Programming Your GPU with OpenMP*

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

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

Eric Stotzer
Texas Instruments
estotzer@ti.com

… and the McIntosh-Smith group at the University of Bristol:
Tom Deakin, **Matt Martineau** and **James Price**

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

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 multiprocessor server.
- 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.

Isambard

- Collaboration between GW4 Alliance (universities of Bristol, Bath, Cardiff, Exeter), the UK Met Office, Cray, and Arm
- ~£3 million, funded by EPSRC
- Expected to be the first large-scale, Arm-based production supercomputer
- **10,000+** Armv8 cores
- Also contains some Intel Xeon Phi (KNL), Intel Xeon (Broadwell), and **NVIDIA P100 GPUs**
- Cray compilers provide the OpenMP implementation for GPU offloading

University of Bristol

University of Bath

University of Exeter

Cardiff University / Prifysgol Caerdydd

Met Office

EPSRC

Cray The Supercomputer Company

ARM

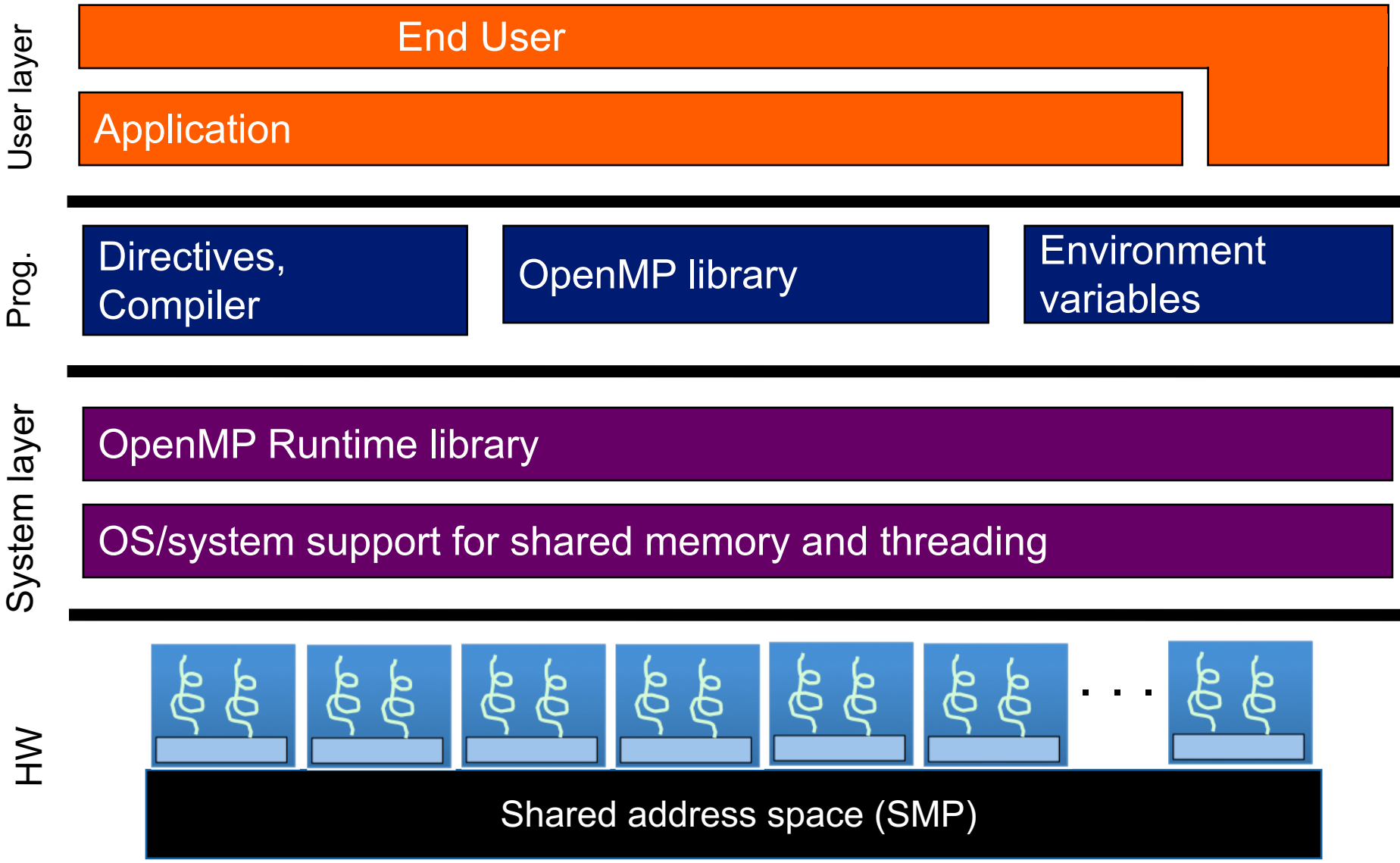
Plan

Module	Concepts	Exercises
OpenMP overview	• OpenMP recap • Hardcore jargon of OpenMP	• None … we'll use demo mode so we can move fast
The device model in OpenMP	• Intro to the Target directive with default data movement	• pi program
Understanding execution	• Intro to nvprof	• nvprof ./pi
Basic memory movement	• The map clause	• Jacobi solver
Working with the target directive	• CPU and GPU execution models and the fundamental combined directive	• Pi with combined directive using nvprof to understand execution
Optimizing memory movement	• Target data regions	• Jacobi with explicit data movement
Optimizing GPU code	• Common GPU optimizations	• Optimized Jacobi solver
Stencil	• Practice everything you've learned!	• 5-point stencil

Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Moving memory to/from the device
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability
- 5-point stencil
- OpenMP 5.0

OpenMP basic definitions: Basic Solution stack

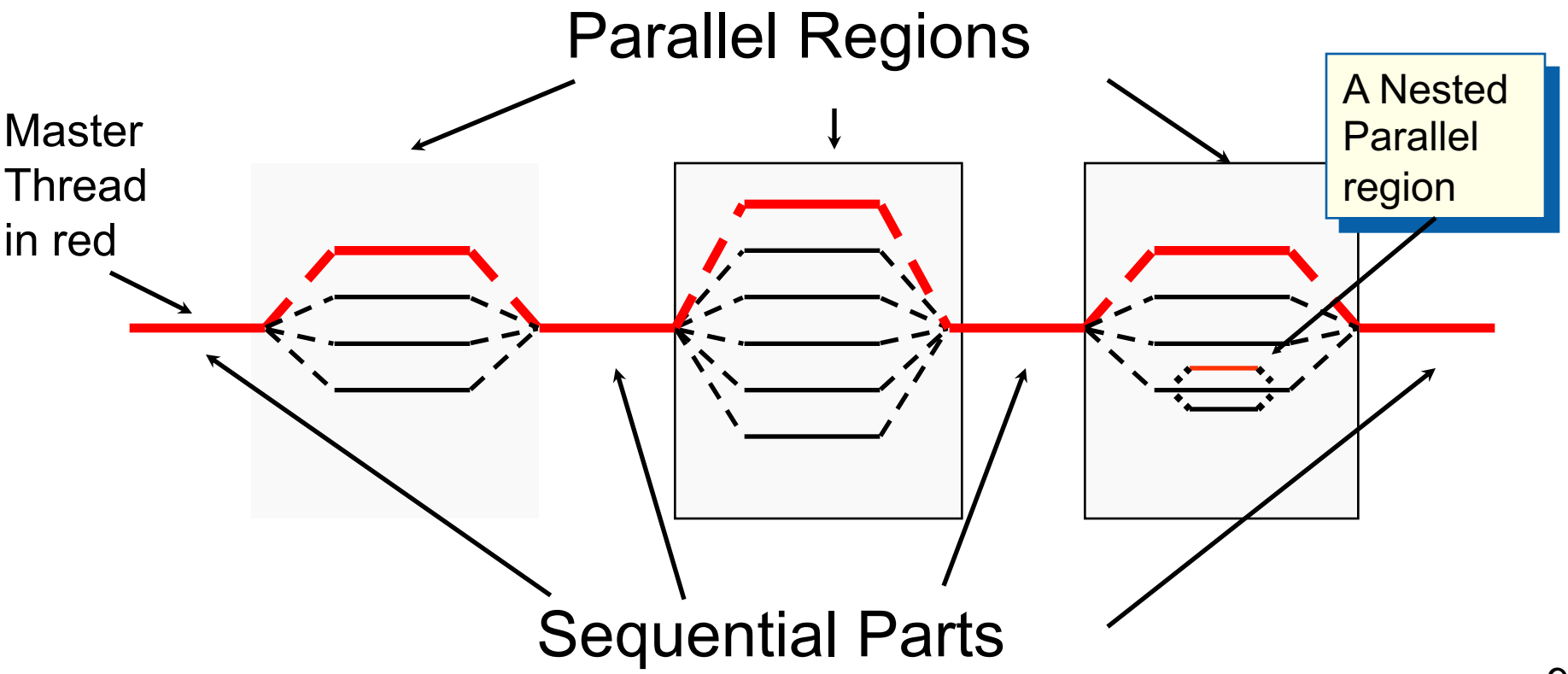


When we launched OpenMP, our focus was on Symmetric Multiprocessor Case ….
i.e. lots of threads with “equal cost access” to memory

OpenMP programming model:

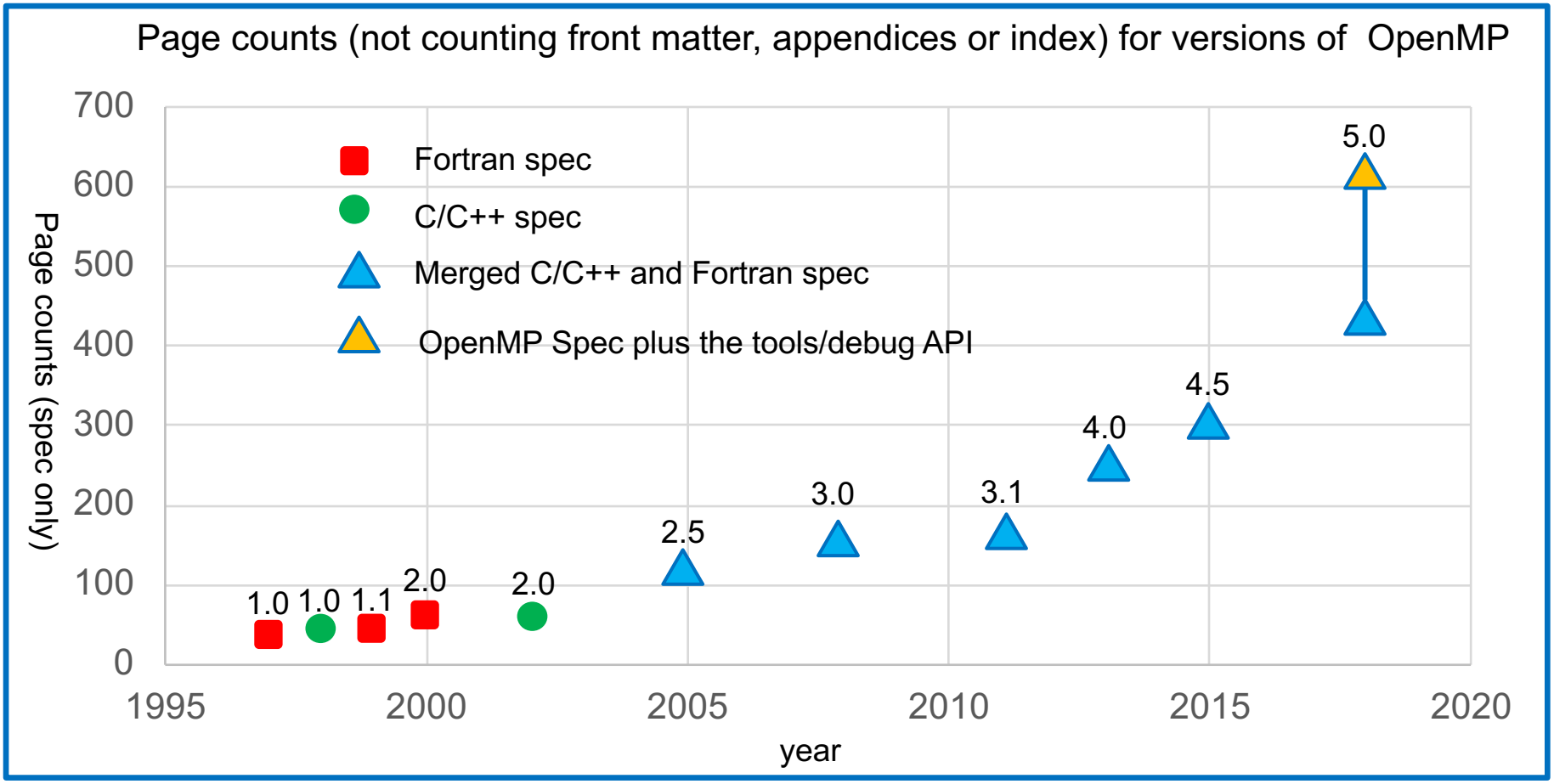
Fork-Join Parallelism:

