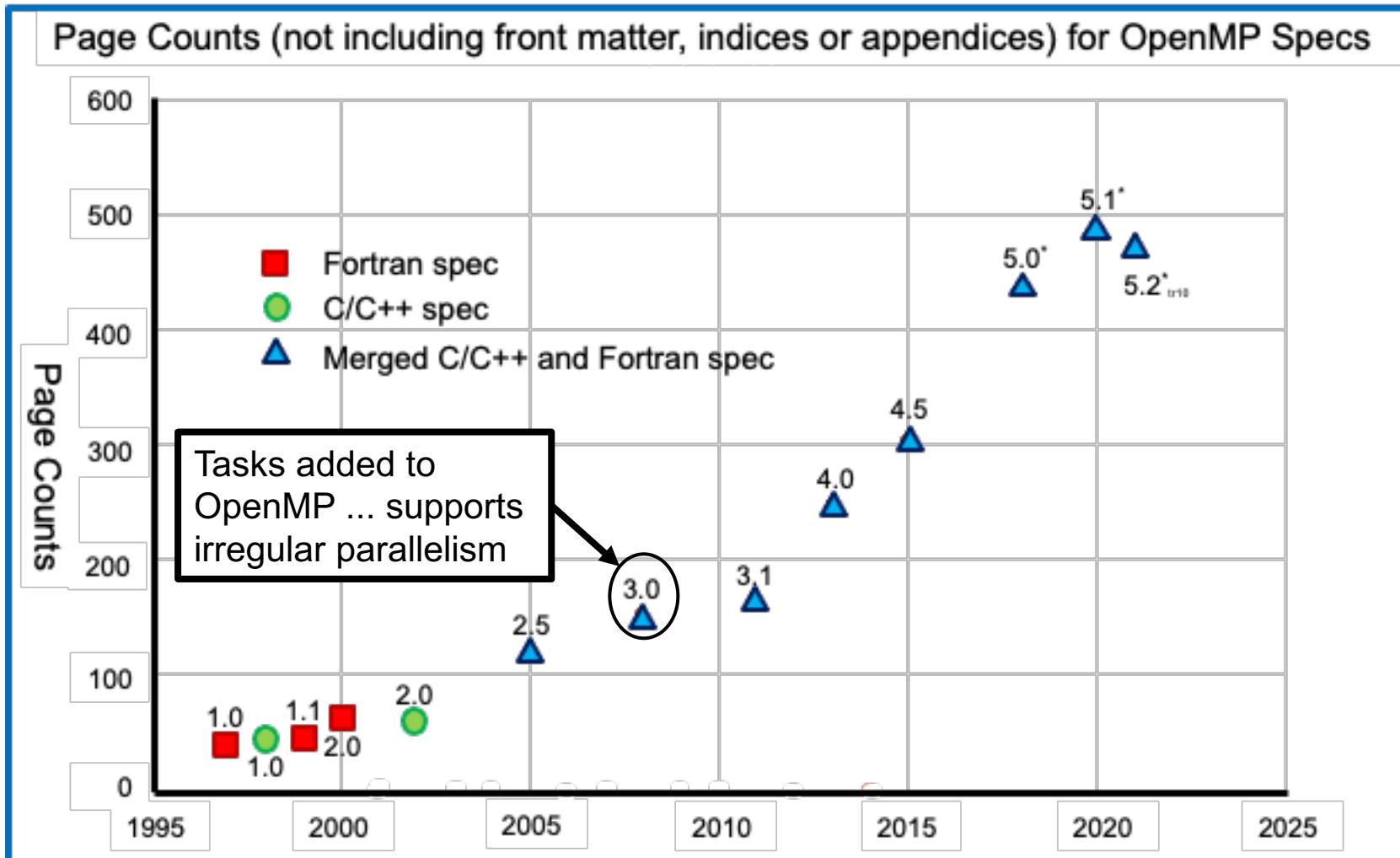
See openmp-tutorial/pi.c

Example: Pi in OpenMP with a loop & reduction

```
#include <omp.h>
static long num_steps = 100000;      double step;
void main ()
{   int i;           double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;           ← Create a scalar local to each thread to hold
        #pragma omp for reduction(+:sum)
        for (i=0; i< num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x), ← Break up loop iterations
                                         and assign them to
                                         threads ... setting up a
                                         reduction into sum.
                                         Note ... the loop index is
                                         local to a thread by default.
        }
    }
    pi = step * sum;
}
```

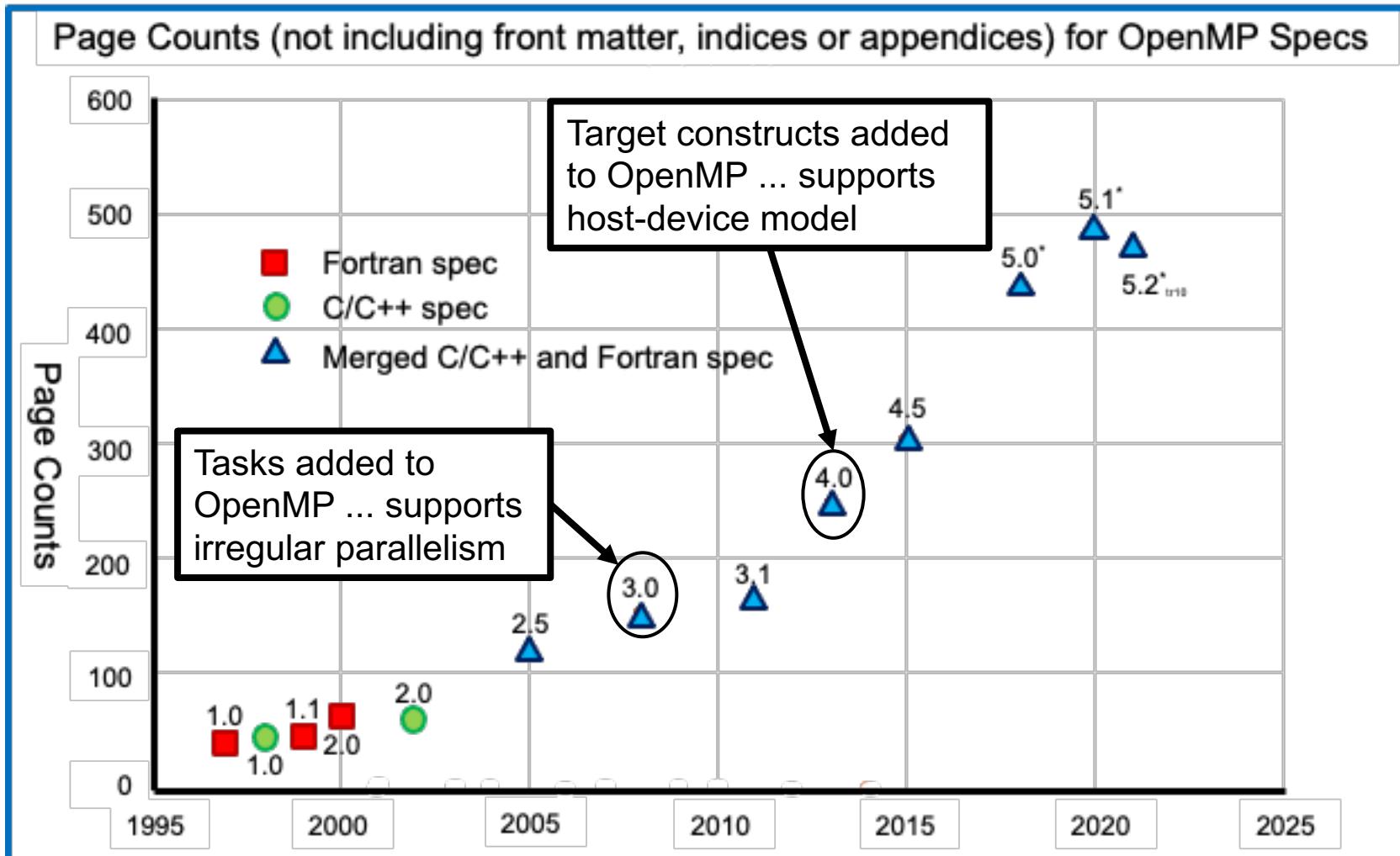
The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.
- The complexity has grown considerably over the years!



The growth of complexity in OpenMP

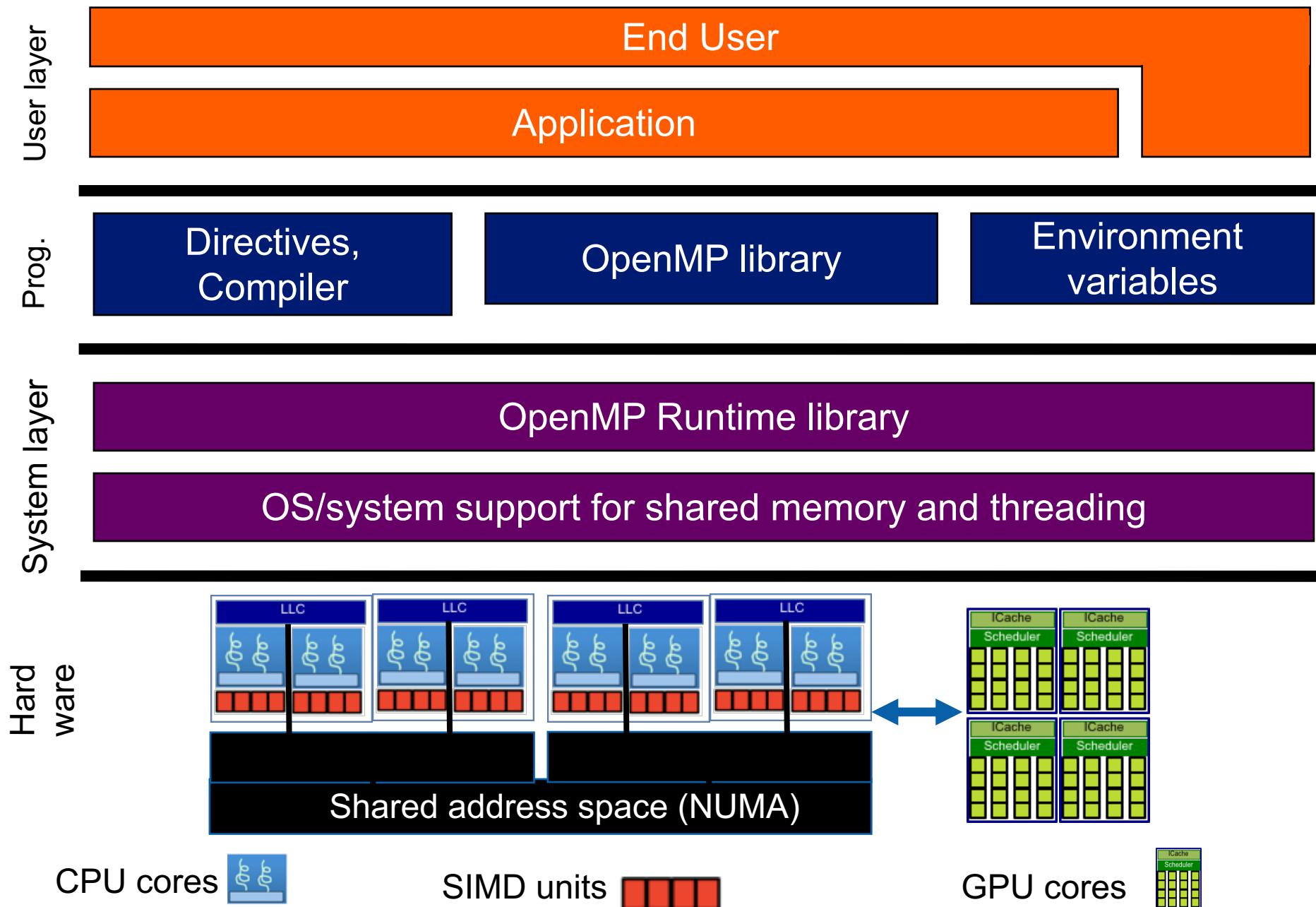
- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.
- The complexity has grown considerably over the years!



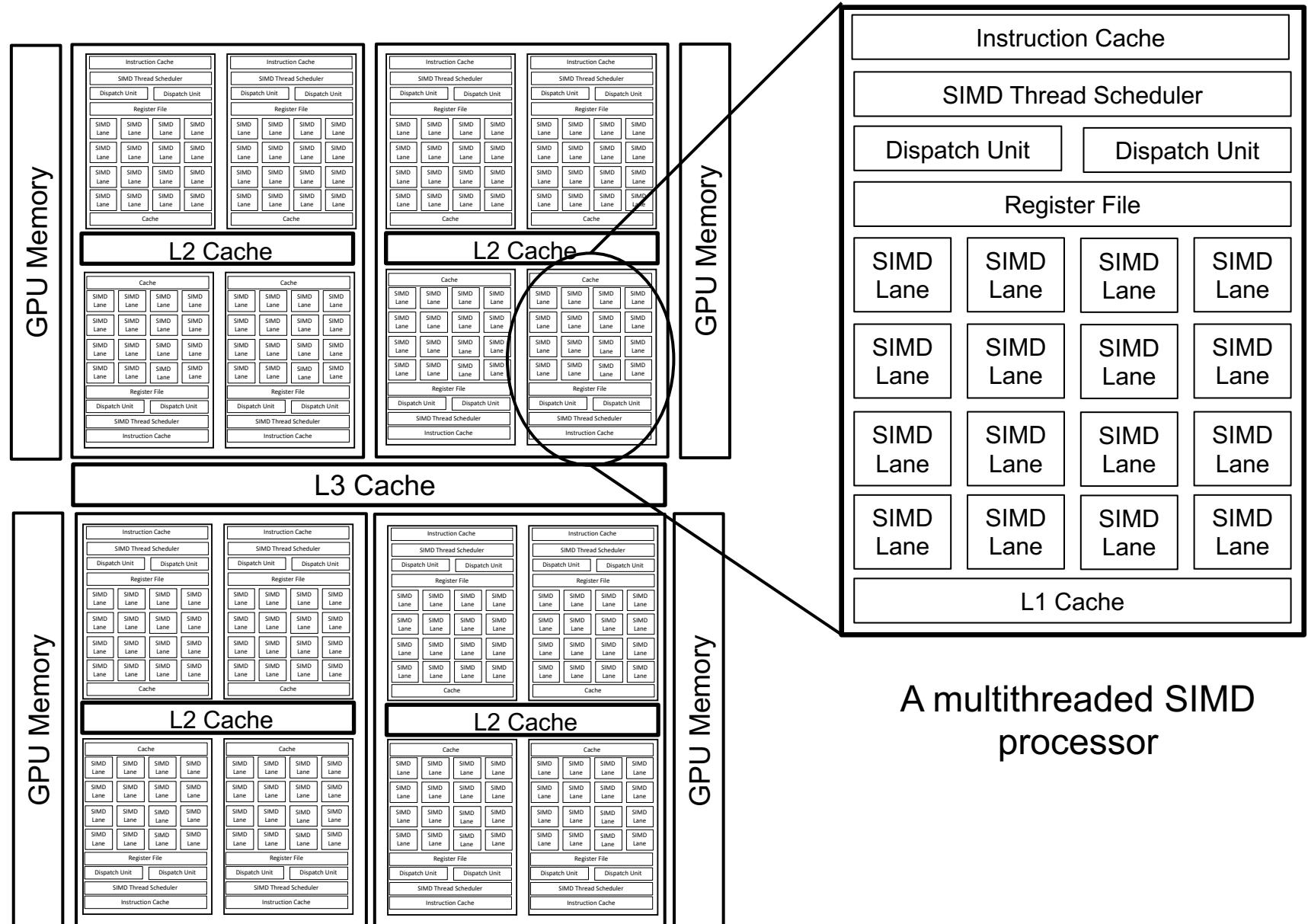
OpenMP programming model

- Up to OpenMP 3.0:
 - Aimed at multi-core CPUs
 - All cores can see all the main memory
 - So OpenMP has one memory space, available to all parallel threads
 - It's SHARED memory programming!
- OpenMP 4.x changes this.
- Added NUMA controls:
 - Available memory doesn't have uniform performance
 - Still shared and available to all CPU cores
- Target device model added:
 - Target device has separate memory space
 - Enables heterogenous programming

