PROGRAMMING YOUR GPU WITH OPENMP
A "HANDS-ON" INTRODUCTION

Tom Deakin (University of Bristol)

Tim Mattson (University of Bristol)

SC24
Atlanta, GA | hpc creates.

Programming Your GPU with OpenMP

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

Tim Mattson
Human Learning Group
tgmattso@gmail.com

This tutorial material includes many contributors, including Eric Stotzer from Mythic Inc, and Harry Waugh, James Price, Matt Martineau and others from the University of Bristol's HPC Research Group.

All the tutorial materials are available online

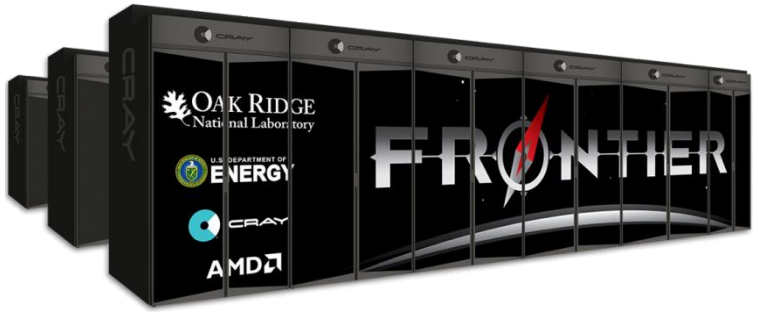
Download the latest version of these slides, and all tutorial materials, at the URL below

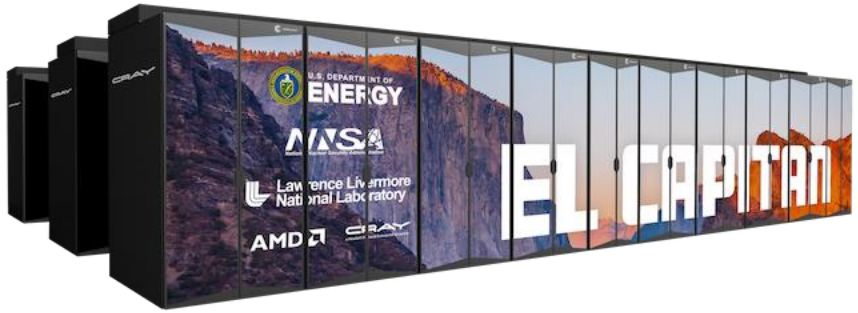
https://github.com/uob-hpc/openmp-tutorial

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





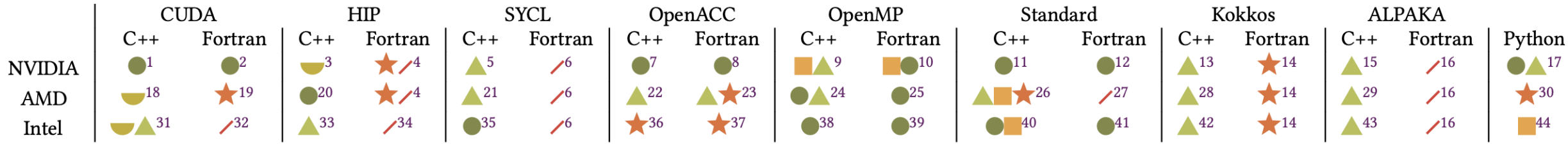
Preliminaries: Part 1

- Disclosures
  - The views expressed in this tutorial are those of the people delivering the tutorial.
    - We are not speaking for our employers.
    - We are not speaking for the OpenMP ARB
- We take these tutorials VERY seriously:
  - Help us improve … tell us how you would make this tutorial better.

Preliminaries: Part 2

- Our plan for the day .. **Active Learning**!
  - We will mix short lectures with short exercises.
  - You will use your laptop to connect to a remote system which includes GPUs.
- Please follow these simple rules
  - Do the exercises that we assign and then change things around and experiment.
    - Embrace active learning!
  - Don't cheat: Do Not look at the solutions before you complete an exercise … even if you get really frustrated.

Why is OpenMP so important?



Full vendor support
Indirect, but comprehensive support, by vendor
Vendor support, but not (yet) entirely comprehensive
Comprehensive support, but not by vendo
Limited, probably indirect support – but at lea some
No direct support available
C++ C++ (sometimes also C)
Fortran Fortran

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

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

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

Agenda

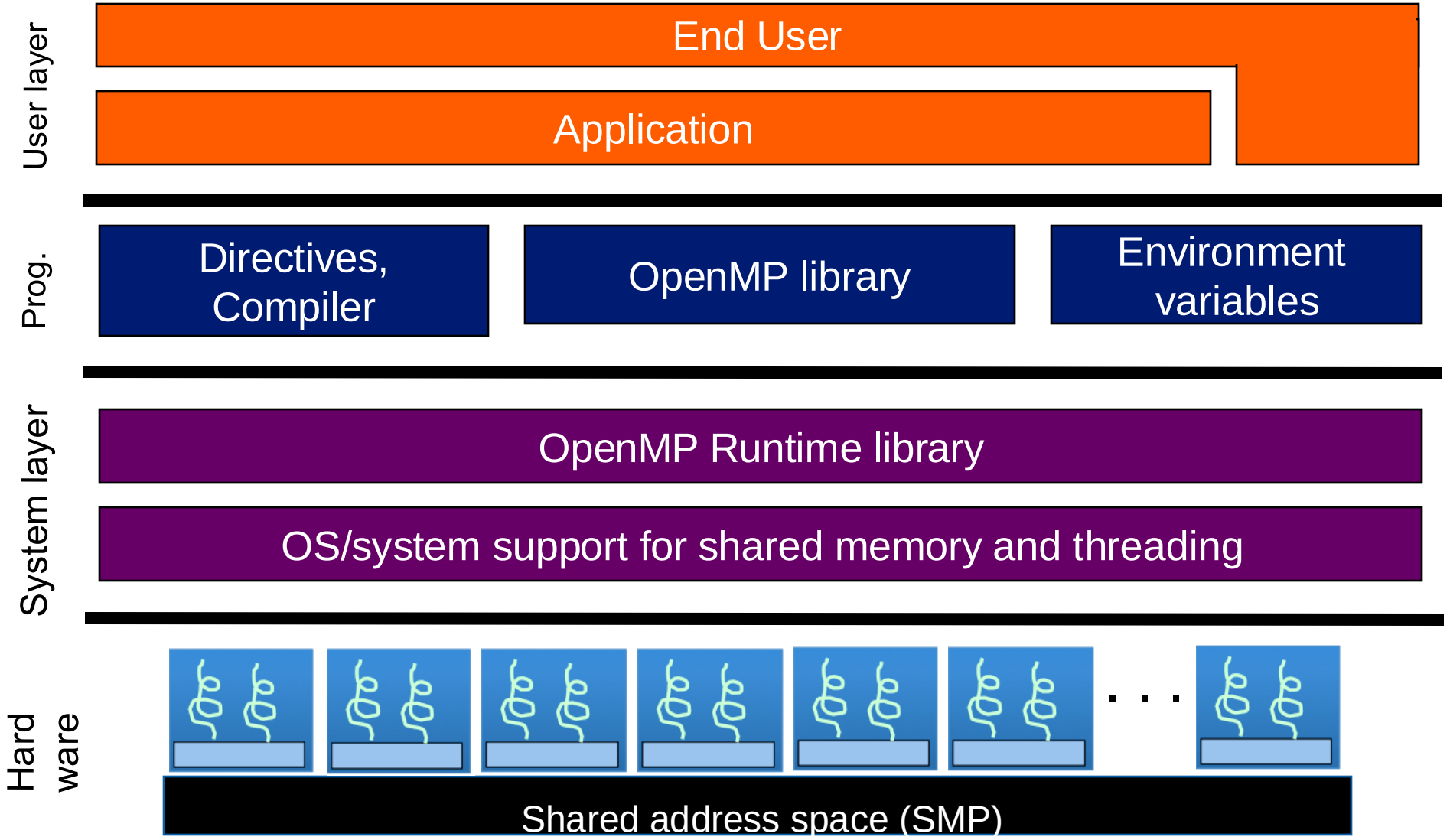
Morning

- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

Afternoon

- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

OpenMP basic definitions: the solution stack

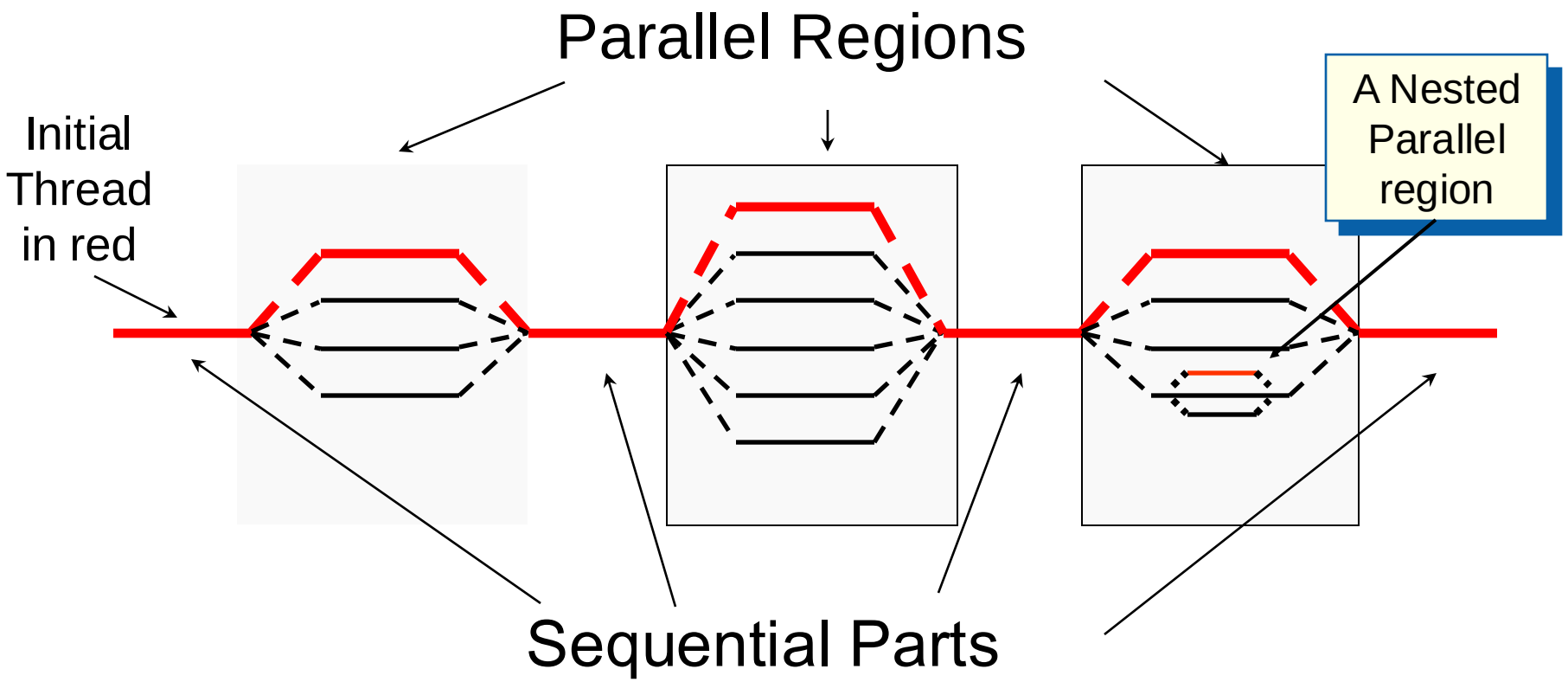


When OpenMP was originally launched, the focus was on **Symmetric Multiprocessing**
…. i.e. lots of threads with “equal cost access” to memory

OpenMP programming model

Fork-Join Parallelism:

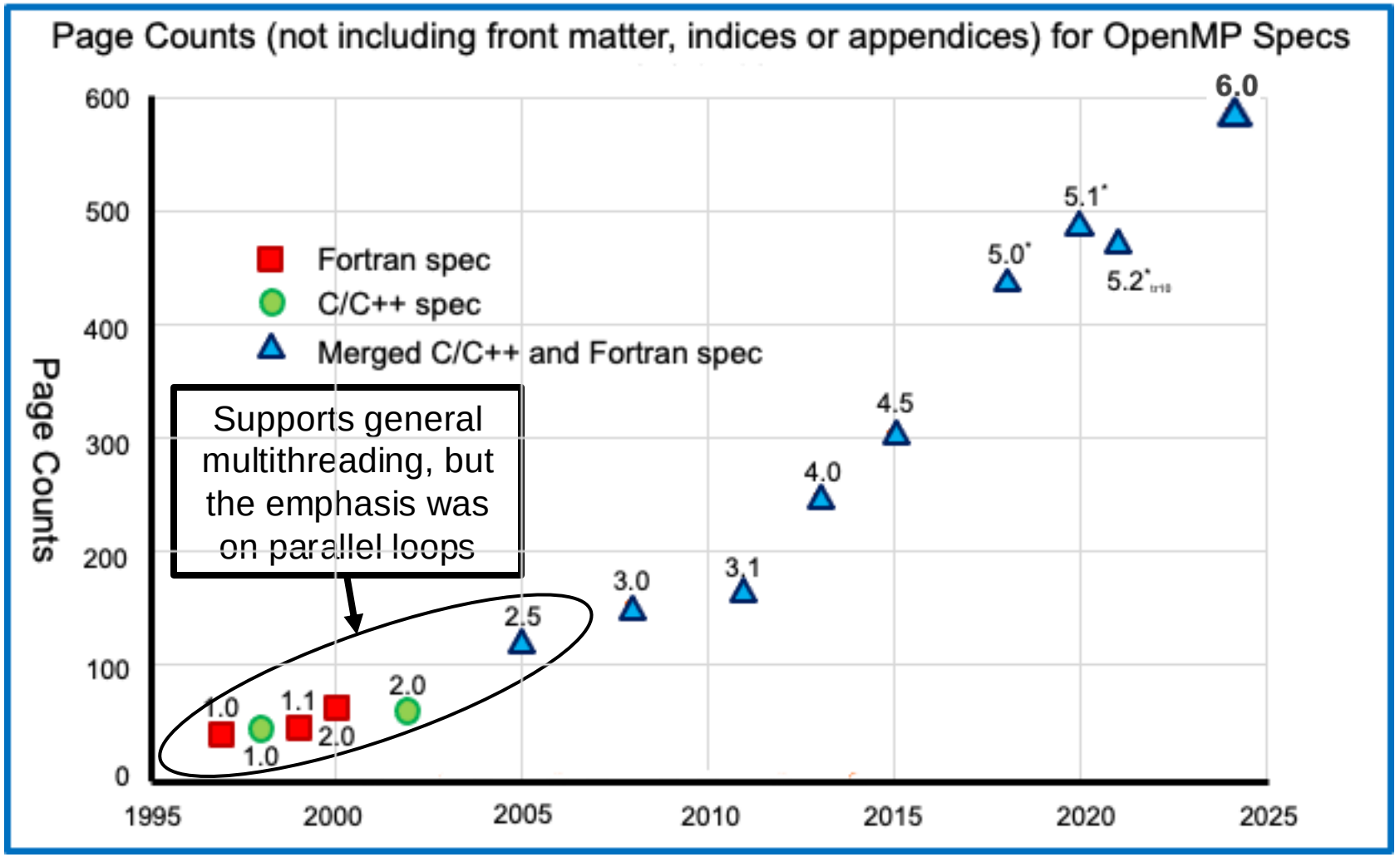
- Initial thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



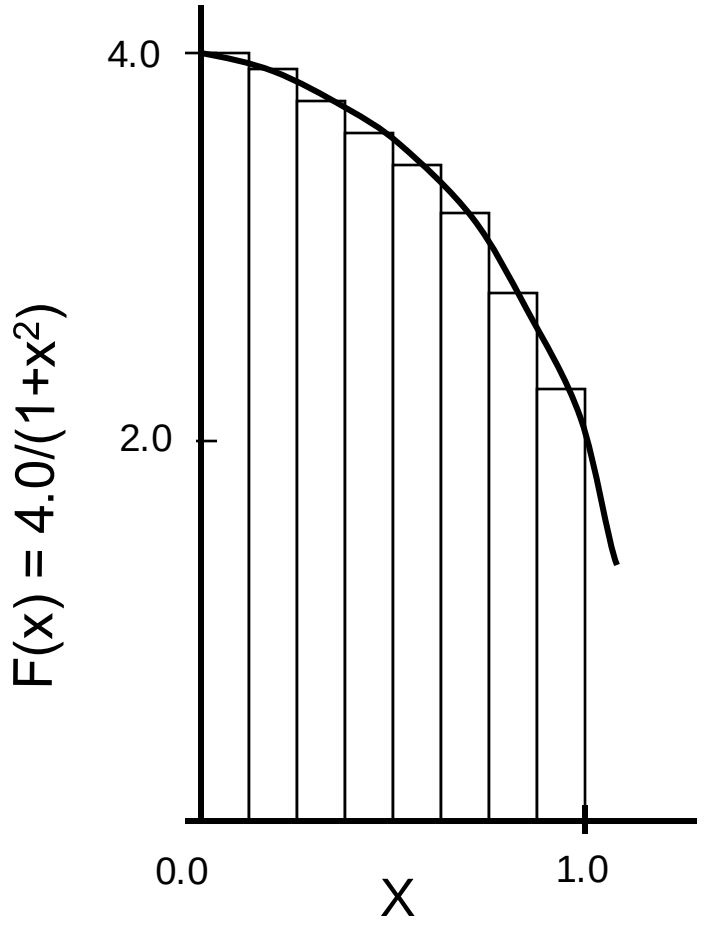
The growth of complexity in OpenMP

- 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 for private(x) reduction(+:sum)
     for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
     }

     pi = step * sum;
}
```

Include file to hold function prototypes, types and other items needed to support OpenMP

Create a team of threads (parallel)… share the work of the for loop between the threads (for).

Note … the loop index (i) of the parallel loop is made local to each thread.

Create a variable local to each thread for the value of x at the center of each interval

Create a copy of the variable, sum, for each thread. Initialize it with the identity of the reduction operator (+ is the operator, 0 is the identity)

Threads wait here until all threads complete their work (a barrier).

Finish the reduction … combine sum from each thread using the reduction operator (+). Then combine with the "original" value of sum using the reduction operator.

Example: combining loops to create more work for the parallel loop.

Matrix multiplication: C = C + A * B where $C \in R^{N,M}$, $A \in R^{N,P}$, $B \in R^{P,M}$

```
void mm_ikj_par(int N, int M, int P, float *A, float *B, float *C)
{

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

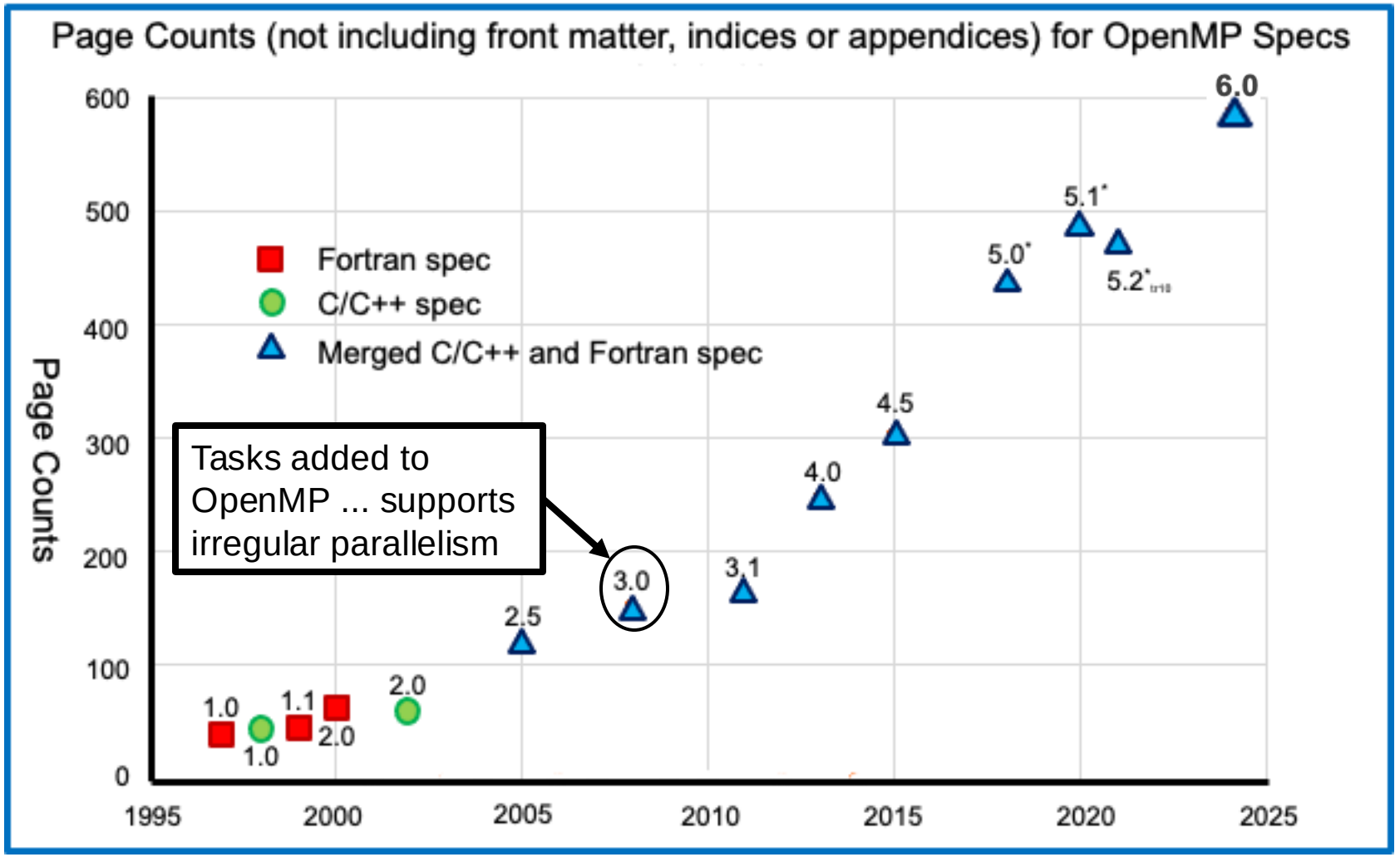
If N and M are not large, you may not have enough work to keep all the threads busy.

collapse(2) will collapse the following 2 nested loops into one larger loop … therefore giving the parallel loop more work to do.