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



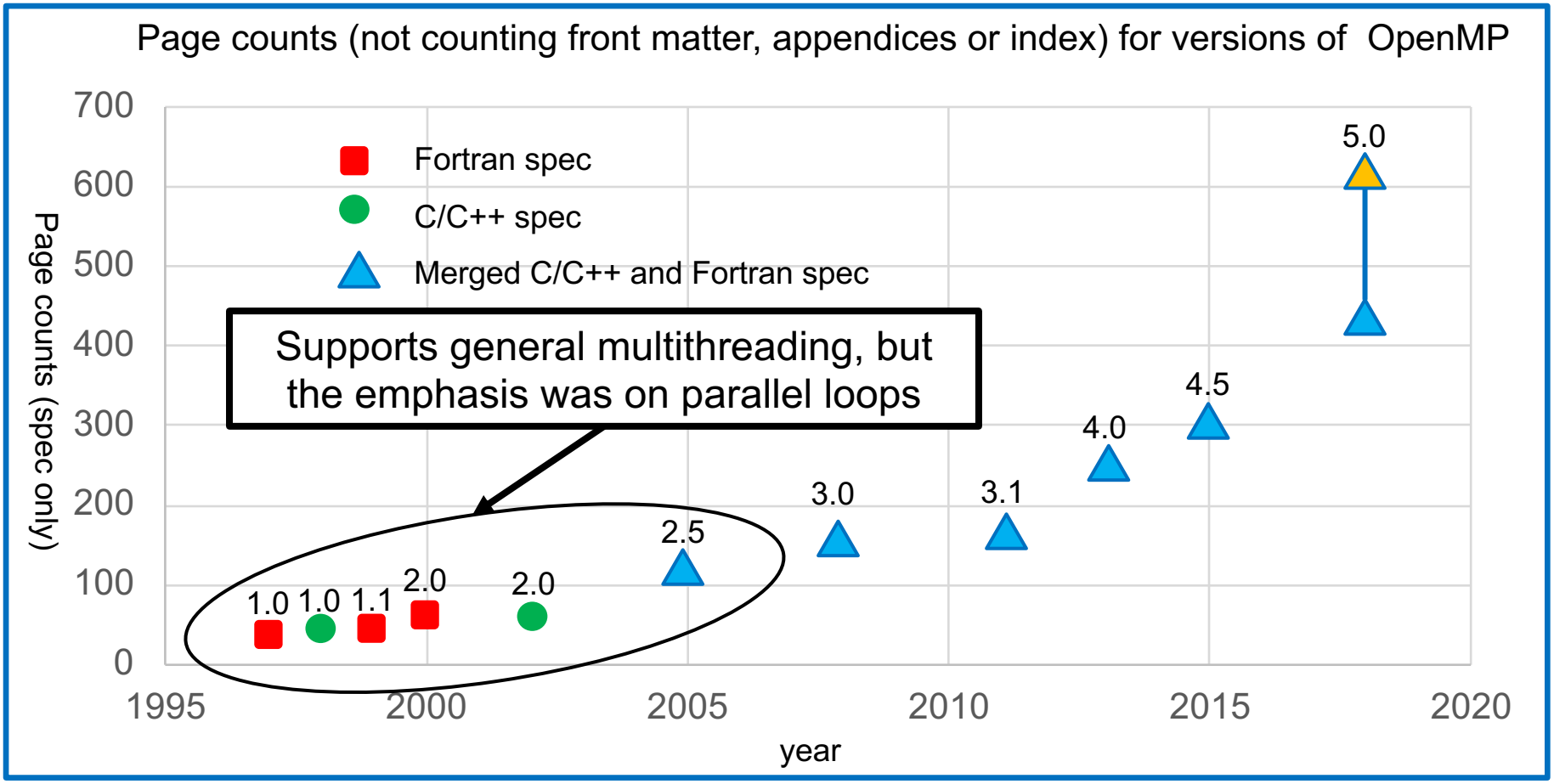
The growth of complexity in OpenMP

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

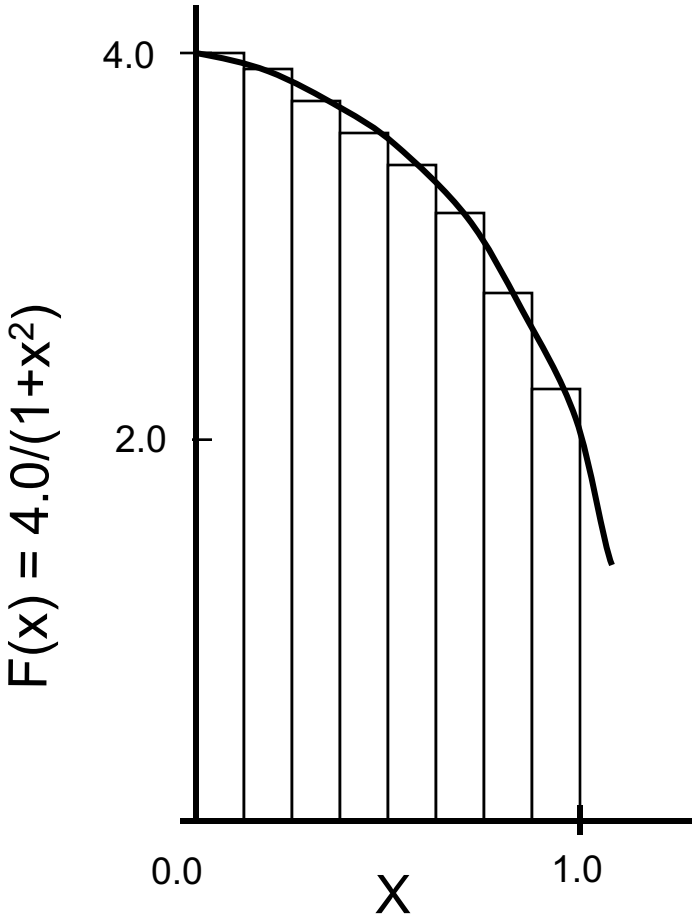


The growth of complexity in OpenMP

- 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 sc18-tutorial/pi.c

Example: Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{    int i;            double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel
     {
          double x;
        #pragma omp for reduction(+:sum)
            for (i=0; i< num_steps; i++){
                  x = (i+0.5)*step;
                  sum = sum + 4.0/(1.0+x*x);
            }
      }
           pi = step * sum;
}
```

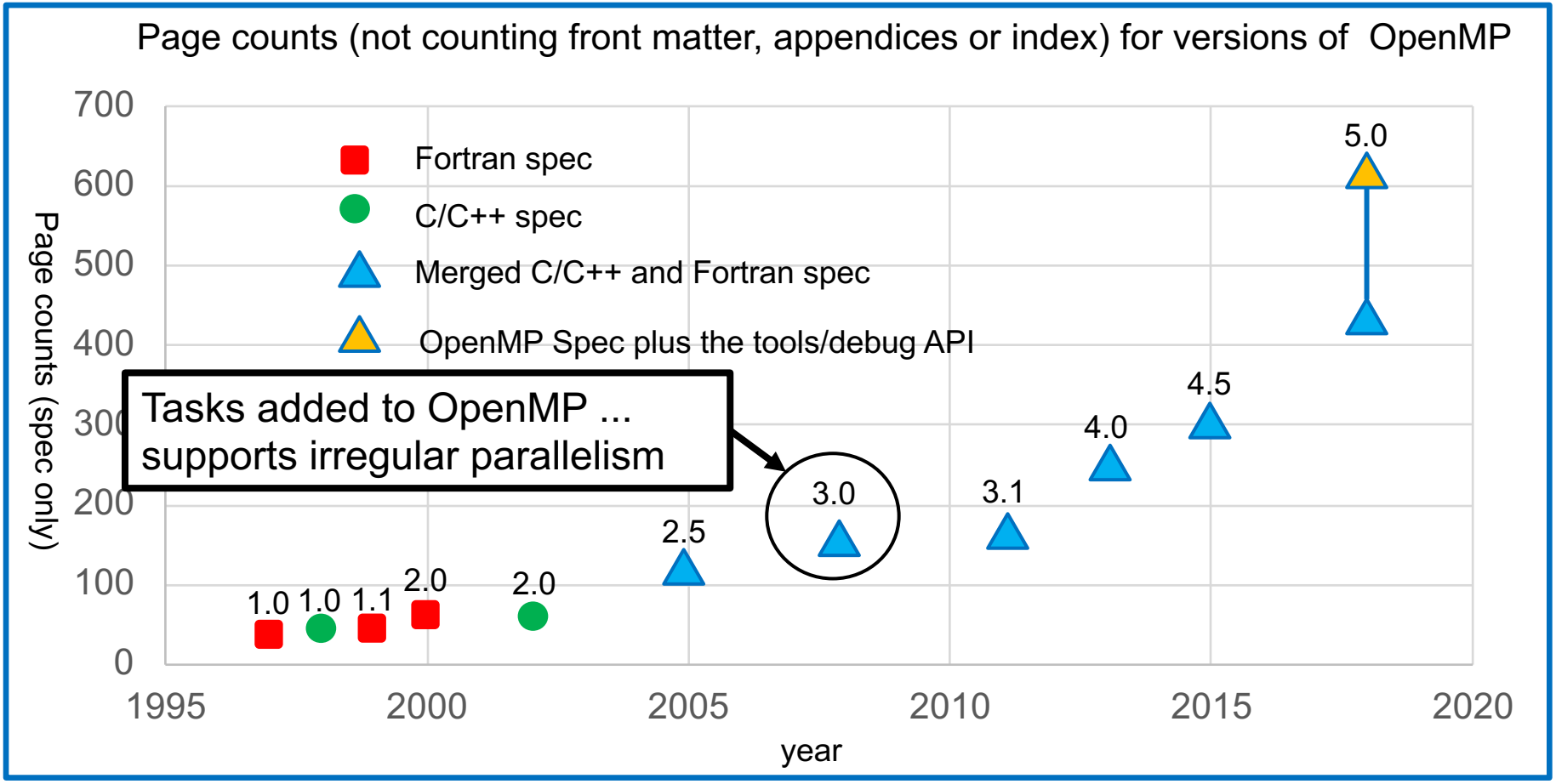
Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop index is local to a thread by default.

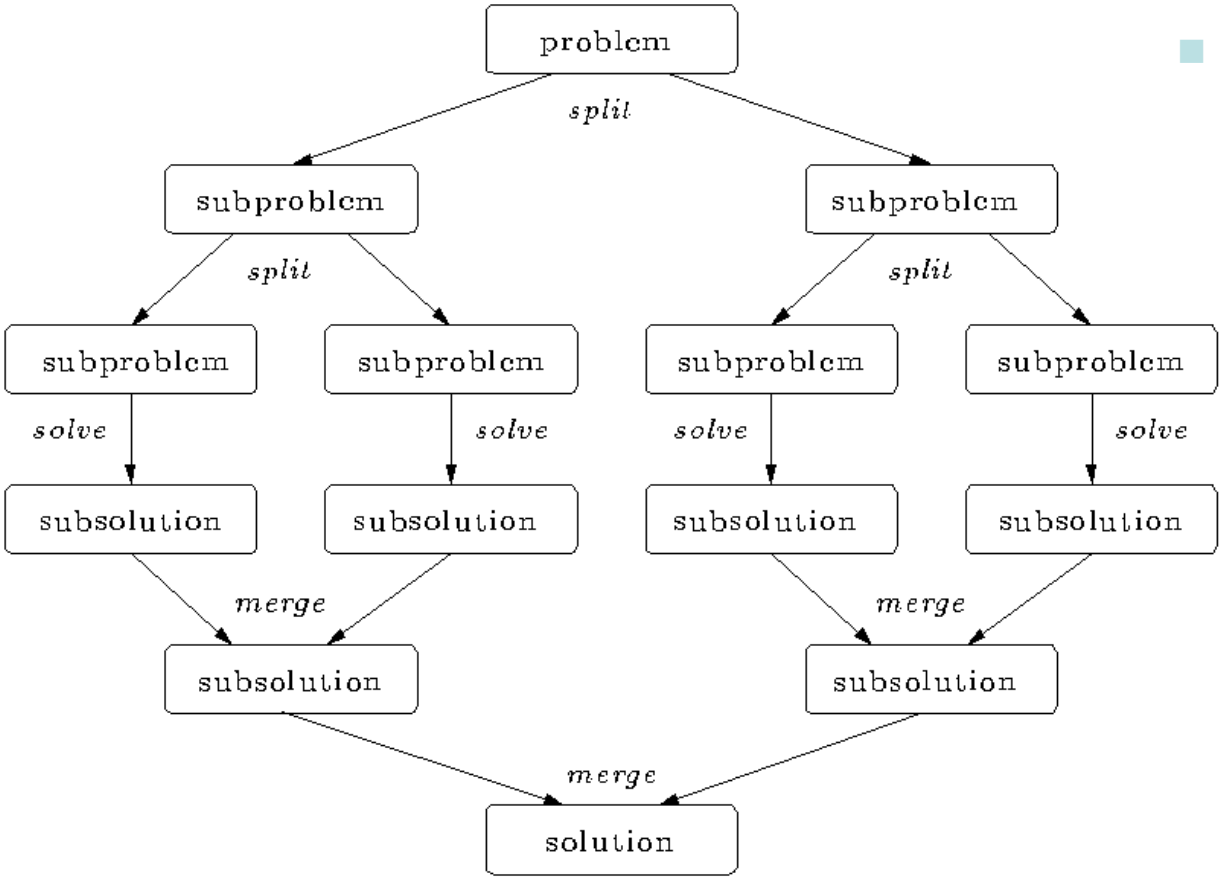
The growth of complexity in OpenMP

- 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!



Divide and conquer

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



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

Program: Divide and conquer pi program

```
 include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
   double x, sum = 0.0,sum1, sum2;
   if (Nfinish-Nstart < MIN_BLK){
      for (i=Nstart;i< Nfinish; i++){
         x = (i+0.5)*step;
         sum = sum + 4.0/(1.0+x*x);
      }
   }
   else{
      iblk = Nfinish-Nstart;
      sum1 = pi_comp(Nstart,         Nfinish-iblk/2,step);
      sum2 = pi_comp(Nfinish-iblk/2, Nfinish,       step);
       sum = sum1 + sum2;
   }
   return sum;
}
```

solve

split