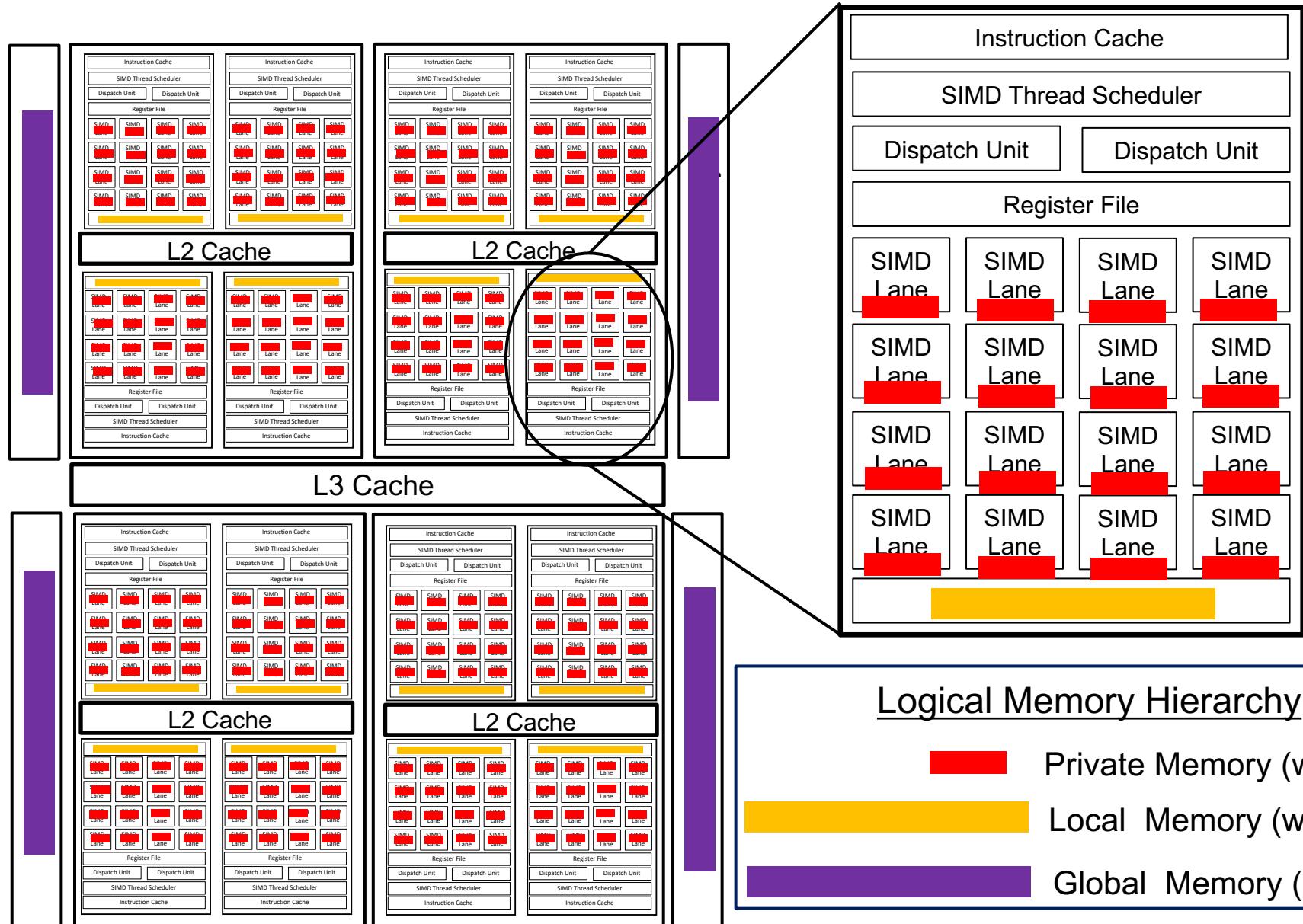
OpenMP basic definitions: the solution stack



A Generic GPU (following Hennessy and Patterson)

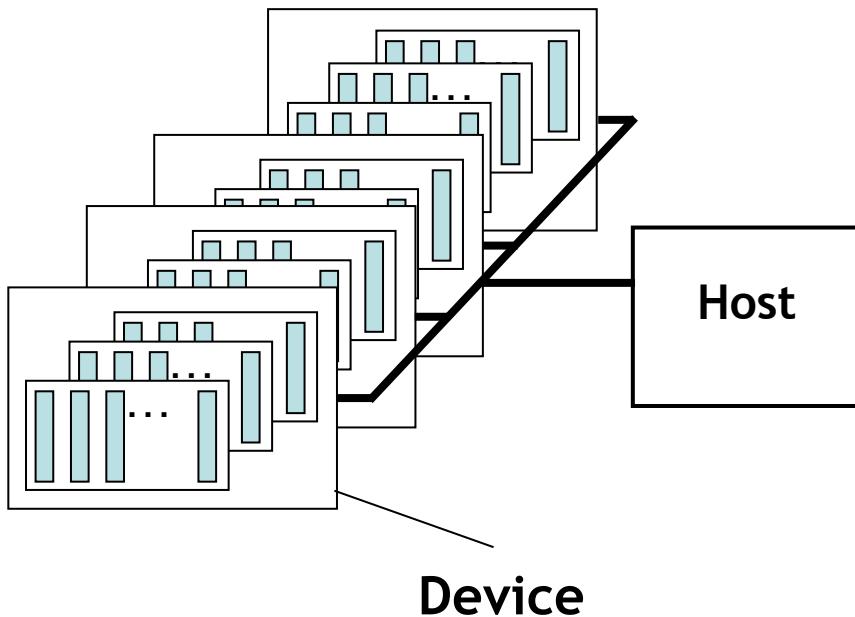


A Generic GPU (following Hennessy and Patterson)



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

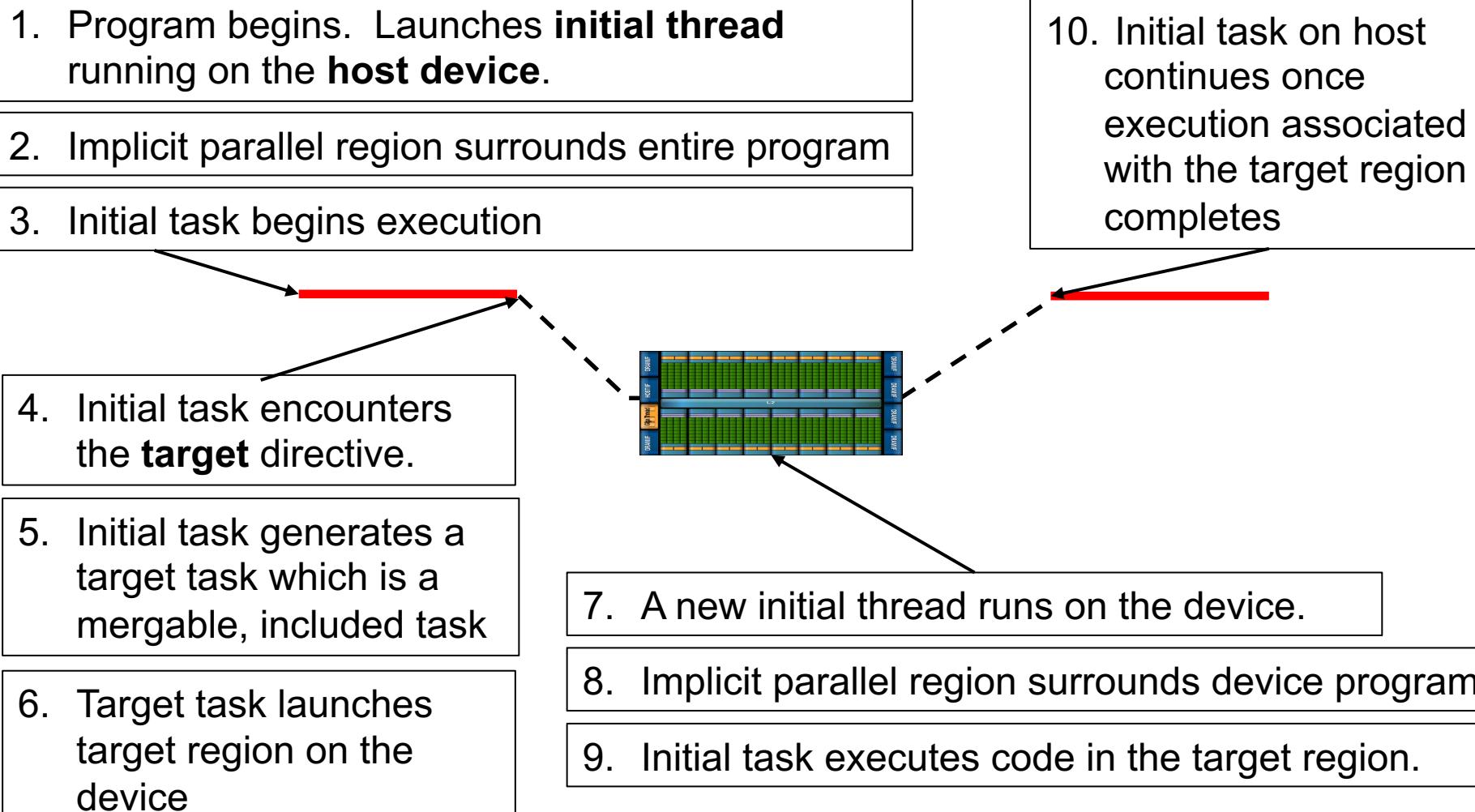


```
#include <omp.h>
#include <stdio.h>
int main()
{
    printf("There are %d devices\n",
           omp_get_num_devices());
}
```

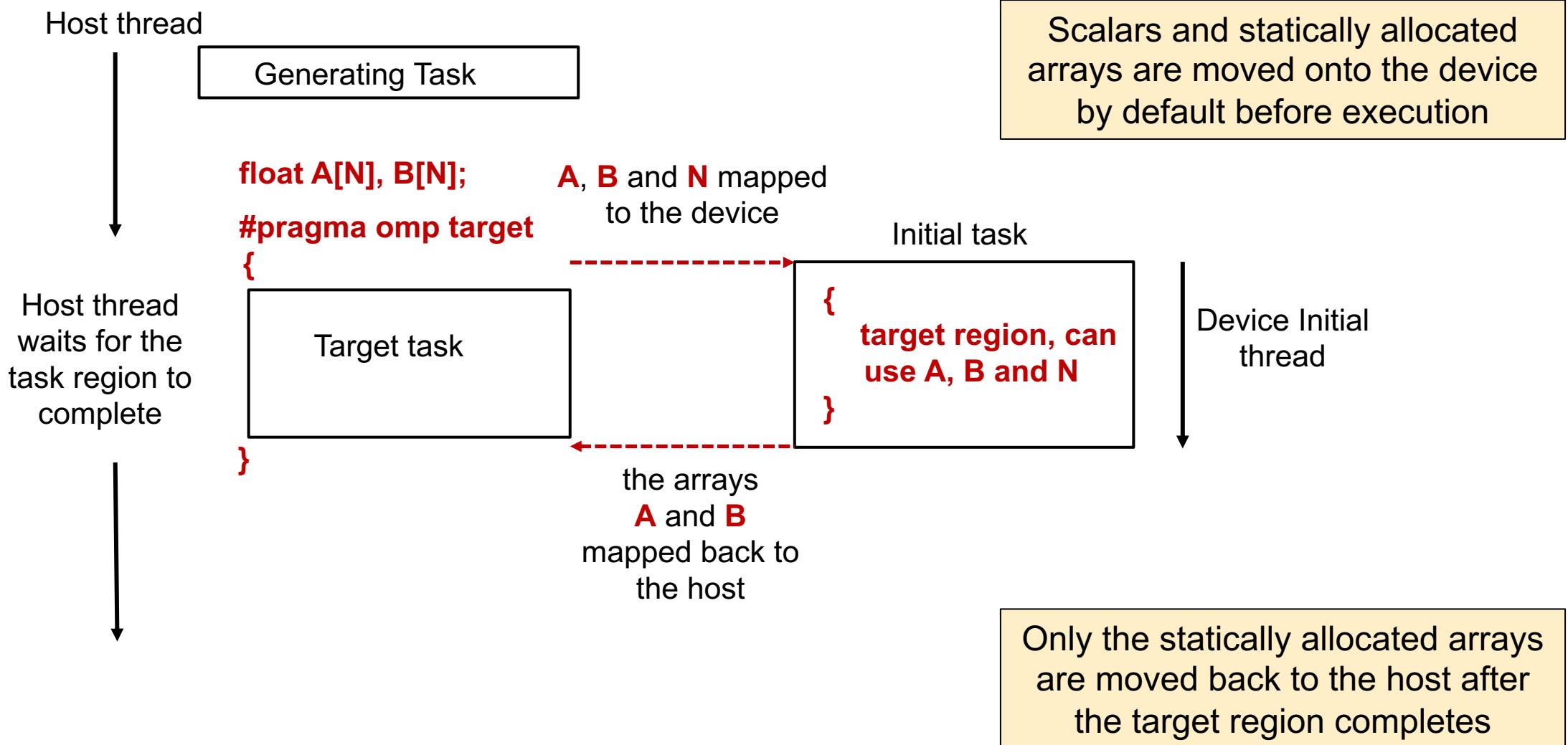
OpenMP with target devices

- The target construct offloads execution to a device.

```
#pragma omp target  
{....} // a structured block of code
```



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



The ‘target data’ environment

- **Remember:** there are distinct memory spaces on host and device.
- OpenMP uses a combination of *implicit* and *explicit* data movement.
- Data may move between the host and the device in well defined places:
 - Firstly, at the beginning and end of a **target** region:

```
#pragma omp target
{
    ...
}
```

// Data may move from host to device here

// and from device to host here

- We'll discuss the other places later...

Default Data Mapping: implicit movement with a target region

- Scalar variables:
 - Examples:
 - int N; double x;
 - OpenMP implicitly maps scalar variables as **firstprivate**
 - A new value per work-item is initialized with the original value (in OpenCL nomenclature, the firstprivate goes in private memory).
 - The variable is not copied back to the host at the end of the target region.
 - In CUDA/OpenCL parlance, a firstprivate scalar can be launched as a parameter to a kernel function without the overhead of setting up a variable in device memory.

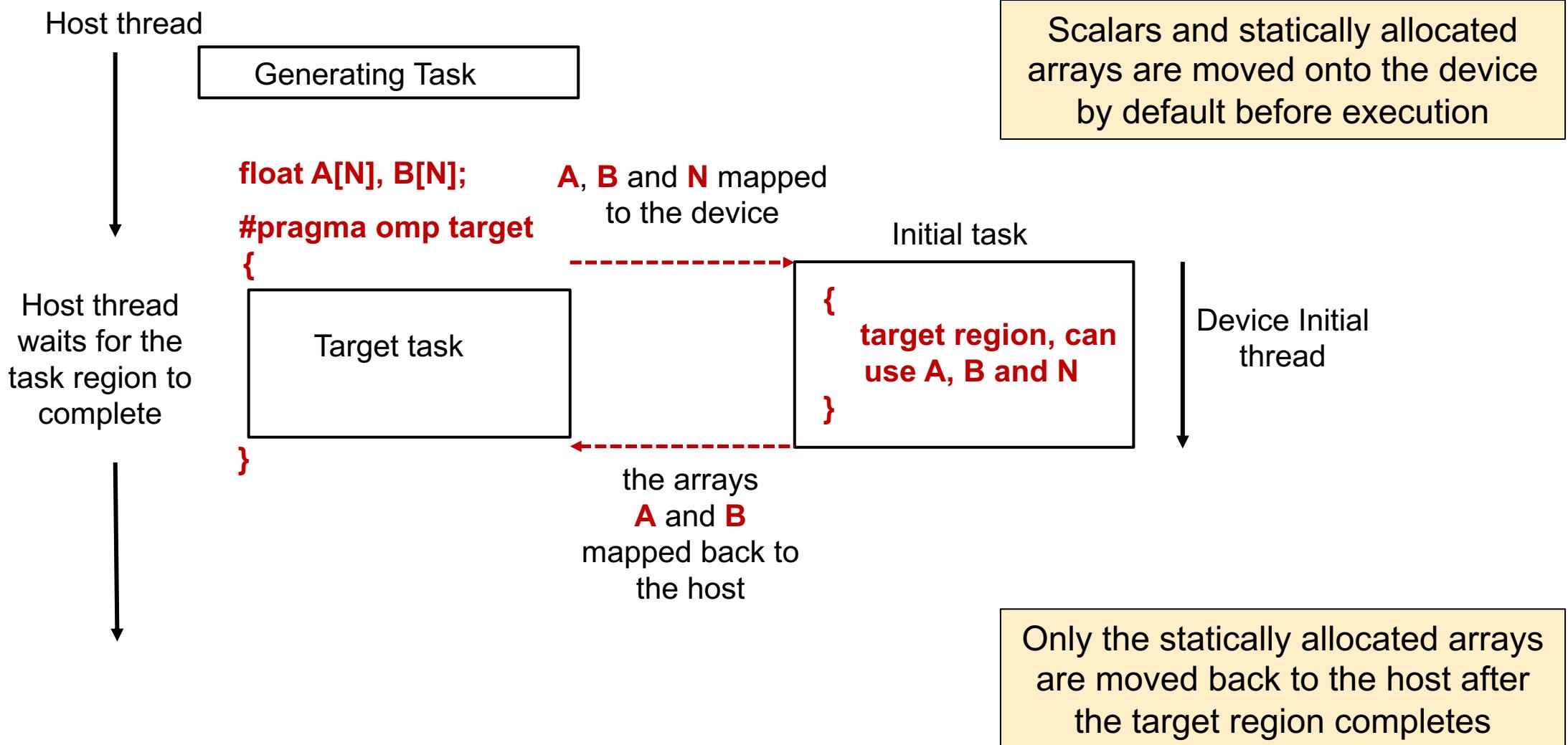
Default Data Mapping: implicit movement with a target region

- Non-scalar variables:
 - Must have a ***complete type***.
 - Example: fixed sized (stack) array:
 - `double A[1000];`
 - Copied to the device at the start of the **target** region, *and* copied back at the end. In OpenCL nomenclature, these are placed in device global memory.
 - A new memory object is created in the target region and initialized with the original data, but it is shared between threads on the device. Data is copied back to the host at the end of the target region.
 - OpenMP calls this mapping **tofrom**

Default Data Mapping: implicit movement with a target region

- Pointers are implicitly copied, but ***not*** the data they point to:
 - *Example: arrays allocated on the heap*
 - `double *A = malloc(sizeof(double)*1000);`
 - The pointer **value** will be mapped (i.e. the address stored in A).
 - But the data it points to ***will not*** be mapped by default.
 - We'll show you how to map a pointer's data shortly.

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.

Commonly used clauses with target

#pragma omp target [clause[,]clause]...
structured-block

if(scalar-expression)

- If the scalar-expression evaluates to false then the target region is executed by the host device in the host data environment.

device(integer-expression)

- The value of the integer-expression selects the device when a device other than the default device is desired.

private(list) firstprivate(list)

- creates variables with the same name as those in the list on the device. In the case of firstprivate, the value of the variable on the host is copied into the private variable created on the device.

map(map-type: list)

- map-type may be **to**, **from**, **tofrom**, or **alloc**. The clause defines how the variables in list are moved between the host and the device. (Lots more on this later)...

nowait

- The target task is deferred which means the host can run code in parallel to the target region on the device.