The growth of complexity in OpenMP

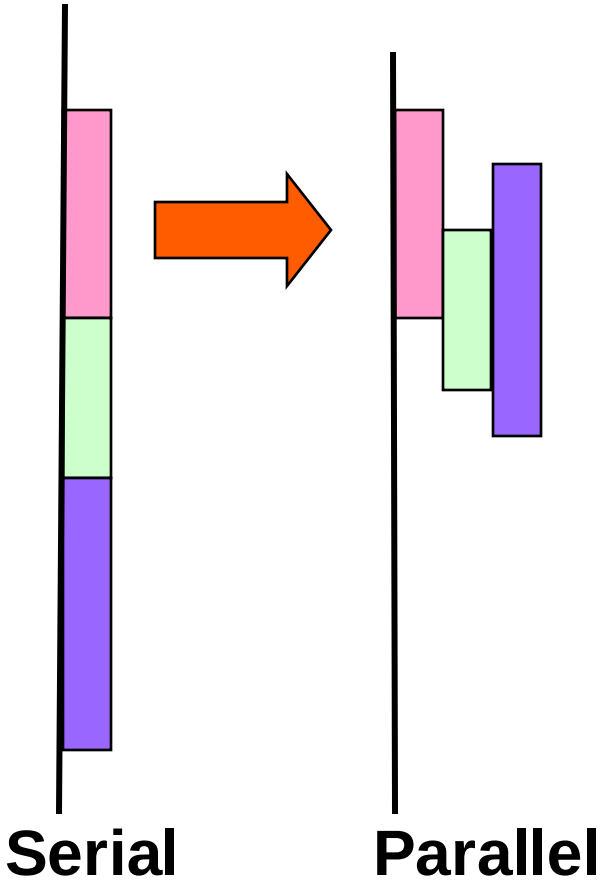
- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.

- The complexity has grown considerably over the years!



What are Tasks?

- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with



- You fill a queue with tasks and then a team of threads does the work … Supports irregular workflows.

```
#pragma omp parallel
{
  #pragma omp single
   {
    p = listhead ;
    while (p) {
       #pragma omp task firstprivate(p)
       {
         process (p);
       }
       p=next (p) ;
     }
   }
}
```

Create a team of threads

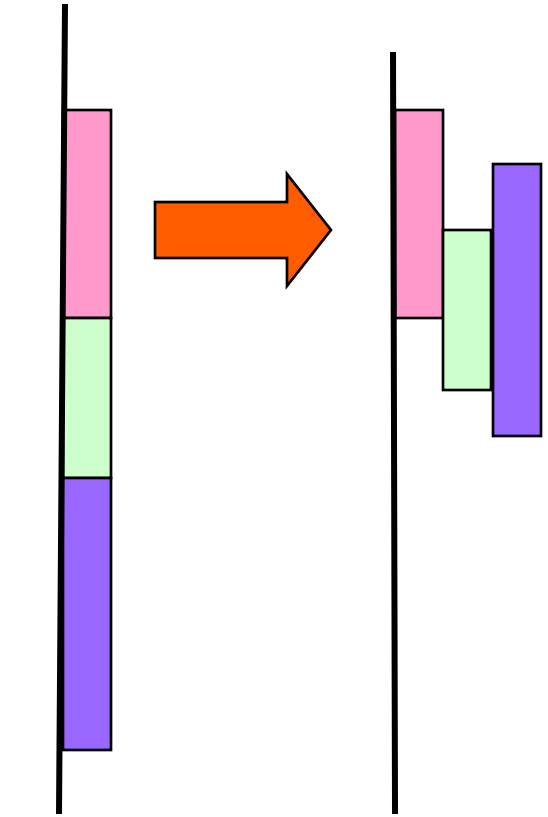
One thread walks the list. The others wait at the barrier at the end of the single construct

Create a task, a private copy of p for each task, and assign p the value of p when the task is created (i.e. firstprivate)

An implied barrier at the end of the single construct

What are Tasks?

- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with



- You fill a queue with tasks and then a team of threads does the work … Supports irregular workflows.

```
#pragma omp parallel
{
  #pragma omp single nowait
   {
    p = listhead ;
    while (p) {
       #pragma omp task firstprivate(p)
       {
         process (p);
       }
       p=next (p) ;

   }
}
```

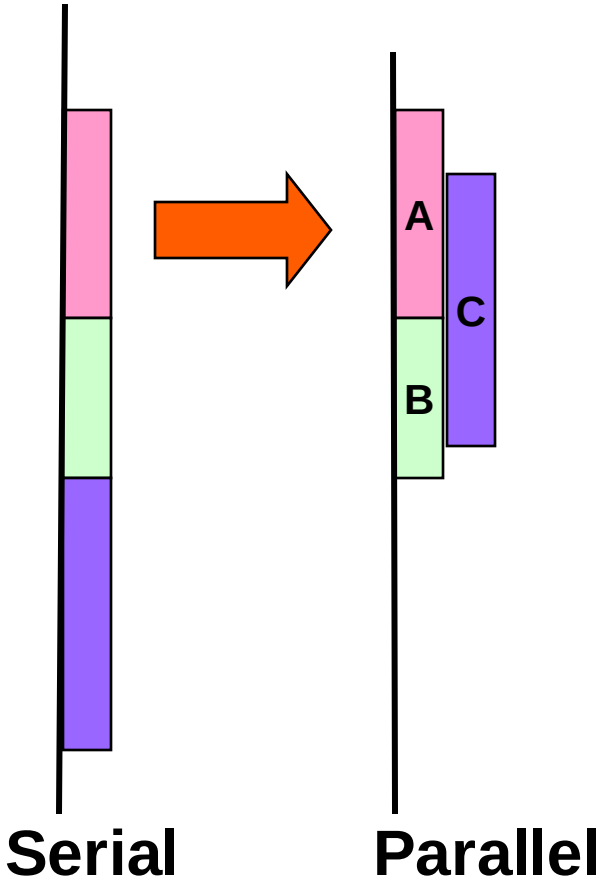
Skip the barrier implied by single

You proceed past the barrier once all tasks, including those nested inside process(), are complete.

An implied barrier at the end of the parallel region

What are Tasks?

- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with



- We can define dependencies between tasks.

```
int a,b,c;
#pragma omp parallel
{
  #pragma omp single nowait
   {
      #pragma omp task depend(out: a)
         taskA (&a);
      #pragma omp task depend(in: a)
         taskB (a,&b);
      #pragma omp task
         taskC (&c);
   }
}
```

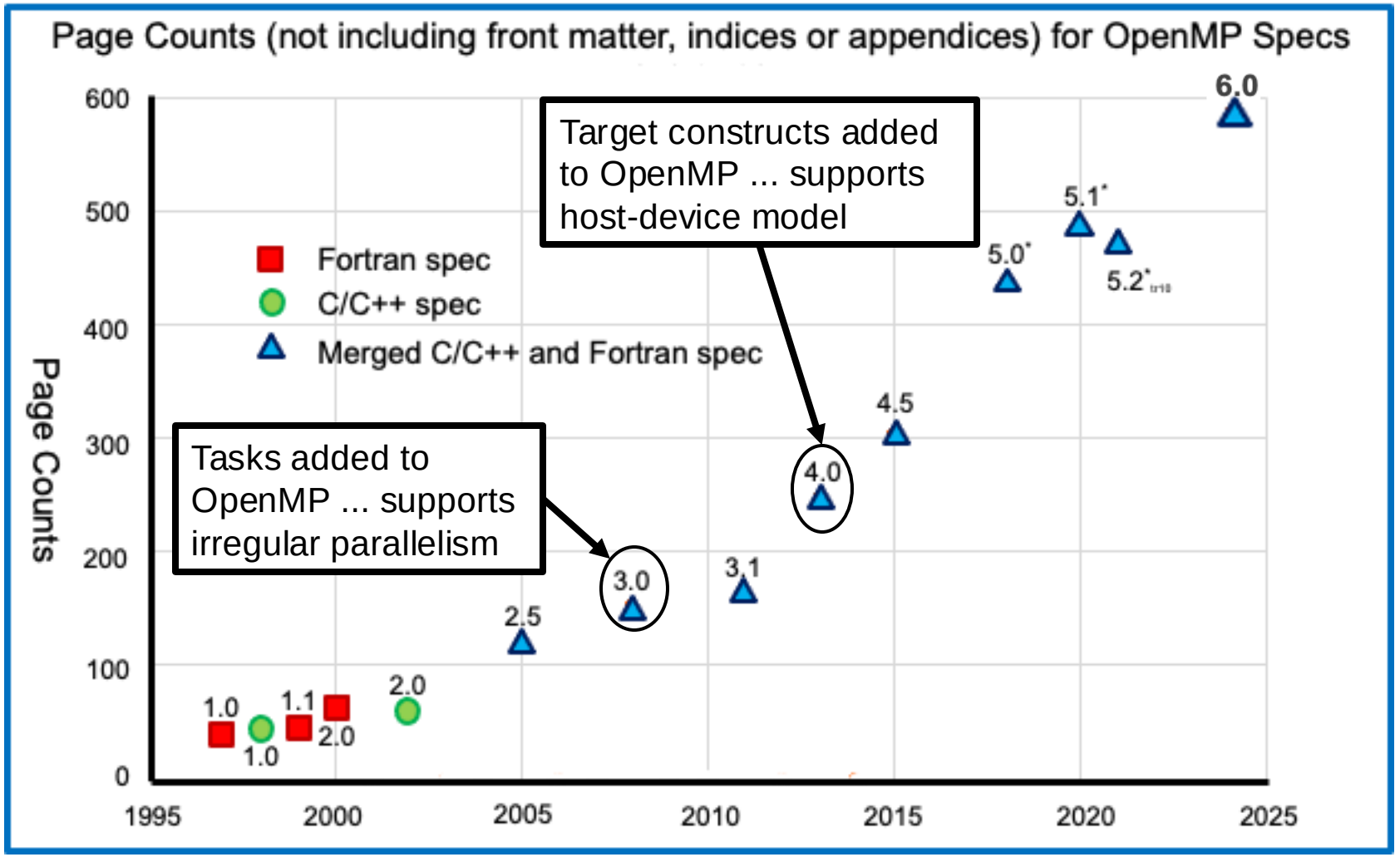
taskB() won't start until taskA() is done.

taskC() is unordered with respect to the other tasks

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!



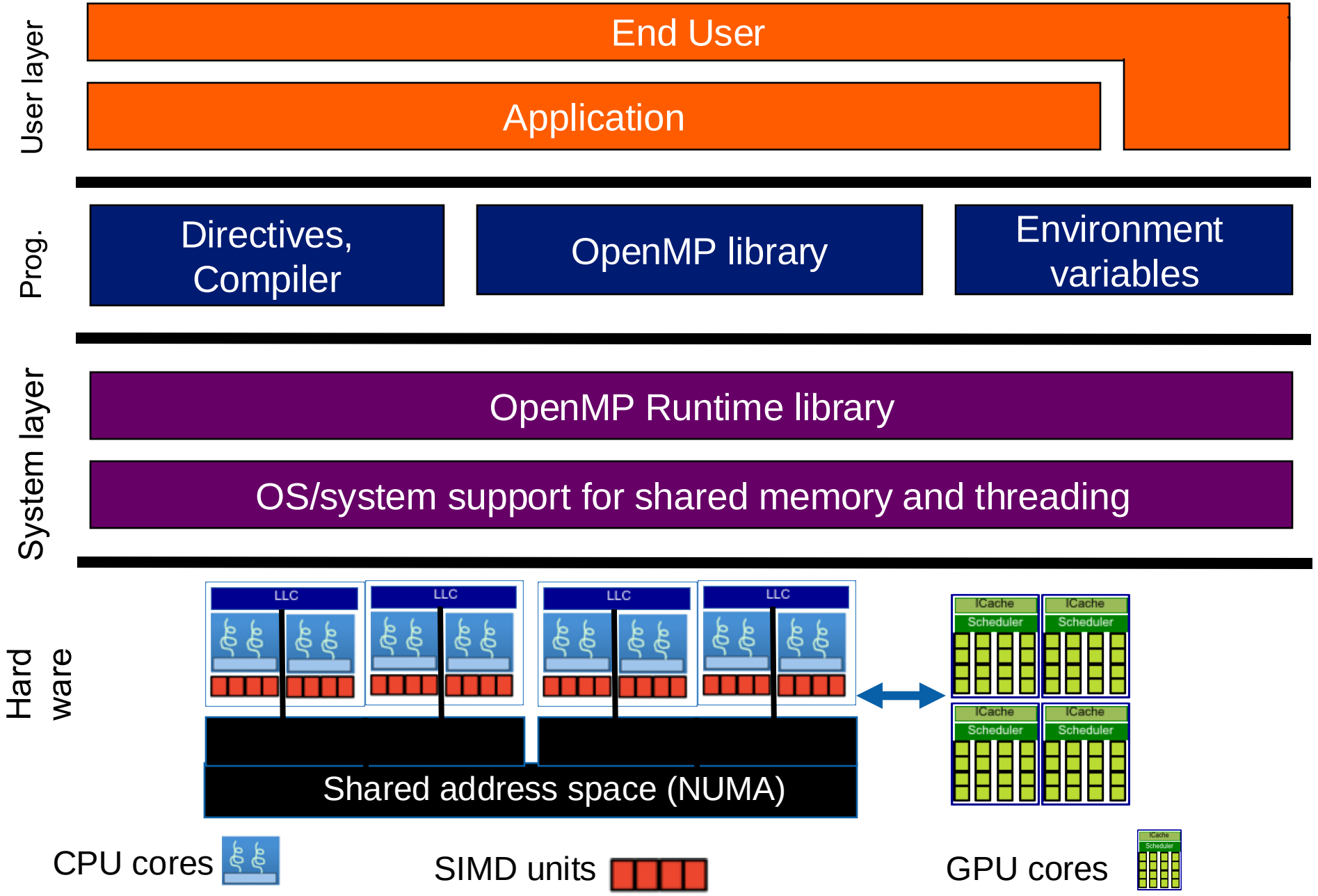
OpenMP programming model

- 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



Live exercise 1

Log in to Cluster and simple CPU vector add in OpenMP

AWS Parallel Cluster

- Thanks to AWS for supporting this tutorial!

- You have access to a ParallelCluster with some NVIDIA Tesla T4 GPUs
- NVIDIA NVHPC compiler toolchain installed

- Register for an account at https://tinyurl.com/sc24ompgpu
- http://sc24ompgpu.s3-website.eu-west-2.amazonaws.com

- Username: train*xy* (two-digit number)
- Password: openmp24

- Headnode IP: 44.204.72.120

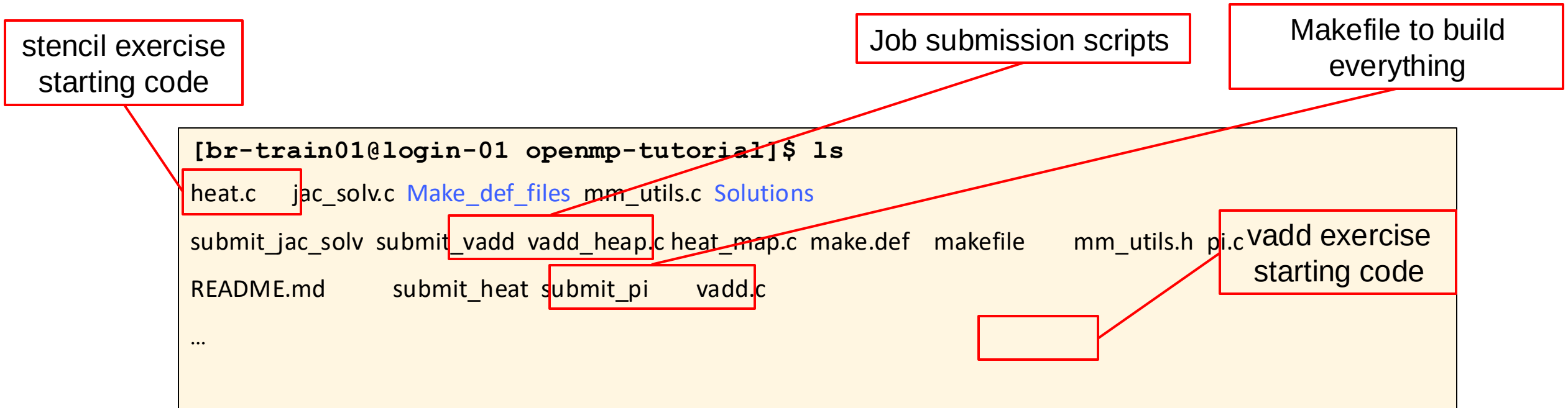
Using our AWS ParallelCluster (1/2)

https://tinyurl.com/sc24ompgpu

ssh trainXX@ 44.204.72.120

openmp24

```
# 1) Log in to the head
ssh trainXX@44.204.72.120


# 2) Change to the directory containing the exercises
cd openmp-tutorial

# 3) List files
ls
```



Using our AWS ParallelCluster (2/2)

https://tinyurl.com/sc24ompgpu
ssh trainXX@ 44.204.72.120
openmp24

```
# Load the compilers (NB: you need both!)
spack load gcc@12 nvhpc

# Build the exercises
make

# Submit a job
sbatch submit_vadd

# Check job status
squeue -u $USER
```

Job status:
- R = **R**unning
- PD = **P**en**d**ing
- CD = **C**omplete**d**
- CF = **C**on**f**iguring

(job will disappear shortly after completion)

```
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
   32    queue0     vadd ec2-user CF       1:10      1 queue0-dy-queue0-compute-resource-0-4
```

```
# Check output from job
# Output file will have the job identifier in name
# This will be different each time you run a job
cat vadd-32.out
```

Exercise: Simple vector add in OpenMP on CPU

Edit file: vadd.c

- Based on a simple parallel pattern: vector addition
- This adds together two arrays, element by element
- We will build on this over the next few exercises
  - Highlights the OpenMP concepts you're learning

- Check you can log into the Cluster
- Take the serial vector add example we've provided, and add OpenMP worksharing directives to run in parallel on the CPU
  - #pragma omp parallel for

https://tinyurl.com/sc24ompgpu

ssh trainXX@ 44.204.72.120

openmp24

Solution: Simple vector add in OpenMP on CPU

Files: Solutions/vadd_par.c, Solutions/submit_vadd_par

```
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 parallel for
   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;
}
```

Agenda

Morning

- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

Afternoon

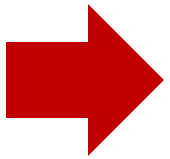
- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

The "BIG idea" Behind GPU programming

Single Instruction Multiple Thread (SIMT) execution model

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

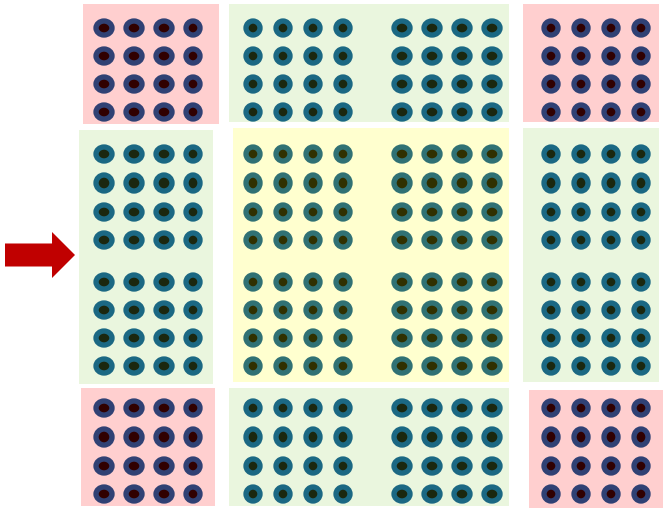
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Create a set of work-items from kernel code

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

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

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

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

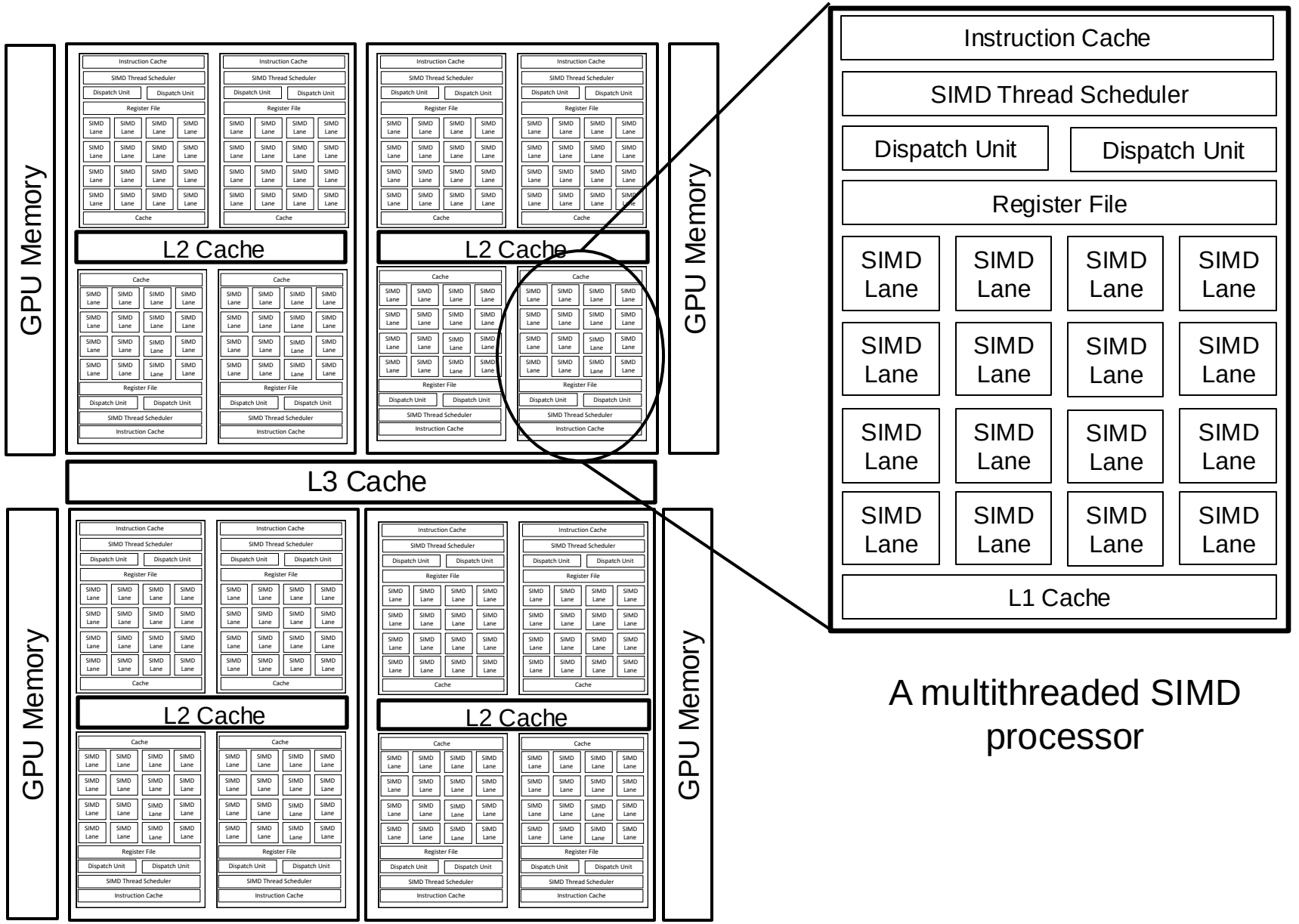
2. Map work-items onto an N dim index space.