```
int main ()
{
  int i;
  double step, pi, sum;
   step = 1.0/(double) num_steps;
    sum = pi_comp(0,num_steps,step);
    pi = step * sum;
}
```

Program: OpenMP tasks

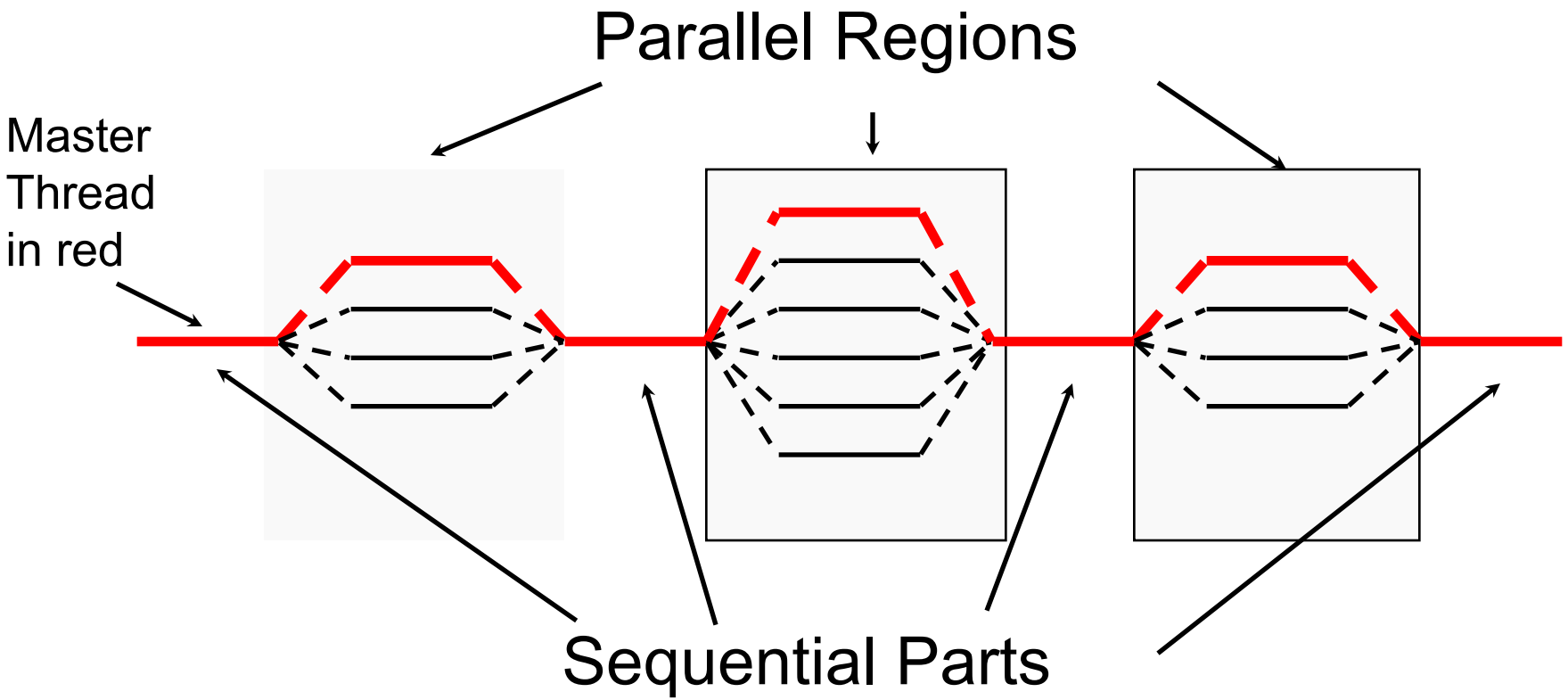
```
 include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
   double x, sum = 0.0,sum1, sum2;
   if (Nfinish-Nstart < MIN_BLK){
      for (i=Nstart;i< Nfinish; i++){
         x = (i+0.5)*step;
         sum = sum + 4.0/(1.0+x*x);
      }
   }
   else{
      iblk = Nfinish-Nstart;
      #pragma omp task shared(sum1)
           sum1 = pi_comp(Nstart,         Nfinish-iblk/2,step);
      #pragma omp task shared(sum2)
            sum2 = pi_comp(Nfinish-iblk/2, Nfinish,       step);
      #pragma omp taskwait
         sum = sum1 + sum2;
   }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
   step = 1.0/(double) num_steps;
   #pragma omp parallel
   {
      #pragma omp single
         sum =
            pi_comp(0,num_steps,step);
    }
     pi = step * sum;
}
```

Adding tasks to OpenMP required major changes to the specification

Fork-Join Parallelism:

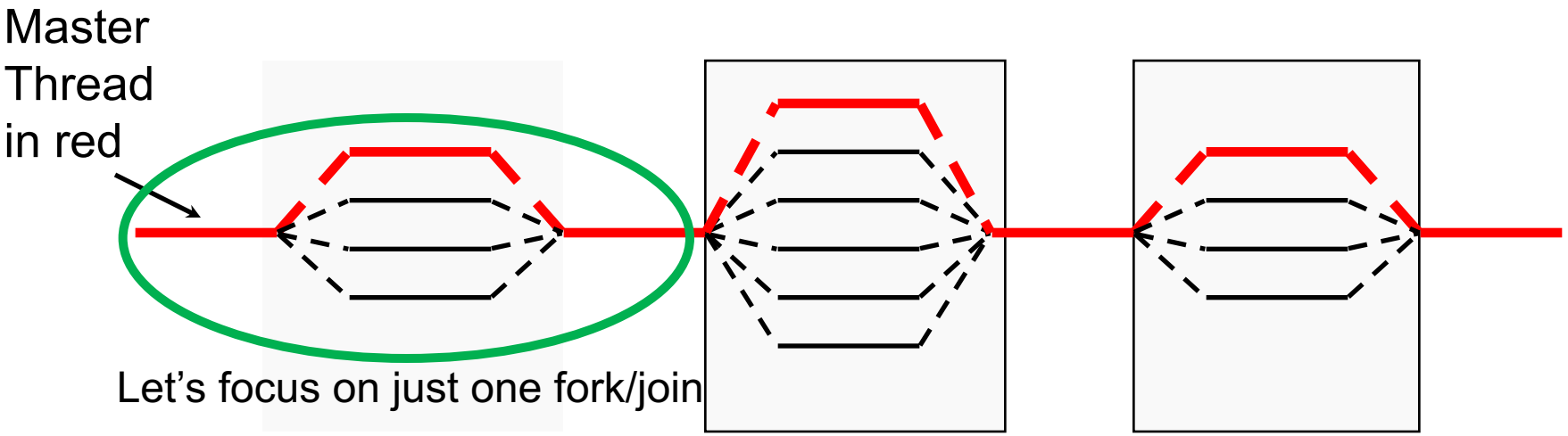
- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



Adding tasks to OpenMP required major changes to the specification

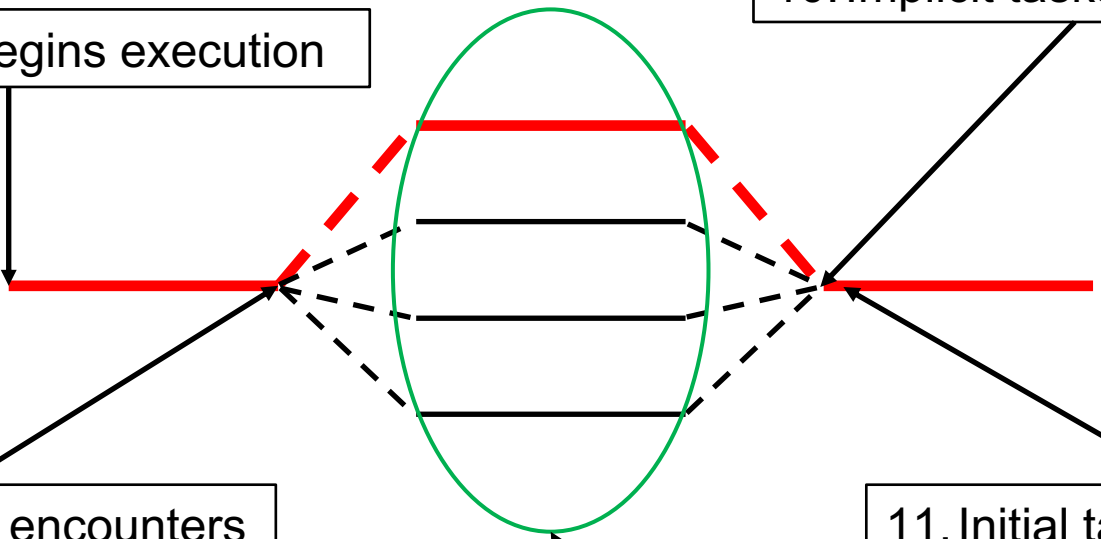
Fork-Join Parallelism:

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



Low Level details of OpenMP

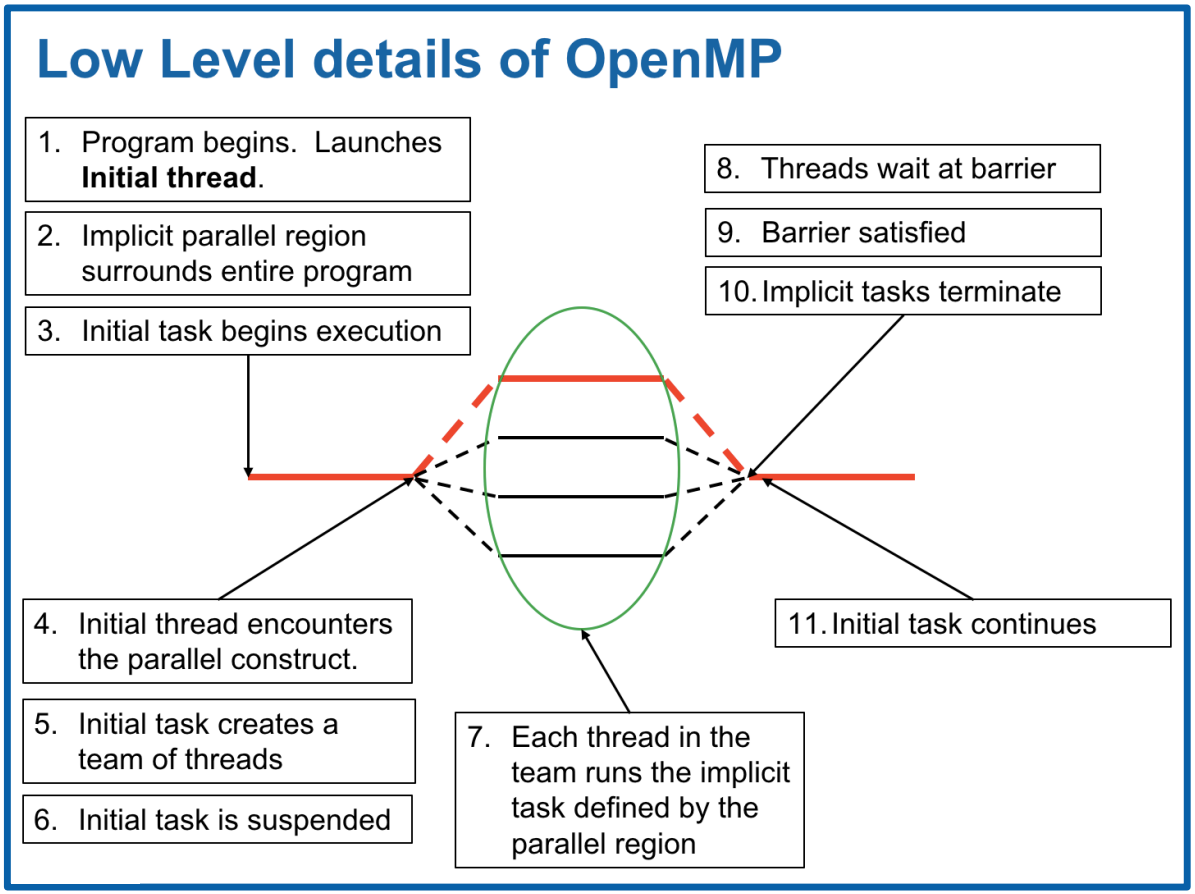
1. Program begins. Launches **Initial thread**.
2. Implicit parallel region surrounds entire program
3. Initial task begins execution
4. Initial thread encounters the parallel construct.
5. Initial task creates a team of threads
6. Initial task is suspended
7. Each thread in the team runs the implicit task defined by the parallel region
8. Threads wait at barrier
9. Barrier satisfied
10. Implicit tasks terminate
11. Initial task continues



Why all this complexity around tasks?

Remember: a language specification is written for people who implement the language … they have ZERO tolerance for ANY ambiguity.

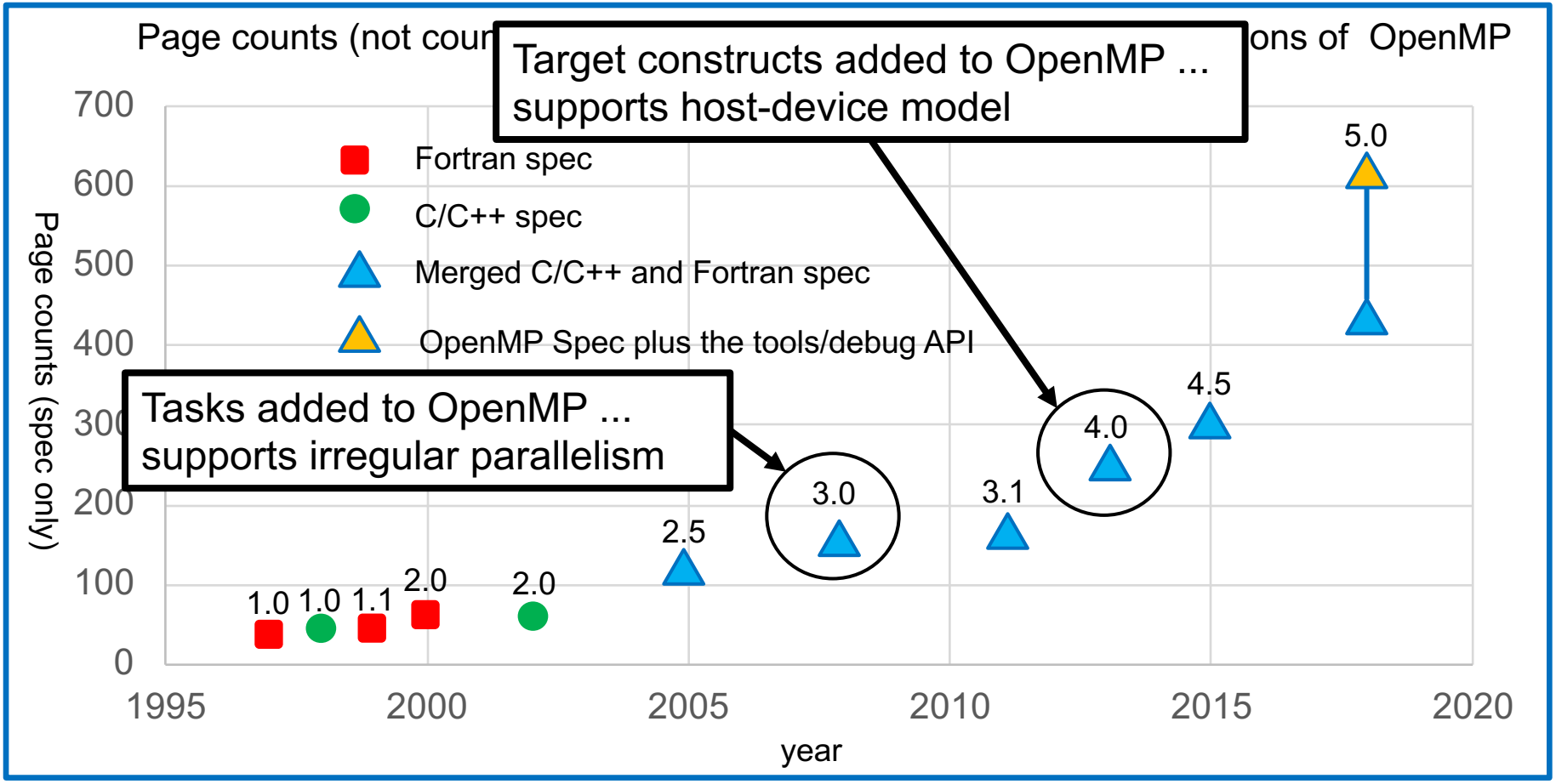
By defining a thread as an execution entity that runs tasks, we can define semantics in terms of tasks and consistently apply it everywhere.



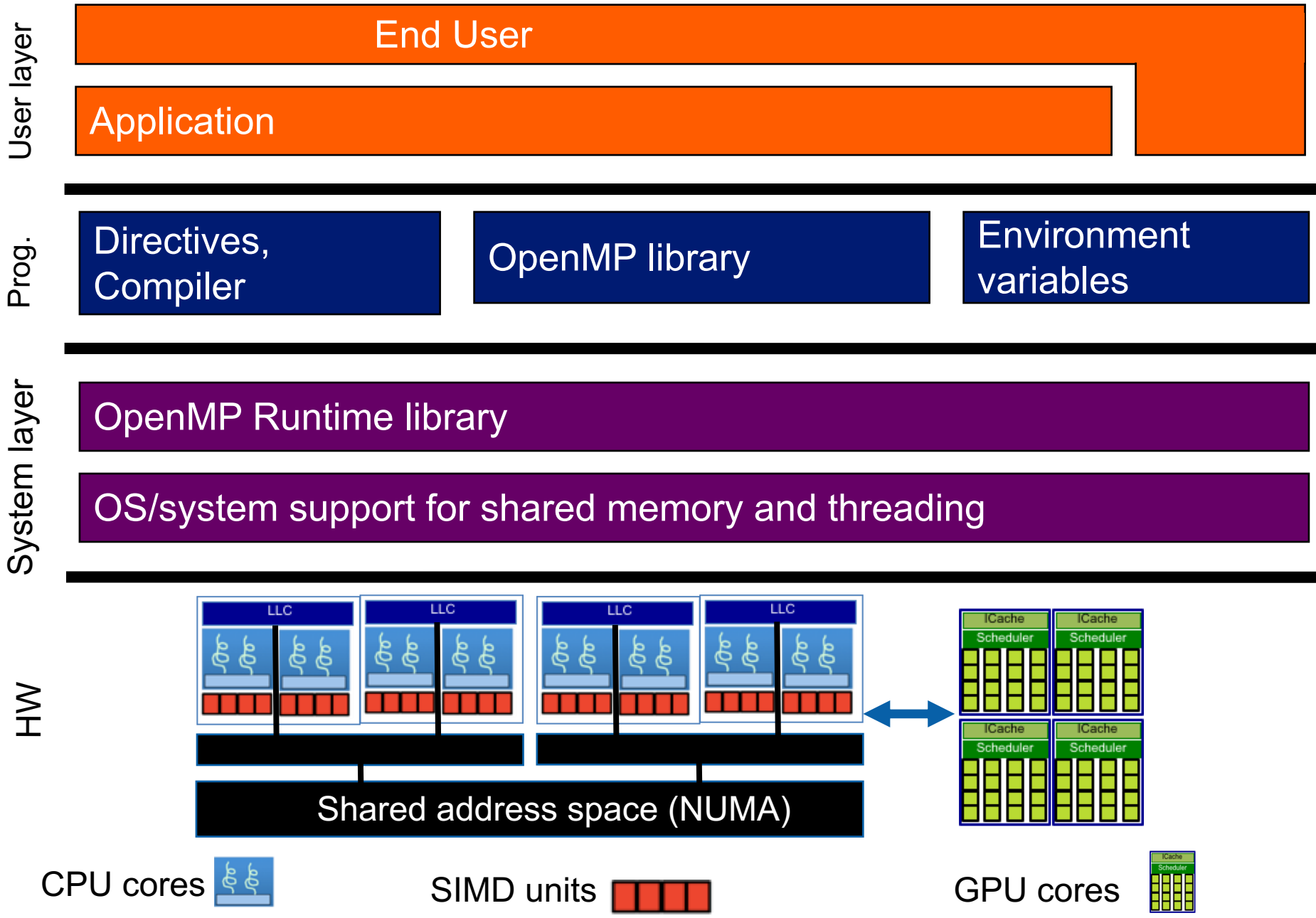
While all these initial threads, implicit tasks, and such are confusing to the programmer, they actually make life easier for people who implement OpenMP.

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 basic definitions: Basic Solution stack

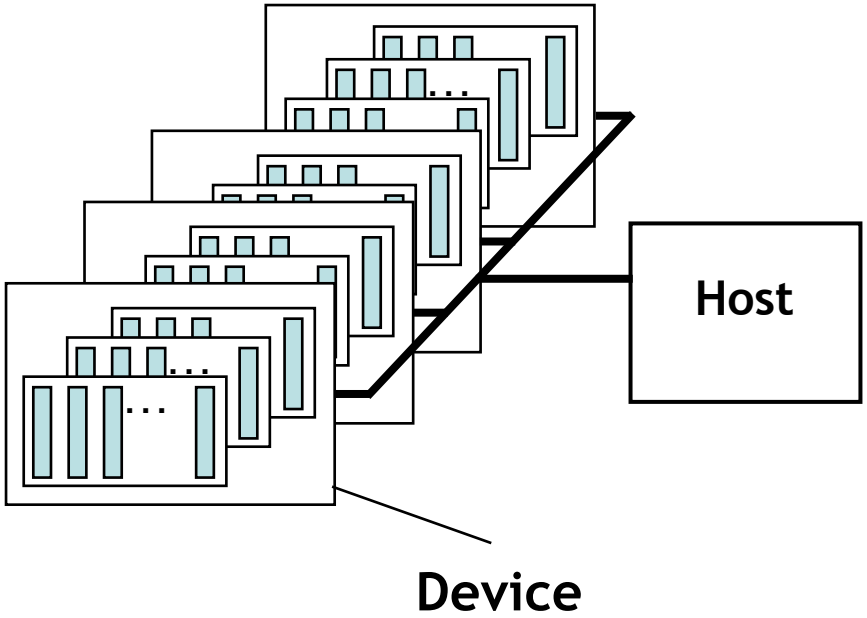


Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Moving memory to/from the device
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability
- 5-point stencil
- OpenMP 5.0

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



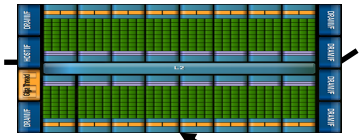

```
#include <omp.h>
#include <stdio.h>
int main()
{
    printf("There are %d devices\n",
            omp_get_num_devices());
}
```

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 thread 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

The target data environment

- Remember: distinct memory spaces on host and device.
- OpenMP uses a combination of *implicit* and *explicit* memory 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 here
   …
} // and here
```

  - We'll discuss the other places later…

Default Data Mapping: implicit movement on a target region

- Scalar variables:
  - Examples:
    - int N; double x;
  - OpenMP 4.5 implicitly maps scalar variables as **firstprivate**
  - A new value per thread and initialized with the original value.
  - The variable *is not* copied back to the host at the end of the target region.
  - OpenMP target regions for GPUs execute with CUDA/OpenMP, and a firstprivate scalar can be launched as a parameter to a kernel function without the overhead of setting up a variable in device memory.

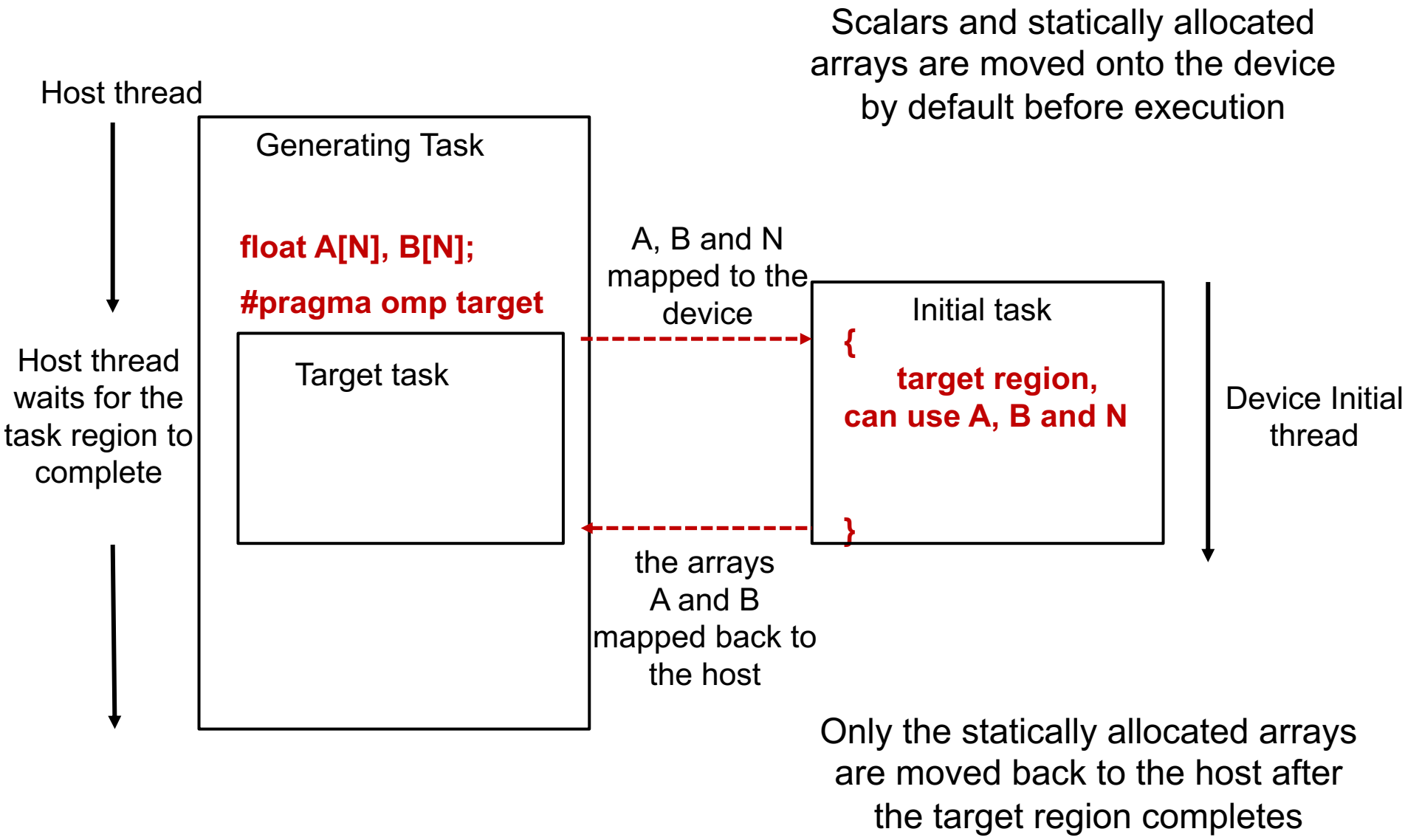
Default Data Mapping: implicit movement on a target region

- 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.
  - A new value 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 on a target region

- Pointers and their data:
  - *Example: arrays allocated on the heap*
    - double *A = malloc(sizeof(double)*1000);
  - The pointer value will be mapped.
  - But the data it points to ***will not*** be mapped by default.
  - In C, OpenMP defines that the pointer is mapped as a zero-length array section.
    - Array section notation: ptr[lower bound : length]
    - Zero length arrays: A[:0]
  - We'll show you how to map a pointer's data very shortly.