Let's run code in parallel on the device

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

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

Edit file: vadd.c

- Make a copy of your parallel vadd.c program for a CPU (i.e. save the CPU version)
 - vadd.c Adds together two arrays, element by element:
$$\text{for}(i=0;i<N;i++) c[i]=a[i]+b[i];$$
- Parallelize your vadd program for a GPU
- Time it for large N and save the result. How does it compare to the CPU version?

- double omp_get_wtime();
- #pragma omp parallel
- #pragma omp for
- #pragma omp parallel 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
```

Solution: Simple vector add in OpenMP on GPU

Files: Solutions/vadd_target.c, Solutions/submit_vadd_target

```
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;
}
```

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 = 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!**
 - Common to misremember this as `begin : end`, but it is **length**
 - 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)**.

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:

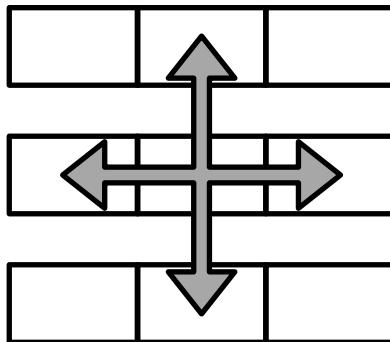
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

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

- 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.

5-point stencil: solve kernel

```
void solve(...) {
    // Finite difference constant multiplier
    const double r = alpha * dt / (dx * dx);
    const double r2 = 1.0 - 4.0*r;

    // Loop over the nxn grid
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {

            // 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);

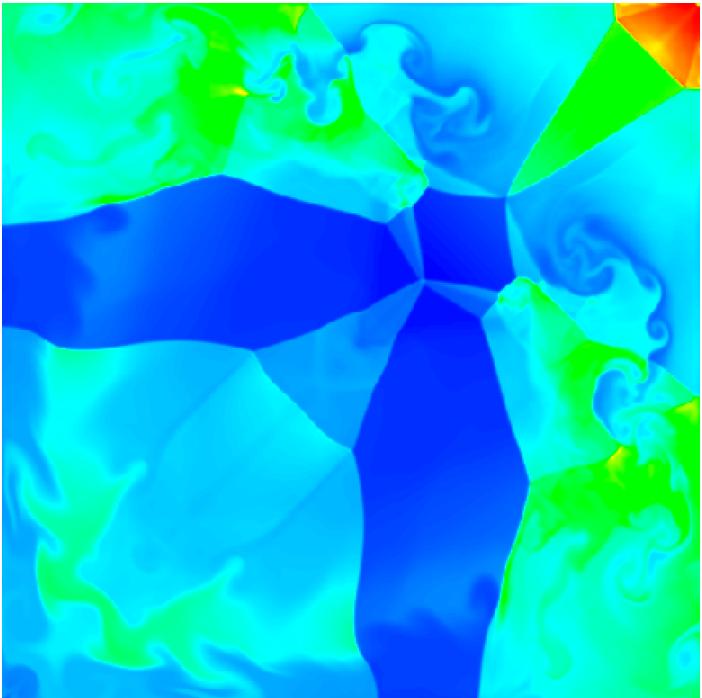
        }
    }
}
```

Profiling GPU code

- Host-to-device transfers are important to optimize
 - Main memory bandwidth of device is typically high
 - P100 has peak of 732 GB/s
 - But memory bandwidth between host and device is usually much lower
 - PCIe 3.0 x16 has peak of 32 GB/s
- Knowing the code, we can predict that all the data movement between the host and device takes a lot of time
- Want to use tools to find this out for certain

CUDA Toolkit

Cray maps their OpenMP constructs onto CUDA so NVIDIA's CUDA toolkit works with the Cray compilers.



We will demonstrate using an OpenMP version of **flow**

Hydrodynamics mini-app solving Euler's compressible equations

Explicit 2D method that uses various stencils, keeping data resident on GPU for entire solve

CUDA Toolkit: NVProf → nsys

Try nsys in your submit_vadd script

Simple profiling: nvprof ./exe <params>

```
> 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]
28.69%	106.32ms	14	7.5942ms	576ns	55.648ms	[CUDA memcpy DtoH]
5.31%	19.682ms	2	9.8411ms	3.8686ms	15.814ms	set_problem\$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

Nvidia has deprecated their wonderful command line tool for profiling GPU programs.

We can still get profiling information with the nsys tool:

```
> nsys ./a.out
```

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

Exercise: parallel stencil (heat)

Files: heat.c

- Take the provided heat stencil code (heat.c)
- Add OpenMP directives to parallelize the loops on the GPU
- Add OpenMP map clauses to copy data between host and device
- Most of the runtime occurs in the solve() routine
- Directives and clauses:
 - #pragma omp target
 - #pragma omp target map
 - #pragma omp loop
 - #pragma omp loop collapse
- Experiment with problem size and the profiler:
 - Where is the bottleneck?

Exercise: heat code inputs

Files: heat.c

- 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.
- Edit submit_heat to change the problem size
- If you try other problems, change the code to report $r < 0.5$.
 - A warning is printed if this is not the case.

Solution: parallel stencil (heat)

Files: Solutions/heat_target.c, Solutions/submit_heat_target

```
// 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 map(tofrom: u[0:n*n], u_tmp[0:n*n])
    #pragma omp loop collapse(2) ←
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            // 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);
        }
    }
}
```

Add the loop directive to the loops
Use collapse clause to increase parallelism

Solution: nsys

```
$ nsys nvprof ./heat_map_target 8000 10
```

```
-----  
Problem input
```

```
Grid size: 8000 x 8000  
Cell width: 1.249844E-01  
Grid length: 1000.000000 x 1000.000000
```

```
Alpha: 1.000000E-01
```

```
Steps: 10  
Total time: 5.000000E-01  
Time step: 5.000000E-02
```

```
-----  
Stability
```

```
r value: 0.320080
```

```
==47637== NVPROF is profiling process 47637, command: ./heat_map_target 8000 10
```

```
Results
```

```
Error (L2norm): 1.499275E-10  
Solve time (s): 4.589534  
Total time (s): 9.635819
```

```
==47637== Profiling application: ./heat_map_target 8000 10
```

```
==47637== Profiling result:
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	53.33%	2.20737s	21	105.11ms	1.4720us	138.77ms	[CUDA memcpy HtoD]
	44.79%	1.85407s	20	92.704ms	49.633ms	121.67ms	[CUDA memcpy DtoH]
	1.88%	77.849ms	10	7.7849ms	7.7600ms	7.8050ms	<u>omp_offloading...</u>

Data movement dominates!

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

Typically lots of iterations!

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

solve() routine uses this pragma:

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

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

```
// Pointer swap  
tmp = u;  
u = u_tmp;  
u_tmp = tmp;
```

```
}
```

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

Next topic: how to keep data resident on
target device between target regions

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 details

- **#pragma omp target update clause[[[,]clause]...]**
- Creates a target task to handle data movement between the host and a device.
- clause is either a motion-clause:
 - to(list)
 - from(list)
- Or one of the following:
 - if(scalar-expression)
 - device(integer-expression)
 - **nowait**
 - **depend (dependence-type : list)**
- **nowait** and **depend** apply to the target task running on the host.

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 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 teams distribute parallel for simd  
    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)
 - **nowait**
 - **depend (dependence-type : list)**
 - map (map-type: list)
- **nowait** and **depend** apply to the target task running on the host.

A note about the nowait clause

- Specify dependencies to ensure the **target enter data** finishes *before* the **target region sibling** task starts:

```
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]) nowait depend(out: A)  
}  
  
int main(void) {  
    int N = 1024; int *A = malloc(sizeof(int) * N);  
    init_array(A, N);  
  
    #pragma omp target teams distribute parallel for simd nowait depend(inout: A)  
    for (int i = 0; i < N; ++i) A[i] = A[i] * A[i];  
  
    #pragma omp taskwait  
  
    #pragma omp target exit data map(from: A[0:N])  
}
```

Notes about Pointer swapping

- Mapping between addresses on host and device is done when the target constructs are **encountered**
- `#pragma omp target data map(from:)`
 - The from location is **fixed** from the start of the target data region
 - If pointers are swapped, data is still copied back to the original pointer

```
void *orig = a;  
#pragma omp target data map(tofrom: a[0:N])  
{  
    a = NULL; // or anything else  
}
```

Data copied back to a's *original location*

- Target exit data map(from:) uses the **current** mapping
 - So if pointers are swapped, it will go to the new address

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.

Exercise

Files: heat.c

- 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.
 - #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

ssh br-trainXX@isambard.gw4.ac.uk
Password: openmpSC22

Solution: Pointer swapping in action

Files: Solutions/heat_target_map.c, Solutions/submit_heat_target_map

```
#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

NVPROF output

Results

Error (L2norm): 1.499275E-10

Solve time (s): 0.161998

Total time (s): 6.185598

==26738== Profiling application: ./heat_data_reg 8000 10

==26738== Profiling result:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	51.67%	161.32ms	10	16.132ms	15.764ms	16.472ms	<code>__omp_offloading_...</code>
	35.66%	111.33ms	3	37.111ms	896ns	56.239ms	[CUDA memcpy HtoD]
	12.67%	39.551ms	1	39.551ms	39.551ms	39.551ms	[CUDA memcpy DtoH]

Data movement summary

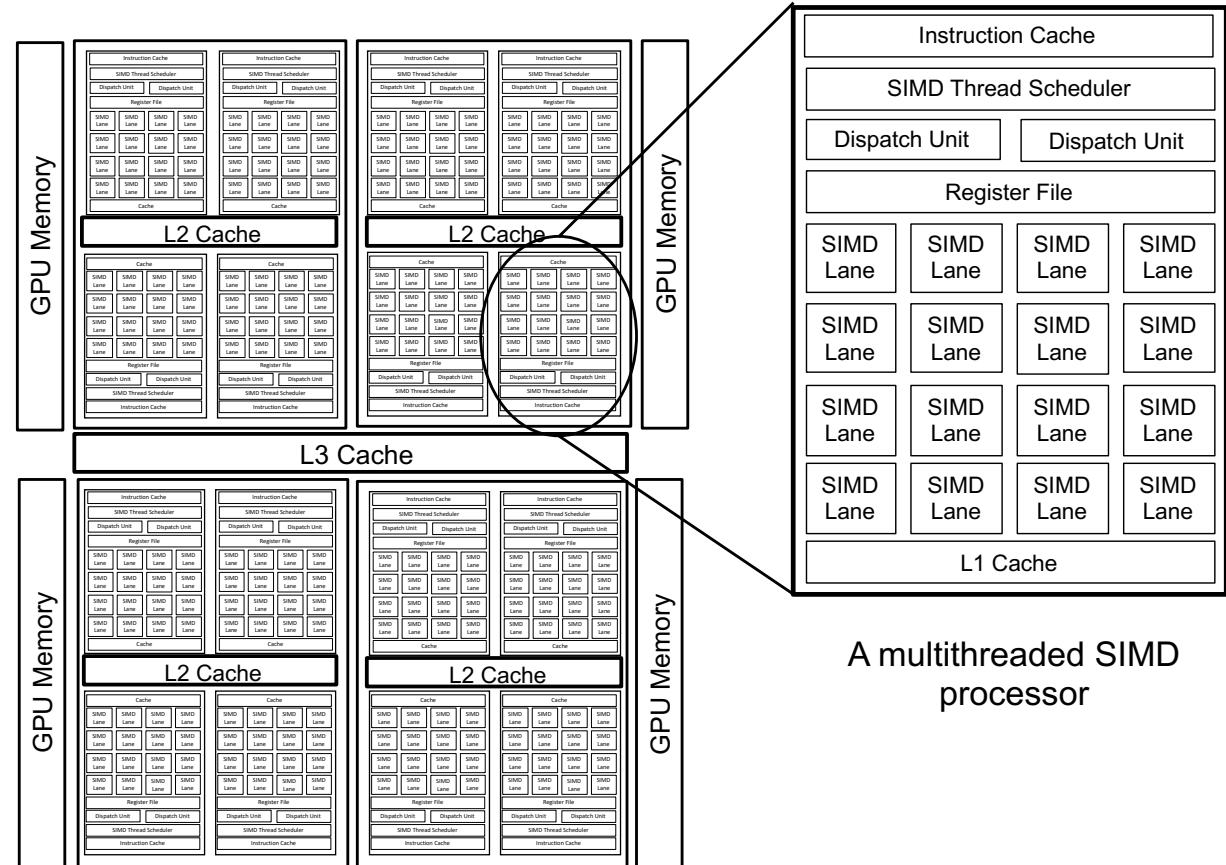
- Data transfers between host/device occur at:
 - Beginning and end of **target** region
 - Beginning and end of **target data** region
 - At the **target enter data** construct
 - At the **target exit data** construct
 - At the **target update** construct
- Can use **target data** and **target enter/exit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

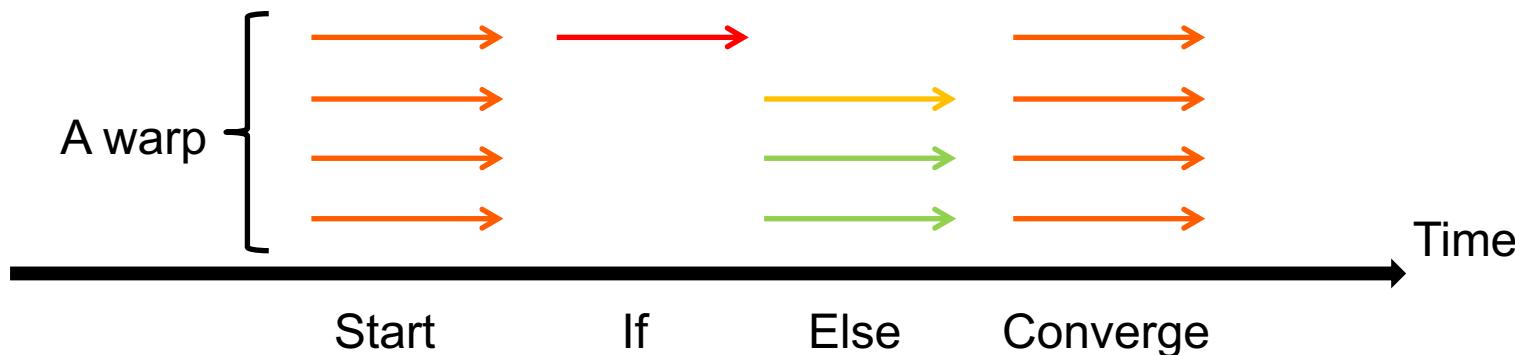


```
#pragma omp parallel for collapse(2)  
for(int i=0;i<N;i++)  
    for(int j=0;j<N;j++)  
        for(int k=0;k<N;k++)  
            *(C+(i*N+j)) += *(A+(i* N +k)) * *(B+(k* N +j));
```

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

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



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];

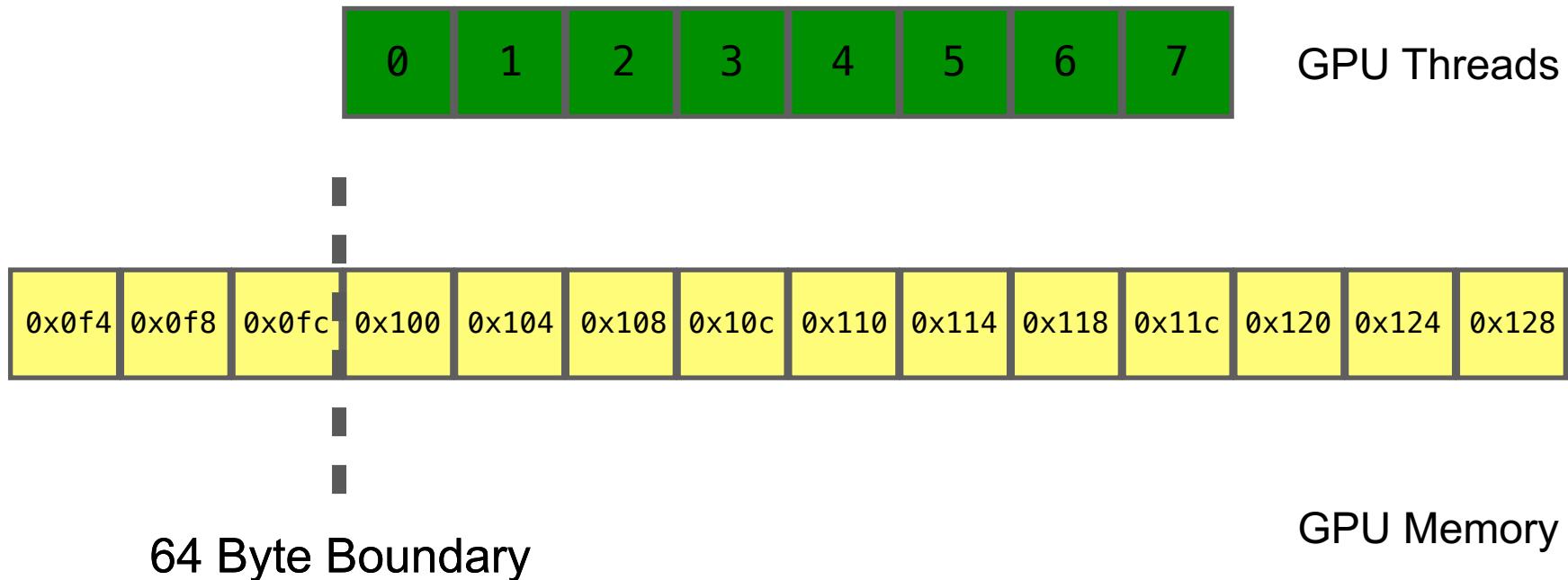
    // still pretty good
    const int c = 3;
    float val2 = memA[id + c];