3. Map data structures onto the same index space

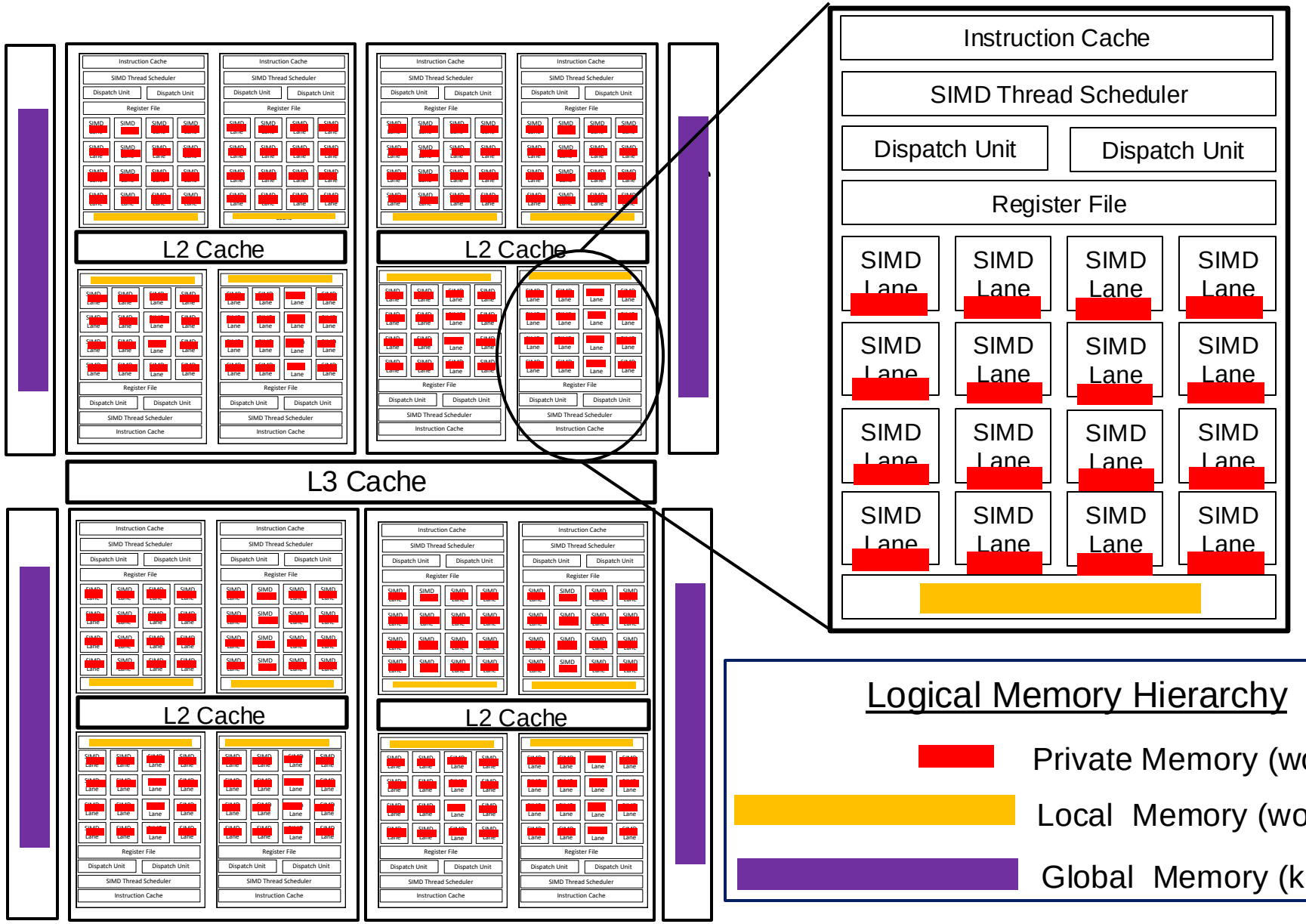
4. Run on hardware designed around the same SIMT execution model

A Generic GPU (following Hennessey and Patterson)



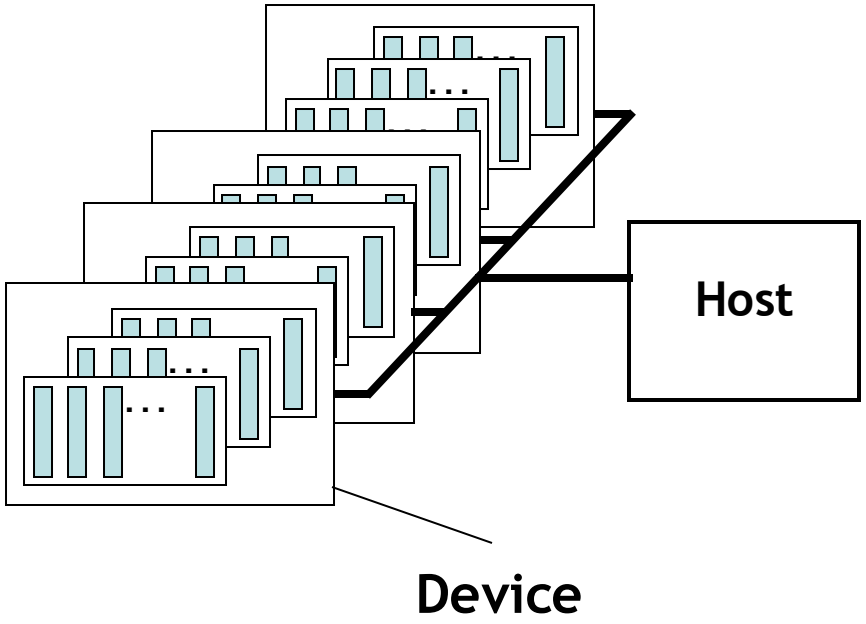
A multithreaded SIMD processor

A Generic GPU (following Hennessey 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



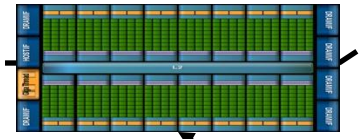

```
#include <omp.h>
#include <stdio.h>
int main()
{
    printf("There are %d devices\n",
            omp_get_num_devices());
}
```

OpenMP with target devices

- The target construct offloads execution to a device.

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

1. Program begins. Launches **initial thread** running on the **host device**.
2. Implicit parallel region surrounds entire program
3. Initial task begins execution

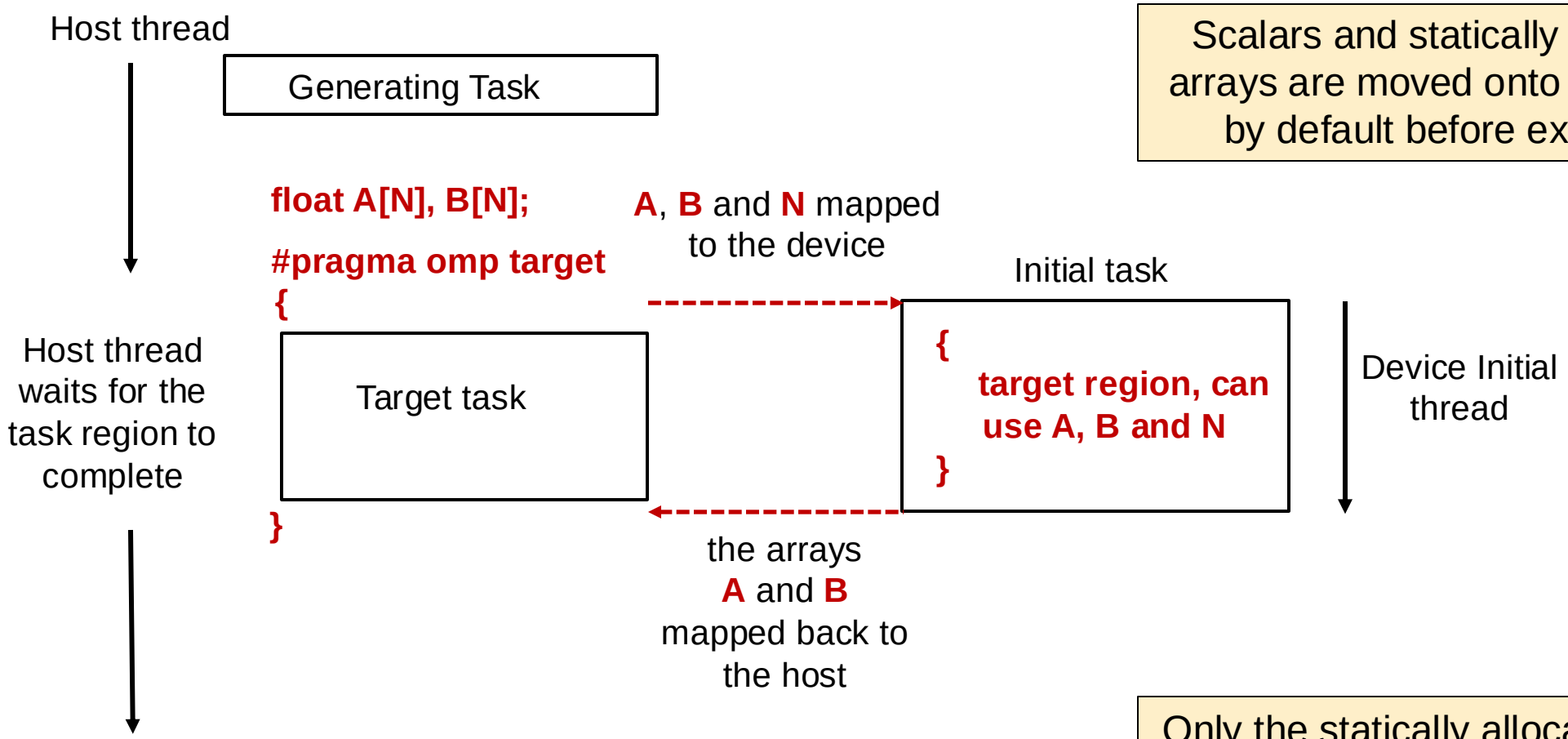


4. Initial task encounters the **target** directive.
5. Initial task generates a target task which is a mergable, included task
6. Target task launches target region on the device

7. A new initial thread runs on the device.
8. Implicit parallel region surrounds device program
9. Initial task executes code in the target region.

10. Initial task on host continues once execution associated with the target region completes

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



Scalars and statically allocated arrays are moved onto the device by default before execution

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

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

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.

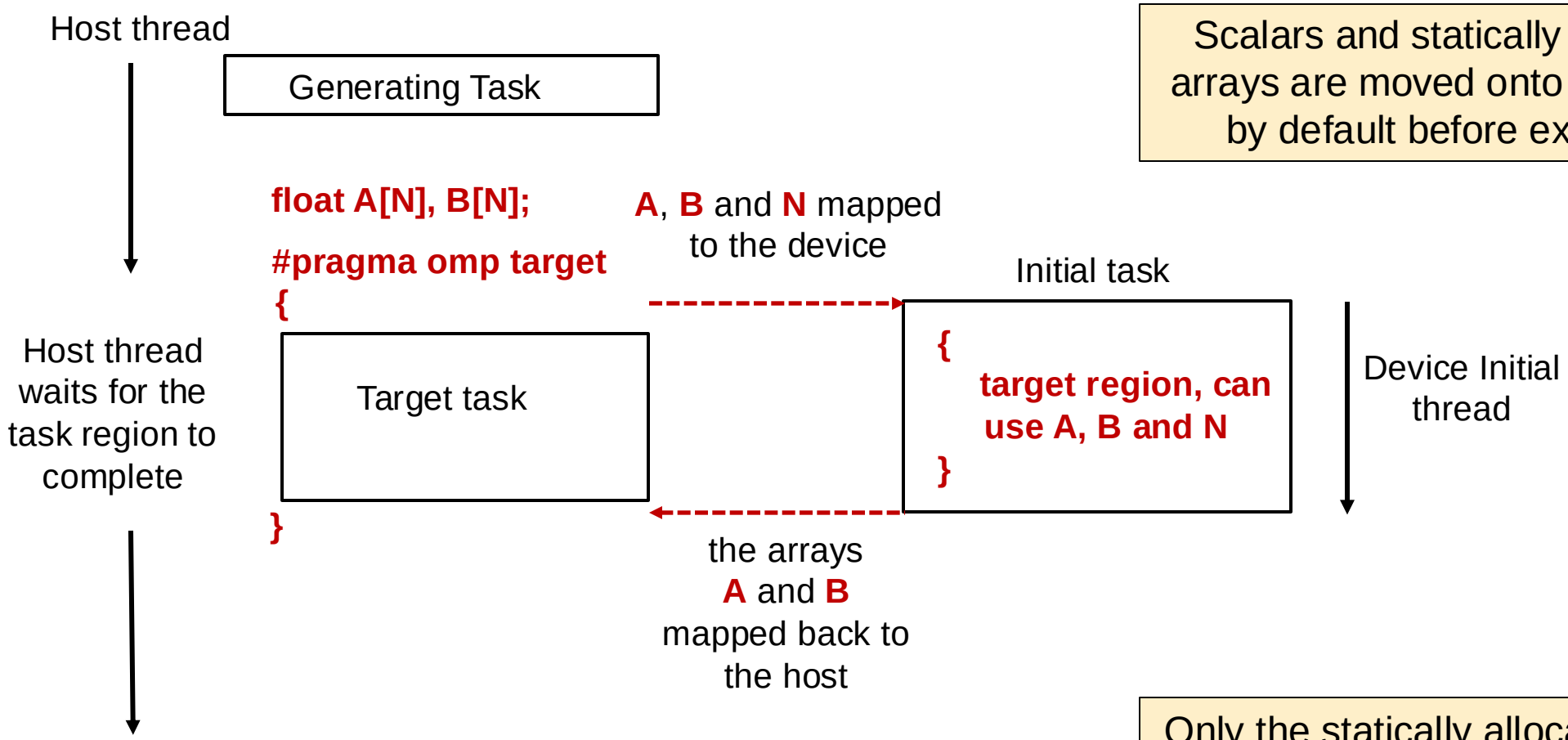
Default Data Mapping:
implicit movement with a target region

- 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 = (double *)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



Scalars and statically allocated arrays are moved onto the device by default before execution

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

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

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.

Loop directive on the CPU

- #pragma omp loop
- #pragma omp parallel loop

- The loop construct says that the iterations of the loop can be run in any order (concurrently)
- It's a contract that says the loop does not contain:
  - OpenMP API calls, calls to procedures with OpenMP directives, and any directive other than parallel/simd/loop

- The loop directive *binds* to the parallel (or teams) region it is found inside
- If not, it binds to the encountering thread, and the compiler can help out thanks to the "as-if" rule.
  - Descriptive parallelism.

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

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.

Later, we will discuss the "Big Ugly Directive" (BUD) where all the details of running a loop on a GPU are exposed. With BUD you don't need to map the reduction variable!

Live exercise 2

Vector add on a GPU

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: 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

https://tinyurl.com/sc24ompgpu

ssh train*XX*@ 44.204.72.120

openmp24

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;
}
```

Agenda

Morning

- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

Afternoon

- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

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  ii=0, 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)**.

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

- Otherwise, the pointer is mapped as a zero-length array (A[0:0])

- 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)

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:

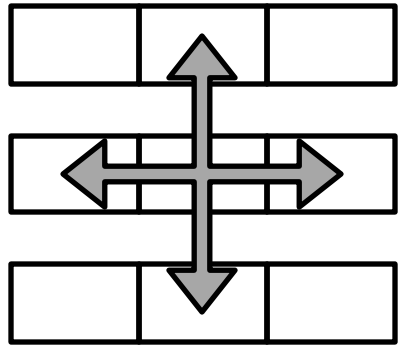
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

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

  - 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);
    }
  }
}
```

Agenda

Morning

- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

Afternoon

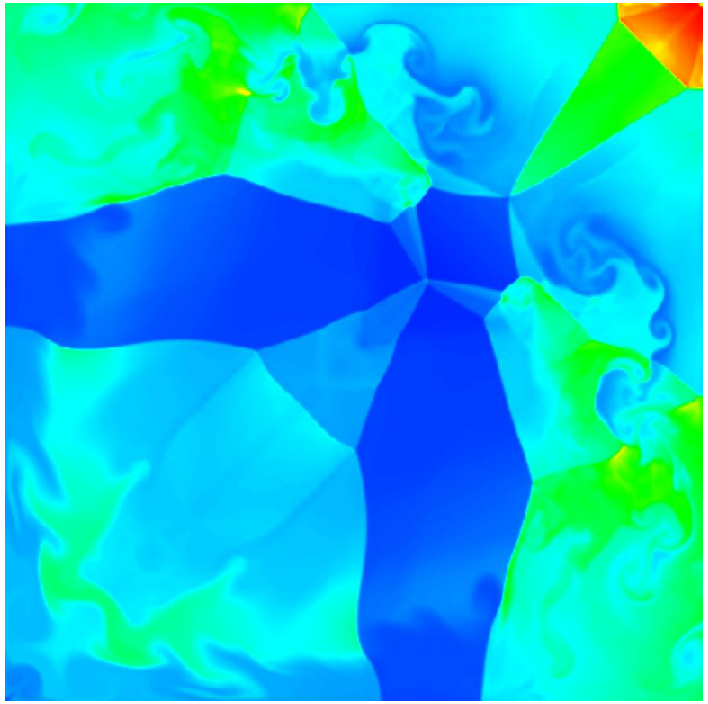
- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

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_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
```

Time to copy data onto GPU

Time to copy data back from GPU

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

Live exercise 3

Parallelising stencil on a GPU

Exercise: parallel stencil (heat)

https://tinyurl.com/sc24ompgpu

ssh trainXX@ 44.204.72.120

openmp24

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?
  - ~~Note, on Isambard, the profile can be run by nsys nvprof --profile-child-processes ./heat~~

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

Exercise: heat code inputs

https://tinyurl.com/sc24ompgpu

ssh trainXX@ 44.204.72.120

openmp24

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  __omp_offloading_...
```

Data movement dominates!

Typically lots of iterations!

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

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

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

Welcome back and recap (5min)

Agenda

Morning

- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

Afternoon

- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

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

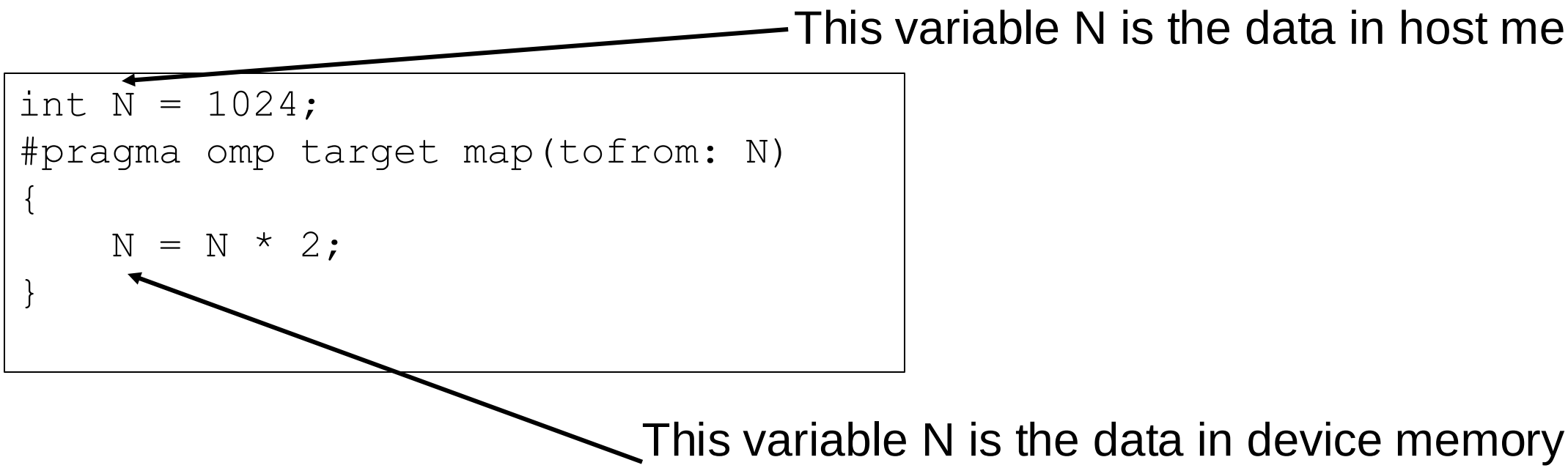
```
#pragma omp target data map(to:A[0:N], B[0:M]) map(from: C[0:P])
{
    #pragma omp target
        {do lots of stuff with A, B and C}

    {do something on the host (not with A,B,C)

    #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

Briefly, original and corresponding variables

- Variables are names for data in memory



To tell them apart, OpenMP gives them names:
The *original variable* and the *corresponding variable*

Mapped variables are reference counted

- Variables that come in original and corresponding pairs are *mapped variables*
- OpenMP reference counts mapped variables in the device data environment
- Provides mechanism for keeping track of mapped variables
- Data transfers are elided if safe (based on the count)

```
int N = 10000;
double *A = malloc( sizeof (double) * N); init(A, N);

#pragma omp target data map(tofrom: A[0:N])
{
    #pragma omp target map(to: A[0:N])
    {
        // Use A
    }
}
```

A[0:N] is mapped to the device data environment for the first time. Space is allocated, reference count incremented to 1. Because count=1 AND map(to), a copy occurs

Target region ends, A[0:N] count decremented. count=1

A[0:N] is mapped again. Reference count incremented to 2. NO COPY! (count != 1)

Target data region ending, A[0:N] count is 1, AND map(from), so a copy occurs. Count decremented, count = 0, so storage deallocated.

Reference counting rules

- On entry:
  - If first time, allocate storage, set count=0
  - Increment count
  - If count==1 and map-type is to or tofrom, copy contents of original variable to corresponding variable
- On exit:
  - If count==1 and map-type if from or tofrom, copy contents of corresponding variable to original variable
  - Decrement count
  - If count==0, deallocate storage

Map-type modifiers (OpenMP 5.x)

- Sometimes we want to ensure a data transfer happens
- map(*map-type-modifier*, *map-type*: list)
- Map-type is to/from/tofrom/etc
- Map-type-modifier is always/close/present
  - Always modifier ensures that data is transferred according to map-type no matter the reference count
  - Present will check the variable is mapped, and if not, gracefully terminate
  - Close means … close (basically implementation defined)

```
#pragma omp target data map(alloc: A[0:N])
{
    #pragma omp target map(always, tofrom: A[0:N])
    { // Use A }

    host_update(A);

    #pragma omp target map(always, tofrom: A[0:N])
    { // Use A }
}
```

Without the always modifier, the program could be incorrect

Back to the 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}

    #pragma omp ??????????
    {do something on the host (with A,B,C)

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

Need a way to move data here
Map(always, …) is one solution…
Alternative: target update directive

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.

Set up the data region ahead of time.