The target data environment



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.

Exercise

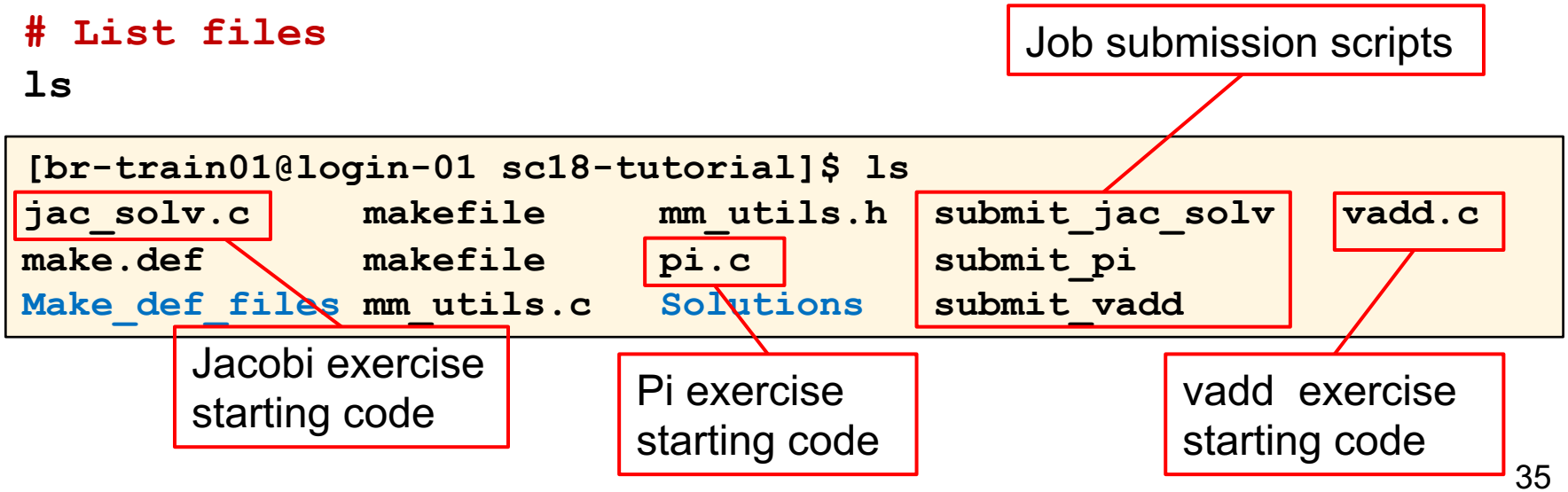
- Use the provided serial PI program (pi.c) which approximates a definitive integral the result of which should equal pi.
- Use the target construct to run the code on a GPU.
- Experiment with the other OpenMP constructs if you have time.

```
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;
}
```

```
#pragma omp target
#pragma omp parallel
#pragma omp for
#pragma omp for reduction(op:list)
#pragma omp for private(list)
```

Using Isambard (1/2)

```
# Log in to the Isambard bastion node
# This node acts as a gateway to the rest of the system
# You will be assigned an account ID (01, 02, …)
# Your password is openmpSC18
ssh br-trainXX@isambard.gw4.ac.uk

# Log in to Isambard Phase 1
ssh phase1

# Change to directory containing exercises
cd sc18-tutorial

# List files
ls
```



Using Isambard (2/2)

```
# Build exercises
make

# Submit job
qsub submit_pi

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

Job status:
- **Q** = **Q**ueued
- **R** = **R**unning
- **C** = **C**omplete

(job will disappear shortly after completion)

```
                                                                  Req'd  Req'd   Elap
Job ID          Username Queue    Jobname    SessID NDS TSK Memory Time  S Time
--------------- -------- -------- ---------- ------ --- --- ------ ----- - -----
9376.master.gw4 br-trai  pascalq  pi         154770   1  36     --  00:02 R 00:00
```

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

OpenMP target directive: the nowait clause

```
#pragma omp target nowait
{
// code defines a target region
}
```

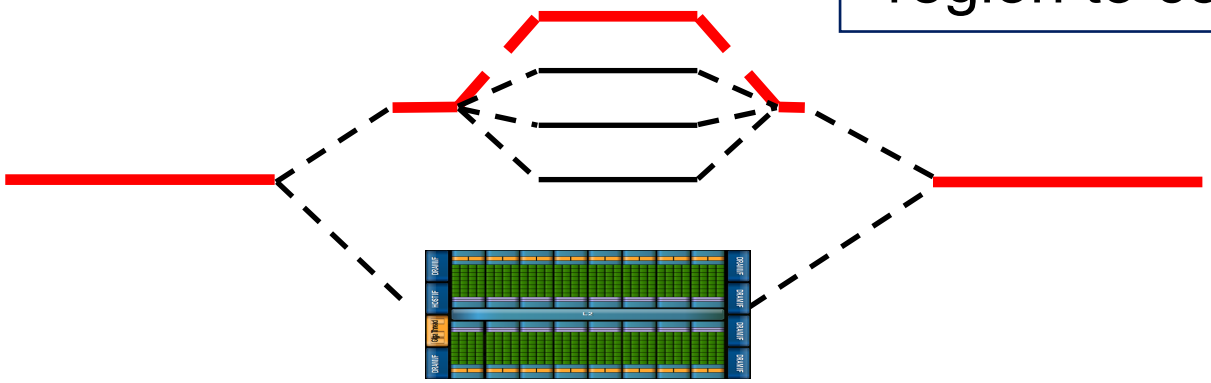
- Make the target task (running on the host) execute as a deferred task

```
#pragma omp parallel for
for(int i=0;i<N;i++) {
    big_stuff(i);
}
```

- The thread's implicit task can define other (potentially parallel) work.

```
#pragma omp taskwait
```

- The implicit task running on the host waits for the target region to complete.



Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Moving memory to/from the device
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability
- 5-point stencil
- OpenMP 5.0

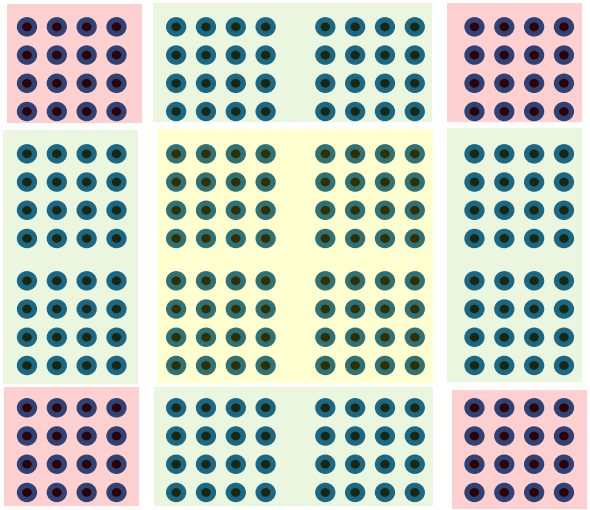
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

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 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

Third Party names are the property of their owners

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

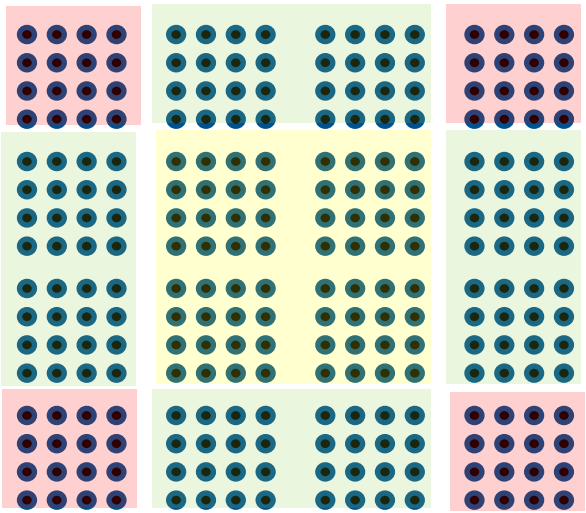
Turn source code into a scalar **work-item** (a CUDA **thread**)

```
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 an N dim index space. Cuda calls a work-group a **thread-block**



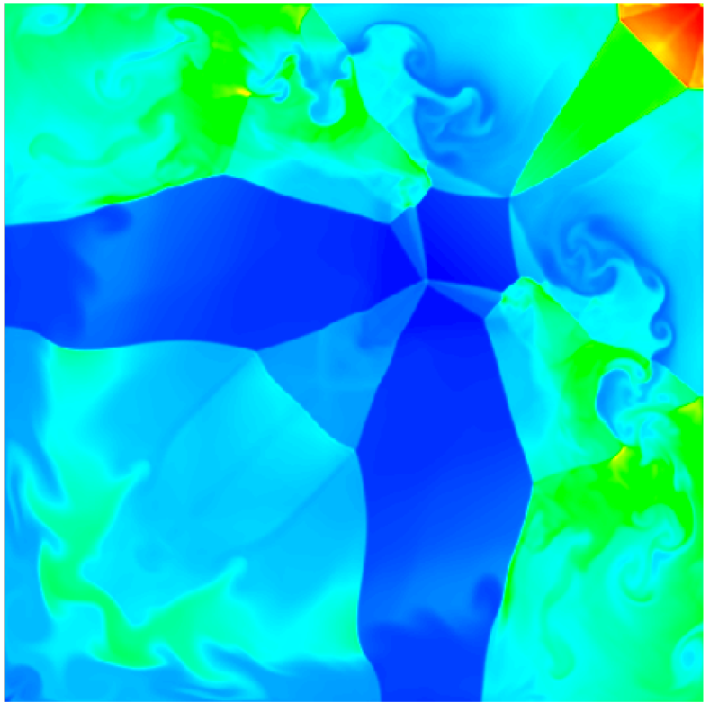
OpenCL index space is called an **NDRange.** CUDA calls this a **Grid**

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** with Cuda)

Third Party names are the property of their owners

CUDA Toolkit

The CUDA toolkit works with code written in OpenMP 4.5 without any changes



We will demonstrate using an OpenMP 4.5 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

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

Profiling data

Timings for computational kernels

CUDA Toolkit: NVProf

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
==188688== Profiling application: ./flow.omp
==188688== Profiling result:
```

Shows block sizes, grid dimensions and register counts for kernels

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

Entries ordered by time

Other CUDA Toolkit tools

- **nvidia-smi** – displays some information about NVIDIA devices on a node
- **cudagdb** – command line debugger in the style of GDB that allows kernel debugging
- **cuda-memcheck** – tool to catch memory access errors in a CUDA application
- **nvvp** – visual optimisation application that can use profiling data from nvprof

Exercise

- Use the Pi program from the previous exercise and explore how it executes on a GPU using nvprof.
  - nvprof ./pi
  - nvprof --print-gpu-trace ./pi

- Is a kernel run on the GPU?
- Where is all the time going?

- You will need to edit the job submission files to submit the job to the queue with nvprof. For example:
  - Replace the line
    - ./pi
  - with the line (for example)
    - nvprof --print-gpu-trace ./pi

Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Moving memory to/from the device
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability
- 5-point stencil
- OpenMP 5.0

Explicit Data Sharing

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

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

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

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 (device to host copy). On entering the region, initial value of the variable 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)**.
- For pointers you must use array section notation ..
  - **map(to:a[0:N]).** Notation is **A[lower-bound : length]**

Moving arrays with the map clause

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

   #pragma omp target map(A[0:N])
   {
     // N, ii and A all exist here
     // The data that A points to DOES exist here!
   }
}
```

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

Copy at start and end of **target** region.

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)

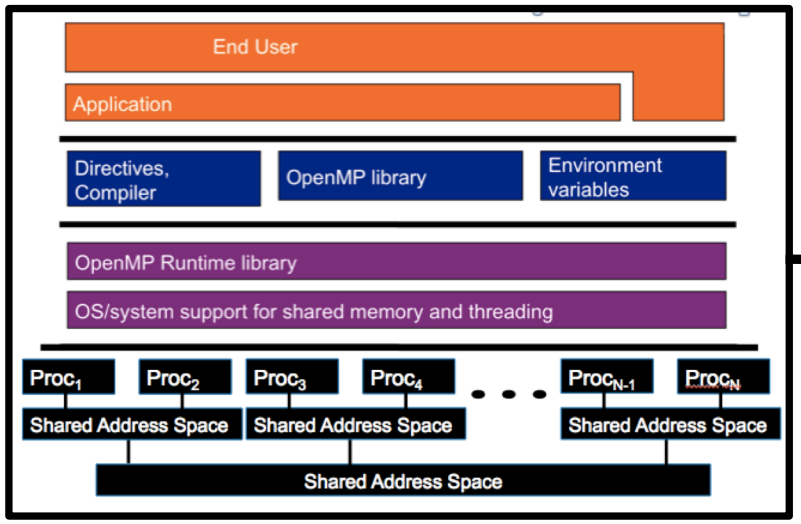
```
    //
    // test convergence
    //
    conv = 0.0;
#pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim])
for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

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

} // end while loop
```

Agenda

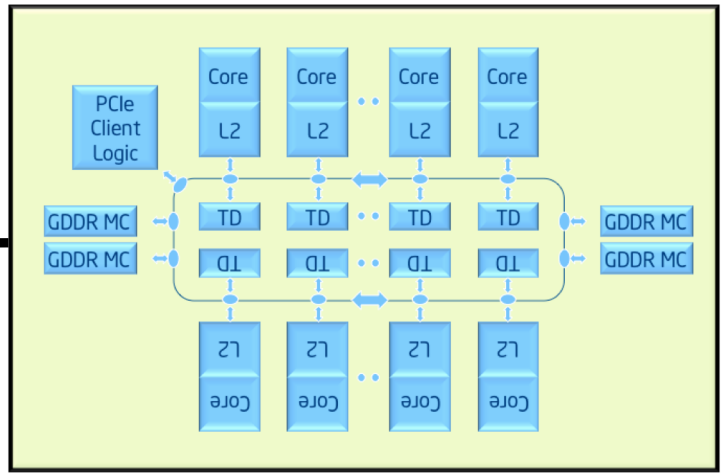
- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Moving memory to/from the device
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability
- 5-point stencil
- OpenMP 5.0

OpenMP device model: Examples

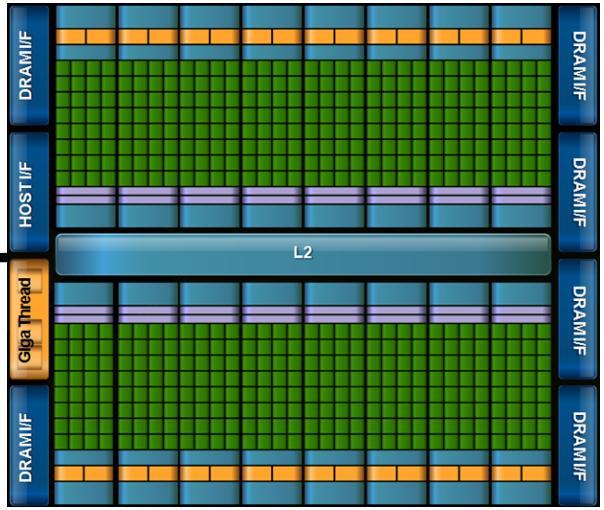
A couple key devices we considered when designing the device model in OpenMP