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

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

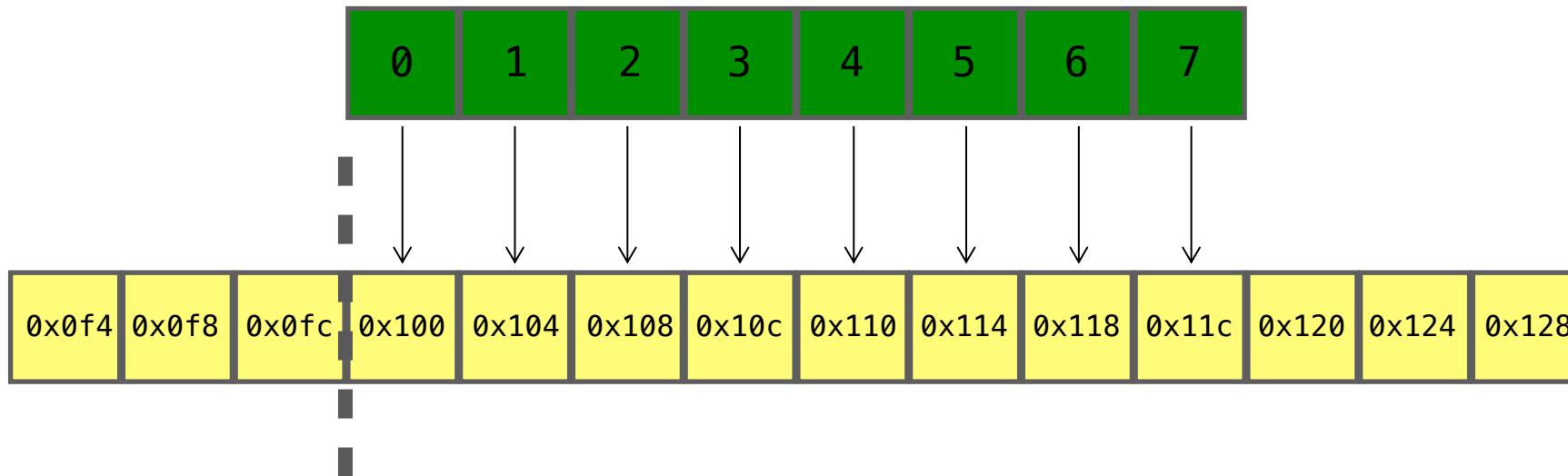
    float val4 = memA[loc];
}
```

Memory access patterns



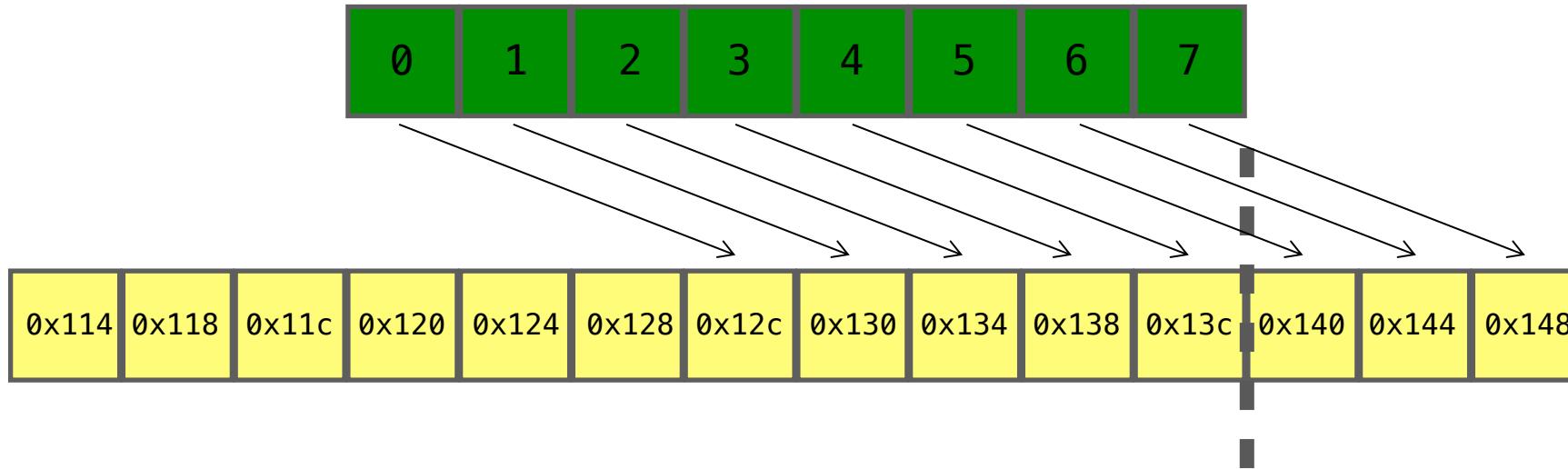
Memory access patterns

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



Memory access patterns

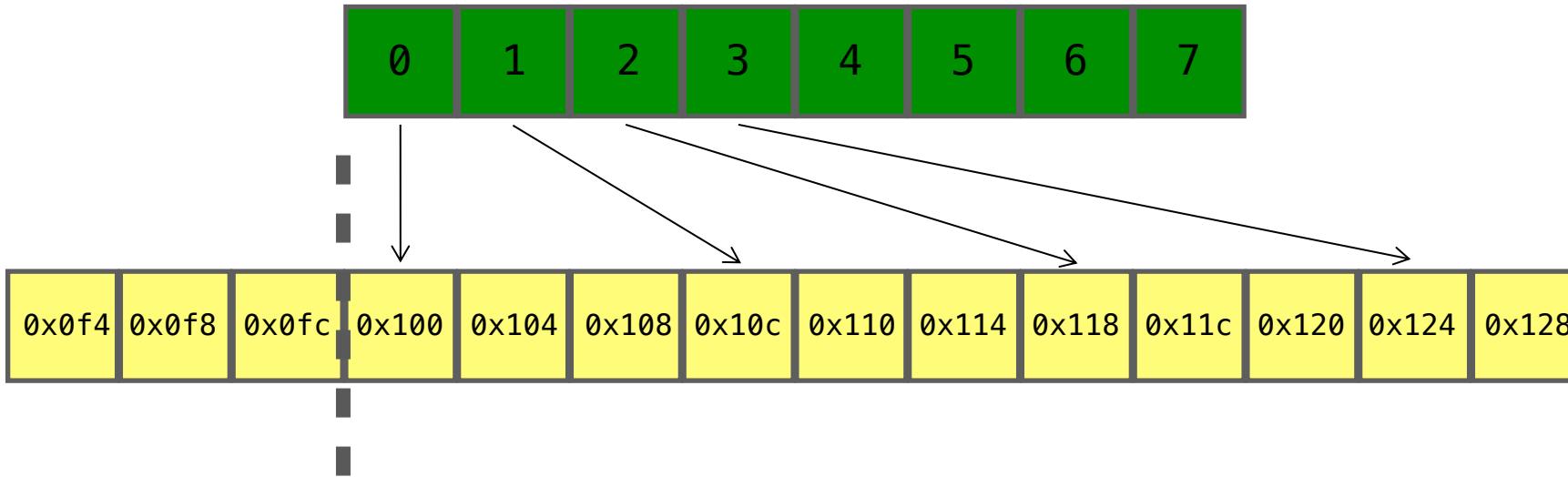
```
const int c = 3;  
float val2 = memA[id + c];
```



64 Byte Boundary

Memory access patterns

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



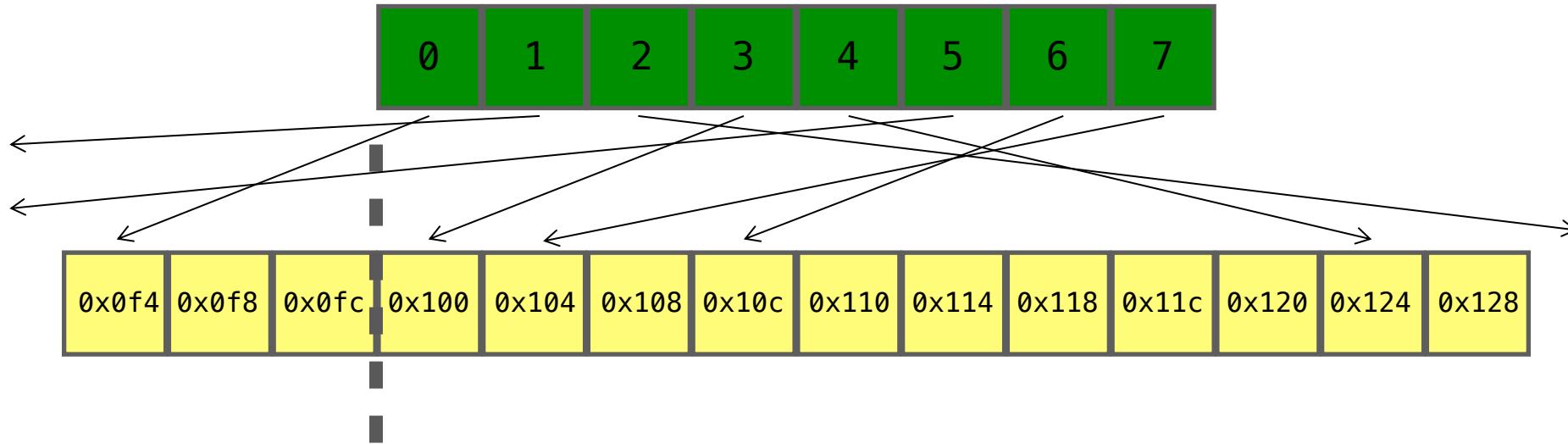
64 Byte Boundary

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

Files: heat.c

- 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: swap loop order

Files: [Solutions/heat_target_map_opt.c](#), [Solutions/submit_heat_target_map_opt](#)

```
// 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;