```
#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}
}
```

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 directive

- Target update directive makes mapped variables consistent irrespective of the reference count
- to() clause: the corresponding variable takes on the value of the original variable.
  - i.e., a host TO device copy

- from() clause: the original variable takes on the value of the corresponding variable

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 omp 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)
  - **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 nowait depend(inout: A)
#pragma omp loop
   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.

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

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

Agenda

Morning

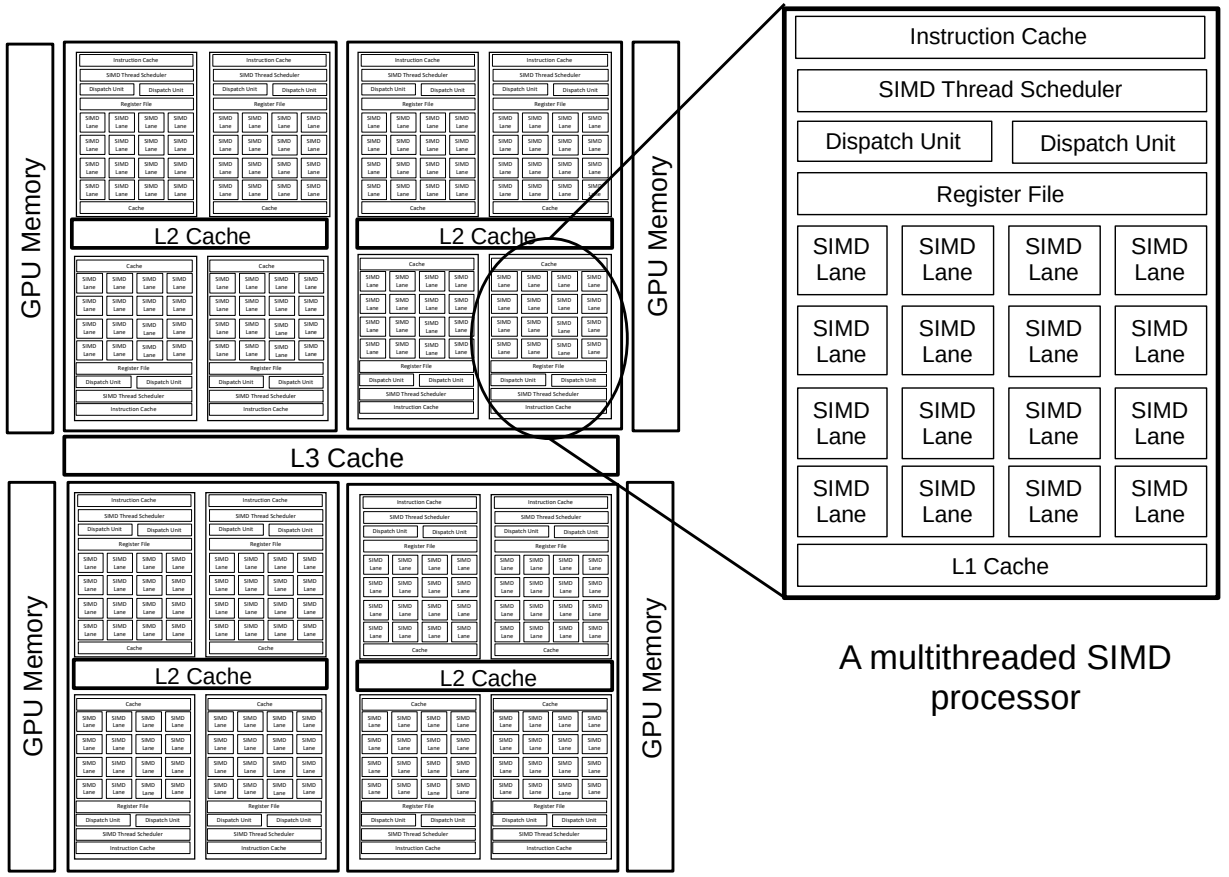
- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

Afternoon

- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

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.



A multithreaded SIMD processor

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

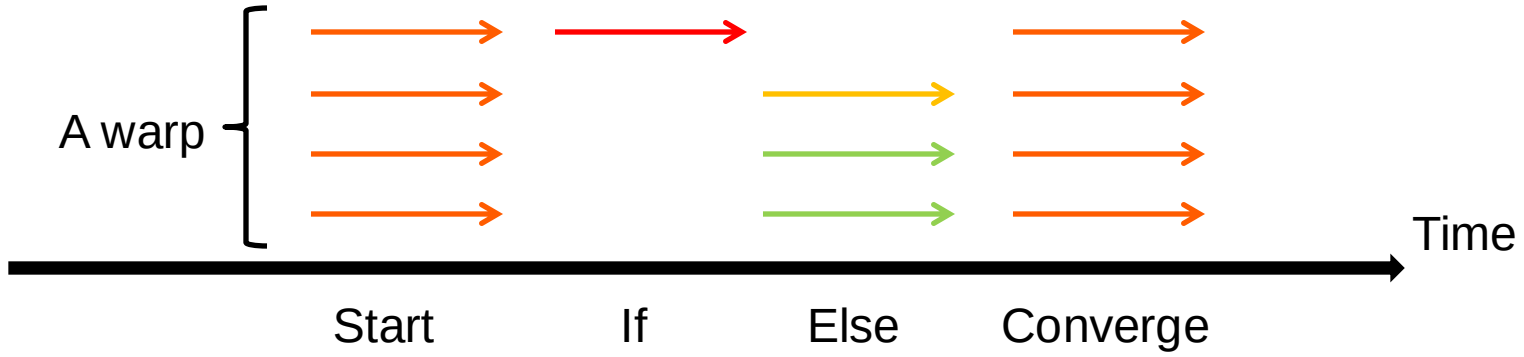
Parallelize i-loop parallelism O(N)

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

Parallelize combined i/j-loops parallelism O(N^2)

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, potentially improving the performance of code that has lots of conditional branches inside the loops

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, this is very simple, but often requires transposing or transforming data on the host before sending it to the GPU

- Sometimes this is about **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;
```

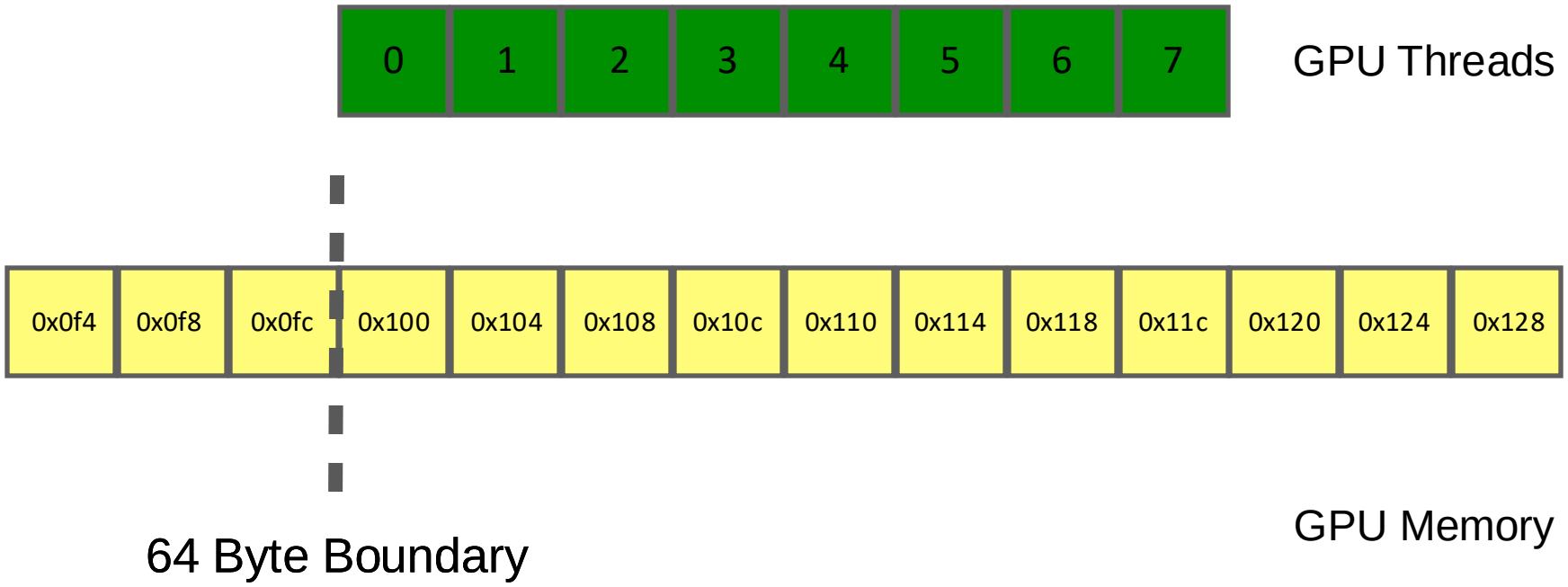
Adjacent work-items/vector-lanes like to access adjacent memory locations

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…

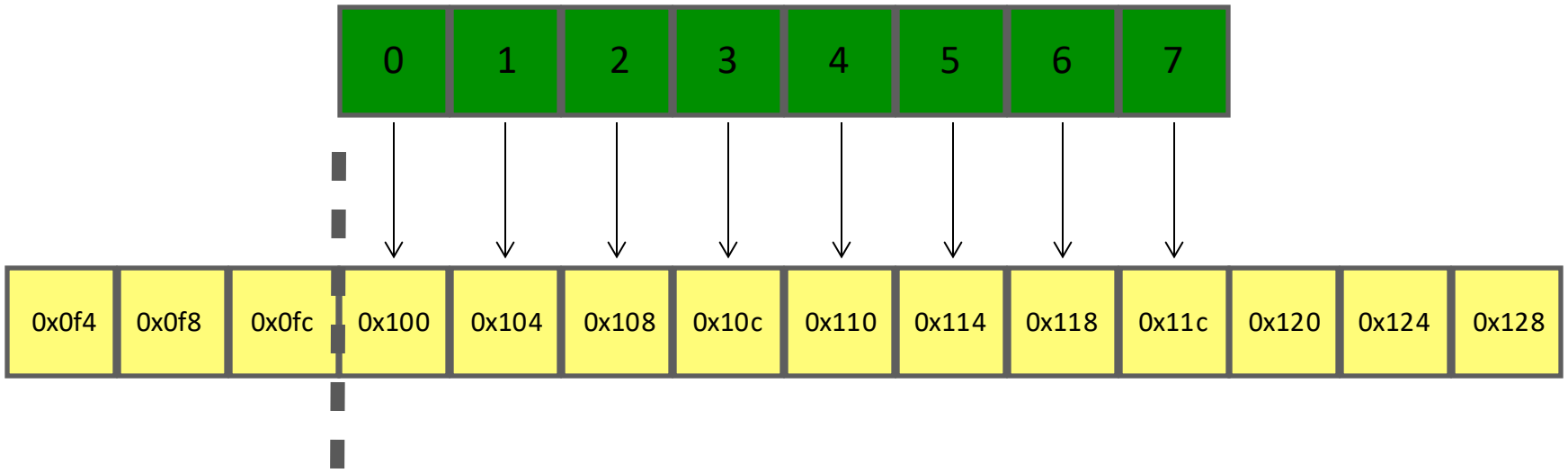
```
for (int id = 0; id < size; id++)
{
  // ideal
     float val1 = memA[id];

  // 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

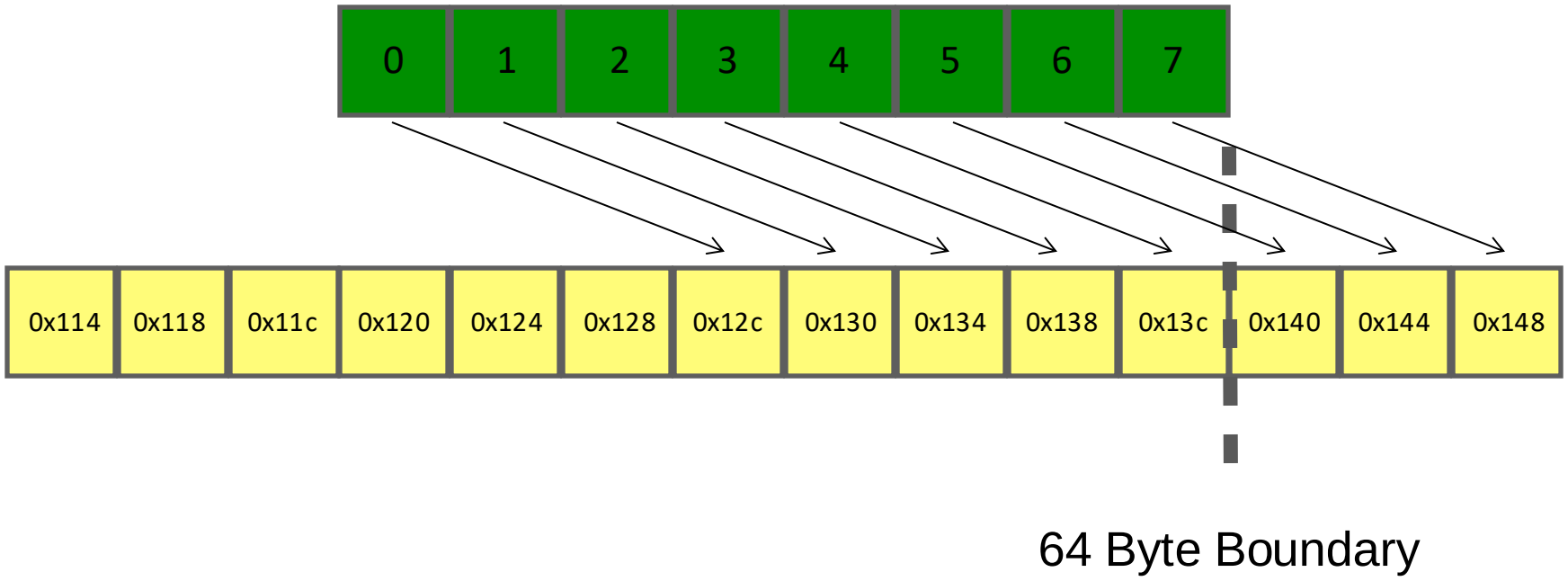
```
float val1 = memA[id];
```



64 Byte Boundary

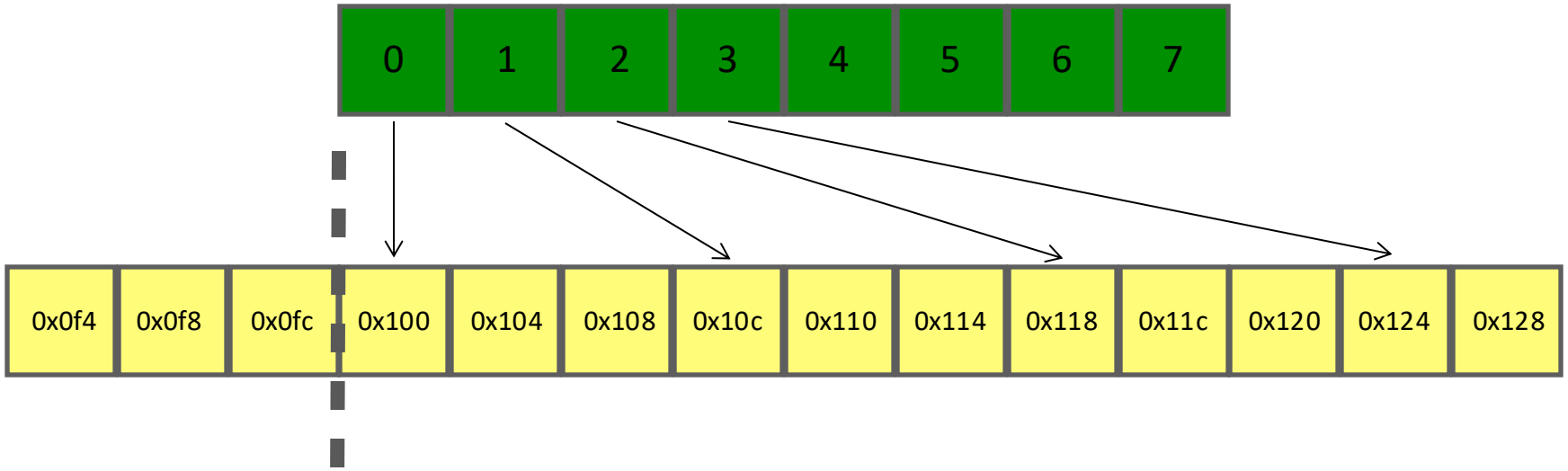
Memory access patterns

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



Memory access patterns

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

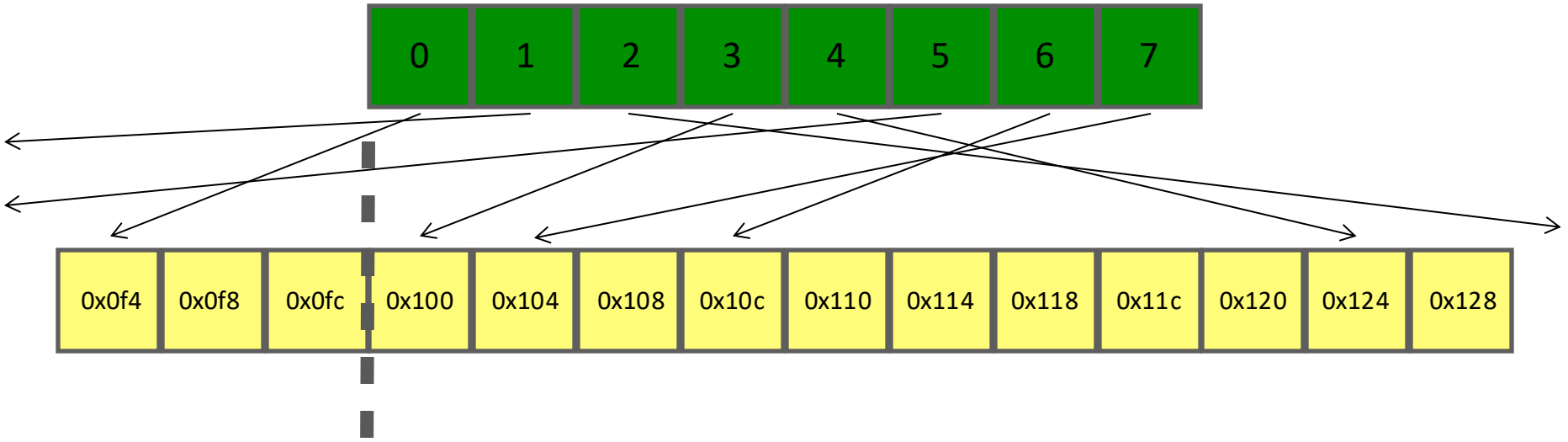


64 Byte Boundary

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

Live exercise 4

Optimising stencil data movement

Optimising stencil kernels

Exercise

Files: heat.c

https://tinyurl.com/sc24ompgpu

ssh train*XX*@ 44.204.72.120

openmp24

- 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 loop

- Then Optimize the stencil ‘solve’ kernel.

- **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: 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])

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

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

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

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

Copy data to device before iteration loop

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

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.

Copy data from device after iteration loop

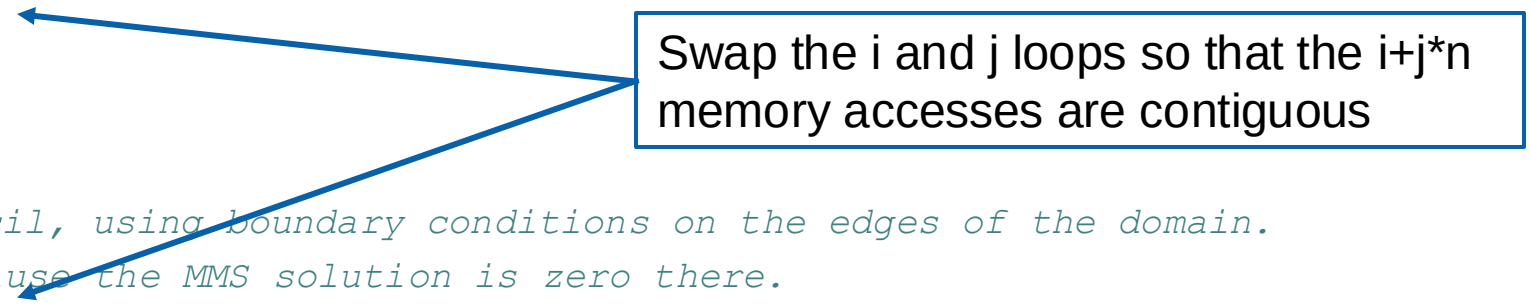
NVPROF output

```
------------------
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  __omp_offloading_...
                   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]
```

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 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);
}}}
```



Agenda

Morning

- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

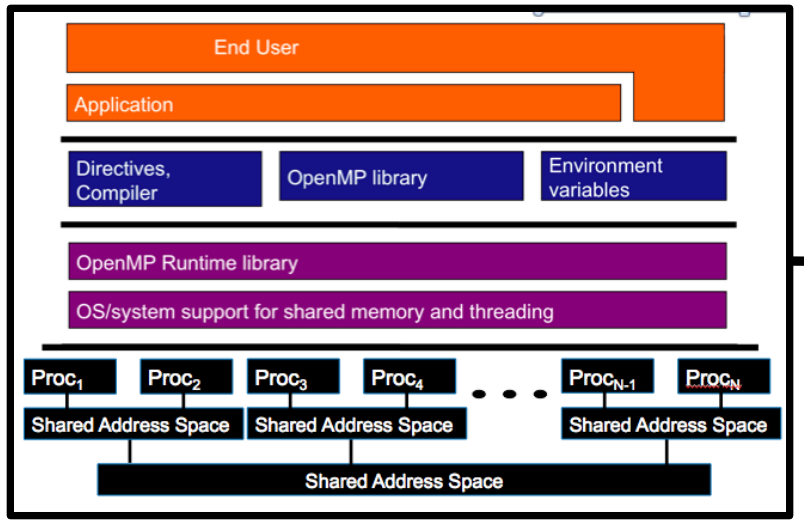
Afternoon

- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

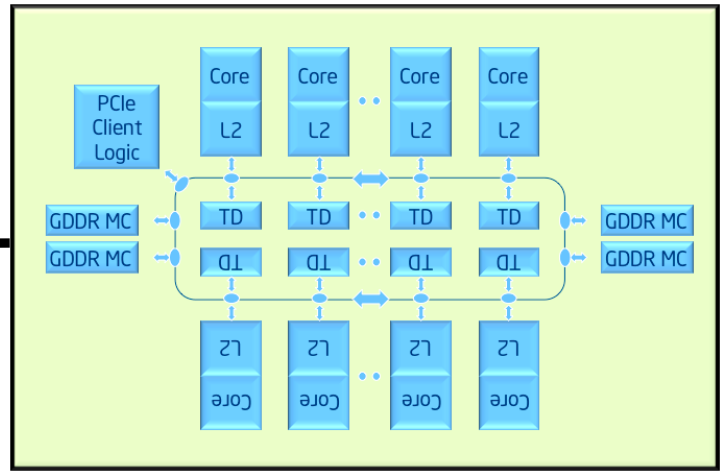
The loop construct is great, but sometimes you want more control.