Host

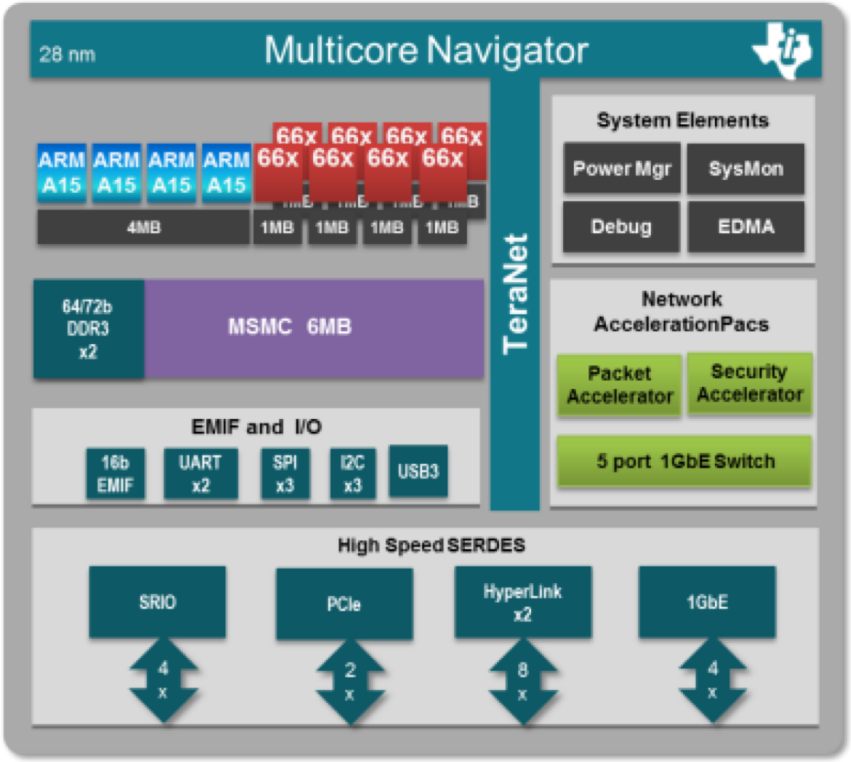


Target Device: Intel® Xeon Phi™ processor



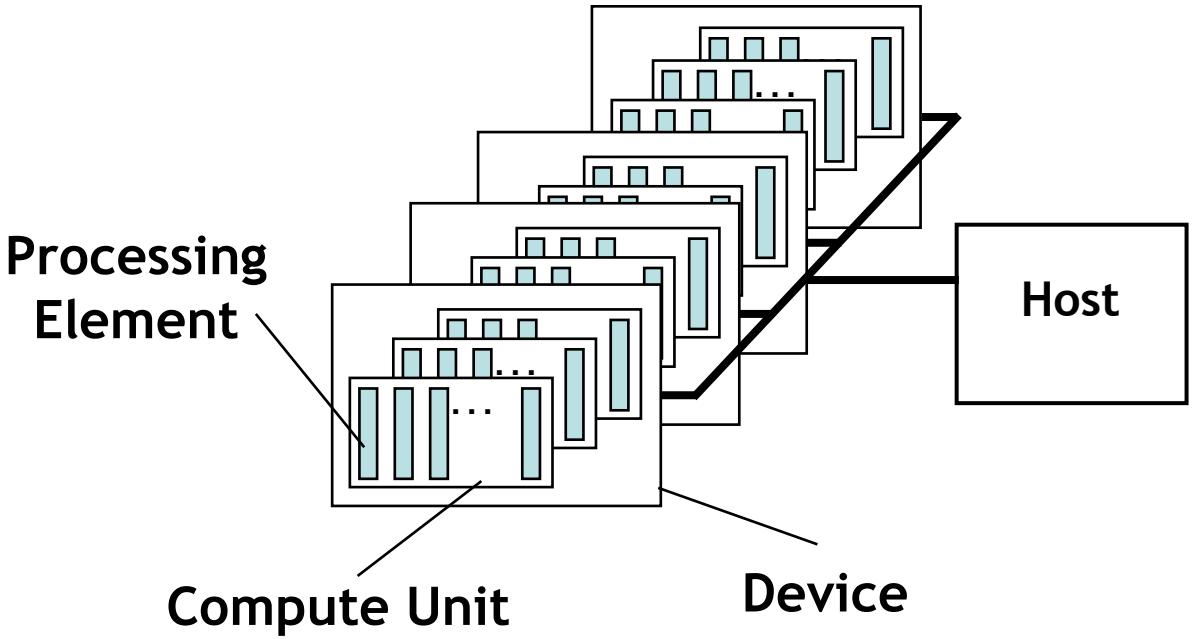
Target Device: GPU

OpenMP Device Model: another example



Heterogeneous SoC

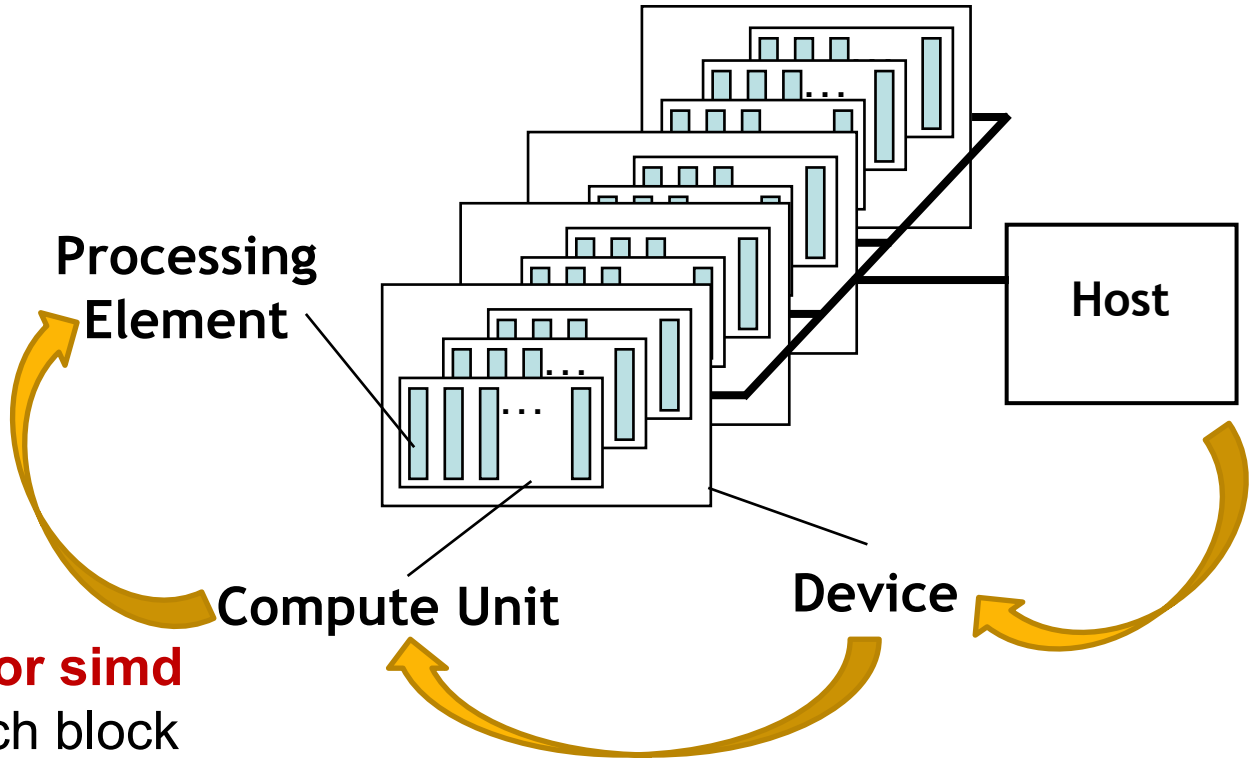
A Generic Host/Device Platform Model



- 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 divided into ***host memory*** and ***device memory***

Third party names are the property of their owners.

Our host/device Platform Model and OpenMP



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

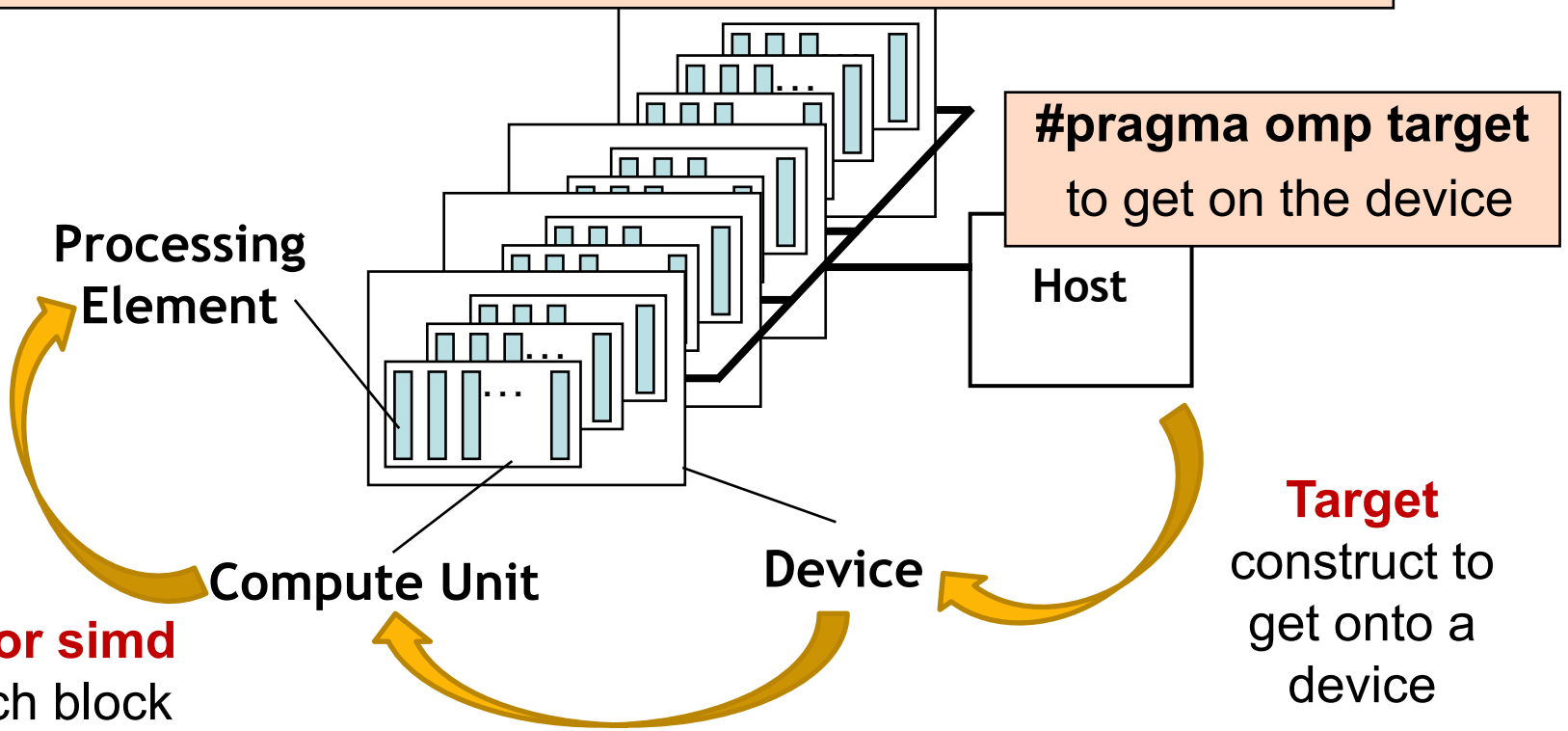
Teams construct to create a league of teams with one team of threads on each compute unit.

Target construct to get onto a device

Distribute construct to assign blocks of loop iterations to teams.

Our host/device Platform Model and OpenMP

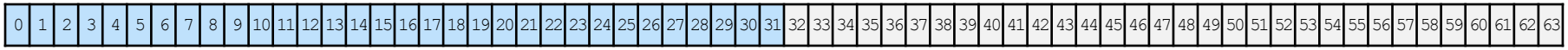
Typical usage ... let the compiler do what's best for the device:



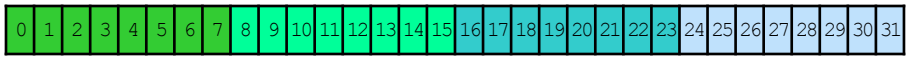
Accelerated Worksharing

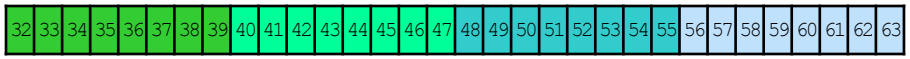
64 iterations assigned to 2 teams; Each team has 4 threads;
Each thread has 2 SIMD lanes

Distribute Iterations across 2 teams

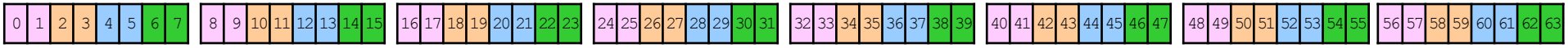


In a team workshare iterations
across 4 threads





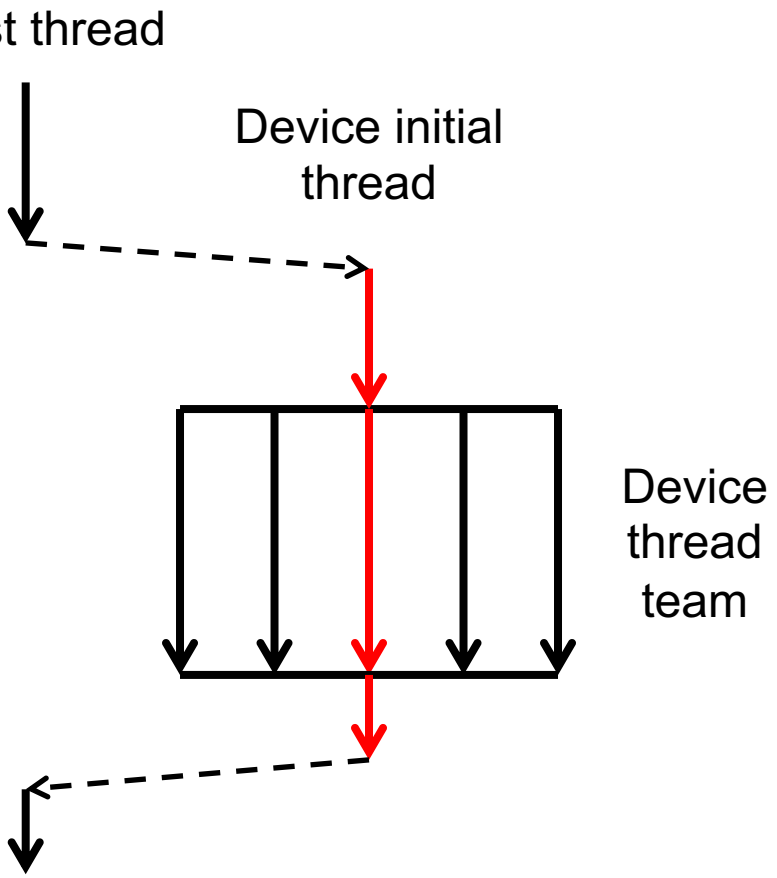
In a thread use SIMD parallelism



Accelerated workshare v1.0

- **Simply add a target construct**

```
#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

Exercise: Accelerate Pi v1.0

- Use the target construct to offload the parallel Pi program to an accelerator.

```
#include <omp.h>
static long num_steps = 100000;
double step;
void main ()
{
      double pi, sum=0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel
      {
          double x,
          #pragma omp for reduction(+:sum)
                for (int i=0; i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
                }
       }
       pi = step * sum;
}
```

Start with a CPU friendly, multithreaded version of the program

- Use Nvprof to compare the offloaded serial pi program we wrote earlier to this offloaded parallel pi program..

teams and distribute constructs

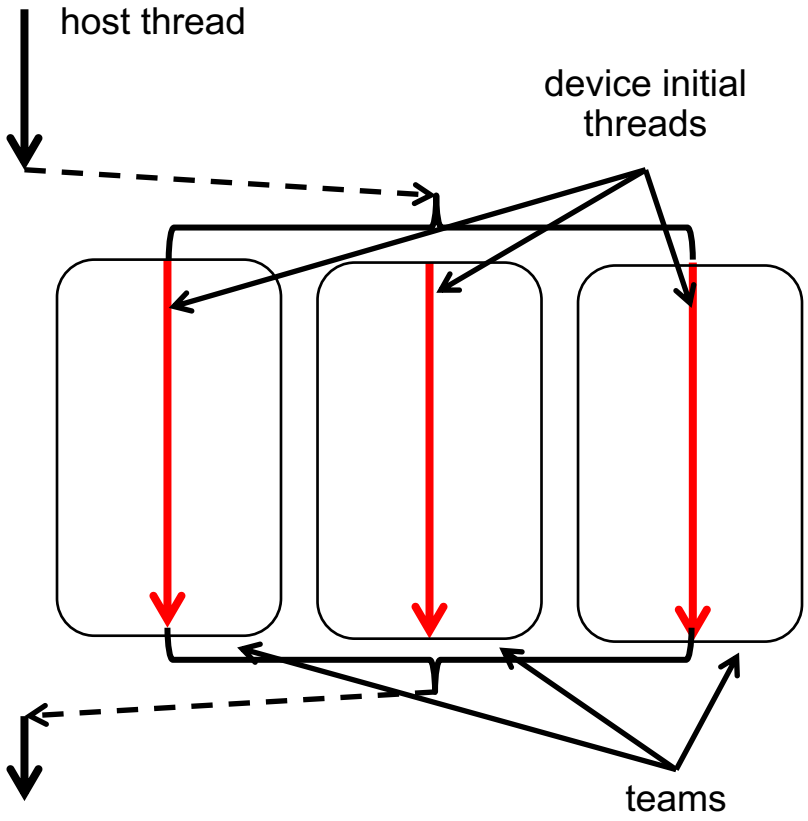
- The **teams** construct
  - Similar to the **parallel** construct
  - It starts a league of thread teams
  - Each team in the league starts as one initial thread – a team of one
  - Threads in different teams cannot synchronize with each other
  - The construct must be “perfectly” nested in a **target**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

- The **distribute** construct
  - Similar to the **for** construct
  - Loop iterations are workshared across the initial threads in a league
  - No implicit barrier at the end of the construct
  - **dist_schedule(*kind[, chunk_size]*)**
    - If specified, scheduling kind must be static
    - Chunks are distributed in round-robin fashion in chunks of size ***chunk_size***
    - If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

Accelerated workshare v2.0

- 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.

Exercise: Accelerate Pi v2.0

- Modify your Pi example to use the **teams** and **distribute** constructs.
- For this exercise do **NOT** use the **parallel** and **for** constructs