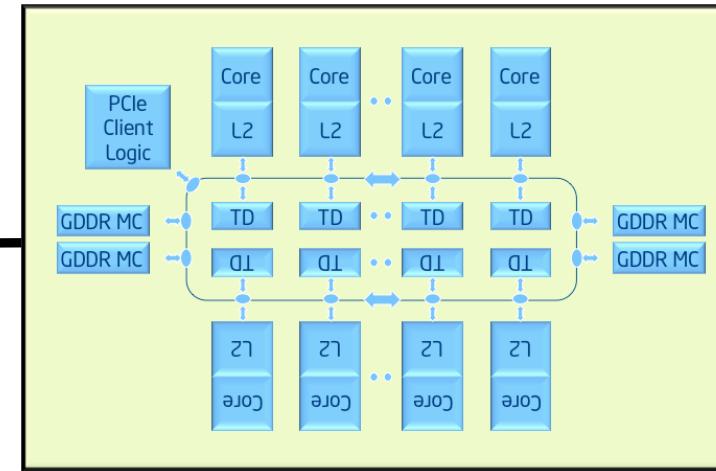
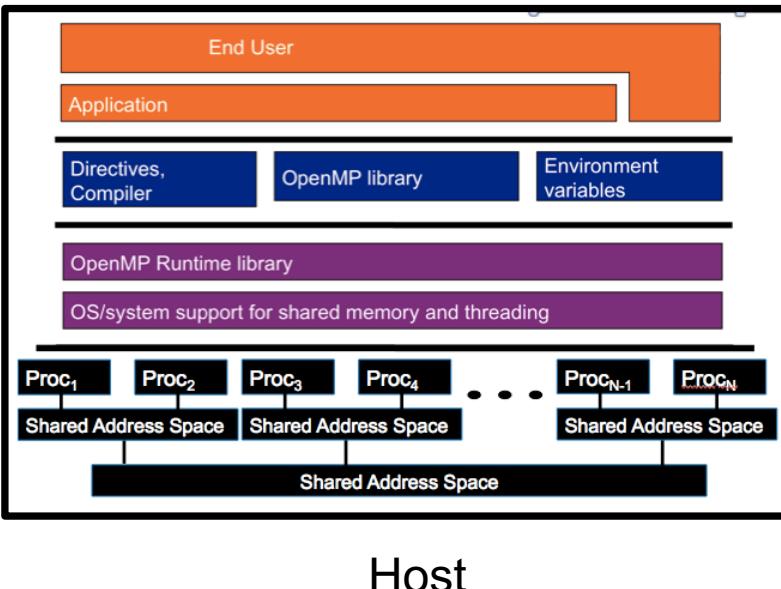
    // Loop over the nxn grid
#pragma omp target
#pragma omp teams distribute parallel for simd collapse(2)
for (int j = 0; j < n; ++j) {
    for (int i = 0; i < n; ++i) {
        // Update the 5-point stencil, using boundary conditions on the edges of the domain.
        // Boundaries are zero because the 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);
    }
}
```

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

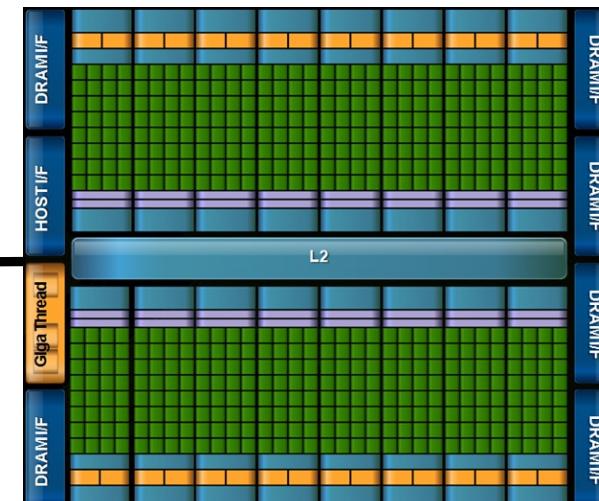
The loop construct is great, but sometimes you want more control.

OpenMP device model: Examples

Some key devices that were considered when designing the device model in OpenMP

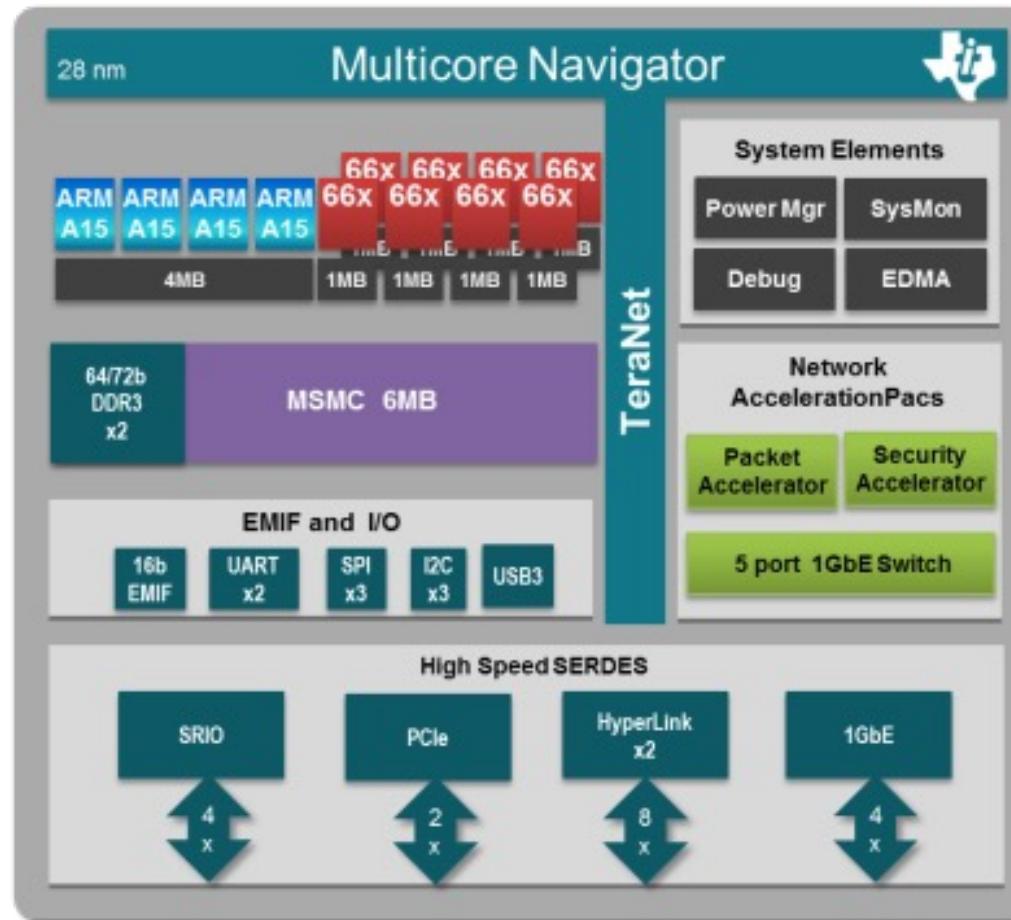


Target Device: Intel® Xeon Phi™ processor



Target Device: GPU

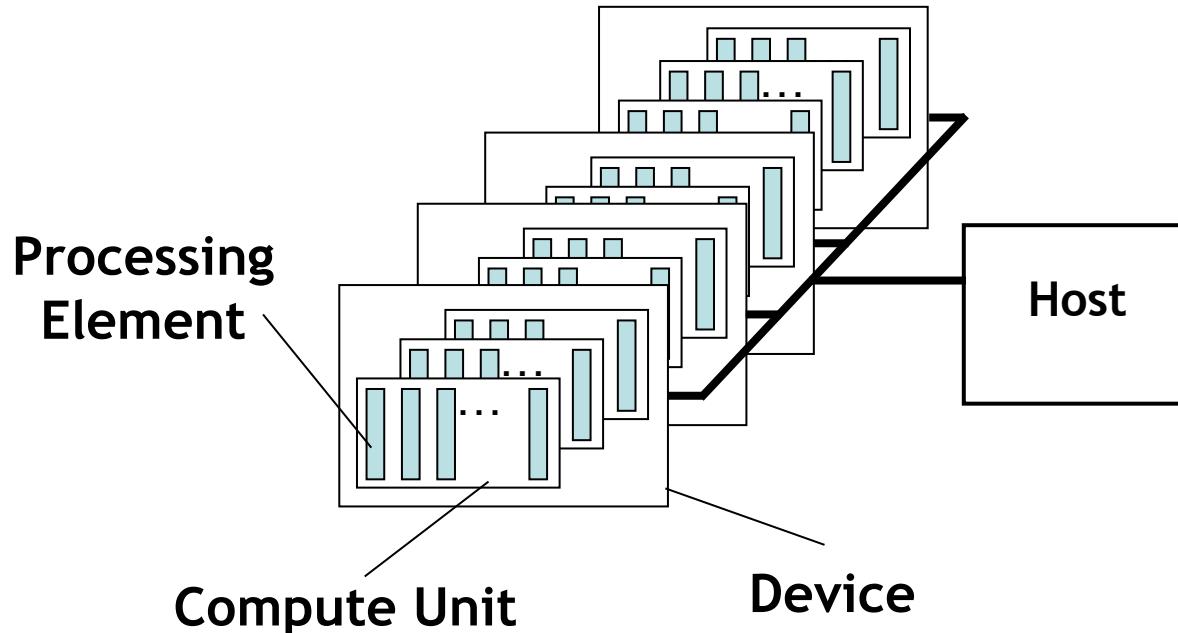
OpenMP Device Model: another example



Heterogeneous System on Chip (SoC)

A Generic Host/Device Platform Model

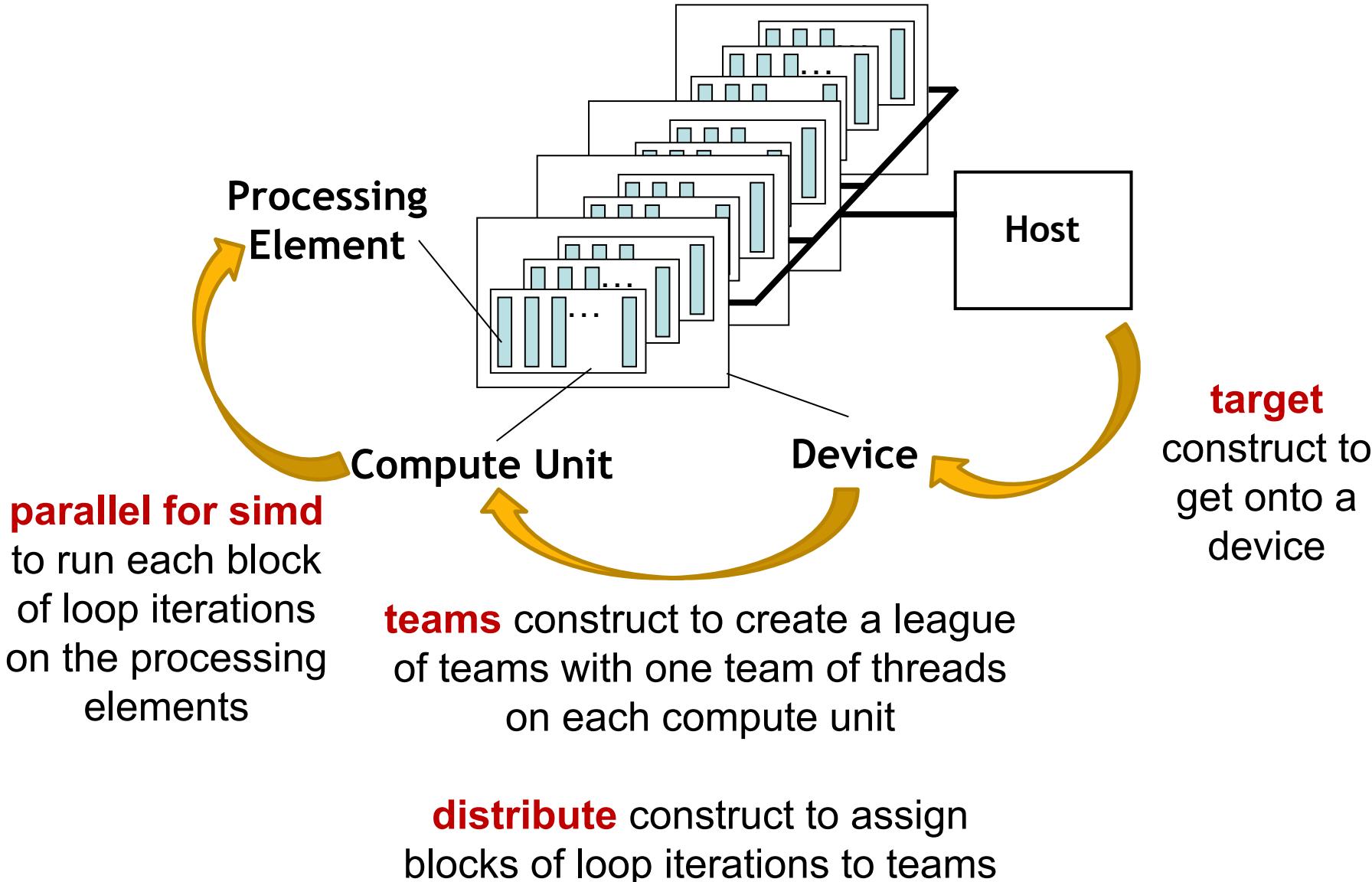
- One **Host** and one or more **Devices**
 - Each Device is composed of one or more **Compute Units**
 - Each Compute Unit is divided into one or more **Processing Elements**
- Memory is divided into **host memory** and **device memory**



Explosion of parallelism

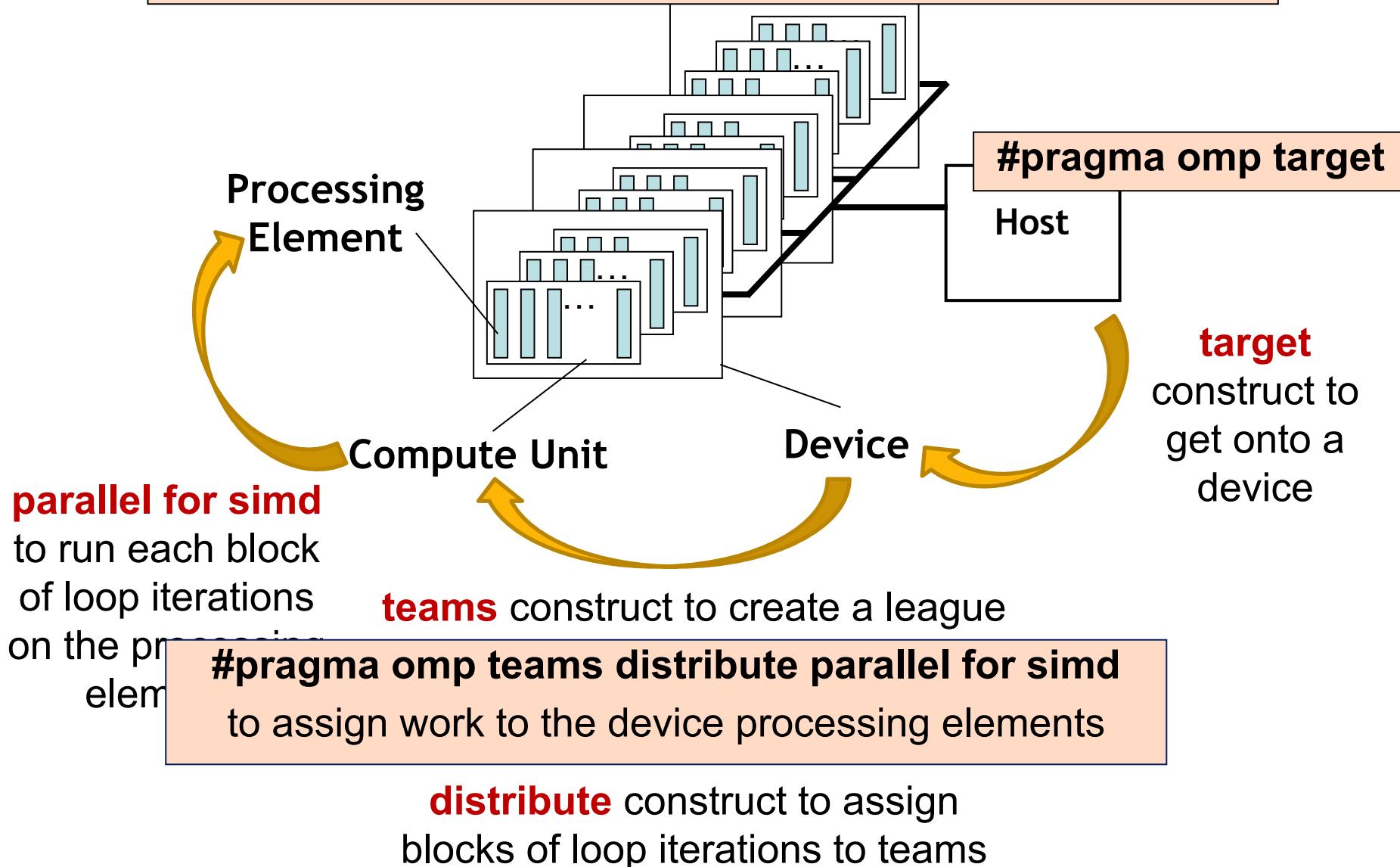
- GPUs are made of many *cores (compute units)*
 - NVIDIA V100 has 80 *Streaming Multiprocessors (SMs)*; these are the compute units
 - NVIDIA A100 has 108 *compute units*
 - Each NVIDIA compute unit has 64 FP32 processing elements
 - GPUs from AMD and Intel have similar structure of compute units and processing elements
- On an A100, that's $108 \times 64 = 6.912$ processing elements available to work in parallel
- Typically you need to expose multiple units of work per processing element for best performance
- Massive amount of (hierarchical) parallelism to exploit

Our host/device Platform Model and OpenMP



Our host/device Platform Model and OpenMP

Typical usage ... let the compiler do what's best for the device:



Implementation details

- OpenMP defines parallelism abstraction
 - Specific terminology is used
- An OpenMP implementation (runtime/compiler) has some freedom in how these are applied to hardware
 - Allows the implementation to make sensible choices to get the best performance
- OpenMP directives operate along spectrum of descriptive and prescriptive control
- Will now explain parallelism in the OpenMP abstraction
 - Will talk about how they correlate with hardware later...

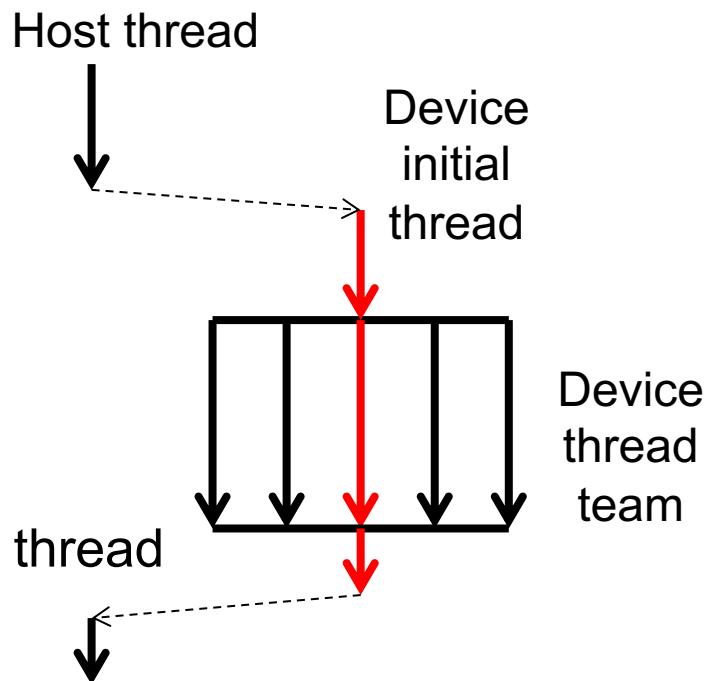
Parallel threads

- Recall fork-join model and parallel regions on a CPU:
 - `#pragma omp parallel`
- Threads are created on entry to parallel region
- All those threads belong to one **team**
- Threads in a team can synchronize:
 - `#pragma omp barrier`

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

```

Transfer control of execution to a **SINGLE** device thread
Only one **team** of threads workshares the loop



‘teams’ and ‘distribute’ constructs

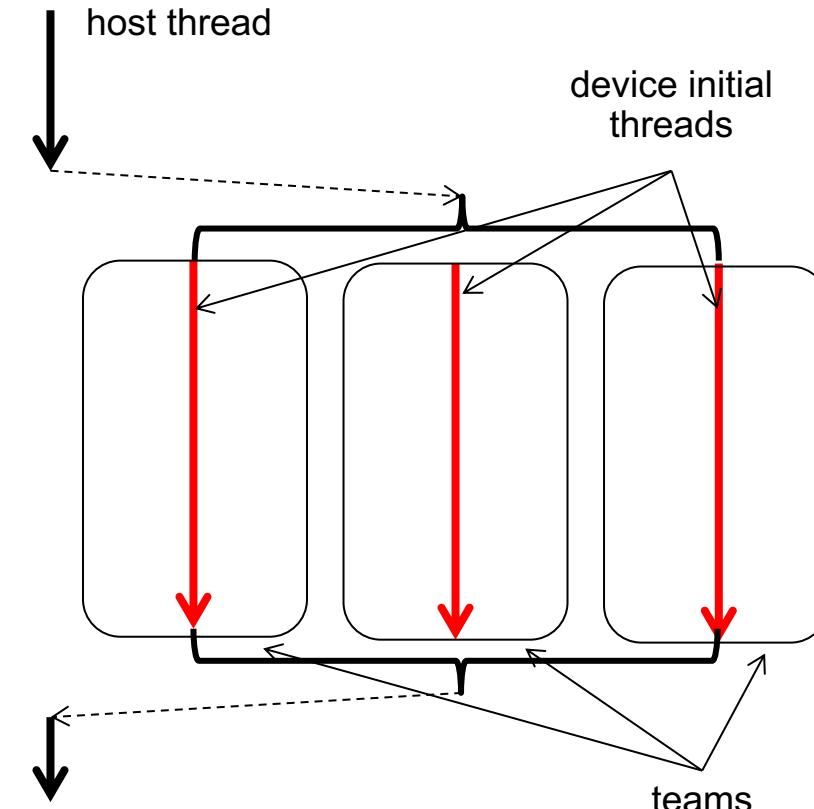
- The **teams** construct
 - Similar to the **parallel** construct
 - It starts a league of *teams*
 - Each team in the league starts with one initial thread – i.e. a team of one thread
 - Threads in different teams **cannot** synchronize with each other
 - The construct must be “perfectly” nested in 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

Multiple teams

- teams construct
- distribute construct

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

```



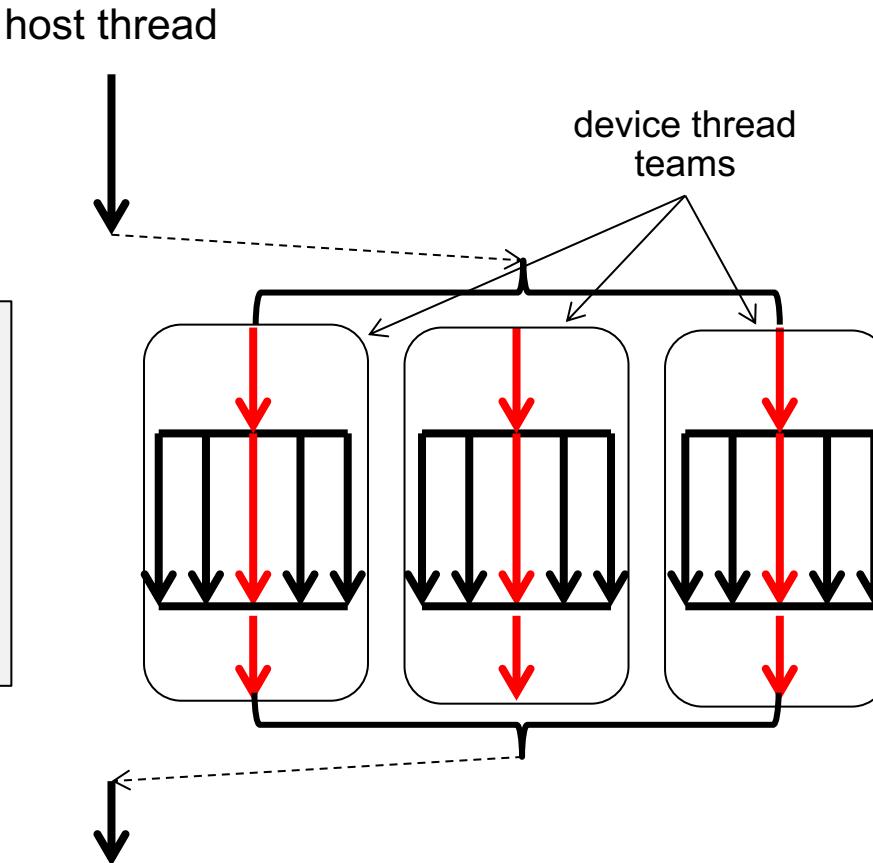
- Transfer execution control to **MULTIPLE** device initial threads
- Workshare loop iterations across the initial threads.

Note: number of teams is implementation defined, good for portable performance. Compilers can choose how they map teams and threads.

Putting it together

- teams distribute
- parallel for simd

```
#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 (one per team)
 - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the master thread in a thread team
 - Workshare loop iterations across the threads in a team (parallel for simd)