OpenMP device model: Examples

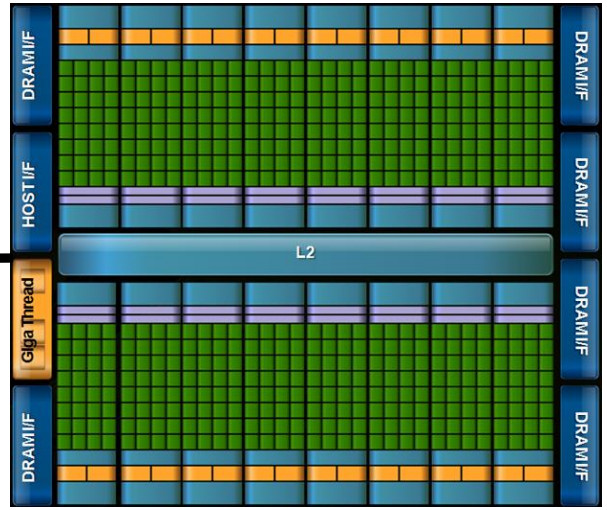
Some key devices that were considered when designing the device model in OpenMP



Host

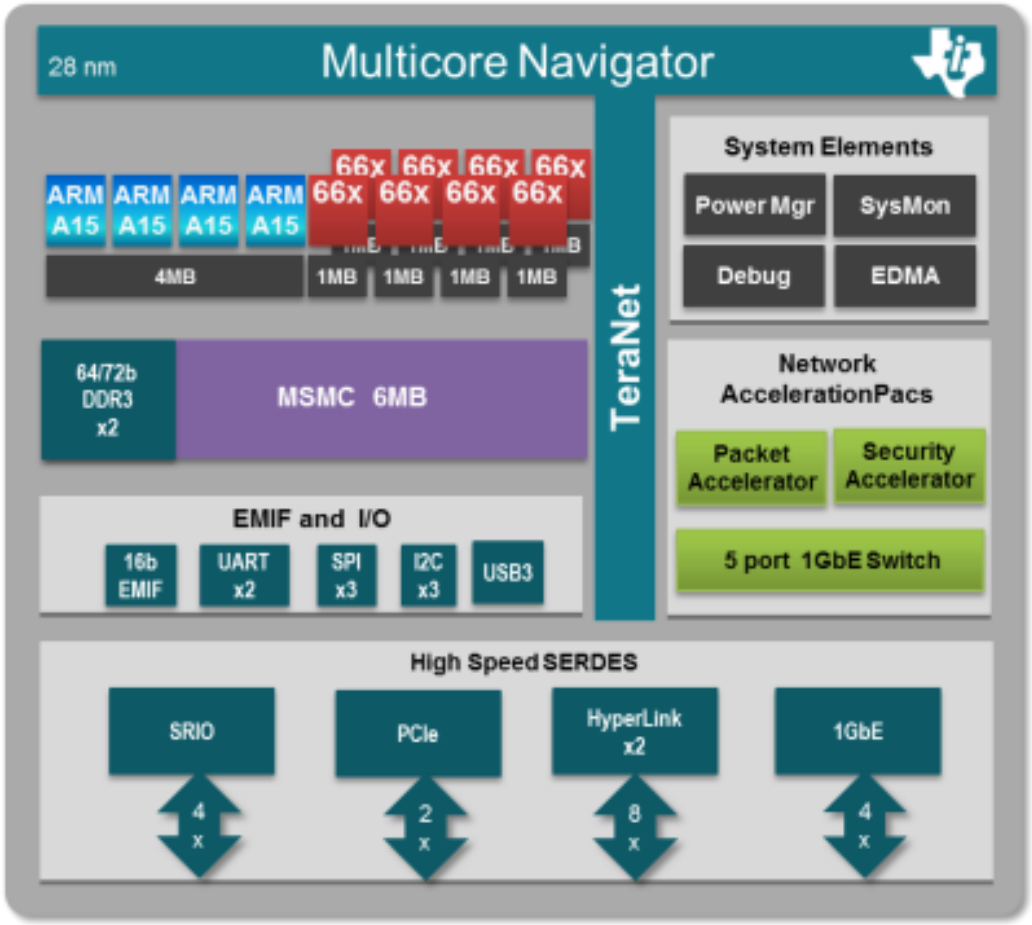


Target Device: Intel® Xeon Phi™ processor



Target Device: GPU

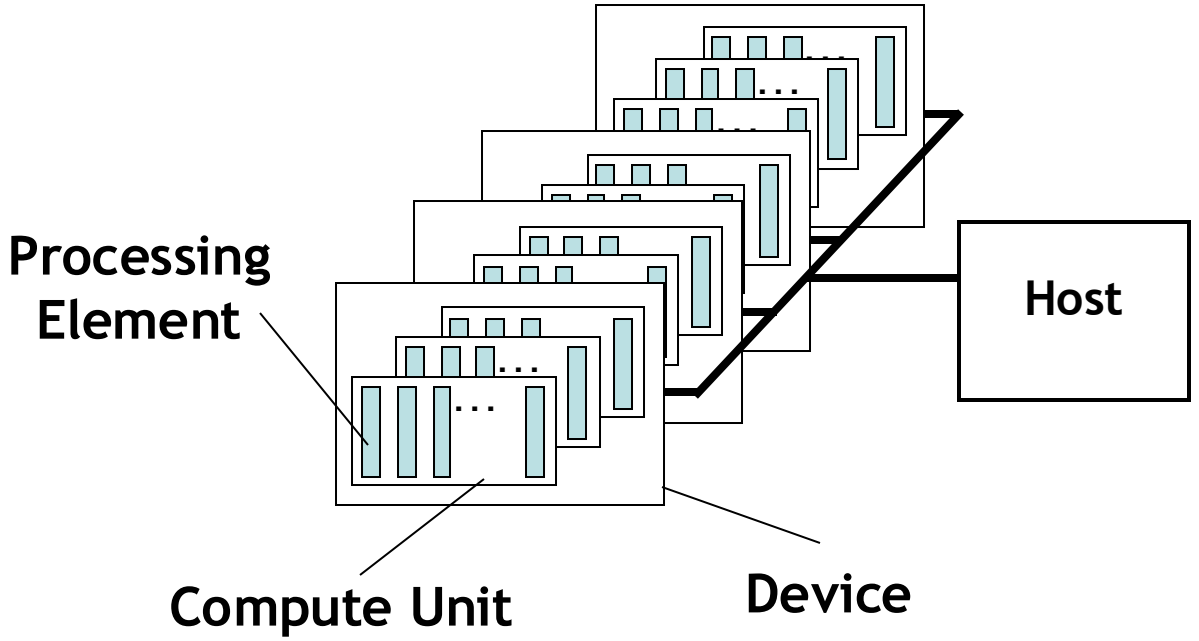
OpenMP Device Model: another example



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

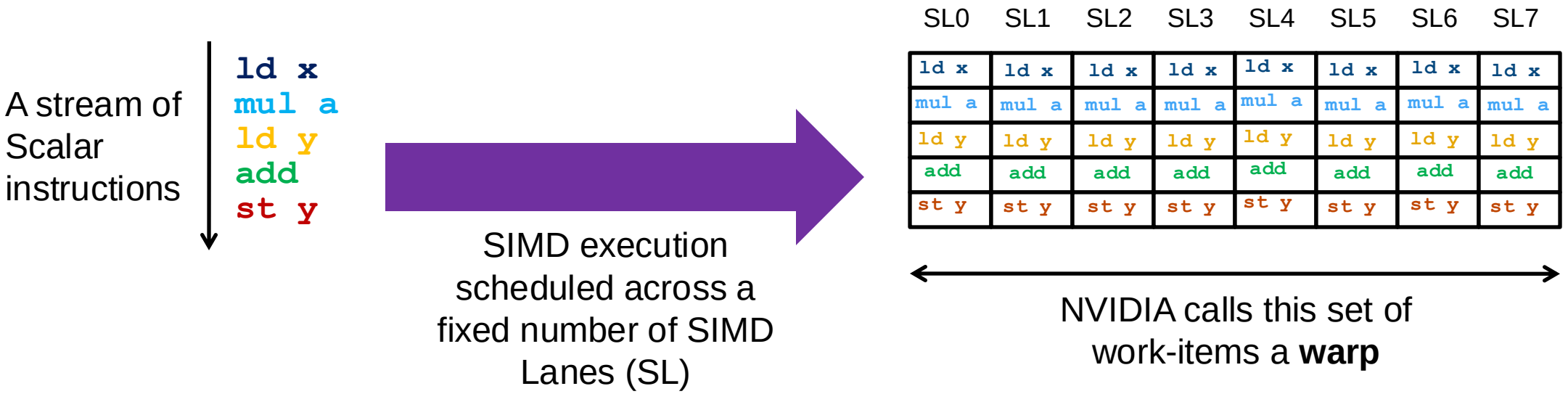
- GPUs are made of many *cores (compute units)*
  - *NVIDIA V100 has 80 Streaming Multiprocessors (SMs); these are the compute units*
  - *NVIDIA A100 has 108 compute units*
  - Each NVIDIA compute unit has 64 FP32 processing elements
  - GPUs from AMD and Intel have similar structure of compute units and processing elements
- On an A100, that's 108 x 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

GPU terminology is Broken (sorry about that)

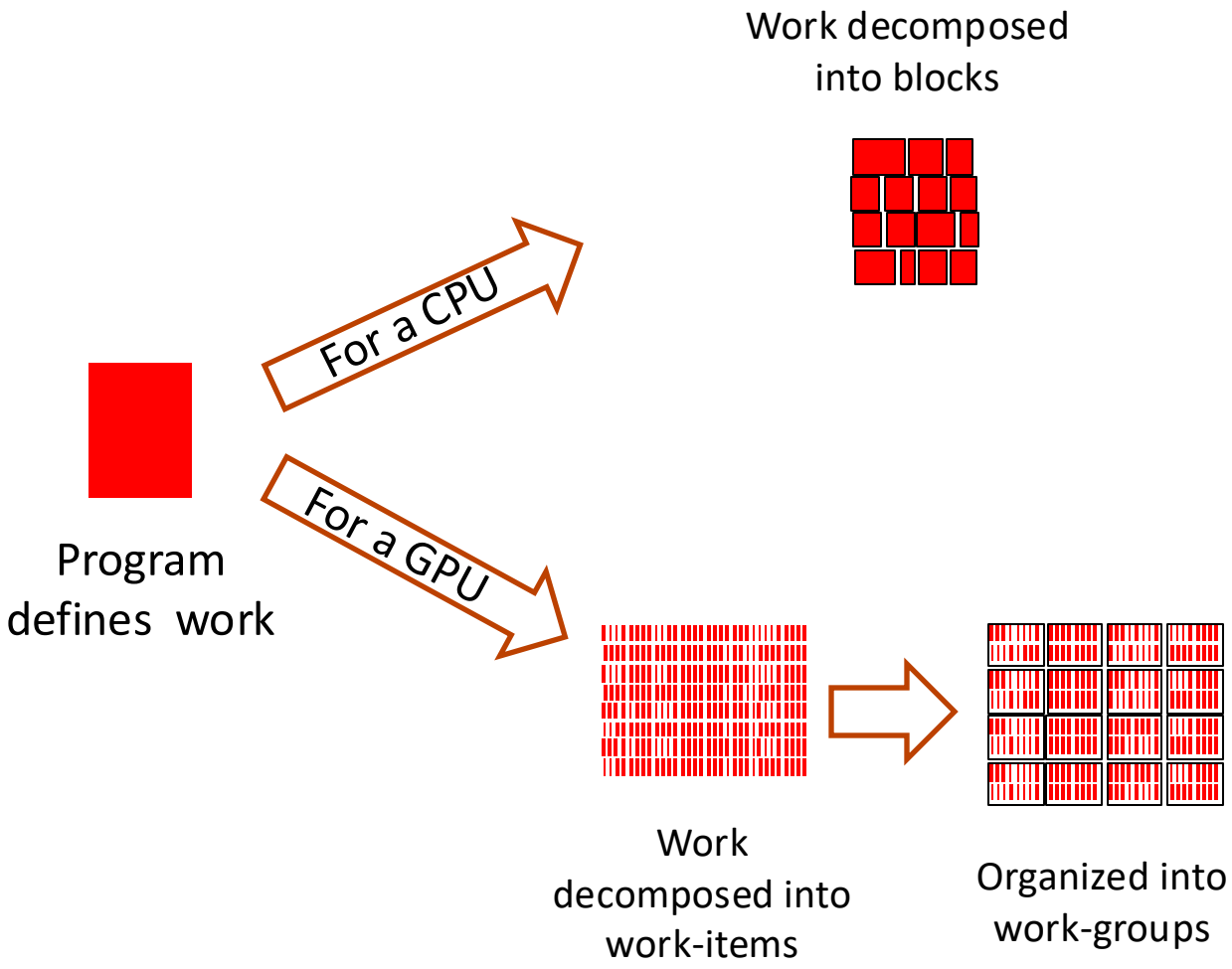
Hennessy and Patterson	CUDA	OpenCL
Multithreaded SIMD Processor	Streaming multiprocessor	Compute Unit
SIMD Thead Scheduler	Warp Scheduler	Work-group scheduler
SIMD Lane	CUDA Core	Processing Element
GPU Memory	Global Memory	Global Memory
Private Memory	Local Memory	Private Memory
Local Memory	Shared Memory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

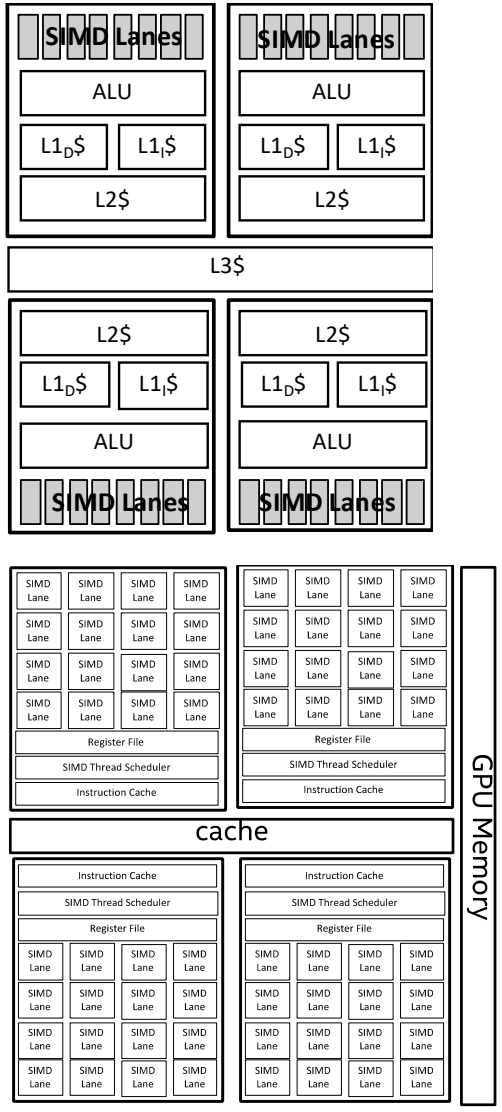
SIMT: Single Instruction, Multiple Thread

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



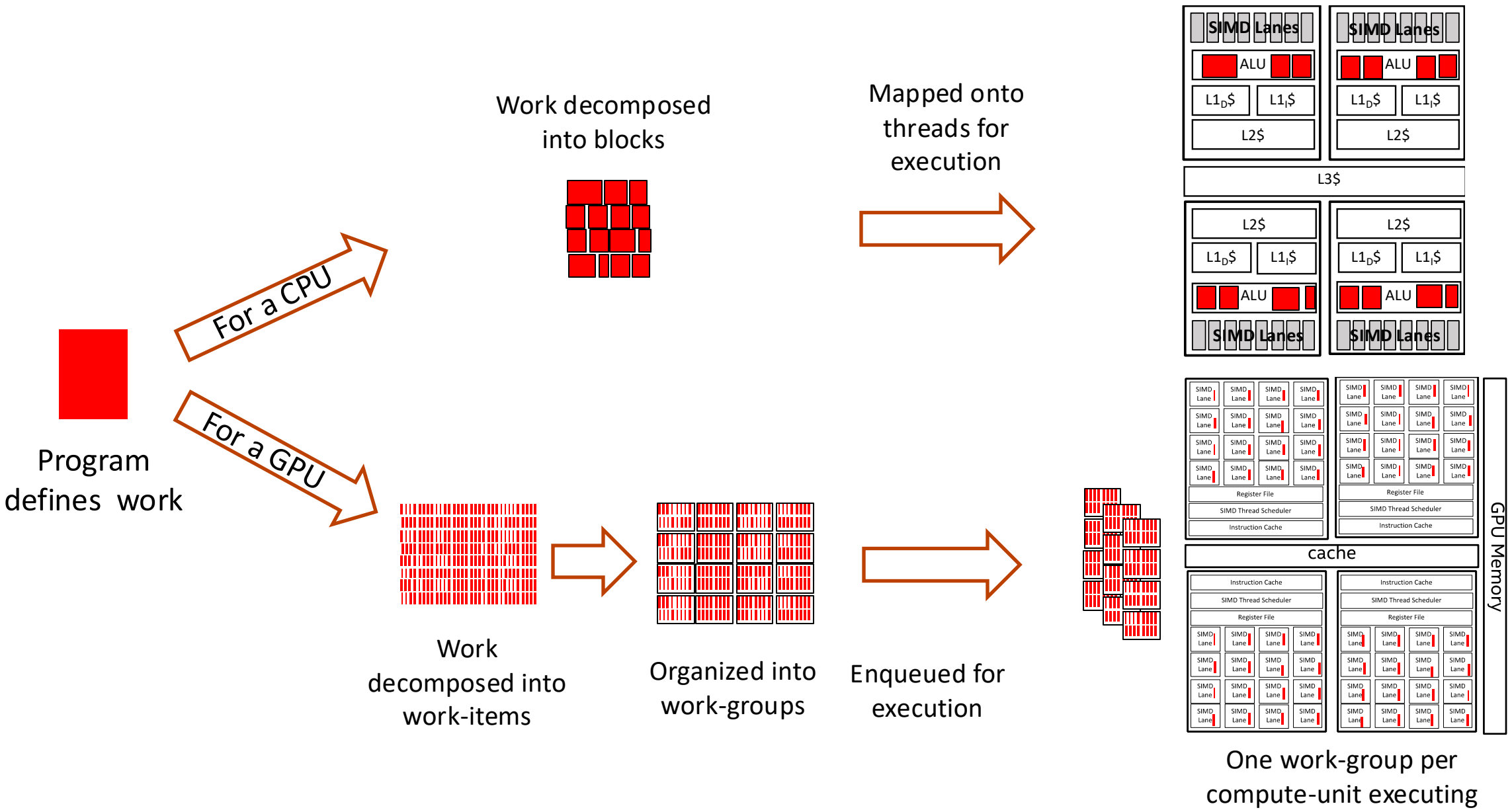
Executing a program on CPUs and GPUs



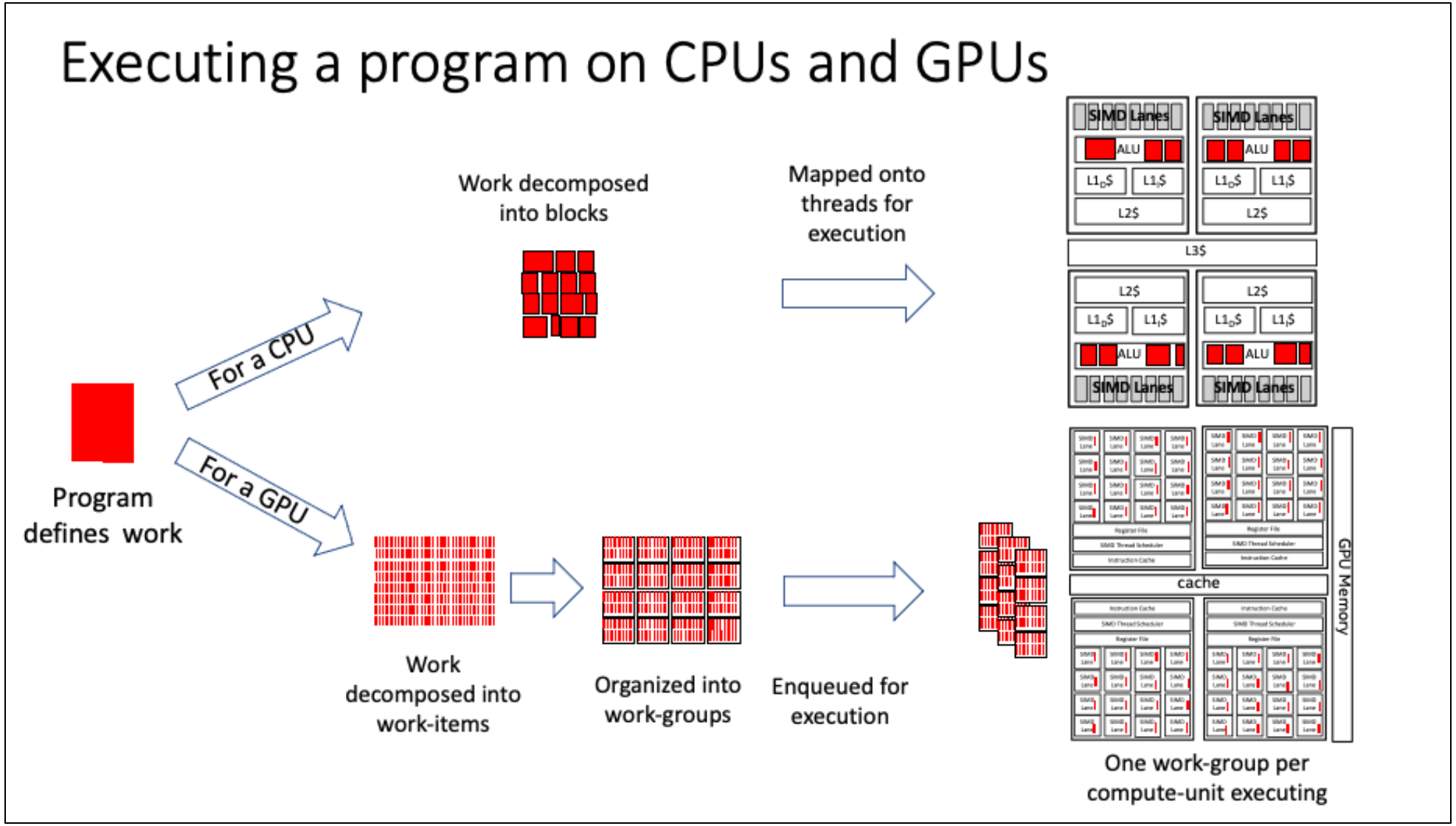


One work-group per compute-unit executing

Executing a program on CPUs and GPUs



CPU/GPU execution model



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.