```
{
    int i;    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;
      #pragma omp for reduction(+:sum)
            for (i=0; i< num_steps; i++){
                    x = (i+0.5)*step;
                    sum = sum + 4.0/(1.0+x*x);
            }
     }
            pi = step * sum;
}
```

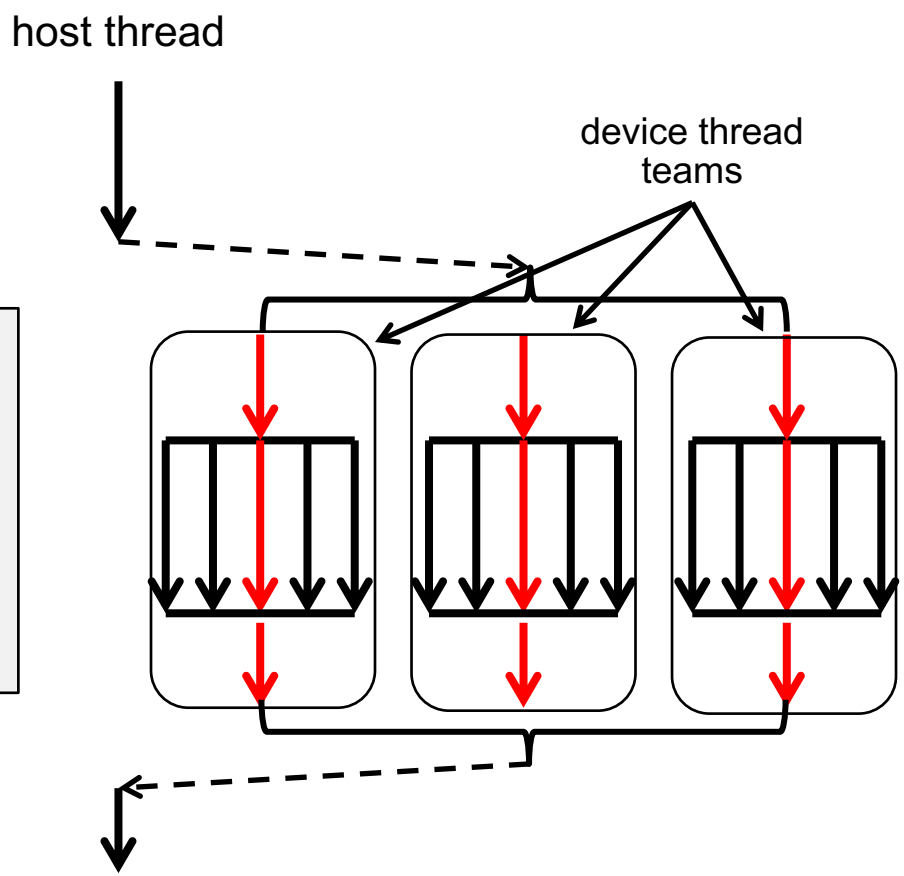
Start with a serial version of the program

- Use Nvprof to understand how this version of the program maps onto a GPU..

Accelerate workshare v3.0

- 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;i++)
   …
```



- Transfer execution control to MULTIPLE device initial threads
  - 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)

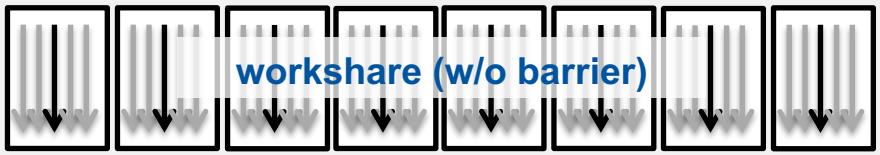
Exercise: Accelerate Pi v3.0

- Modify your Pi example to use **teams**, **distribute** and **parallel for simd** constructs.
  - Hint: omp_get_num_teams() returns the number of teams executing a teams region

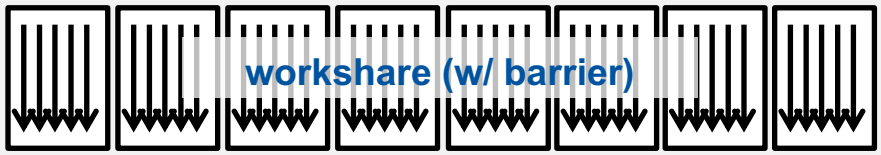
```
{
    int i;    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp target map(sum)
    #pragma omp teams reduction(+: sum)
    {
       double x;
       #pragma omp distribute
             for (int i=0; i< num_steps; i++){
                   x = (i+0.5)*step;
                   sum = sum + 4.0/(1.0+x*x);
             }
     }
     pi = step * sum;
}
```

Multiple thread teams in parallel

```
#pragma omp target map(sum)
#pragma omp teams reduction(+:sum)
{
 int block_size = num_steps/omp_get_num_teams();
#pragma omp distribute dist_sched(static, 1)
  for (int ii = 0; ii < num_steps; ii += block_size){
```




```
#pragma omp parallel for simd reduction(+: sum)
    for (int i = ii; i < MIN(ii+block_size, num_steps); i++) {
```




```
        sum = …
  }}
}
```

Combined 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);
       }
}
```

Accelerate Pi v4.0

- Modify your Pi example to use the **teams distribute parallel for simd** construct.
  - Feel free to update your Pi code and try it…

```
{    int i;     double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp target map(sum)
     #pragma omp teams distribute parallel for simd \
                          reduction(+: sum) private(x)
               for (i=0; i< num_steps; i++){
                          x = (i+0.5)*step;
                          sum = sum + 4.0/(1.0+x*x);
               }
      pi = step * sum;
}
```

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
  - **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).

Exercise:

- Modify your Jacobi solver to use the **teams distribute parallel for simd** construct.
- Compare the runtimes of our jacobi solvers and use Nvprof to understand how the two program execute.

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])
#pragma omp teams distribute parallel for simd private(i,j)
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)
#pragma omp teams distribute parallel for simd  \
                          private(i,tmp) reduction(+: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
```

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)
#pragma omp teams distribute parallel for simd  \
                        private(i,tmp) reduction(+:conv)
for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

} // end while loop
```

This worked but the performance was awful. Why?

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs

Cray® XC40™ Supercomputer running Cray® Compiling Environment 8.5.3.
Intel® Xeon ® CPU E5-2697 v2 @ 2.70GHz with 32 GB DDR3. NVIDIA® Tesla® K20X, 6GB.

Third party names are the property of their owners.

Data movement dominates!!!

```
while((conv > TOLERANCE) && (iters<MAX_ITERS))
  { iters++;
    xnew = iters % s ? x2 : x1;
    xold   = iters % s ? x1 : x2;

    #pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                    map(to:A[0:Ndim*Ndim], b[0:Ndim] )
     #pragma omp teams distribute parallel for simd private(i,j)
     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;
    #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                                map(tofrom:conv)
      #pragma omp teams distribute parallel for private(i,tmp) reduction(+:conv)
      for (i=0; i<Ndim; i++){
         tmp  = xnew[i]-xold[i];
         conv += tmp*tmp;
      }
    conv = sqrt((double)conv);
  }
```

Typically over 4000 iterations!

For each iteration, copy to device $(3*Ndim+Ndim^2)*sizeof(TYPE)$ bytes

For each iteration, copy from device 2*Ndim*sizeof(TYPE) bytes

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

Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Moving memory to/from the device
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability
- 5-point stencil
- OpenMP 5.0

Target data directive

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

```
#pragma omp target data map(to: A,B) map(from: C)
{….}  // a structured block of code
```

- Data is copied into the device data environment at the beginning of the directive and at the end
- Inside the **target data** region, multiple **target** regions can work with the single data region

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

    {do something on the host}

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

Target update directive

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

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

    #pragma omp update from(A)

    host_do_something_with(A)

    #pragma omp update to(A)

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

Set up the data region ahead of time.

Copy A on the device to A on the host.

Copy A on the host to A on the device.

Target enter/exit data constructs

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

- Can achieve similar behavior with two standalone directives:
  #pragma omp target enter data map(…)
  #pragma omp target exit data map(…)

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

Target enter/exit data example

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

int main(void) {

   int N = 1024;
   int *A = malloc(sizeof(int) * N);
   init_array(A, N);

   #pragma omp target teams distribute parallel for simd
   for (int i = 0; i < N; ++i)
      A[i] = A[i] * A[i];

 #pragma omp target exit data map(from: A[0:N])
}
```

Exercise

- Modify your parallel jacobi_solver from the last exercise.
- Use the target data construct to create a data region. Manage 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
  - #pragma omp target map(to:list) map(from:list) map(tofrom:list)
  - #pragma omp parallel for reduction(+:var) private(list)

Jacobi Solver (Par Targ Data, 1/2)

```
#pragma omp target enter data map(to: xold[0:Ndim], xnew[0:Ndim], \
                                      A[0:Ndim*Ndim], b[0:Ndim] )
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;

#pragma omp target
      #pragma omp teams distribute parallel for simd private(j)
    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 Data, 2/2)

```
// test convergence
conv = 0.0;
#pragma omp target map(tofrom: conv)
{
#pragma omp teams distribute parallel for simd  \
                        private(tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
 } // end target region
   conv = sqrt((double)conv);
   TYPE* tmp = xold;
   xold = xnew;
   xnew = tmp;

} // end while loop
 #pragma omp target exit data map(from: xold[0:Ndim], xnew[0:Ndim])
```

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs

Third party names are the property of their owners.

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 motion-clause or one of the following:
  - if(scalar-expression)
  - device(integer-expression)
  - nowait
  - depend (dependence-type : list)
- The motion-clause is one of the following:
  - to(list)
  - from(list)
- nowait and depend apply to the target task running on the host.

Target enter/exit data details

- **#pragma omp target enter data clause[[[,]clause]...]**
- Creates a target task to handle data movement between the host and a device.

- clause is one of the following:
  - if(scalar-expression)
  - device(integer-expression)
  - **nowait**
  - **depend (dependence-type : list)**
  - map (map-type: list)

- **nowait** and **depend** apply to the target task running on the host.

A note about the nowait clause

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

```
void init_array(int *A, int N) {
  for (int i = 0; i < N; ++i) A[i] = i;
 #pragma omp target enter data map(to: A[0:N]) nowait depend(out: A)
}

int main(void) {
  int N = 1024; int *A = malloc(sizeof(int) * N);
  init_array(A, N);

  #pragma omp target teams distribute parallel for simd    \
                                                nowait depend(inout: A)
  for (int i = 0; i < N; ++i) A[i] = A[i] * A[i];

  #pragma omp taskwait

  #pragma omp target exit data map(from: A[0:N])
}
```

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.

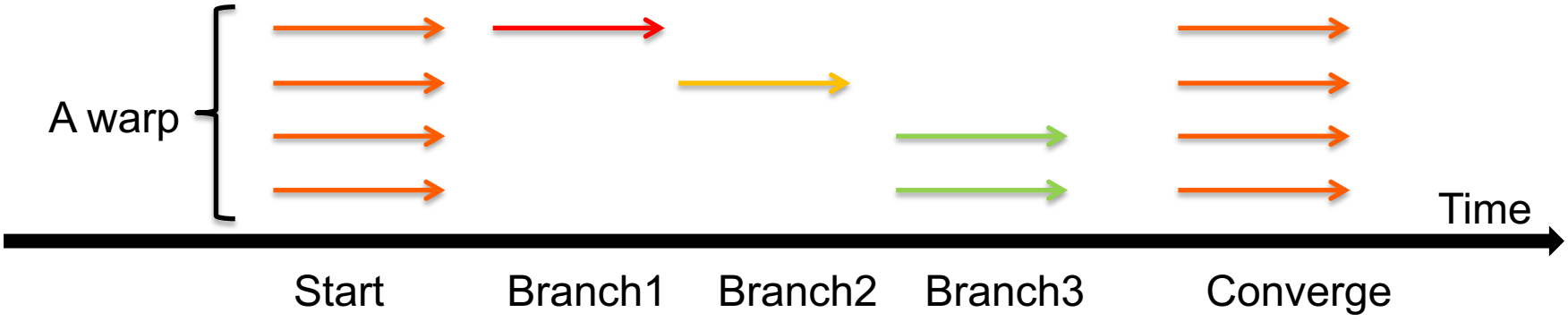
- Use the **target update** construct to transfer data on the fly within a **target data** region or between **target enter/exit data** directives.

Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Moving memory to/from the device
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability
- 5-point stencil
- OpenMP 5.0

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 SPMD pattern.
- Branch behavior
  - Each branch will be executed serially
  - Work-items not following the current branch will be disabled



Branching

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

Branching

Conditional execution

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

Selection and masking

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

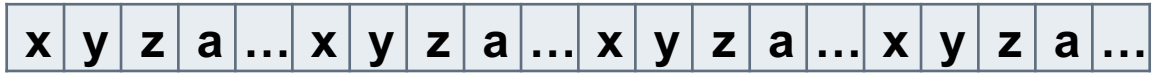
Coalesced Access

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

Memory layout is critical to performance

- "Structure of Arrays vs. Array of Structures"
  - Array of Structures (AoS) more natural to code

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



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

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

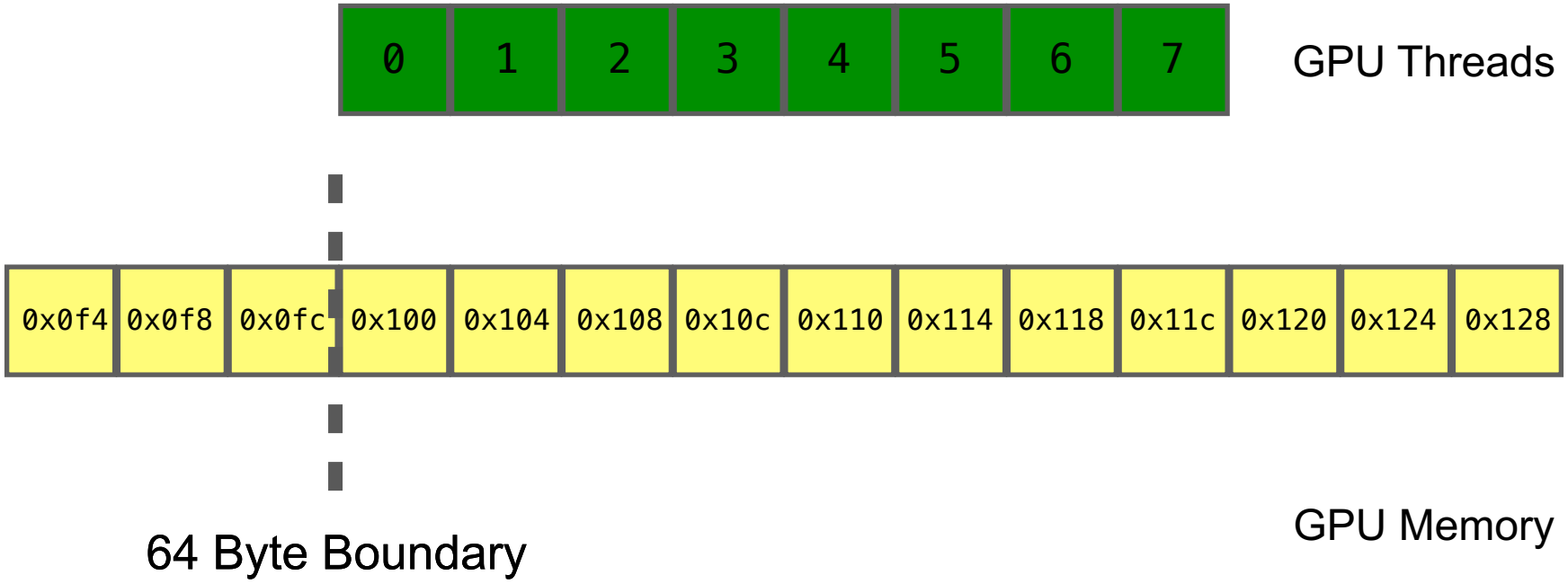
Adjacent work-items/vector-lanes like to access adjacent memory locations

Coalescence

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

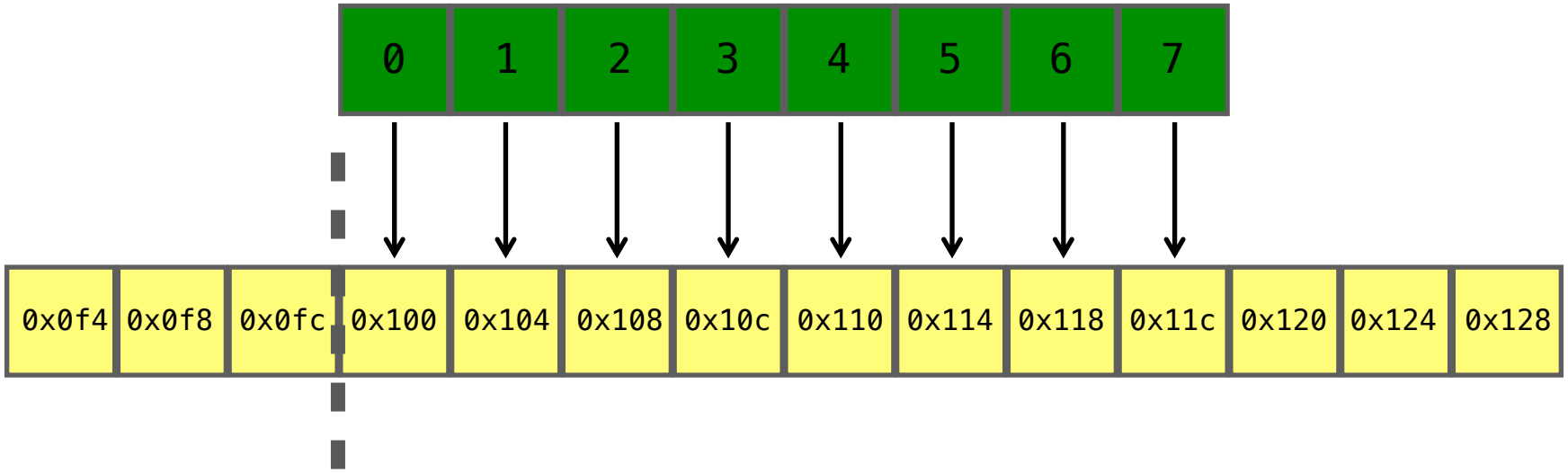
```
for (int id = 0; id < size; id++)
{
  // ideal
     float val1 = memA[id];