Composite Constructs

- The distribution patterns can be cumbersome
- OpenMP defines composite constructs for typical code patterns
 - **distribute simd**
 - **distribute parallel for**
 - **distribute parallel for simd**
 - ... plus additional combinations for **teams** and **target**
- Let the compiler figure out the loop tiling

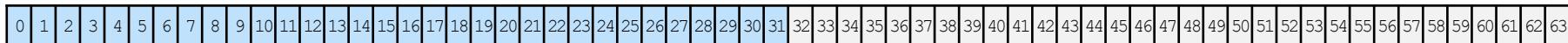
```
#pragma omp target teams
{
    #pragma omp distribute parallel for simd
    for (int i = 0; i < n; i++) {
        F(i) = G(i);
    }
}
```

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**

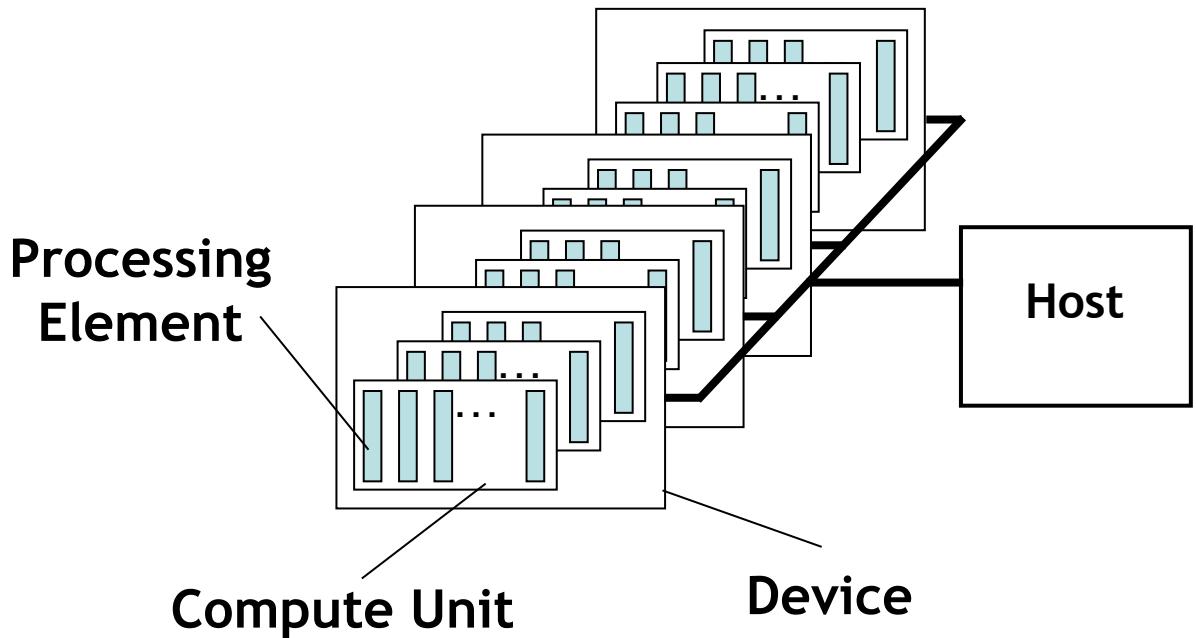
There is MUCH more ... beyond what have time to cover

- 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.

Recall OpenMP device model

Uses OpenCL terminology, but describes a generic GPU:

- Two levels of parallelism:
 - Compute Units
 - Processing Elements
- Processing elements in a compute unit typically operate in lock-step
 - But not necessarily, e.g. NVIDIA Volta architecture
 - Performance typically lower when they don't



GPU terminology is Broken (sorry about that)

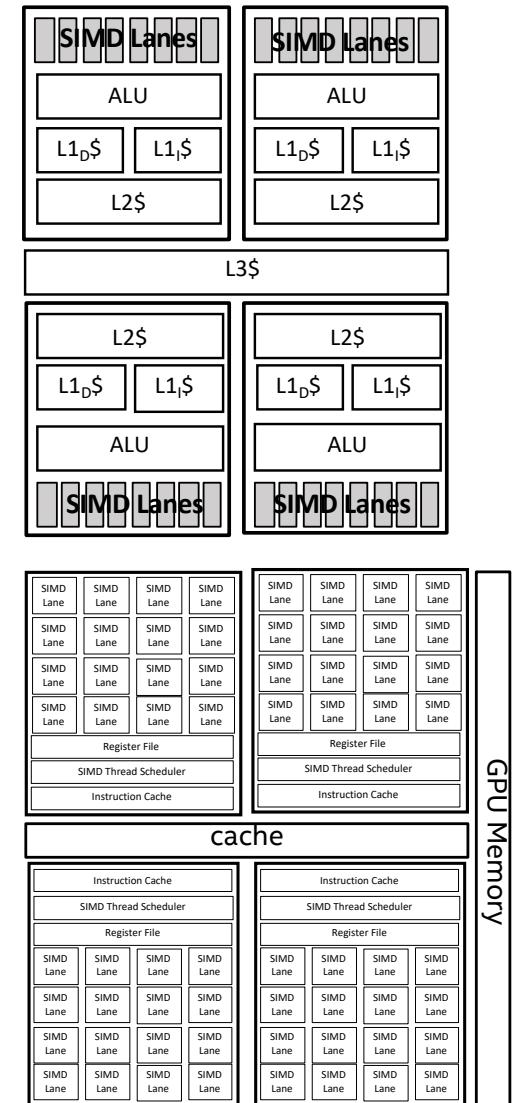
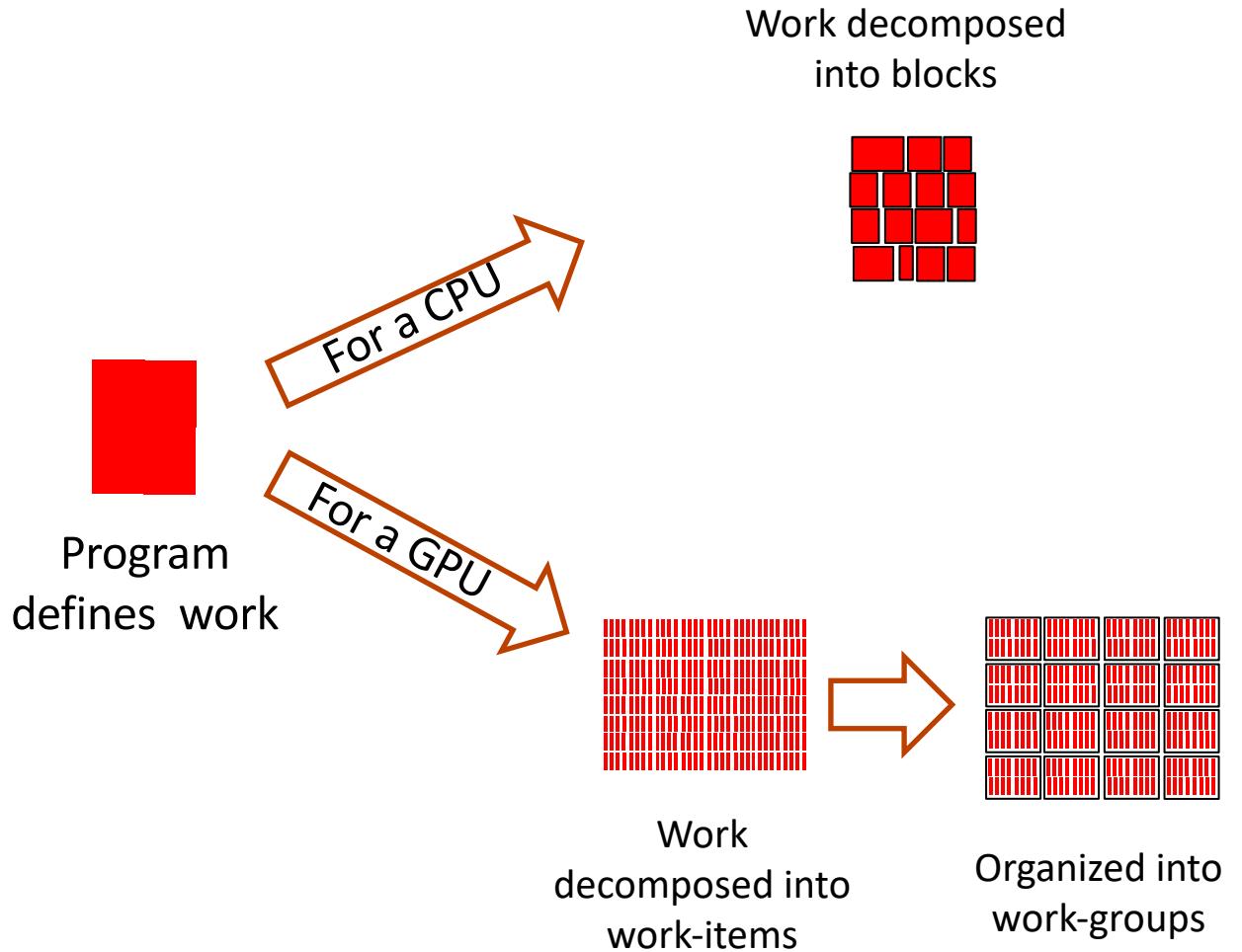
Hennessy and Patterson	CUDA	OpenCL
Multithreaded SIMD Processor	Streaming multiprocessor	Compute Unit
SIMD Thread Scheduler	Warp Scheduler	Work-group scheduler
SIMD Lane	CUDA Core	Processing Element
GPU Memory	Global Memory	Global Memory
Private Memory	Local Memory	Private Memory
Local Memory	Shared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

SIMT: Single Instruction, Multiple Thread

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

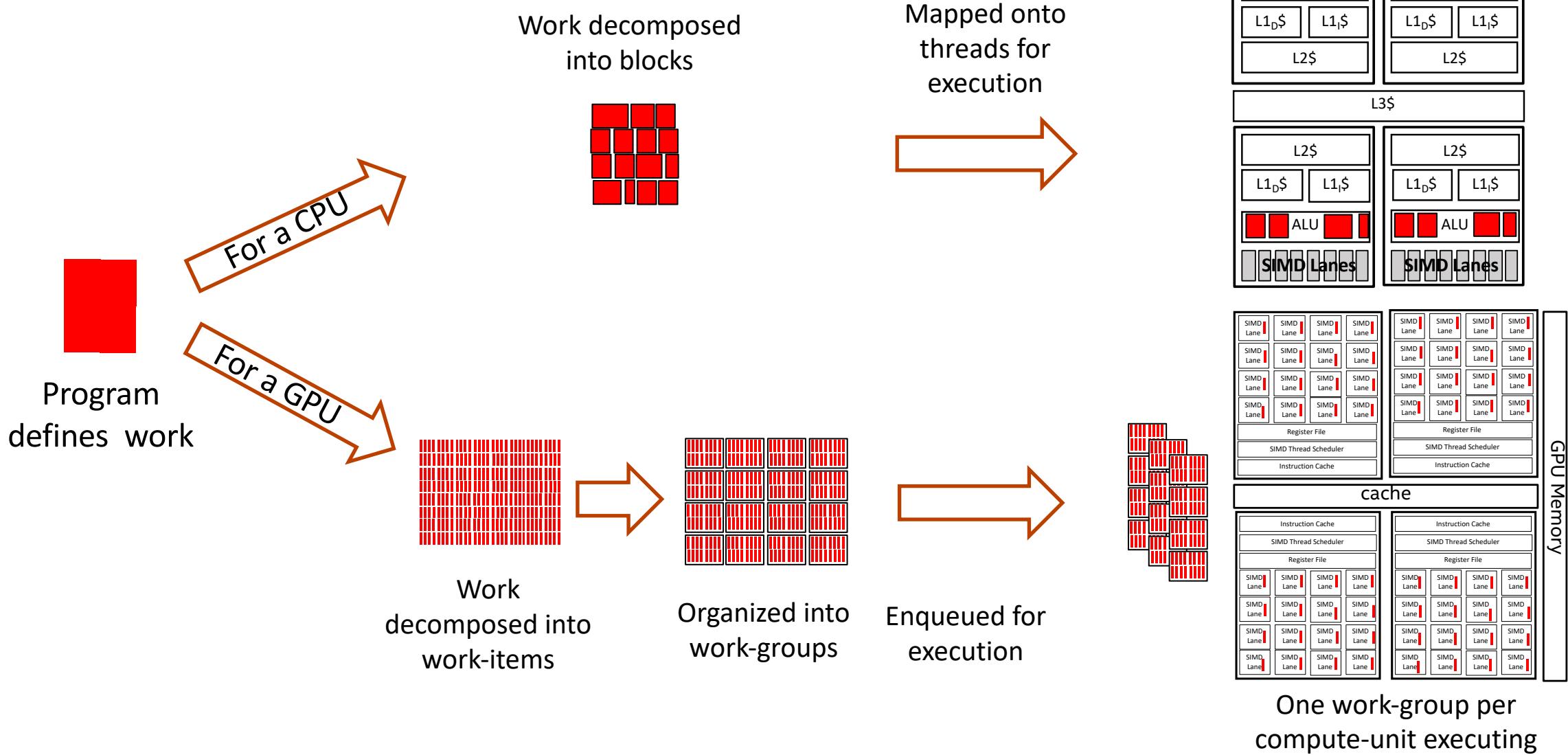


Executing a program on CPUs and GPUs



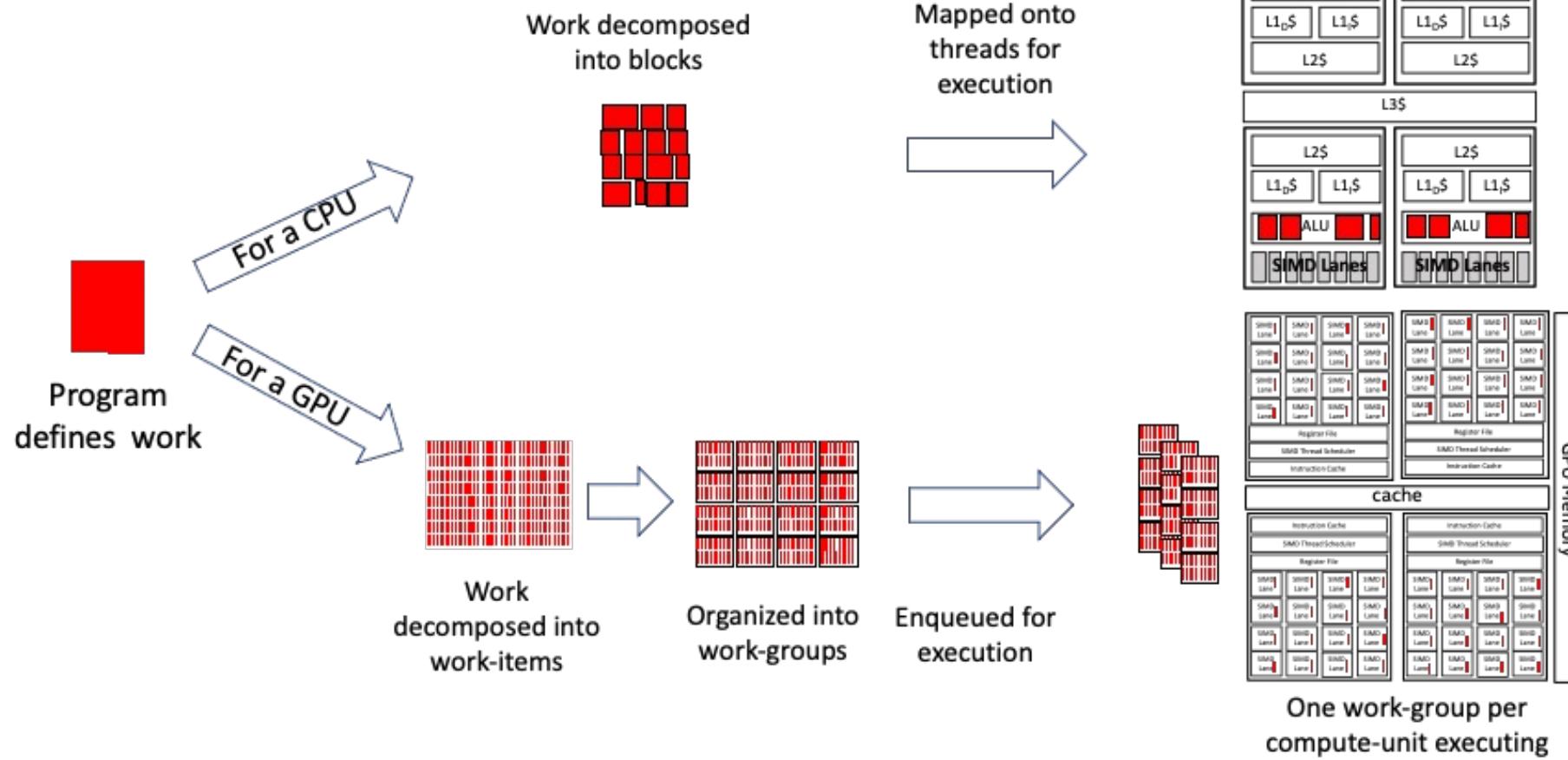
One work-group per
compute-unit executing

Executing a program on CPUs and GPUs



CPU/GPU execution model

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.