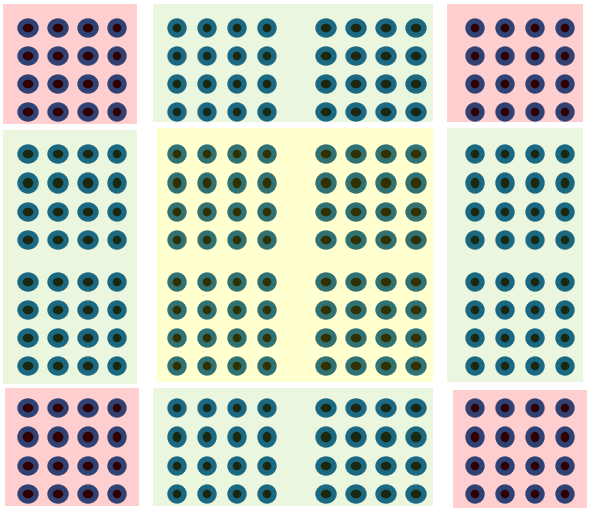
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

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[local_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



3. Map data structures onto the same index space

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

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

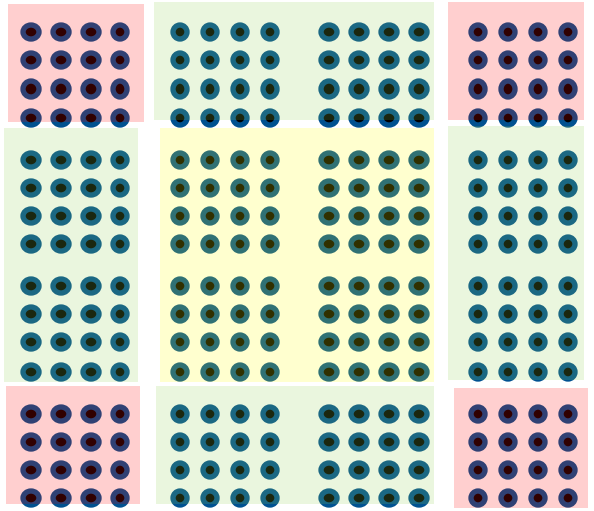
Turn source code into a scalar **work-item** (a CUDA **thread**)

```
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[local_id] = accum;
  barrier(CLK_LOCAL_MEM_FENCE);
  reduce(l_sums, p_sums);
}
```

This code defines a **kernel**

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

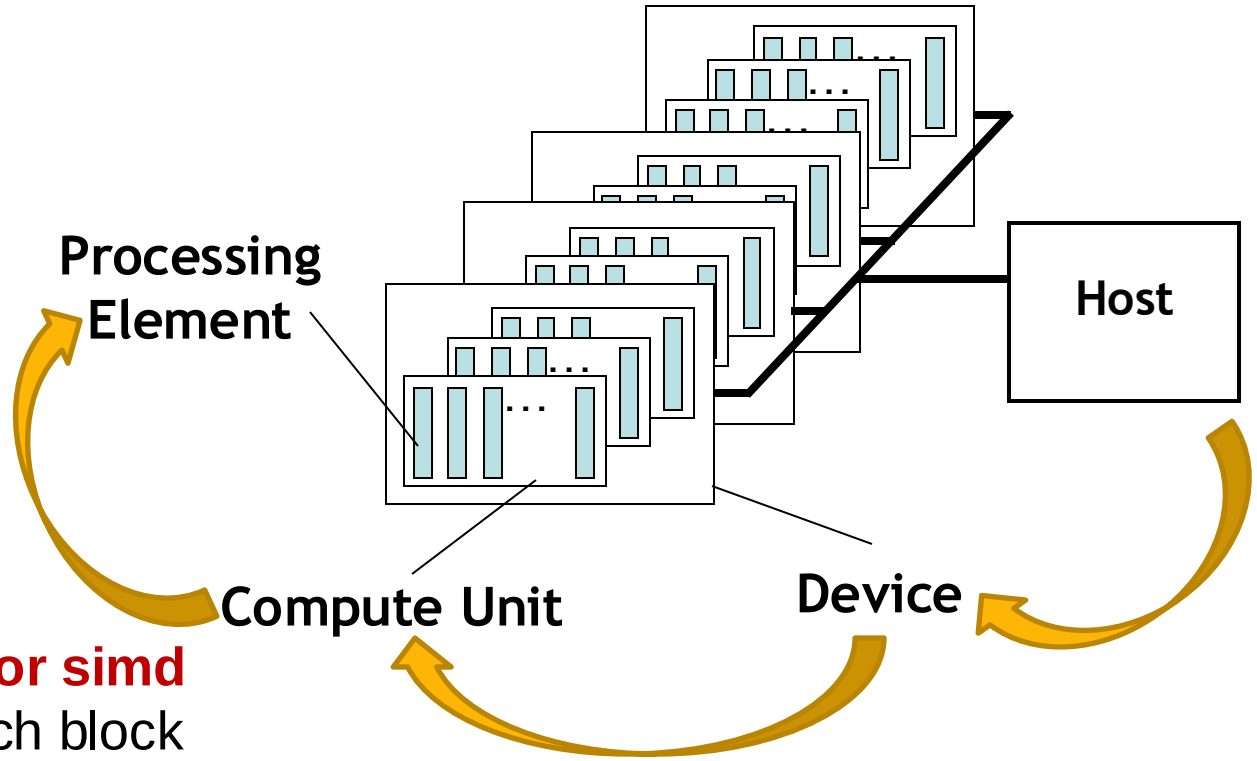
Organize work-items into **work-groups** and map onto an N dimensional index space. CUDA calls a work-group a **thread-block**



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

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

Our host/device Platform Model and OpenMP



target construct to get onto a device

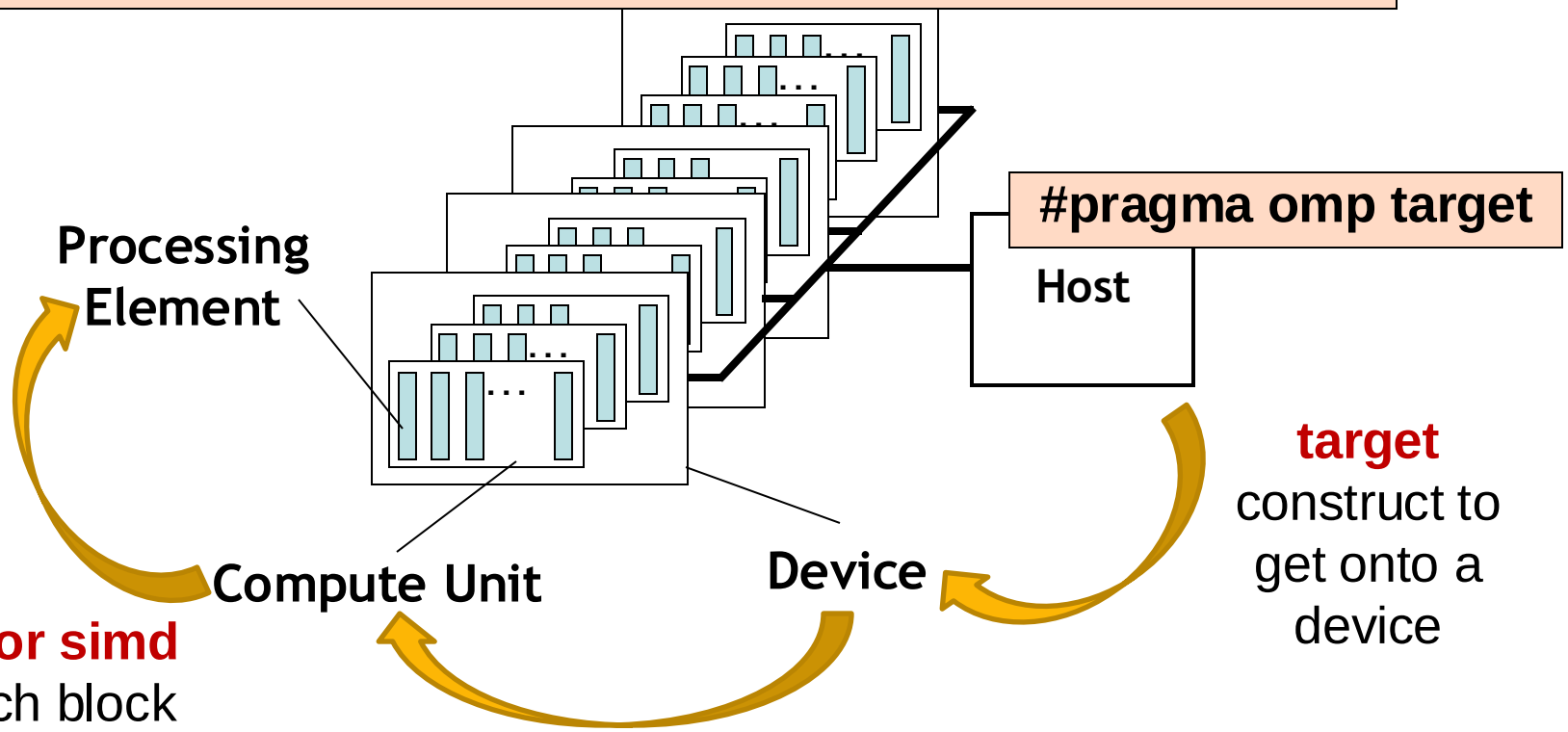
parallel for simd to run each block of loop iterations on the processing elements

teams construct to create a league of teams with one team of threads on each compute unit

distribute construct to assign blocks of loop iterations to teams

Our host/device Platform Model and OpenMP

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



target construct to get onto a device

parallel for simd to run each block of loop iterations on the pr... elem...

teams construct to create a league

#pragma omp teams distribute parallel for simd
to assign work to the device processing elements

distribute construct to assign blocks of loop iterations to teams

Implementation details

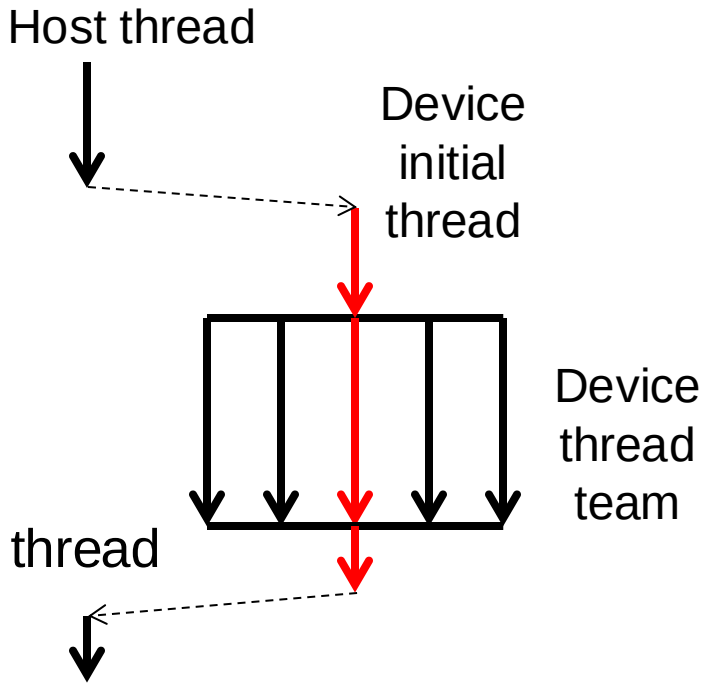
- OpenMP defines parallelism abstraction
  - Specific terminology is used
- An OpenMP implementation (runtime/compiler) has some freedom in how these are applied to hardware
  - Allows the implementation to make sensible choices to get the best performance
- OpenMP directives operate along spectrum of 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

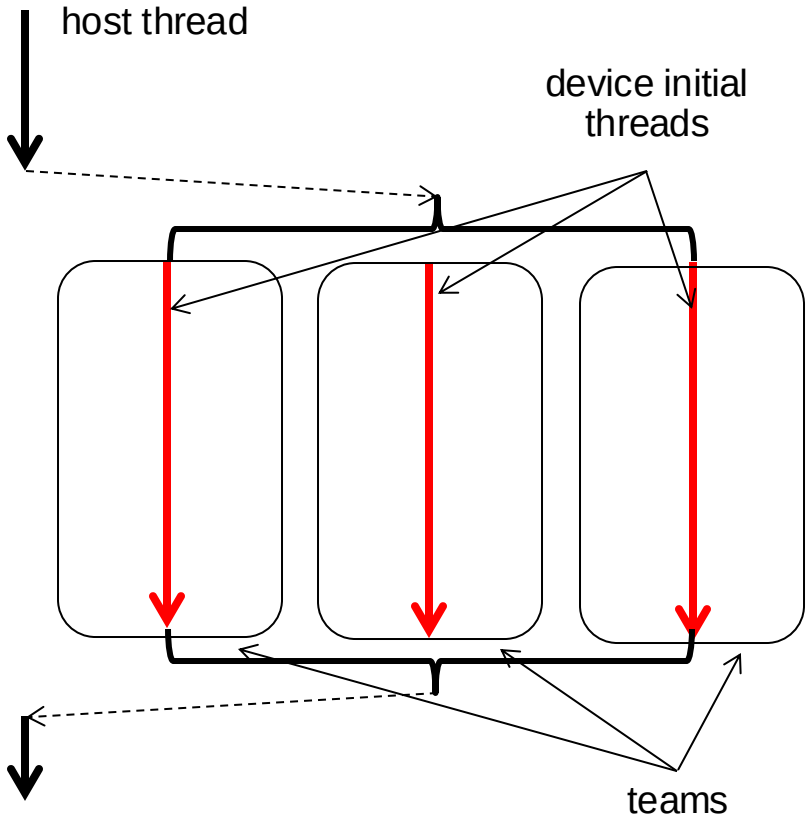
- The **teams** construct
  - Similar to the **parallel** construct
  - It starts a league of *teams*
  - Each team in the league starts with one initial thread – i.e. a team of one thread
  - Threads in different teams **cannot** synchronize with each other
  - The construct must be “perfectly” nested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



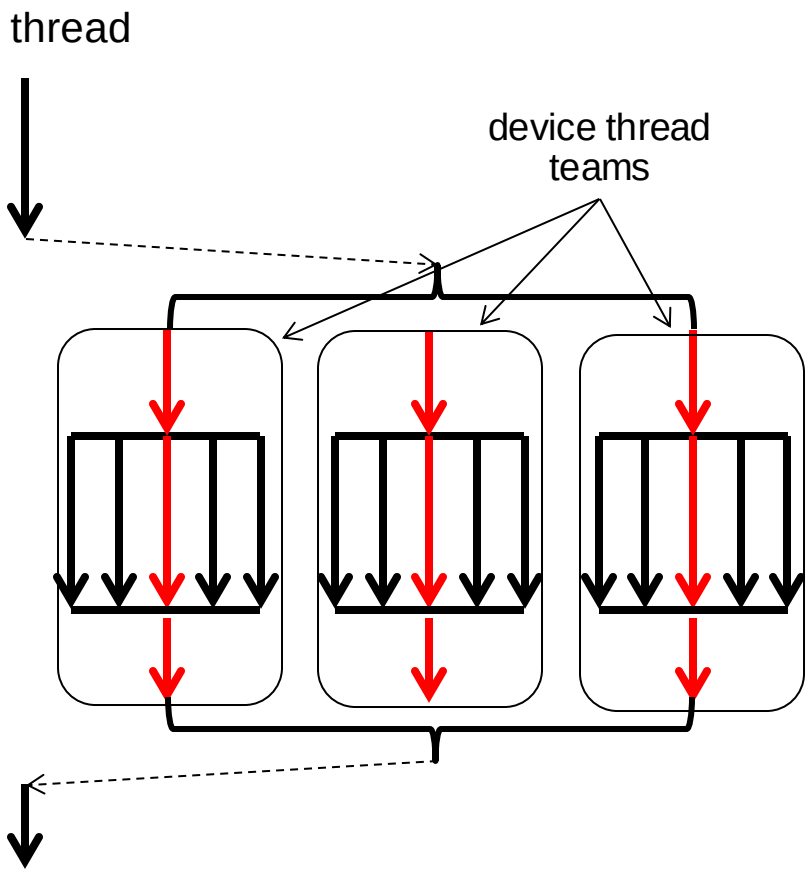
- Transfer execution control to MULTIPLE device initial threads
- Workshare loop iterations across the initial threads.

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

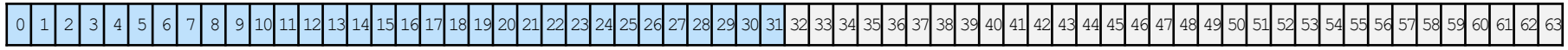
```
#pragma omp target teams
{
      #pragma omp distribute parallel for simd
      for (int i = 0; i < n; i++) {
            F(i) = G(i);
      }
}
```

Worksharing example

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

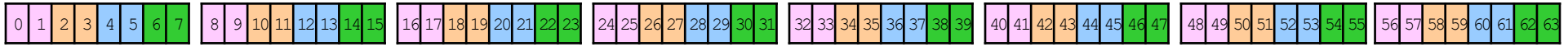
64 iterations assigned to 2 teams;
Each team has 4 threads;
Each thread has 2 SIMD lanes

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***

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:
   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
```

Shows block sizes, grid dimensions and register counts for kernels

Entries ordered by time

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 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.

Agenda

Morning

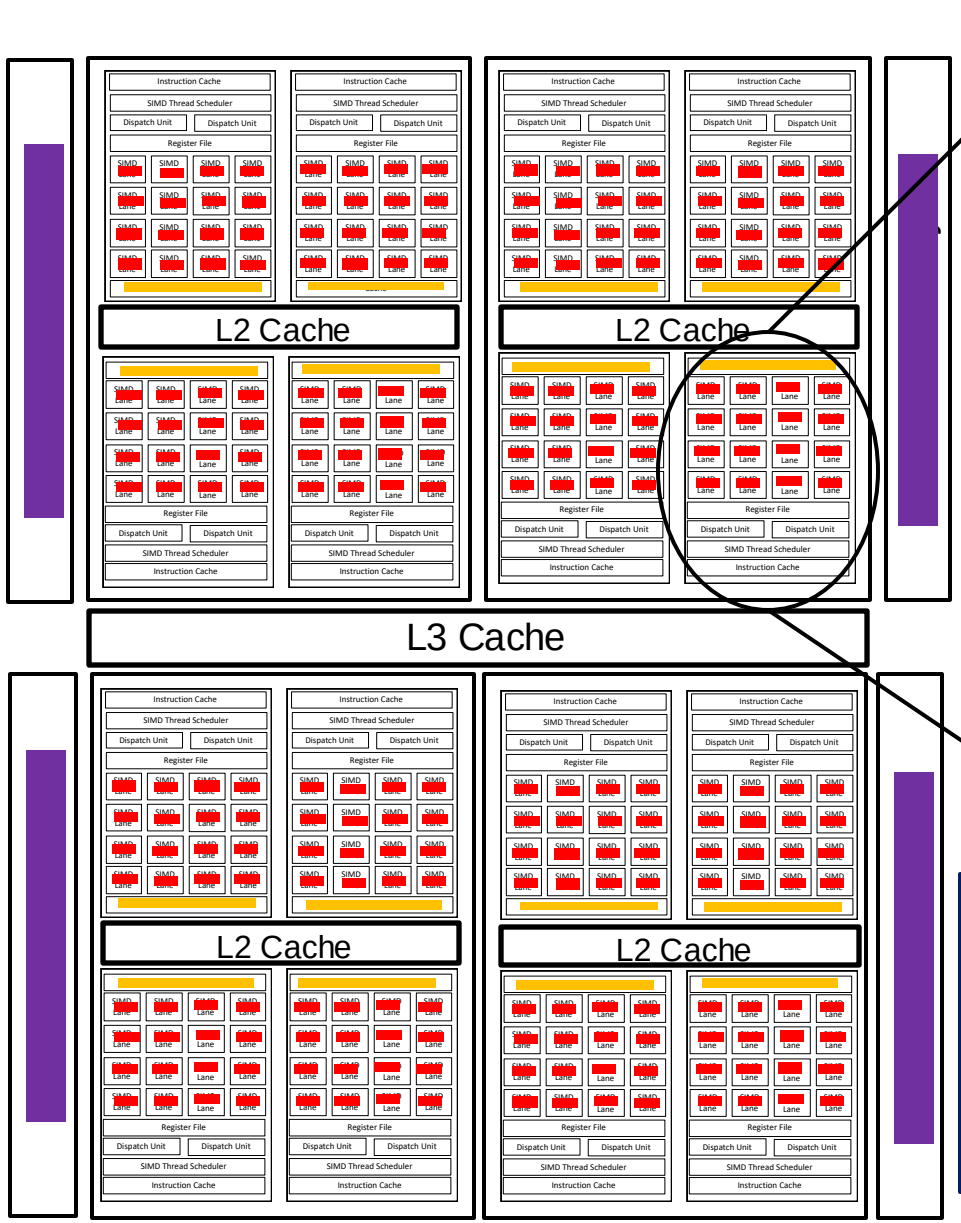
- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

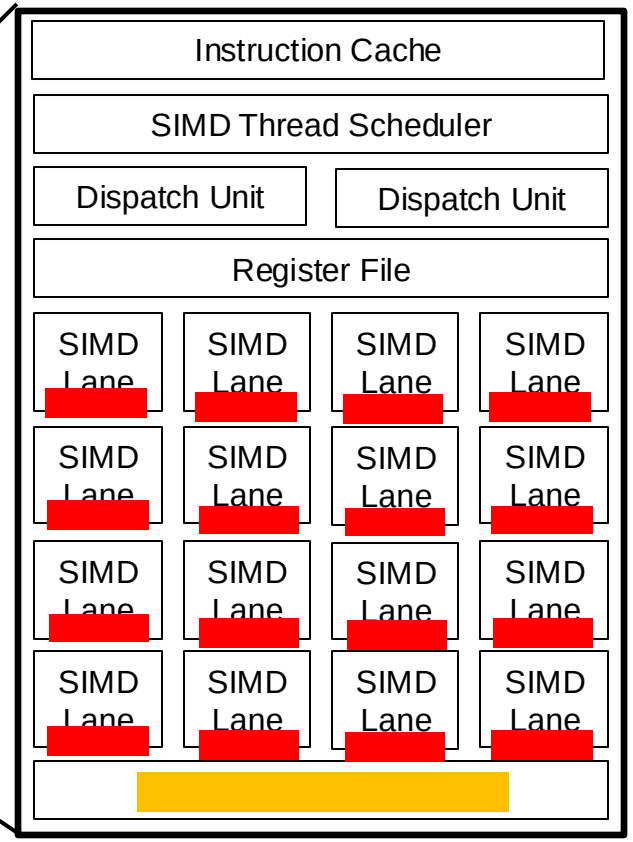
Afternoon

- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

Back to the Generic GPU (following Hennessey and Patterson)

Each multithreaded SIMD processor typically has some programable scratchpad memory





Logical Memory Hierarchy

- Private Memory (work-item)
- Local Memory (work-group)
- Global Memory (kernel)

OpenMP memory spaces and allocators

- OpenMP provides control of **where** data is allocated using **allocators**
- Originally designed for systems with deeper memory hierarchy than just “main memory”
  - E.g., Combination of DDR and HBM like on Intel Xeon Phi or Intel Sapphire Rapids
  - omp_high_bw_mem_alloc allocates in the high bandwidth memory space
  - omp_low_lat_mem_alloc allocates in the low latency memory space
  - The system and OpenMP implementation define what those spaces mean on that platform
  - These pre-defined allocators used in omp_alloc() library call and allocate() clause

OpenMP allocators for team-only memory

- omp_pteam_mem_alloc
  - Data allocated in this memory space is available to threads in the team where one of those threads allocated the data
- omp_cgroup_mem_alloc
  - Data allocated in this memory space is available to threads in the team where one of those threads allocated the data, AND, and child threads
  - OpenMP threads that satisfy that at said to be in a *contention group*

- On current implementations of OpenMP targeting GPUs, these target the same local/shared memory to each team

Putting data in team-only memory

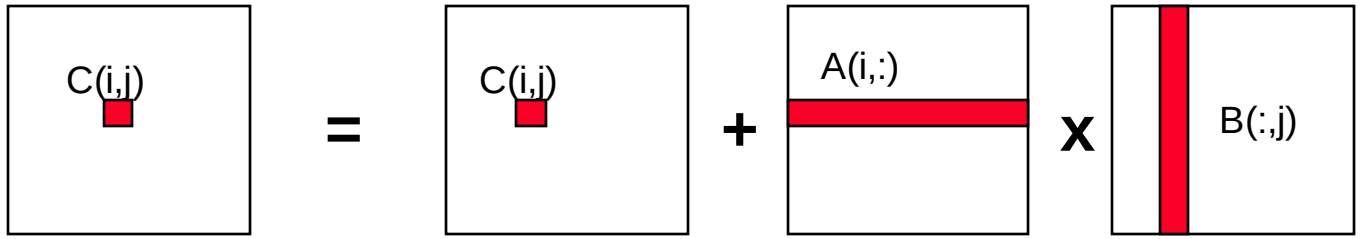
- Three step process:

1. Add the uses_allocators(omp_pteam_mem_alloc) clause to the target construct
2. Privatize the variables with the private() clause on the teams construct
3. Use the allocate() clause to select the allocator used for teams-private variables