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

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

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

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

Memory access patterns



Memory access patterns

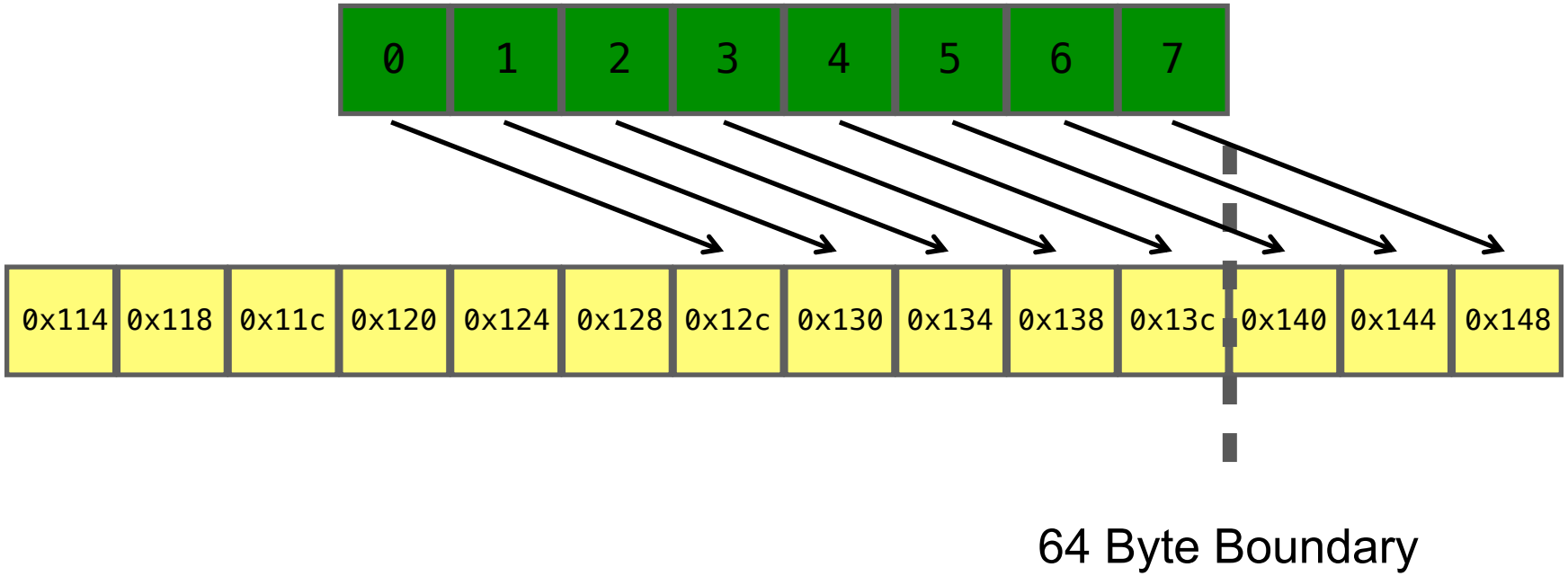
```
float val1 = memA[id];
```



64 Byte Boundary

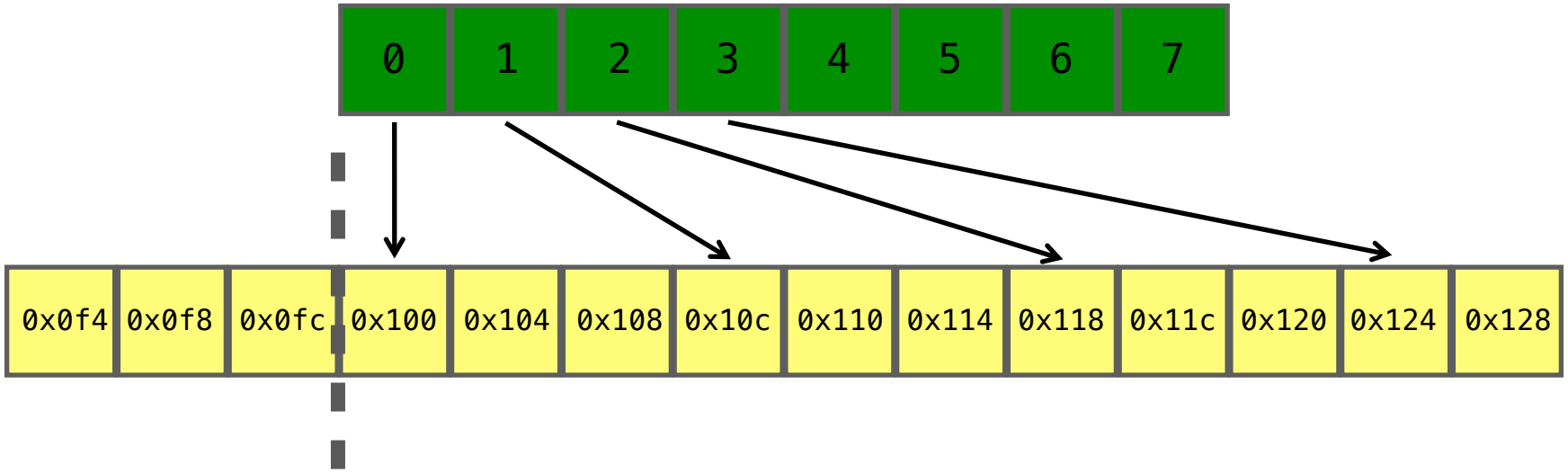
Memory access patterns

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



Memory access patterns

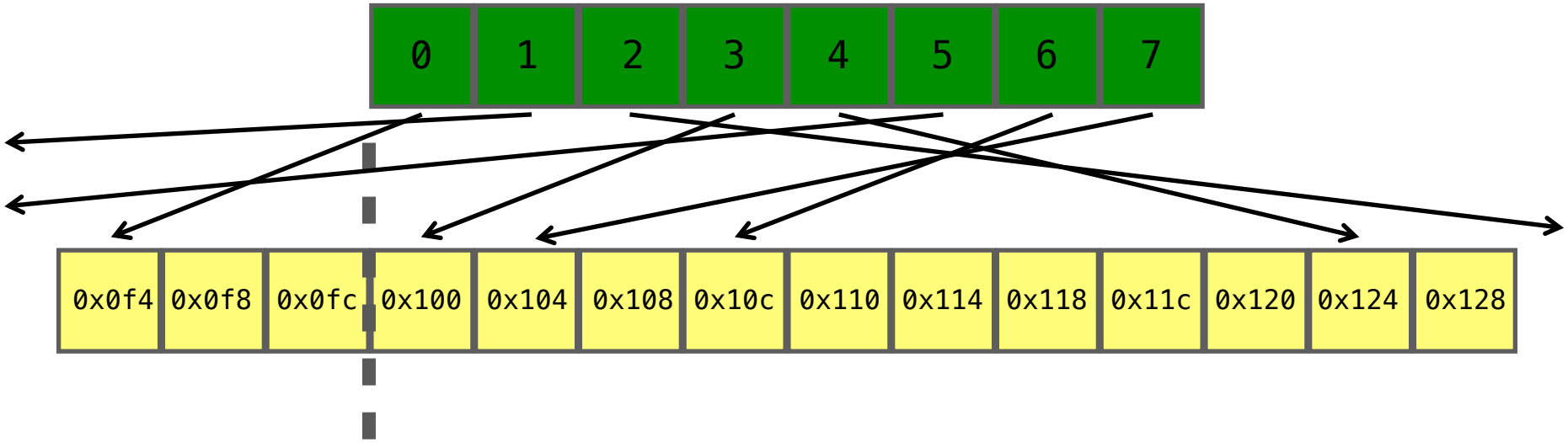
```
float val3 = memA[3*id];
```



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

Memory access patterns

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

float val4 = memA[loc];
```



64 Byte Boundary

Exercise

- Modify your parallel jacobi_solver from the last exercise.
- Experiment with the optimizations we've discussed.
  - #pragma omp target
  - #pragma omp target data
  - #pragma omp target map(to:list) map(from:list) map(tofrom:list)
  - #pragma omp parallel for reduction(+:var) private(list)

- Note: if you want to generate a transposed A matrix to try a different memory layout, you can use the following function from mm_utils
  - void init_diag_dom_near_identity_matrix_colmaj(int Ndim, TYPE *A)

Jacobi Solver (Par Targ Data, 1/2)

```
#pragma omp target enter data map(to: xold[0:Ndim], xnew[0:Ndim], \
                                      A[0:Ndim*Ndim], b[0:Ndim] )
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;

#pragma omp target
      #pragma omp teams distribute parallel for simd private(j)
    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            xnew[i]+= (A[i*Ndim + j]*xold[j])*((TYPE)(i != j));
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (Par Targ Data, 2/2)

```
// test convergence
conv = 0.0;
#pragma omp target map(tofrom: conv)
 #pragma omp teams distribute parallel for simd  \
                        private(tmp) reduction(+: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
```

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs
	Above plus reduced branching	13.74 secs

```
#pragma omp target exit data map(from: xold[0:Ndim], xnew[0:Ndim])
```

Cray® XC40™ Supercomputer running Cray® Compiling Environment 8.5.3.
Intel® Xeon ® CPU E5-2697 v2 @ 2.70GHz with 32 GB DDR3. NVIDIA® Tesla® K20X, 6GB.

Third party names are the property of their owners.

Jacobi Solver (Targ Data/branchless/coalesced mem, 1/2)

```
#pragma omp target enter data map(to: xold[0:Ndim], xnew[0:Ndim], \
                                      A[0:Ndim*Ndim], b[0:Ndim] )
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
#pragma omp target
      #pragma omp teams distribute parallel for simd private(j)
    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
             xnew[i]+= (A[j*Ndim + i]*xold[j])*((TYPE)(i != j));
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

We replaced the original code with a poor memory access pattern

```
xnew[i]+= (A[i*Ndim + j]*xold[j])
```

With the more efficient

```
xnew[i]+= (A[j*Ndim + i]*xold[j])
```

Jacobi Solver (Targ Data/branchless/coalesced mem, 2/2)

```
    //
    // test convergence
    conv = 0.0;
#pragma omp target map(tofrom: conv)
 #pragma omp teams distribute parallel for simd  \
                        private(tmp) reduction(+: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
```

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs
	Above plus reduced branching	13.74 secs
	Above plus improved mem access	7.64 secs