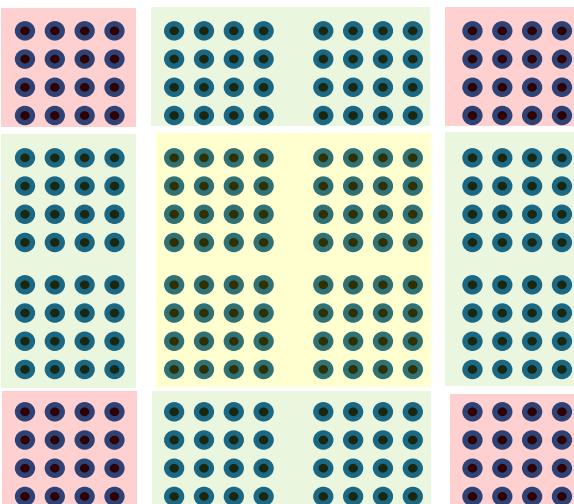
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Turn source code into a scalar work-item

```
extern void reduce( __local float*, __global float* );  
  
__kernel void pi( const int niters, float step_size,  
    __local float* l_sums, __global float* p_sums)  
{  
    int n_wrk_items = get_local_size(0);  
    int loc_id    = get_local_id(0);  
    int grp_id   = get_group_id(0);  
    float x, accum = 0.0f; int i, istart, iend;  
  
    istart = (grp_id * n_wrk_items + loc_id) * niters;  
    iend   = istart+niters;  
  
    for(i= istart; i<iend; i++){  
        x = (i+0.5f)*step_size; accum += 4.0f/(1.0f+x*x); }  
  
    l_sums[loc_id] = accum;  
    barrier(CLK_LOCAL_MEM_FENCE);  
    reduce(l_sums, p_sums);  
}
```

This is OpenCL kernel code ... the sort of code the OpenMP compiler generates on your behalf

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



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



3. Map data structures onto the same index space

How do we execute code on a GPU: OpenCL and CUDA nomenclature

Turn source code into a scalar **work-item** (a CUDA **thread**)

```
extern void reduce( __local float*, __global float*);  
  
__kernel void pi( const int niters, float step_size,  
                 __local float* l_sums, __global float* p_sums)  
{  
    int n_wrk_items = get_local_size(0);  
    int loc_id    = get_local_id(0);  
    int grp_id   = get_group_id(0);  
    float x, accum = 0.0f;  int i,istart,iend;  
  
    istart = (grp_id * n_wrk_items + loc_id) * niters;  
    iend   = istart+niters;  
  
    for(i= istart; i<iend; i++){  
        x = (i+0.5f)*step_size;  accum += 4.0f/(1.0f+x*x); }  
  
    l_sums[loc_id] = accum;  
    barrier(CLK_LOCAL_MEM_FENCE);  
    reduce(l_sums, p_sums);  
}
```

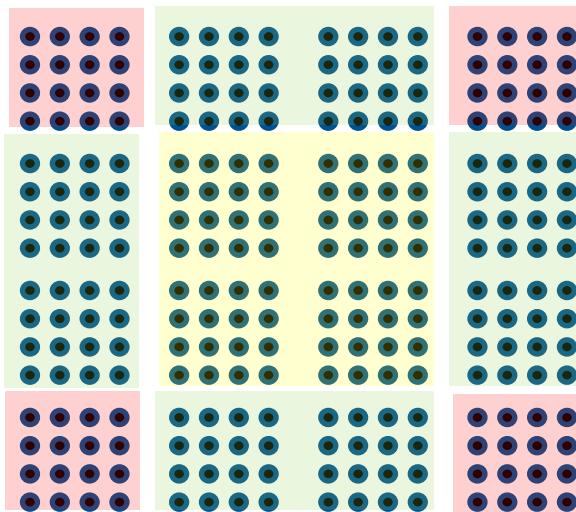
This code defines a **kernel**

It's called SIMD, but GPUs are really vector-architectures with a block of work-items executing together (a **subgroup** in OpenCL or a **warp** in CUDA)

Submit a kernel
to an OpenCL
command queue or a
CUDA **stream**



Organize work-items into
work-groups and map onto an N
dimensional index space. CUDA calls
a work-group a **thread-block**



OpenCL index space is
called an **NDRange**. CUDA
calls this a **Grid**

OpenMP: mapping the parallelism

- OpenMP defines three levels of parallelism:
 1. Teams
 2. Parallel threads
 3. SIMD
- But GPU hardware really has two levels of parallelism:
 1. Compute units
 2. Processing elements
- Implementations have flexibility in how they associate OpenMP concepts to the underlying hardware
- LLVM-based compilers, including Cray CCE ≥ 9 , **usually** associate:
 - OpenMP teams to compute units
 - OpenMP threads to processing elements
 - OpenMP SIMD is ignored
- Cray classic compiler maps SIMD to processing elements instead

How is this parallelism applied?

- Consider:

```
#pragma omp teams distribute
```

- Loop iterations distributed between teams
- Remember, you can't synchronize between teams
- So all iterations are **independent**
- Implementations can, and will, share the work across the whole GPU:
 - OpenMP teams being mapped to processing elements
 - Doesn't matter how the work-items are grouped into work-groups (compute units) as no synchronisation
- Behaves somewhat like SIMD auto-vectorization

What parallelism are you getting?

- With more than one possible mapping, sometimes you need to find out what is really happening.
- Compiler documentation:
 - Cray: `man intro_openmp`
- Compiler output:
 - In CCE 10, `-fsave-loopmark` flag
- Profiling:
 - `$ nvprof --print-gpu-trace`
 - Look for the number of threads per block, and number of blocks
 - Combine that with knowledge of pragma and number of loop iterations

CUDA Toolkit: NVProf/nsys

Trace profiling: nvprof --print-gpu-trace ./exe <params>

```
> nvprof --print-gpu-trace ./flow.omp4 flow.params
Problem dimensions 4000x4000 for 1 iterations.
==188688== NVPROF is profiling process 188688, command: ./flow.omp4 flow.params
```

Iteration 1

Timestep: 1.816932845523e-04

PASSED validation.

Wallclock 0.0325s, Elapsed Simulation Time 0.0

==188688== Profiling application: ./flow.omp4

==188688== Profiling result:

Shows block sizes, grid dimensions and register counts for kernels

Start	Duration	Grid Size	Block Size	Regs*	Name
577.84ms	4.7040us	-	-	-	[CUDA memcpy HtoD]
578.84ms	960ns	-	-	-	[CUDA memcpy HtoD]
578.90ms	3.0720us	(32 1 1)	(128 1 1)	10	allocate_data\$ck_L30_1
578.97ms	4.6720us	-	-	-	[CUDA memcpy HtoD]
578.98ms	1.2480us	(32 1 1)	(128 1 1)	10	allocate_data\$ck_L30_1
579.00ms	4.7040us	-	-	-	[CUDA memcpy HtoD]
579.01ms	1.2160us	(32 1 1)	(128 1 1)	10	allocate_data\$ck_L30_1
579.04ms	4.7040us	-	-	-	[CUDA memcpy HtoD]
579.05ms	1.2160us	(32 1 1)	(128 1 1)	10	allocate_data\$ck_L30_1
579.08ms	4.7040us	-	-	-	[CUDA memcpy HtoD]
579.09ms	1.2160us	(32 1 1)	(128 1 1)	10	allocate_data\$ck_L30_1

Entries ordered by time

Some OpenMP performance portability results

- To test performance we use a mixture of synthetic benchmarks and mini-apps.
- We compare against device-specific code written in **OpenMP 3.0** and **CUDA**.
- We eventually use OpenMP 4.x to run on *every diverse architecture that we believe is currently supported*.
- Our initial expectations were low – but initial results we produced in 2016 were promising.

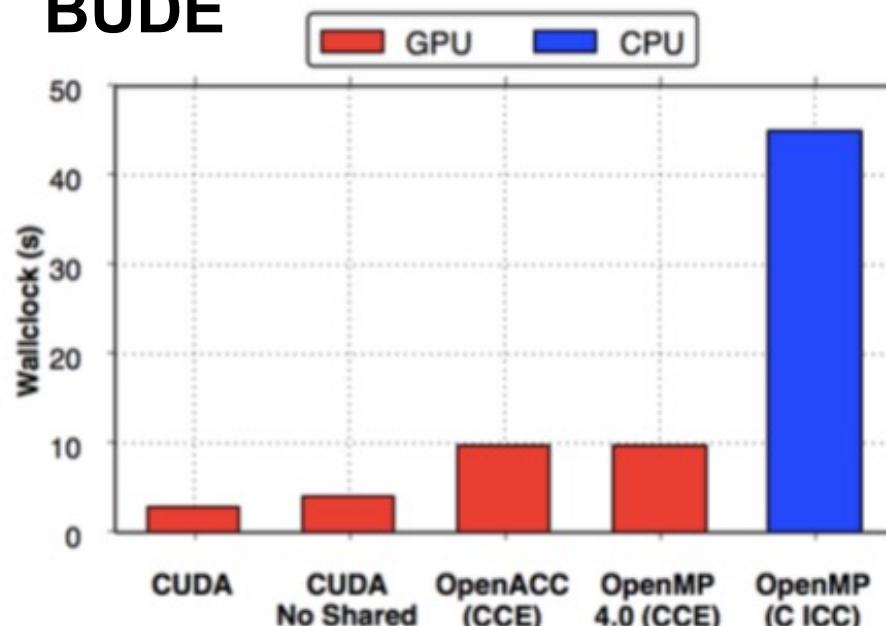
Performance?

* CCE 8.4.3, ICC 15.0.3, PGI 15.01, CUDA 7.0 on an NVIDIA® K20X, and Intel® Xeon® Haswell 16 Core Processor (E5-2698 v3 @ 2.30GHz)

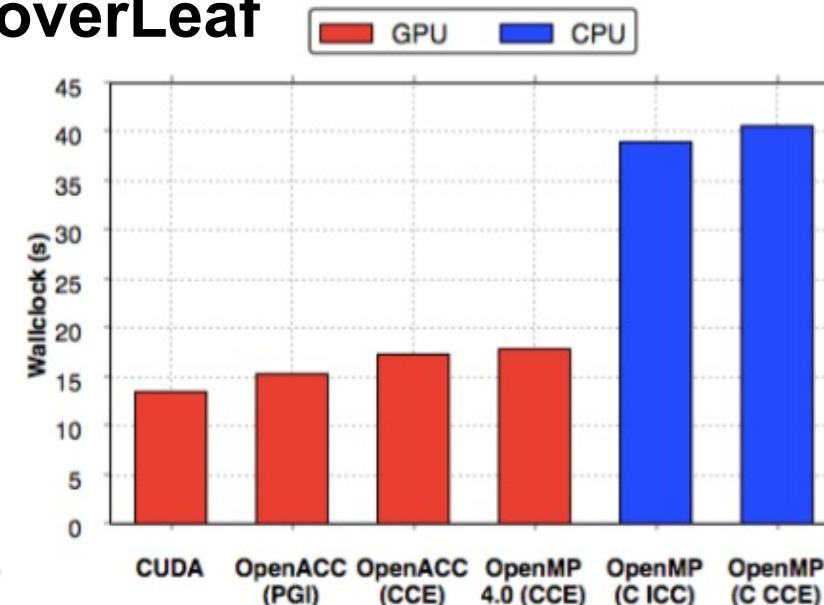
Immediately we see impressive performance compared to CUDA

Clearly the Cray compiler leverages the existing OpenACC backend

BUDE



CloverLeaf



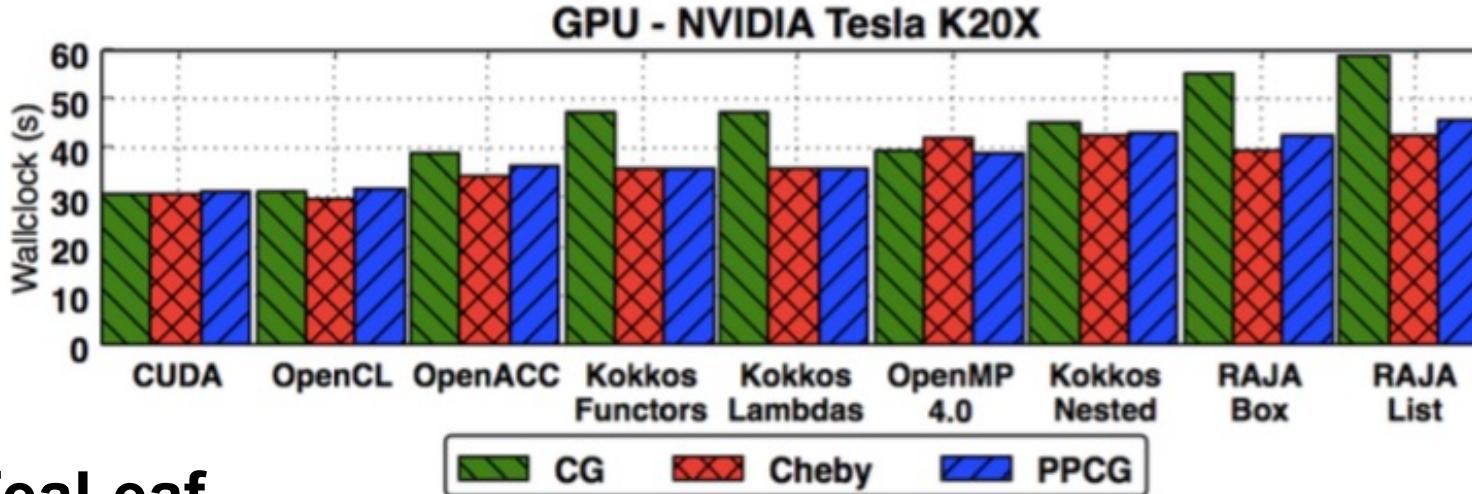
Even with OpenMP 4.5 there is still no way of targeting shared memory directly.

This is set to come in with OpenMP 5.0, and Clang supports targeting address spaces directly

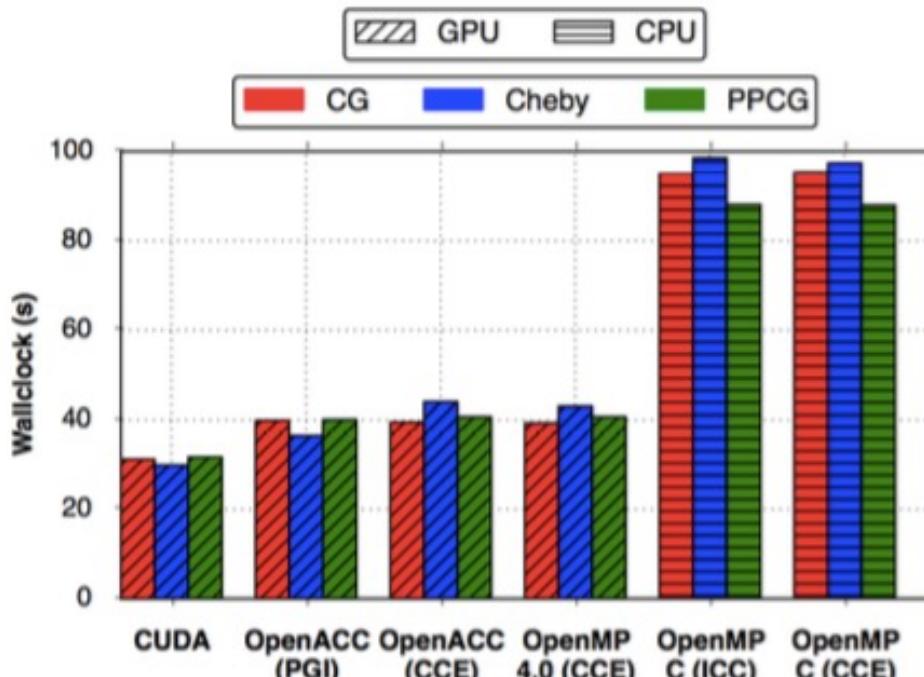
Martineau, M., McIntosh-Smith, S. Gaudin, W., *Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model*, 2016, HIPS'16

Performance?

* CCE 8.4.3, ICC 15.0.3, PGI 15.01, CUDA 7.0 on an NVIDIA® K20X, and Intel® Xeon® Haswell 16 Core Processor (E5-2698 v3 @ 2.30GHz)



TeaLeaf



Third party names are the property of their owners.