```
double arr[32];

#pragma omp target uses_allocators(omp_pteam_mem_alloc)
#pragma omp teams allocate(omp_pteam_mem_alloc: arr) private(arr)
#pragma omp distribute
for (int i = 0; i < N; ++i) {
  #pragma omp parallel for
  for (int j = 0; j < M; ++j) {
    arr[i] = 1.0;
  }
}
```

Optimizing matrix multiplication

- Matrix Multiplication cost determined by FLOPS and memory movement:
  - $2*n^3 = O(n^3)$ FLOPS
  - Operates on $3*n^2 = O(n^2)$ numbers
- To optimize matrix multiplication, we must ensure that for every memory access we execute as many FLOPS as possible.
- Outer product algorithms are faster, but for pedagogical reasons, let's stick to the simple dot-product algorithm.



Dot product of a row of A and a column of B for each element of C

- If we tile/block the loops, we can improve the data reuse

From HandsOnOpenCL

Matrix multiplication

```
for (int i = 0; i < Ndim; ++i) {
  for (int j = 0; j < Mdim; ++j) {
    for (int k = 0; k < Pdim; ++k) {
      C[i * Mdim + j] += A[i * Pdim + k] * B[k * Mdim + j];
  }
}
```

Blocked matrix multiplication

```
#define BSIZE 32
int Nblk = Ndim / BSIZE; // Num. blocks
int Mblk = Mdim / BSIZE;
int Pblk = Pdim / BSIZE;

for (int ib = 0; ib < Nblk; ++ib) { // Loop over blocks of C
  for (int jb = 0; jb < Mblk; ++jb) {
    for (int kb = 0; kb < Pblk; ++kb) { // blocks of rows of A

      for (int i = ib*Bsize; i < (ib+1)*Bsize; ++i) { // Mat mul of a block
        for (int j = jb*Bsize; j < (jb+1)*Bsize; ++j) {
          for (int k = kb*Bsize; k < (kb+1)*Bsize; ++k) {
            C[i * Mdim + j] += A[i * Pdim + k] * B[k * Mdim + j];
}}}}}}
```

Live exercise 6

Optimising matrix multiply

Exercise

Files: mm_*.c

https://tinyurl.com/sc24ompgpu

ssh trainXX@ 44.204.72.120

openmp24

- Optimize the matrix multiplication kernel by tiling the loop.
- Make sure you control the number of threads and teams!

- Allocate blocks of the matrix in team-only memory using the allocate(), uses_allocators() and private() clauses.

- Try different input sizes to see the effect of the optimizations.

- Keep an eye on the FLOP/s rate as reported by the application.

- Note: You need to use the LLVM compiler which supports the allocate() directive.
  - module load llvm-main (NOT llvm)
  - cp Make_def_files/linux_llvm.sh make.def

Solution (from the book!)

```
#define BSIZE 32
int Nblk = Ndim / BSIZE; // Num. blocks
int Mblk = Mdim / BSIZE;
int Pblk = Pdim / BSIZE;

double Ablk[Bsize*Bsize];
double Bblk[Bsize*Bsize];

#pragma omp target uses_allocators(omp_pteam_mem_alloc)
#pragma omp teams distribute collapse(2) \
    num_teams(Nblk*Mblk) thread_limit(Bsize*Bsize) \
    allocate(omp_pteam_mem_alloc: Awrk, Bwrk) private(Awrk, Bwrk)
for (int ib = 0; ib < Nblk; ++ib) { // Loop over blocks of C
  for (int jb = 0; jb < Mblk; ++jb) {
    for (int kb = 0; kb < Pblk; ++kb) { // blocks of rows of A

// continued
      for (int i = ib*Bsize; i < (ib+1)*Bsize; ++i) { // Mat mul of a block
        for (int j = jb*Bsize; j < (jb+1)*Bsize; ++j) {
          for (int k = kb*Bsize; k < (kb+1)*Bsize; ++k) {
            C[i * Mdim + j] += A[i * Pdim + k] * B[k * Mdim + j];
}}}}}}
```

Solution continued (from the book!)

```
#pragma omp parallel num_threads(Bsize*Bsize)
{
  // Copy blocks into pteam memory
  #pragma omp for collapse(2)
  for (int i = ib*Bsize; i < (ib+1)*Bsize; ++i)
    for (int k = kb*Bsize; k < (kb+1)*Bsize; ++k)
      Awrk[(i%Bsize)*Bsize + (k%Bsize)] = A[i*Pdim+k];

  #pragma omp for collapse(2)
  for (int j = jb*Bsize; j < (jb+1)*Bsize; ++j)
    for (int k = kb*Bsize; k < (kb+1)*Bsize; ++k)
      Bwrk[(k%Bsize)*Bsize + (j%Bsize)] = B[k*Mdim+j];

  #pragma omp for collapse(2)
  for (int i = ib*Bsize; i < (ib+1)*Bsize; ++i) { // Mat mul of a block
        for (int j = jb*Bsize; j < (jb+1)*Bsize; ++j) {
          for (int k = kb*Bsize; k < (kb+1)*Bsize; ++k) {
            C[i * Mdim + j] += \
              Awrk[(i%Bsize)*Bsize + (k%Bsize)] *
              Bwrk[(k%Bsize)*Bsize + (j%Bsize)];
}}} } }}} }
```

Agenda

Morning

- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

Afternoon

- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

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?

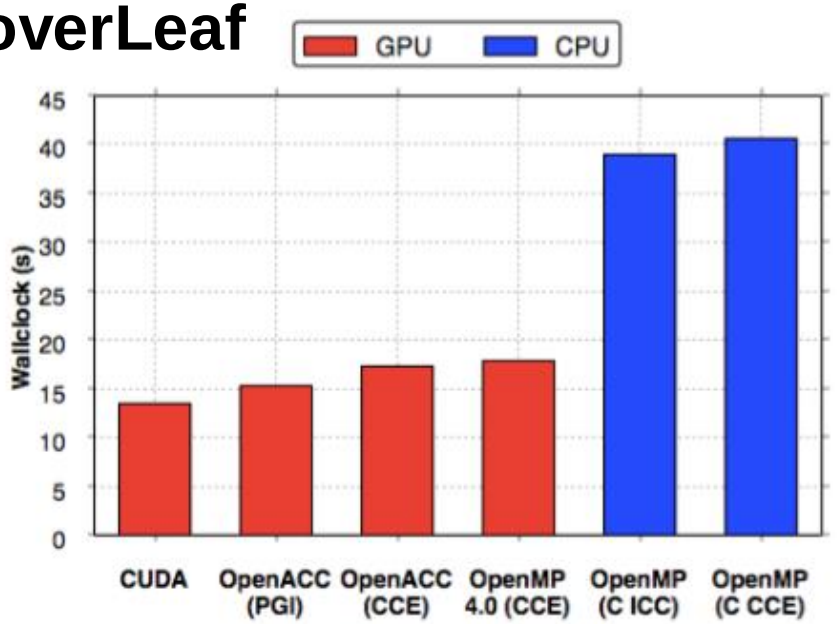
* CCE 8.4.3, ICC 15.0.3, PGI 15.01, CUDA 7.0 on an NVIDIA® K20X, and Intel® Xeon® Haswell 16 Core Processor (E5-2698 v3 @ 2.30GHz)

2016

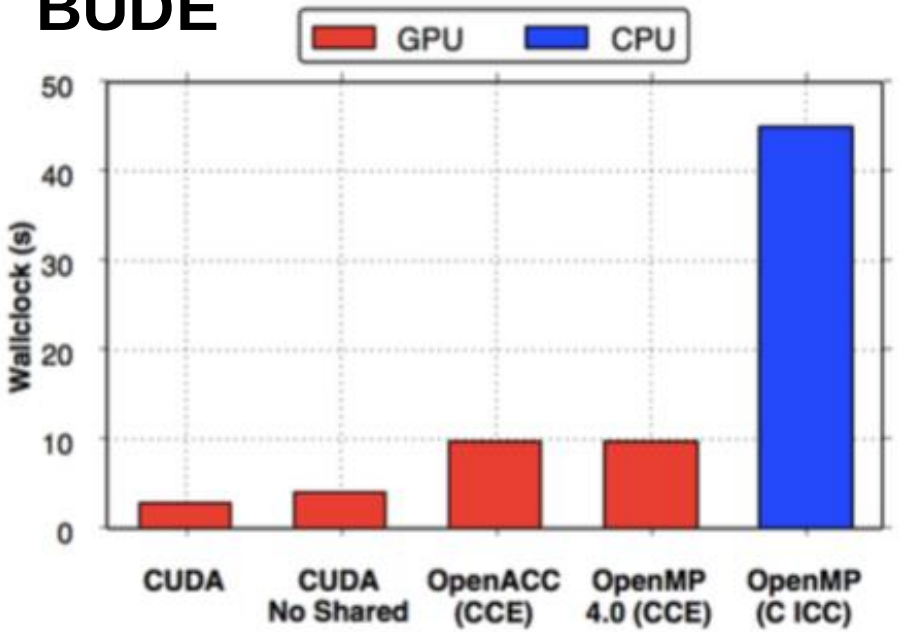
Immediately we see impressive performance compared to CUDA

Clearly the Cray compiler leverages the existing OpenACC backend

CloverLeaf



BUDE



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

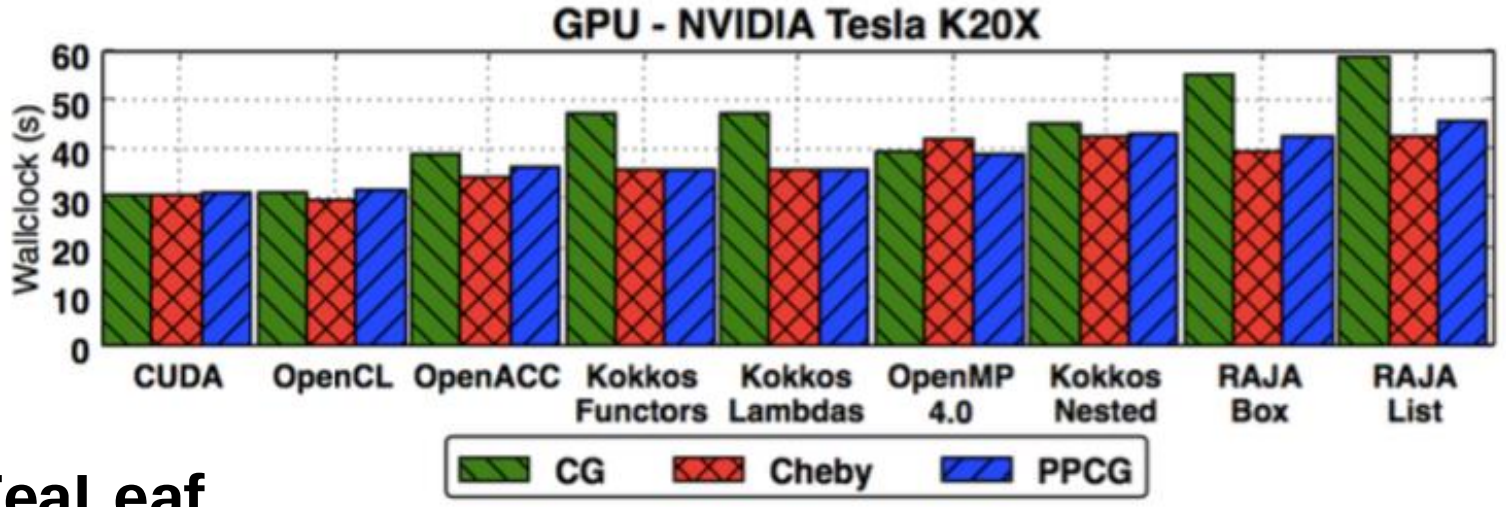
This is set to come in with OpenMP 5.0, and Clang supports targeting address spaces directly

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

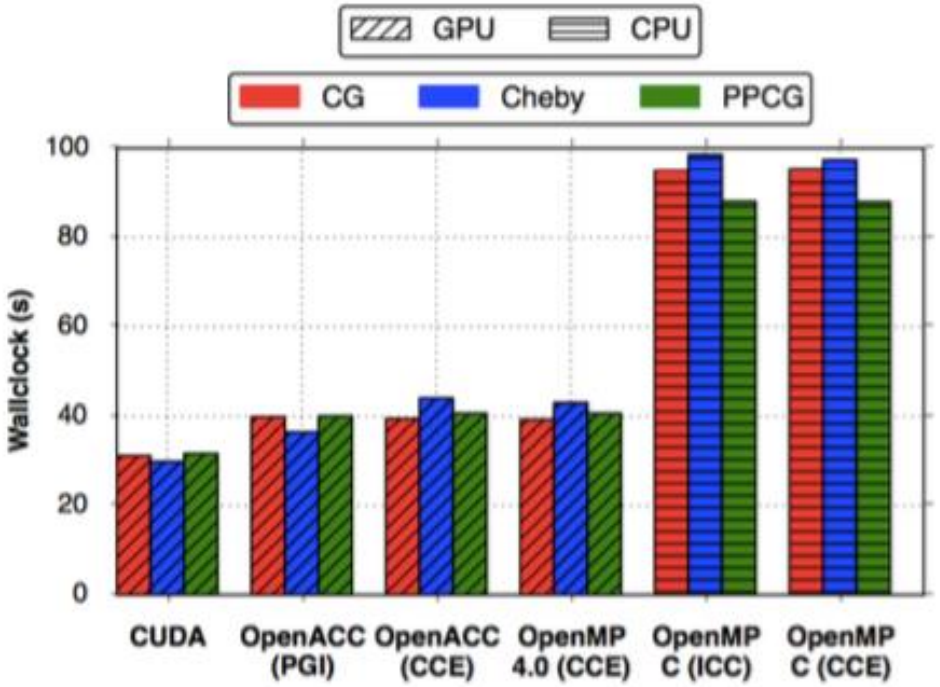
Performance?

* 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)

2016



TeaLeaf



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

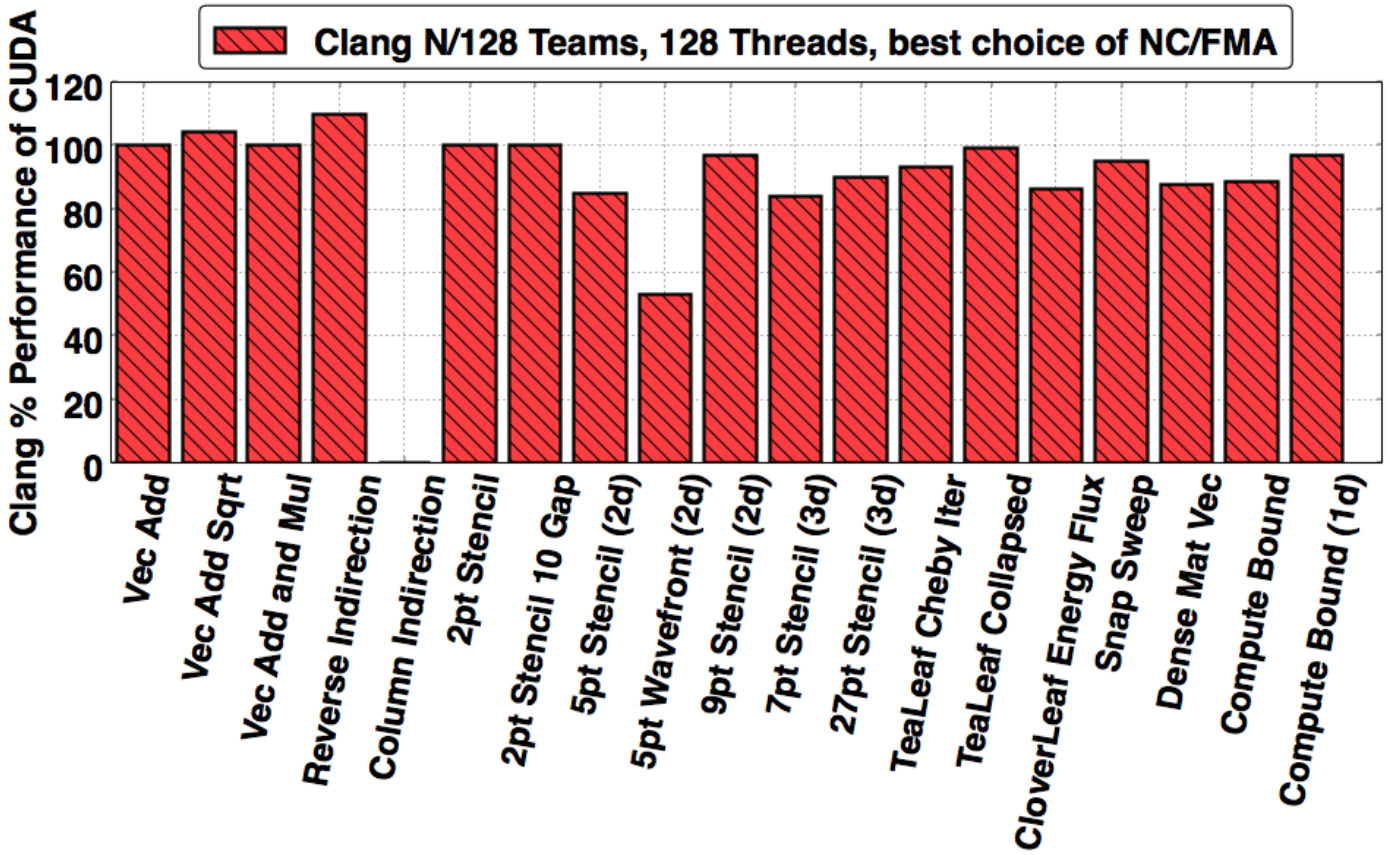
We have seen these figures continually improve as the languages have matured

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

Can you do better?

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

2016



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?

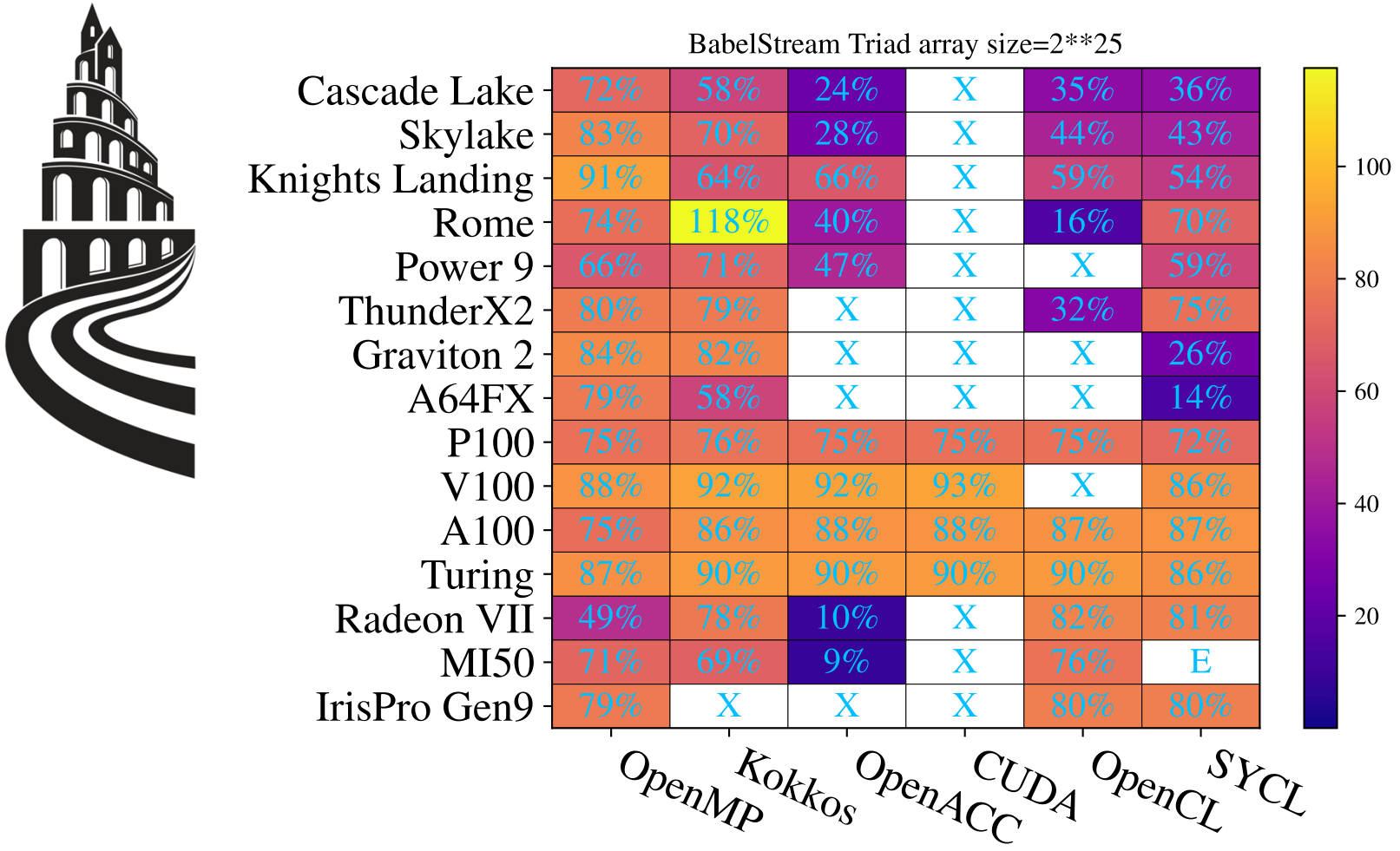
- Up until this point we had implicitly proven a good level of portability as we had successfully run OpenMP 4.x on many devices (Intel® CPU, Intel Xeon® Phi™ processors, NVIDIA® GPUs).

- The compiler support continually changes, improving performance, correctness and introducing new architectures.

- We keep tracking this improvement over time.

OpenMP in The Matrix

2020



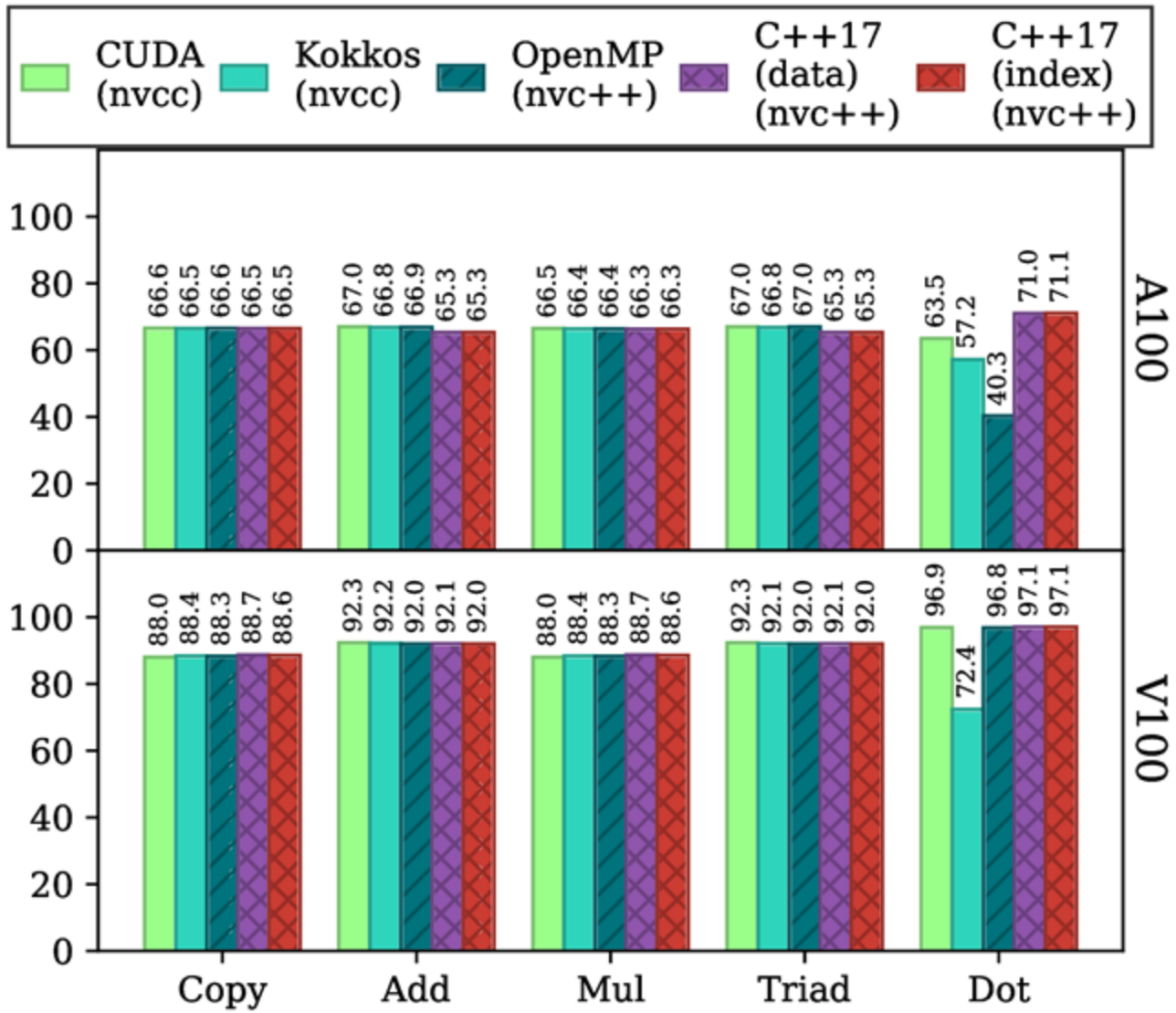
OpenMP has wide support, and **good** performance across all platforms

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

OpenMP/CUDA/Kokkos/C++/oh my!

2022

BabelStream kernels on NVIDIA A100 and V100 GPUs



Essentially no performance difference for most kernels

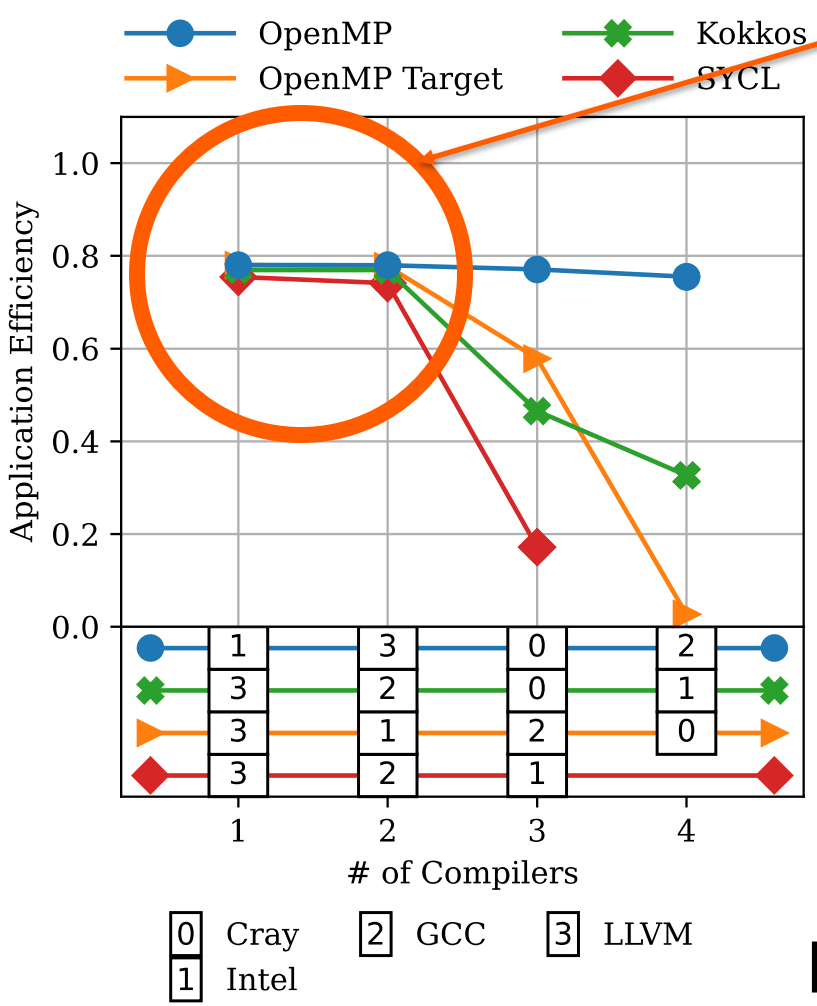
Reduction performance improving over time

From doi.org/10.1109/PMBS56514.2022.00009

Target everything?

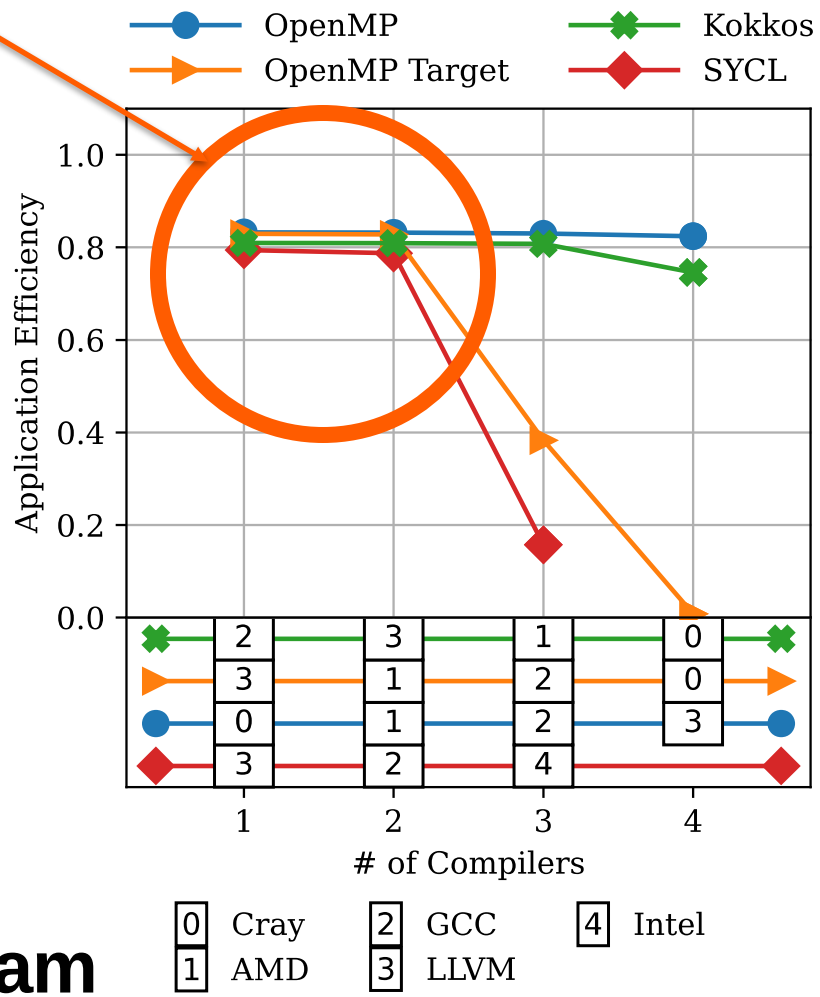
2022

Possible to get good performance with #pragma omp target on CPUs



Icelake

BabelStream



Milan

Deakin, et al, Heterogeneous Programming for the Homogeneous Majority, P3HPC at IEEE/ACM Supercomputing 2022

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.

Agenda

Morning

- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

Afternoon

- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

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 Most of OpenMP 5.x supported
- **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 and AMD 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/

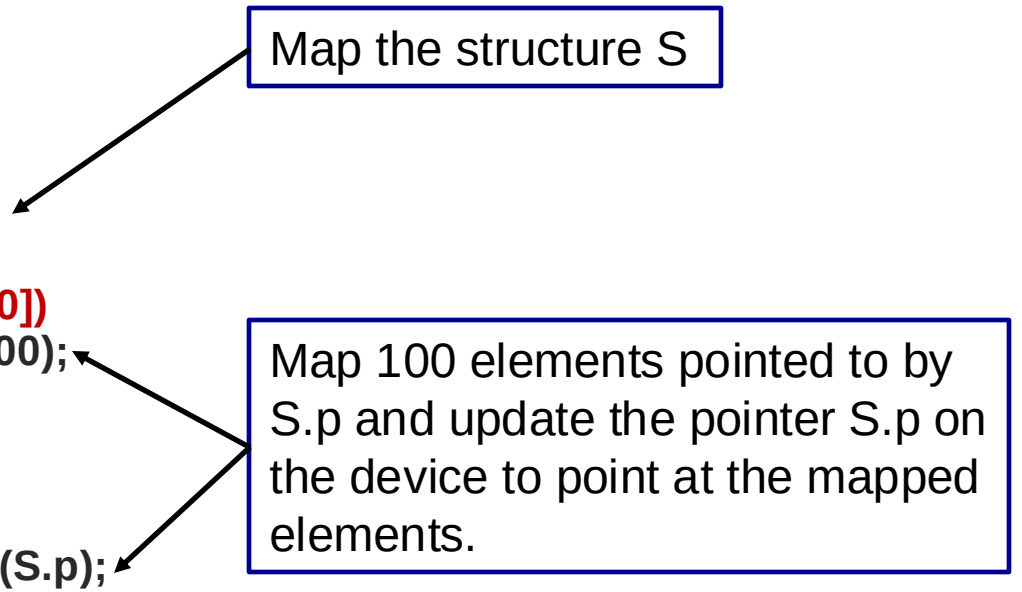
OpenMP 5.x and ecosystem

- OpenMP 5 adds features to make writing performance portable programs simpler.
- Highlighting some applicable to target offload:
  - 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);
```