We found that Cray's OpenMP 4.0 implementation achieved great performance on a K20x

We have seen these figures continually improve as the languages have matured

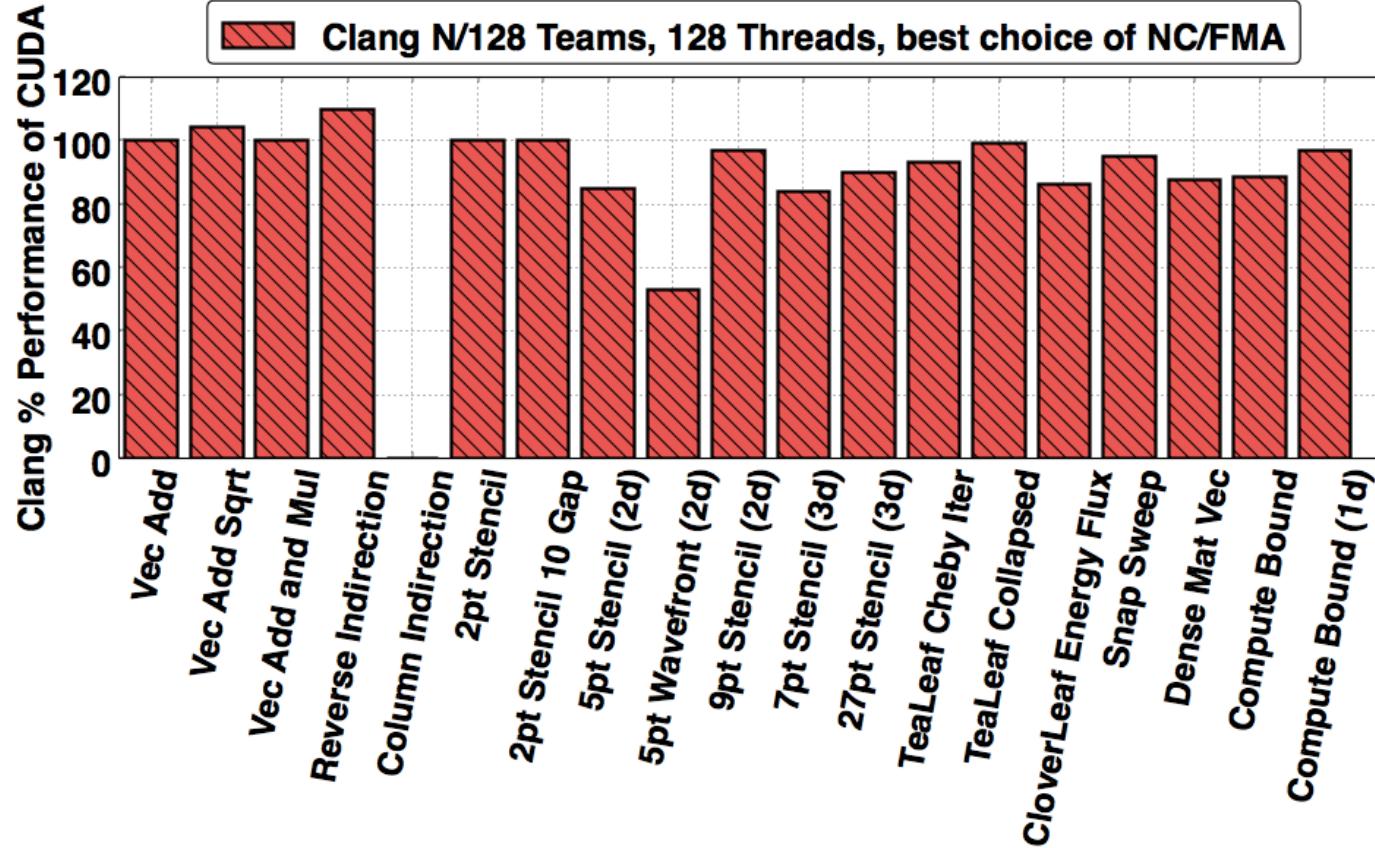
Martineau, M., McIntosh-Smith, S. Gaudin, W., Assessing the Performance Portability of Modern Parallel Programming Models using TeaLeaf, 2016, CC-PE

How do you get good performance?

- Our findings so far: You can achieve good performance on GPUs with OpenMP 4.x.
- We achieved this by:
 - Keeping data resident on the device for the greatest possible time.
 - Collapsing loops with the **collapse** clause, so there was a large enough iteration space to saturate the device (ideally $>10^4$ iterations to keep a modern GPU busy).
 - Using the BUD: `teams distribute parallel for simd`
 - ~~Using the **simd** directive to vectorize inner loops.~~
 - ~~Using **schedule(static, 1)** for coalescence (obsolete).~~
 - Using ***nvprof*** of course.

Can you do better?

* Clang copy <https://github.com/clang-ykt>,
CUDA 8.0, NVIDIA K40m



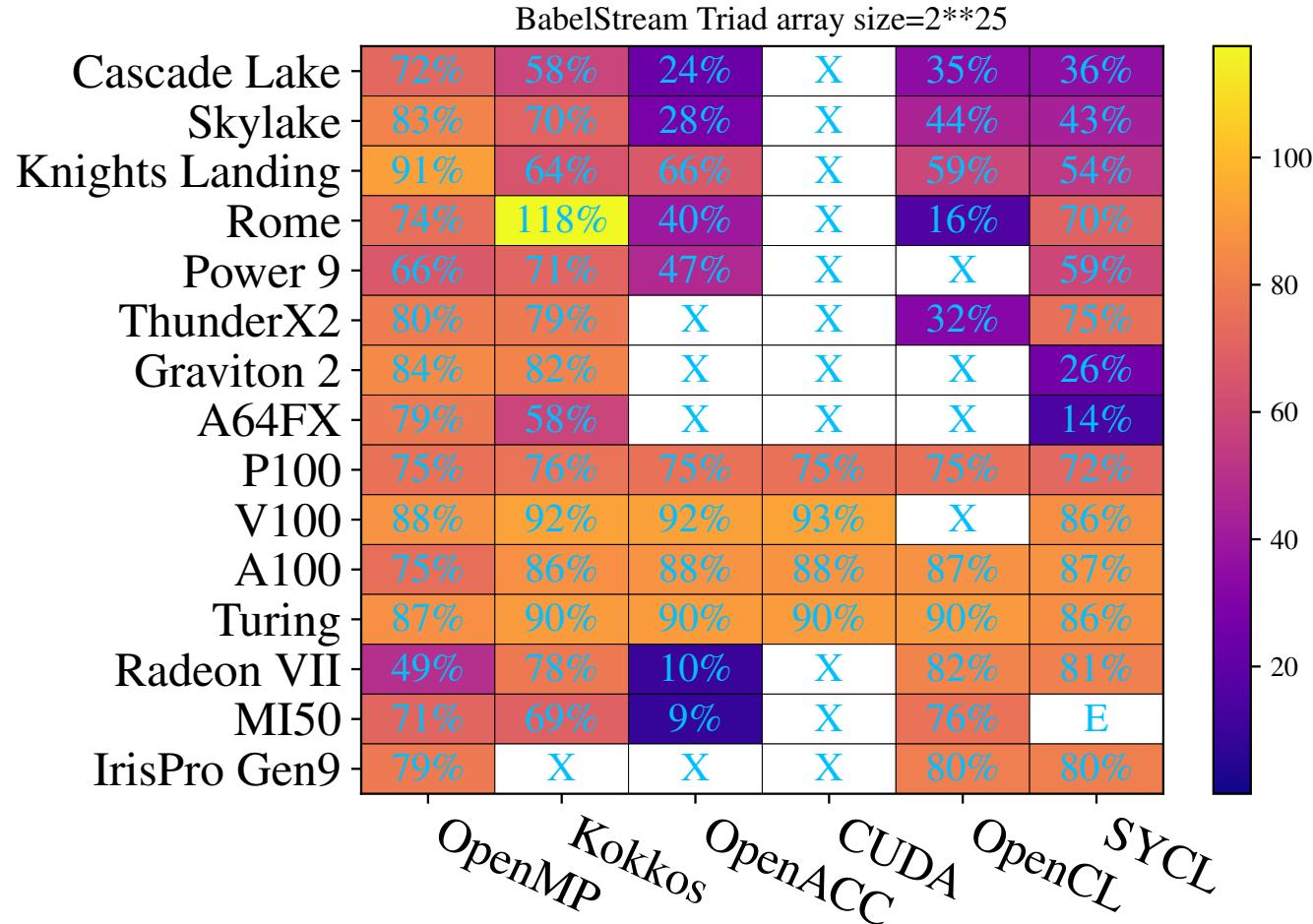
Through extensive tuning of the compiler implementation we were able to execute CloverLeaf mini-app within 9% absolute runtime of hand optimized CUDA code...

Martineau, M., Bertolli, C., McIntosh-Smith, S., et al. *Broad Spectrum Performance Analysis of OpenMP 4.5 on GPUs*, 2016, PMBS'16

Good. Performance... and Portability?

- Up until this point we had implicitly proven a good level of portability as we had successfully run OpenMP 4.x on many devices (Intel® CPU, Intel Xeon® Phi™ processors, NVIDIA® GPUs).
- The compiler support continually changes, improving performance, correctness and introducing new architectures.
- We keep tracking this improvement over time.

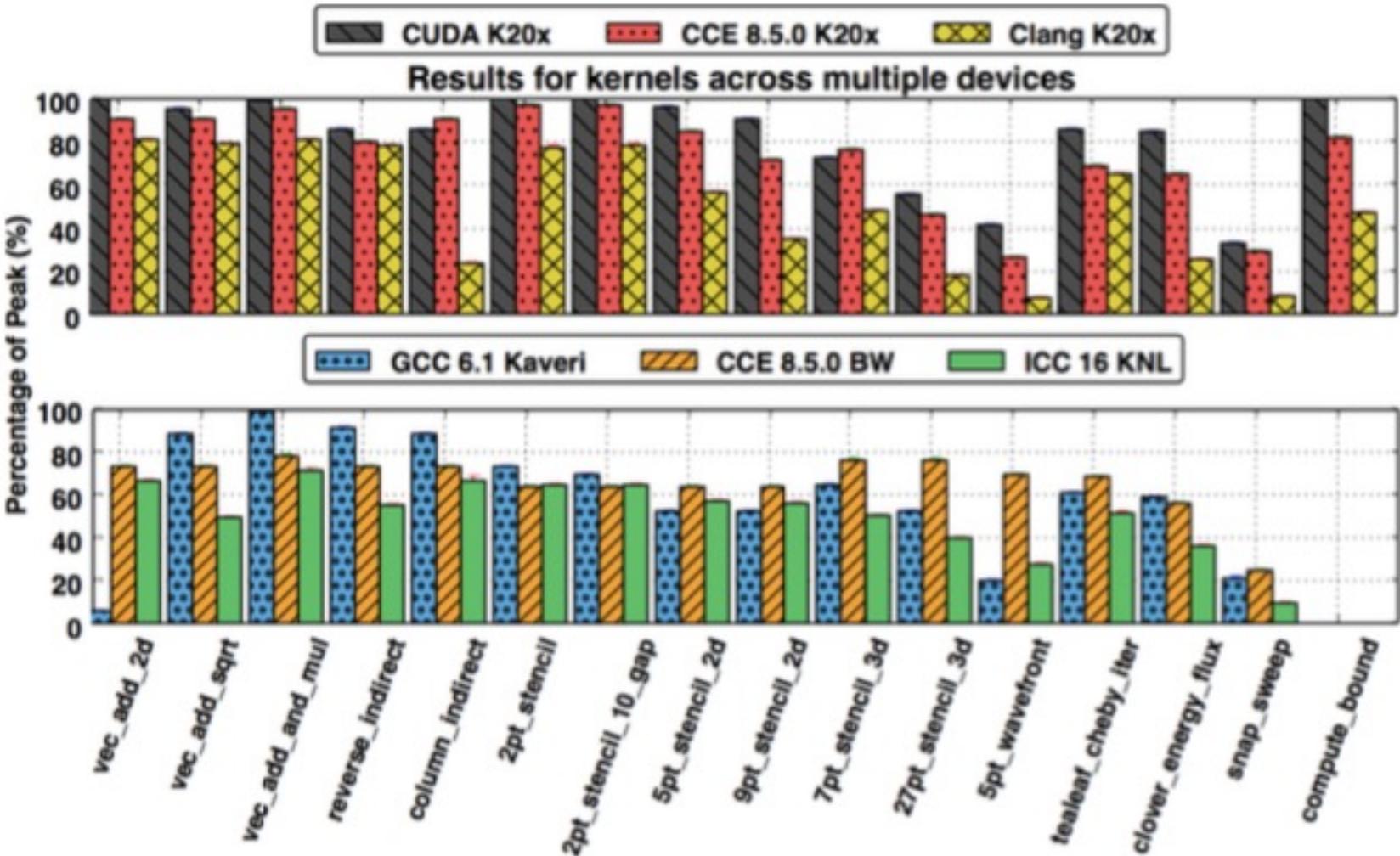
OpenMP in The Matrix



OpenMP has wide support,
and **good** performance
across all platforms

Deakin, T., et al., *Tracking Performance Portability on the Yellow Brick Road to Exascale*,
P3HPC Workshop, 2020. <http://p3hpc.org>

Nice - but beware of the caveat.



There is a **MAJOR** caveat - the directives were not identical.

The worst case scenario

```
// CCE targeting NVIDIA GPU
#pragma omp target teams distribute simd
for(...) {
}

// Clang targeting NVIDIA GPU
#pragma omp target teams distribute parallel for schedule(static, 1)
for(...) {
}

// GCC 6.1 target AMD GPU
#pragma omp target teams distribute parallel for
for(...) {
}

// ICC targeting Intel Xeon Phi
#pragma omp target if(offload)
#pragma omp parallel for simd
for(...) {
```

Four different ways of writing for the same kernel...

The answer:

- If you can, just use `#pragma omp loop!`
- If you can't, use most compilers would accept the combined construct:
 - `#pragma omp target teams distribute parallel for`
- This *does not* generalize to all algorithms unfortunately, but the majority can be adapted.
- The construct makes a lot of guarantees to the compiler and it is very easy to reason about for good performance.

Caveats

If you can - just use loop!

- *Real applications* will have algorithms that are structured such that they can't immediately use the combined construct.
- The handling of **clauses**, such as **collapse**, can be tricky from a performance portability perspective.
- Don't be misguided... performance is possible without using the combined construct, but it likely won't be consistent across architectures.

Performance Portability

If you can - just use loop!

- Feature complete implementations will allow you to write performant code, and they will allow you to write portable code.
- To get both will likely require algorithmic changes, and a careful approach to using OpenMP 4.5 in your application.
- Avoid setting **num_teams(nt)** and **thread_limit(tl)** if you can, this is definitely not going to be performance portable.
- Use **collapse(n)** in all situations where you expect the trip count of the outer loop to be small, but be aware that it can have a negative effect on CPU performance.
- Use the combined construct whenever you can.

Compiler Support for OpenMP target

- **NVIDIA** support much of OpenMP for GPUs in NVHPC
- **Intel** began support for OpenMP 4.0 targeting their Intel Xeon Phi coprocessors in 2013 (compiler version 15.0). Compiler version 17.0 and later versions support OpenMP 4.5. Compiler in oneAPI supports offload to Intel GPUs.
- **Cray** provided the first vendor supported implementation targeting NVIDIA GPUs in late 2015. CCE 9 moved to LLVM base. The latest version of CCE now supports all of OpenMP 4.5 and some of OpenMP 5.
- **AMD** AOMP compiler supports offload to AMD GPUs.
- **IBM** has recently completed a compiler implementation using Clang, that fully supports OpenMP 4.5. This is being introduced into the Clang main trunk.
- **LLVM/Clang** supports OpenMP 4.5 offload to NVIDIA GPUs. Used as base for many compilers.
- **GCC 6.1** introduced support for OpenMP 4.5. **GCC 10** can target Intel Xeon Phi, AMD GCN GPUs and NVIDIA GPUs.
- **PGI** compilers don't currently support OpenMP on GPUs (but they do for CPUs).

OpenMP compiler information: <https://www.openmp.org/resources/openmp-compilers-tools/>

OpenMP 5.x and ecosystem

- OpenMP 5 adds features to make writing performance portable programs simpler.
- Highlighting some applicable to target:
 - Interop
 - Mappers
 - Unified Shared Memory (USM) and requires
 - Function variants
 - Reverse offload
 - `OMP_TARGET_OFFLOAD`
 - Reduction result mapping
 - Reduction variables now implicitly map(tofrom)

OpenMP 5.0: Pointer attachment

- Map pointer variables and initialize them to point to device memory.

```
struct {  
    char *p;  
    int a;  
} S;  
S.p = malloc(100);
```

```
#pragma omp target data map(S)  
{  
#pragma omp target map(S.p[:100])  
{ // attach(S.p) = device_malloc(100);  
...  
}  
} // device_free(S.p[:100]), detach(S.p);  
}
```

```
free(S.p);
```

Map the structure S

Map 100 elements pointed to by S.p and update the pointer S.p on the device to point at the mapped elements.

OpenMP 5.0: #pragma omp declare mapper

- The **declare mapper** directive declares a user-defined *mapper* for a given type.
- A *mapper* defines a method for mapping complex data structures to a target device.
- A mapper may be used to implement a *deep copy* of pointer structure elements.

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;
```

```
#pragma omp declare mapper(myvec_t v)\n    use_by_default map(v, v.data[:v.len])  
size_t num = 50;  
myvec_t *v = alloc_array_of_myvec(num);
```

```
#pragma omp target map(v[:50])\n{\n    do_something_with_v(&v);\n}
```

Declare a mapper that declares how a structure variable of type myvect_t is mapped.

Use the mapper for myvec_t to map an array of type myvec_t

OpenMP 5.0: #pragma omp requires

- Code requires specific features, e.g. shared memory between host and devices.

```
typedef struct mypoints {  
    struct myvec * x;  
    struct myvec scratch;  
    double useless_data[500000];  
} mypoints_t;
```

```
#pragma omp requires unified_shared_memory
```

```
mypoints_t p = new_mypoints_t();
```

```
#pragma omp target  
{  
    do_something_with_p(&p);  
}
```

This code assumes that the host and device share memory.

No map clauses. All of p is shared between the host and device.

OpenMP 5.0: function variants

- Declare a device specific version (*variant*) of a function.
 - The variant is optimized for the device.

```
double a[N], b[N], c[N];
```

```
#pragma omp declare variant(fastFUNC) match(target)  
double FUNC(double, double);
```

```
#pragma omp target  
for (int i=0; i<N; i++)  
    a[i] = FUNC(b[i], c[i]);
```

Declare fastFUNC as a variant for FUNC when executing in a target region.

Call fastFUNC here instead of FUNC

OpenMP 5.0: reverse offload

- Execute a region of code back on the host from within a target region.
 - A target device may not be able to execute this code.

```
double a[N], b[N], c[N];
```

```
#pragma omp target map(to:b,c) map(from:a)
{
```

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

```
#pragma omp target device(ancestor:1)
    printf_array(a);
```

```
...
```

```
}
```

Execute printf_array back on the host

OpenMP 5.0: accelerators miscellaneous

- Implicit **declare target** directives
 - No need to put **omp declare target** on every function if compiler can determine function is used on a target device.
- Allow **declare target** on C++ classes with virtual members.
- **defaultmap(*implicit-behavior*[:*variable-category*)**
 - E.g. **defaultmap(to:aggregate)**, **defaultmap(alloc:scalar)**
- **OMP_TARGET_OFFLOAD MANDATORY | DISABLED**
 - A new environment variable that controls device constructs.
- C/C++ array shaping
 - E.g. **int *p; #pragma omp map(([10][1024*1024])p[i])**
- Many other clarifications...