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)\
      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])
{
  do_something_with_v(&v);
}
```

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]);
```

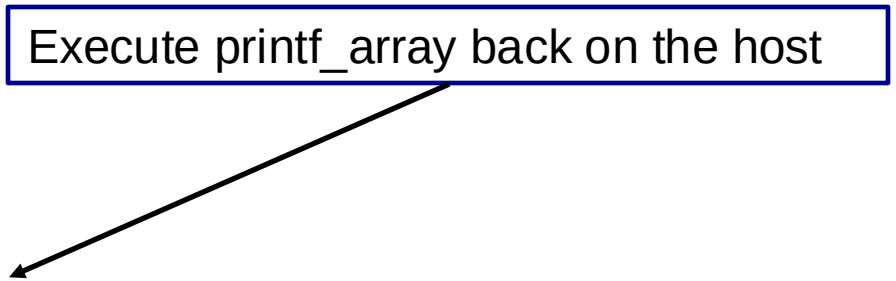
Declare fastFUNC as a variant for FUNC when executing in a target region.

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);

  …
}
```



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…

Agenda

Morning

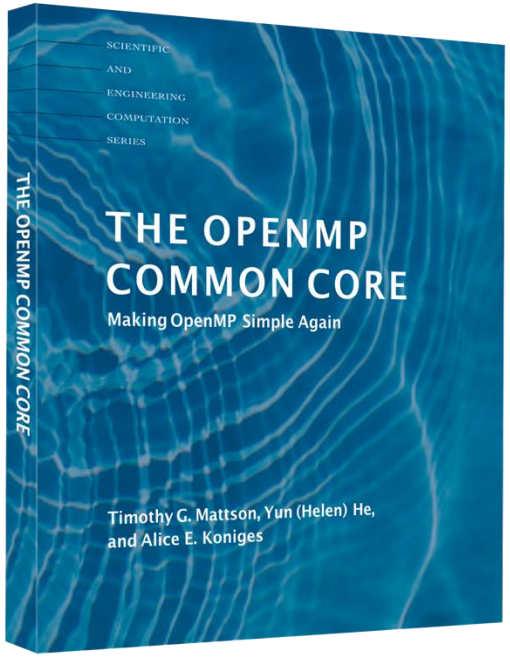
- Introduction
- OpenMP overview
- Live exercise 1
- Device model
- Moving data implicitly
- Loop directive
- Live exercise 2
- **Coffee break, 30 mins**
- Moving data explicitly
- Profiling offloaded code
- Live exercise 3

Afternoon

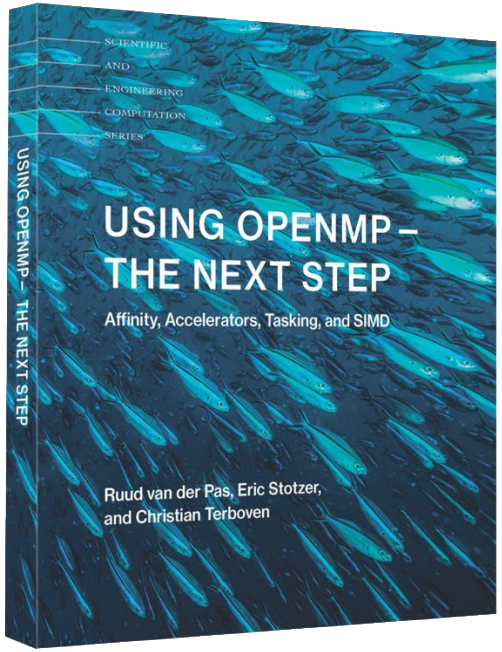
- Welcome back and recap
- Controlling data movement
- Optimising GPU
- Live exercise 4
- **Coffee break, 30 mins**
- BUD – “Big Ugly Directive”
- Team-only memory
- Live exercise 5
- Performance portability
- OpenMP 5 and ecosystem
- QA, discussion, time to finish exercises

To learn more about OpenMP

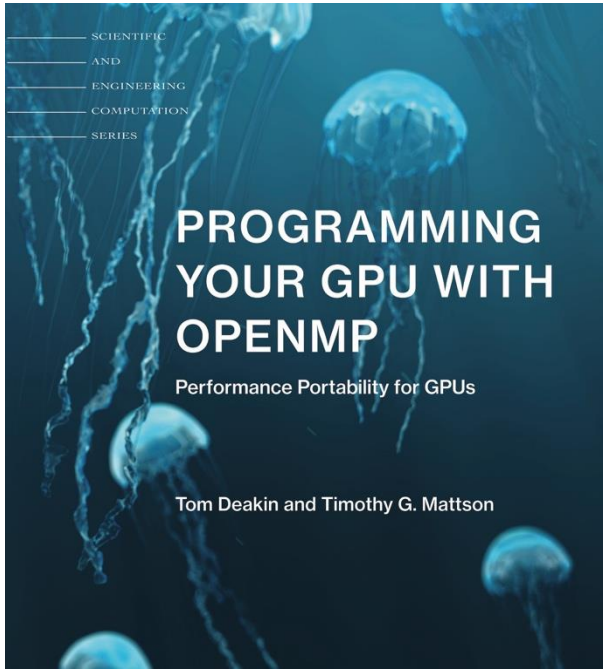
The OpenMP web site has a great deal of material to help you with OpenMP www.openmp.org
Reading the spec is painful … but each spec has a collection of examples. Study the examples, don't try to read the specs.
Since the specs are written ONLY for implementors … programmers need the OpenMP Books to master 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)
Coming in November 2023

Programming Your GPU with OpenMP

Thank you for joining us!

Please Evaluate this Session – give feedback via the SC Schedule page.

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

Tim Mattson
Human Learning Group
tgmattso@gmail.com

Tutorial materials are also available here: https://github.com/uob-hpc/openmp-tutorial

Live Q&A and Discussion

Appendices

- OpenMP and C++

OpenMP and C++

- OpenMP and C++ version compatibility
- Mapping class member variables and functions

OpenMP 4.x C++ support

- The OpenMP API specification refers to ISO/IEC 14882:1998 as **C++98!**
- Think programming “C in C++”.

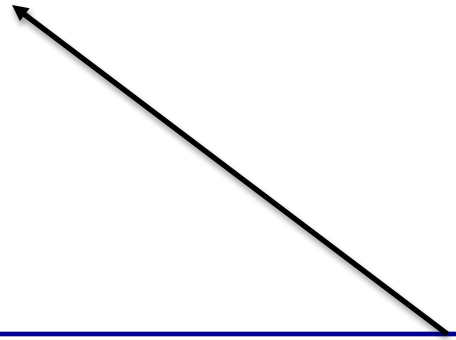
```
foo(std::vector<double> &x)
{
  #pragma omp target data map(x)
 { … }
}
```


You cannot map STL containers!

```
foo(std::vector<double> &x)
{
  double *pv = &x[0];
  #pragma omp target data map(pv[:x.size()])
 { … }
}
```

Mapping class member variables and functions

```
struct typeX { int a; };
class typeY {
  int a;
public:
  int foo() { return a^0x01;}
};
#pragma omp declare target to(typeY::foo)

#pragma omp declare target
struct typeX varX; // ok
#pragma omp end declare target
class typeY varY; // ok

void foo()
{
#pragma omp target map(varY)
{
  varX.a = 100; // ok
  varY.foo(); // ok
}}
```



The member function typeY::foo() can be accessed on a target device as long as it appears in a declare target directive and is not virtual.

Inspired by OpenMP 4.5 Examples Document

Mapping dynamically allocated class member variables

```
class Matrix
{
  Matrix(int n) {
  len = n;
  v = new double[len];
  #pragma omp target enter data map(alloc:v[0:len])
 }

 ~Matrix() {
  #pragma omp target exit data map(delete:v[0:len])
  delete[] v;
 }

private:
  double* v;
  int len;
};
```

Use delete map type since the corresponding host data is free'd after the deconstructor.

Inspired by OpenMP 4.5 Examples Document

OpenMP 5.0 C++ support

- OpenMP starts to support *modern* C++
- The OpenMP API specification refers to ISO/IEC 14882:{2011,2014,2017} as C++11, C++14, and C++17 respectively.

- The use of the following features may result in unspecified behavior.
  - Alignment support
  - Standard layout types
  - Allowing move constructs to throw
  - Defining move special member functions
  - Concurrency
  - Data-dependency ordering: atomics and memory model
  - Additions to the standard library
  - Thread-local storage
  - Dynamic initialization and destruction with concurrency
  - C++11 library
  - Sized deallocation (C++14)
  - What signal handlers can do (C++14)

Our running example: Jacobi solver

- An iterative method to solve a system of linear equations
  - Given a matrix A and a vector b find the vector x such that $Ax=b$
- The basic algorithm:
  - Write A as a lower triangular (L), upper triangular (U) and diagonal matrix

    $Ax = (L+D+U)x = b$
  - Carry out multiplications and rearrange

    $Dx=b-(L+U)x \rightarrow x = (b-(L+U)x)/D$
  - Iteratively compute a new x using the x from the previous iteration

    $X_{new} = (b-(L+U)x_{old})/D$
- Advantage: we can easily test if the answer is correct by multiplying our final x by A and comparing to b
- Disadvantage: It takes many iterations and only works for diagonally dominant matrices

Jacobi Solver

Iteratively update xnew until the value stabilizes (i.e. change less than a preset TOL)

```
<<< allocate and initialize the matrix A >>>
<<< and vectors x1, x2 and b           >>>

while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;

    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

```
    // test convergence
    conv = 0.0;
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

    // swap pointers for next
    // iteration
    TYPE* tmp = xold;
    xold = xnew;
    xnew = tmp;

} // end while loop
```

Exercise: Jacobi solver

- Start from the provided jacobi_solver program. Verify that you can run it serially.
- Parallelize for a CPU using the *parallel for* construct on the major loops
- Use the target directive to run on a GPU.
  - #pragma omp target
  - #pragma omp target map(to:list) map(from:list) map(tofrom:list)

Jacobi Solver (Par Targ, 1/2)

```
while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;

#pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                   map(to:A[0:Ndim*Ndim], b[0:Ndim] )
for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (Par Targ, 2/2)

```
      //
      // test convergence
      //
      conv = 0.0;
 #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]), map(tofrom:conv)
for (i=0; i<Ndim; i++){
          tmp  = xnew[i]-xold[i];
          conv += tmp*tmp;
      }
      conv = sqrt((double)conv);

     TYPE* tmp = xold;
     xold = xnew;
     xnew = tmp;

} // end while loop
```