```
#pragma omp target exit data map(from: xold[0:Ndim], xnew[0:Ndim])
```

Cray® XC40™ Supercomputer running Cray® Compiling Environment 8.5.3.
Intel® Xeon ® CPU E5-2697 v2 @ 2.70GHz with 32 GB DDR3. NVIDIA® Tesla® K20X, 6GB.

Third party names are the property of their owners.

Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Moving memory to/from the device
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability
- 5-point stencil
- OpenMP 5.0

Compiler Support

- **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.
- **Cray** provided the first vendor supported implementation targeting NVIDIA GPUs late 2015. The latest version of CCE now supports all of OpenMP 4.5.
- **IBM** has recently completed a compiler implementation using Clang, that fully supports OpenMP 4.5. This is being introduced into the Clang main trunk.
- **Clang** 7.0 supports OpenMP 4.5 offload to NVIDIA GPUs.
- **GCC 6.1** introduced support for OpenMP 4.5, and can target Intel Xeon Phi, or HSA-enabled AMD GPUs. V7 added support for NVIDIA GPUs.
- **PGI** compilers don't currently support OpenMP on GPUs (but they do for CPUs).

OpenMP compiler information: http://www.openmp.org/resources/openmp-compilers/

Third party names are the property of their owners.

Performance?

- 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 - we were able to achieve great performance on Intel Xeon Phi Knights Corner, but didn't know what to expect on GPU.

Third party names are the property of their owners.

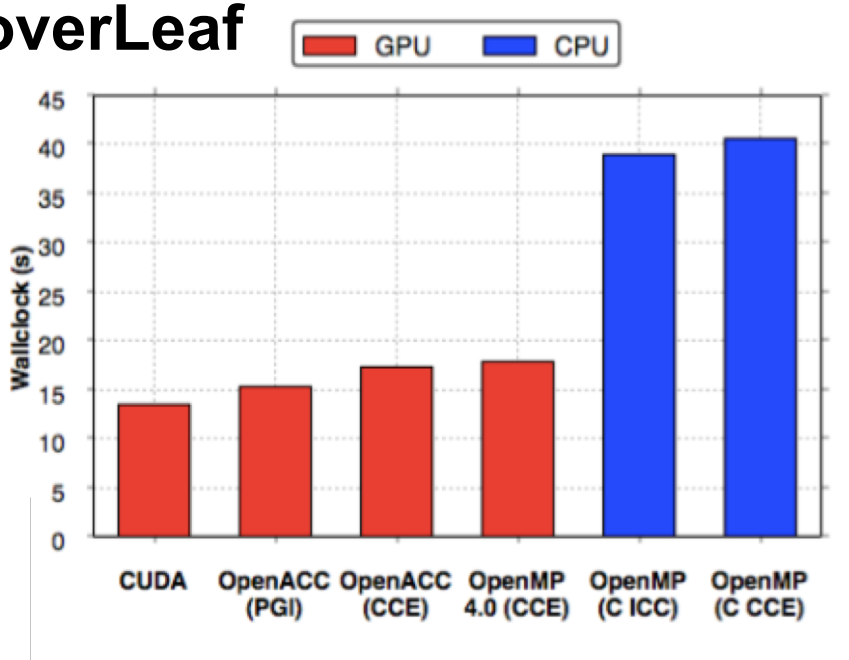
Performance?

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

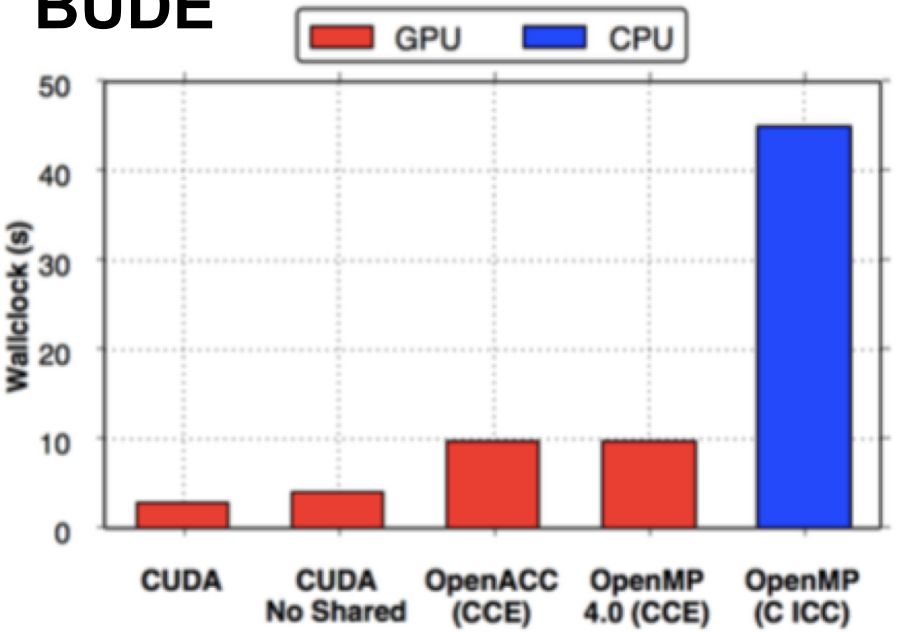
Immediately we see impressive performance compared to CUDA

Clearly the Cray compiler leverages the existing OpenACC backend

CloverLeaf



BUDE



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

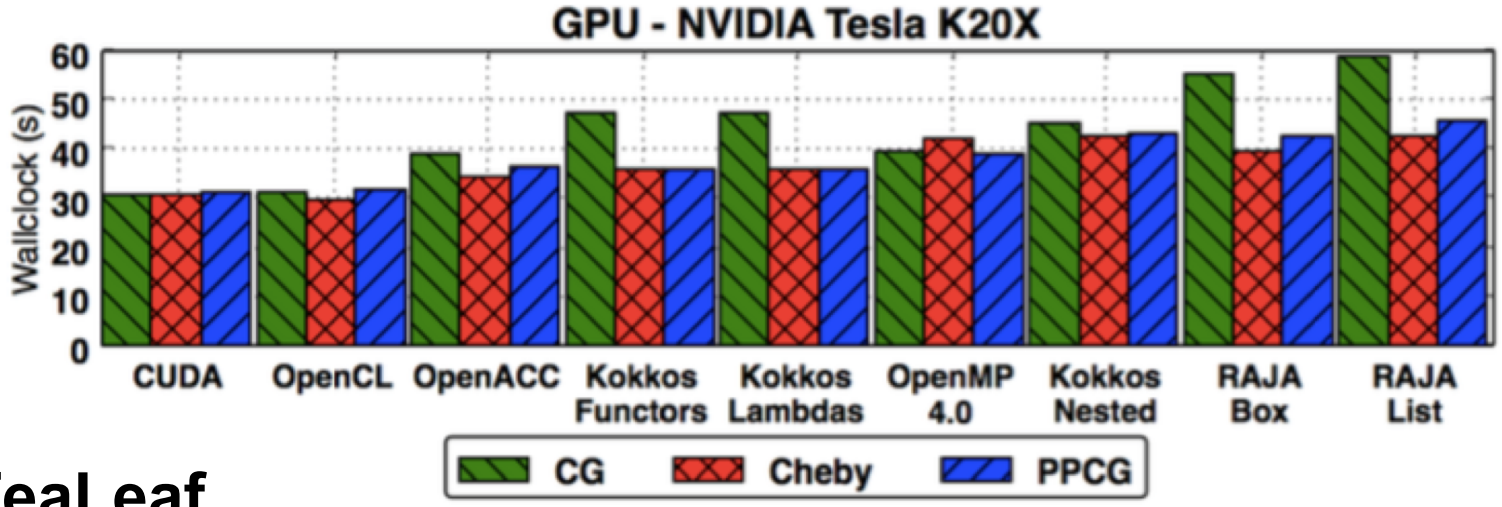
This is set to come in with OpenMP 5.0, and Clang supports targeting address spaces directly

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

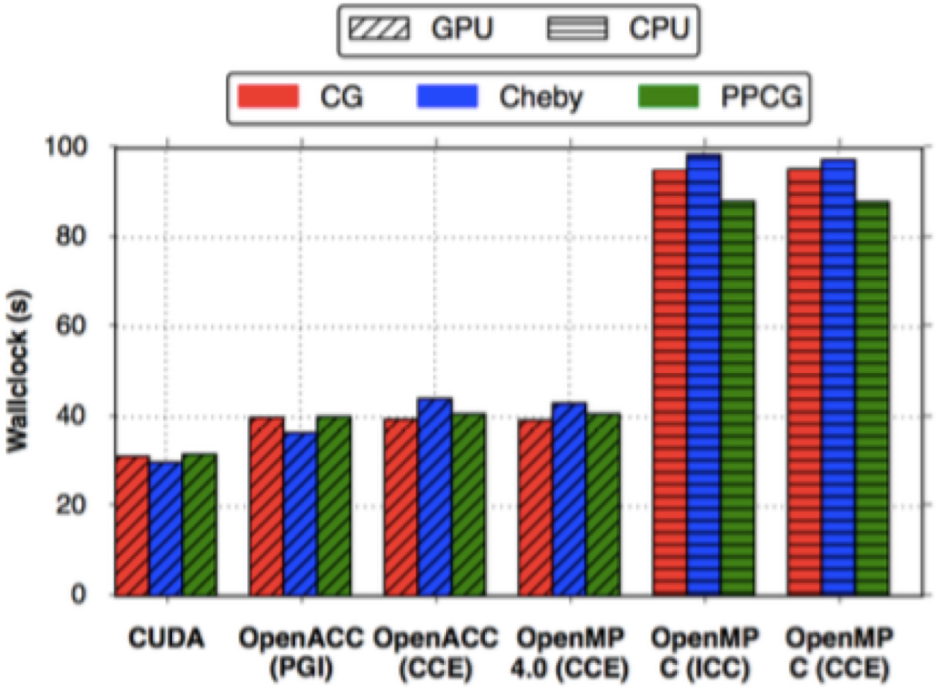
Third party names are the property of their owners.

Performance?

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



TeaLeaf



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

We see these figures continually improving as the languages mature

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

Third party names are the property of their owners.

How do you get good performance?

- Our finding so far: *You can achieve good performance on GPUs with OpenMP 4.x.*

- We achieved this by:

  - Keeping data resident on the device for the greatest possible time.

  - Collapsing loops with the **collapse** clause, so there was a large enough iteration space to saturate the device.

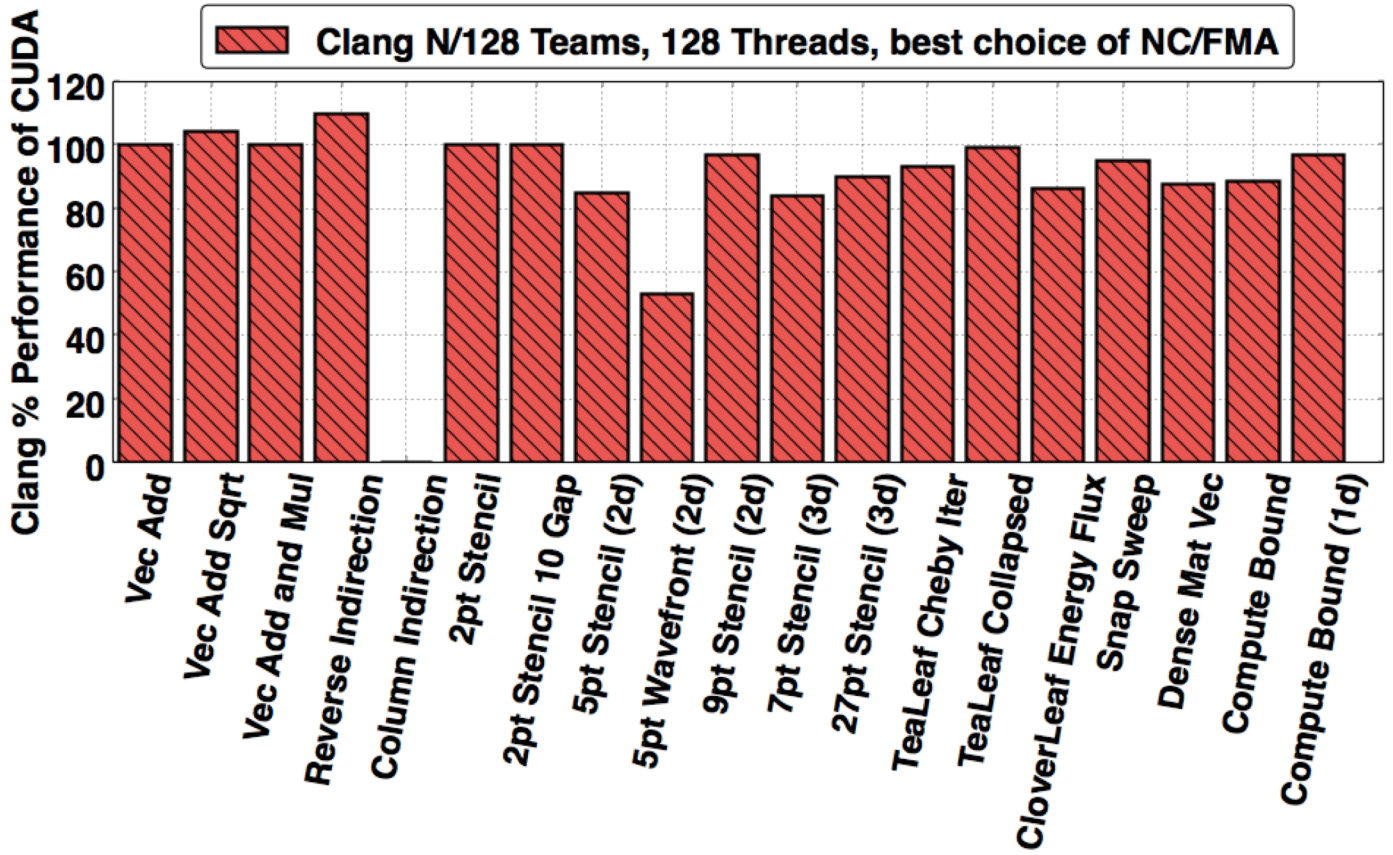
  - Using the **simd** directive to vectorize inner loops.

  - Using **schedule(static, 1)** for coalescence (obsolete).

  - Using *nvprof* of course.

Can you do better?

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



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

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

Third party names are the property of their owners.

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).

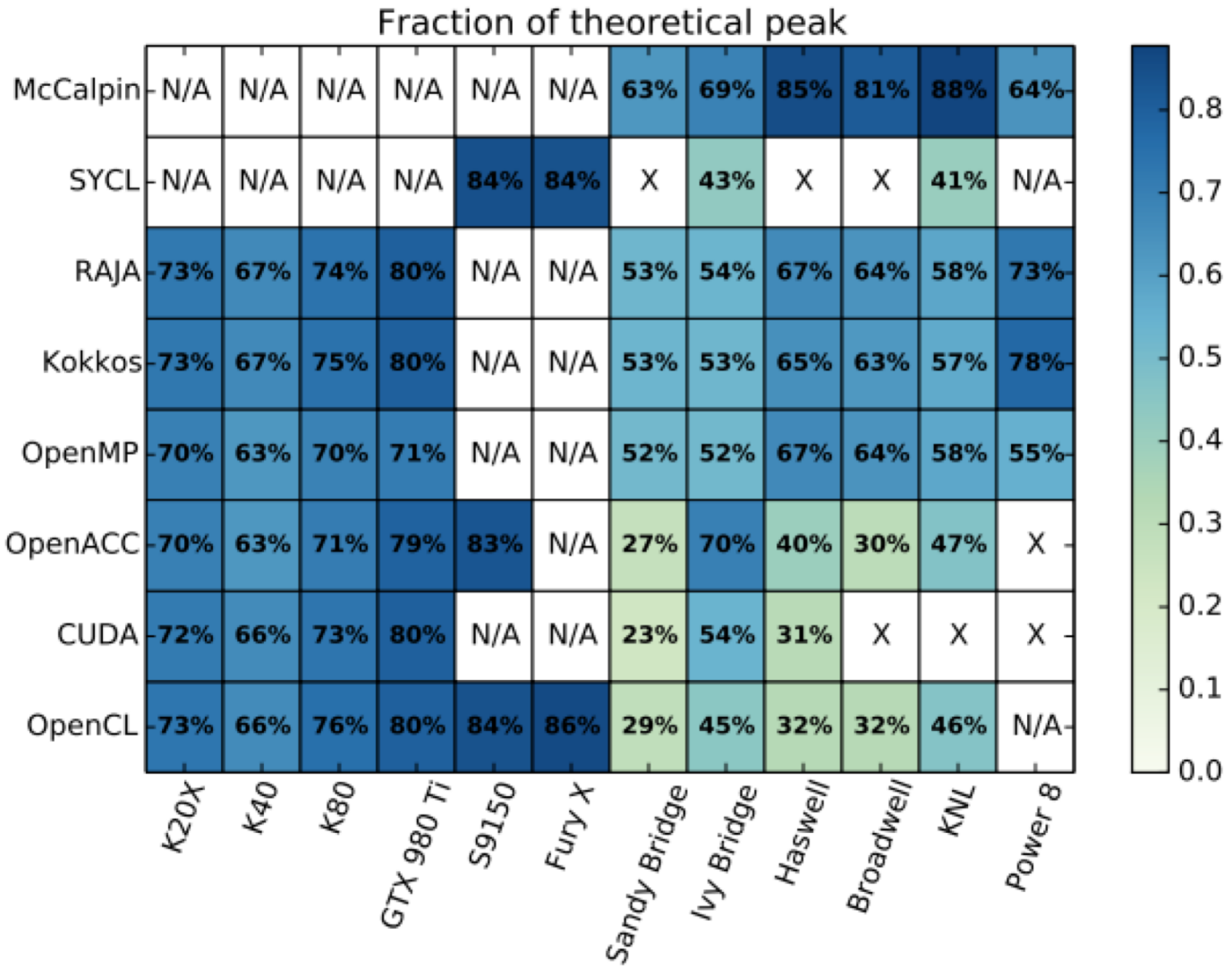
- The compiler support continually changes, improving performance, correctness and introducing new architectures.

- We keep tracking this improvement over time.

Third party names are the property of their owners.

OpenMP in The Matrix.

System details on the following slide



On those supported target architectures, OpenMP achieves good performance

Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S., *GPU-STREAM v2.0 Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models, ISC'16*

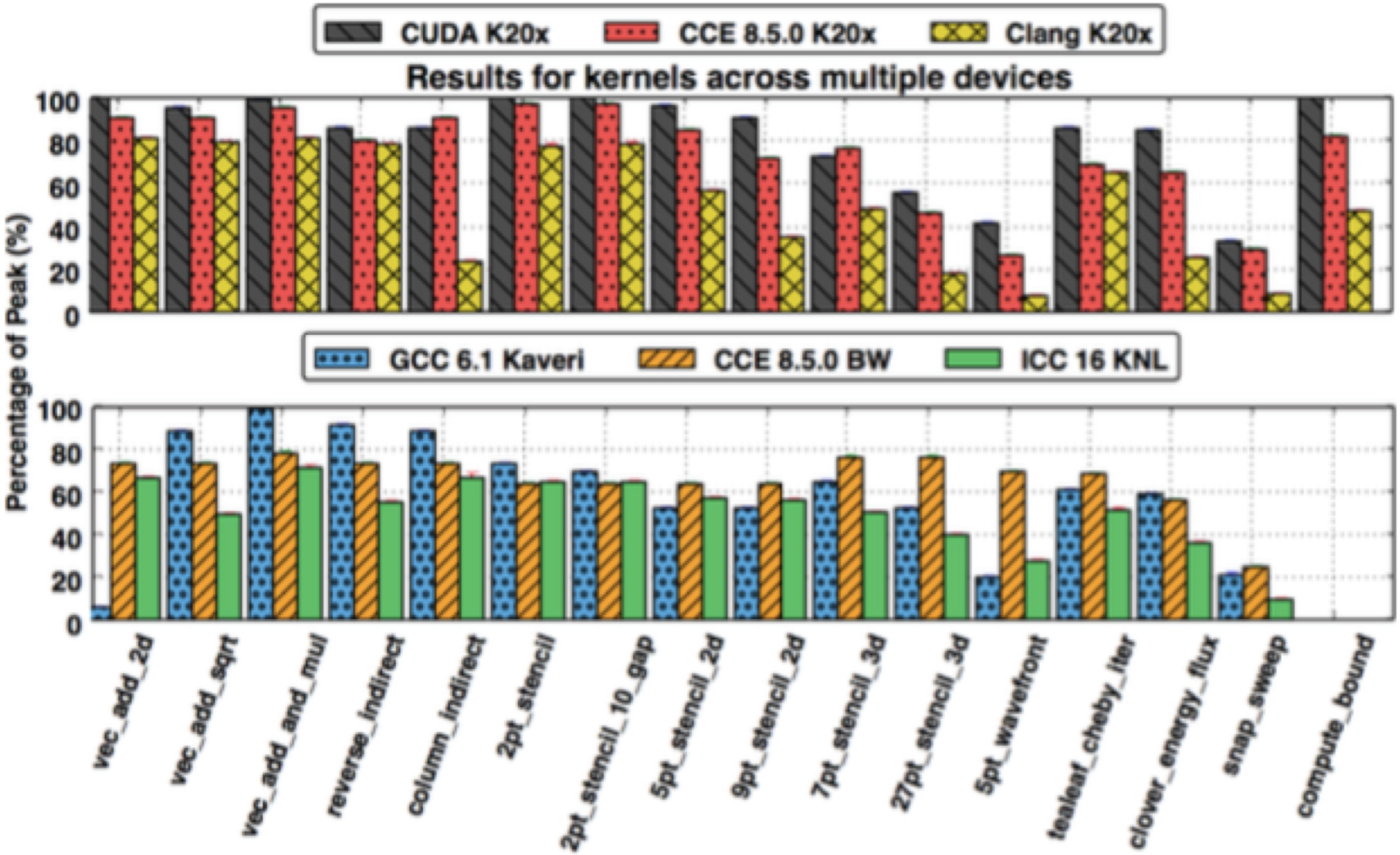
Third party names are the property of their owners.

System details

Abbreviation	System details
K20X	Cray® XC40, NVIDIA® K20X GPU, Cray compilers version 8.5, gnu 5.3, CUDA 7.5
K40	Cray® CS cluster, NVIDIA® K40 GPU, Cray compilers version 8.4, gnu 4.9, CUDA 7.5
K80	Cray® CS cluster, NVIDIA® K40 GPU, Cray compilers version 8.4, gnu 4.9, CUDA 7.5
S9150	AMD® S9150 GPU. Codeplay® copmputeCpp compiler 2016.05 pre-release. AMD-APP OpenCL 1.2 (1912.5)drivers for SyCL. PGI® Accelerator)TM) 16.4 OpenACC
GTX 980 Ti	NVIDA® GTX 980 Ti. Clang-ykt fork of Clang for OpenMP. PGI® Accelerator™ 16.4 OpenACC. CUDA 7.5
Fury X	AMD® Fury X GPU (based on the Fiji architecture).
Sandy Bridge	Intel® Xeon® E5-2670 CPU. Intel® compilers release 16.0. PGI® Accelerator)TM) 16.4 OpenACC and CUDA-x86. Intel® OpenCL runtime 15.1. Codeplay® copmputeCpp compiler 2016.05 pre-release
Ivy Bridge	Intel® Xeon® E5-2697 CPU. Gnu 4.8 for RAJA and Kokkos, Intel® compiler version 16.0 for stream, Intel® OpenCL runtime 15.1. Codeplay® copmputeCpp compiler 2016.05 pre-release.
Haswell	Cray® XC40, Intel® Xeon® E5-2698 CPU. Intel® compilers release 16.0. PGI® Accelerator)TM) 16.3 OpenACC and CUDA-x86. Gnu 4.8 for RAJA and Kokkos
Broadwell	Cray® XC40, Intel® Xeon® E5-2699 CPU. Intel® compilers release 16.0. PGI® Accelerator)TM) 16.3 OpenACC and CUDA-x86. Gnu 4.8 for RAJA and Kokkos
KNL	Intel® Xeon® Phi™7210 (knights landing) Intel® compilers release 16.0. PGI® Accelerator)TM) 16.4 OpenACC with target specified as AVX2.
Power 8	IBM® Power 8 processor with the XL 13.1 compiler.

Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S., *GPU-STREAM v2.0 Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models, ISC'16*

Nice - but beware of the caveat.



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

Martineau, M., McIntosh-Smith, S. Gaudin, W., *Pragmatic Performance Portability with OpenMP 4.x, 2016, IWOMP'16*

Third party names are the property of their owners.

The worst case scenarios.

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

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

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

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

Four different ways of writing for the same kernel…

The answer:

```
#pragma omp target teams distribute parallel for simd
for(...) {
}
```

If you can - just use the combined construct!

- The 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

```
#pragma omp target teams distribute parallel for simd
for(...) {
}
```

If you can - just use the combined construct!

- *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

```
#pragma omp target teams distribute parallel for simd
for(...) {
}
```

If you can - just use the combined construct!

- 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 short, but be aware that it can have a negative effect on CPU performance.
- Use the combined construct whenever you can.

Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Moving memory to/from the device
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability
- 5-point stencil
- OpenMP 5.0

5-point stencil: the heat program

- Heat equation models the 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:

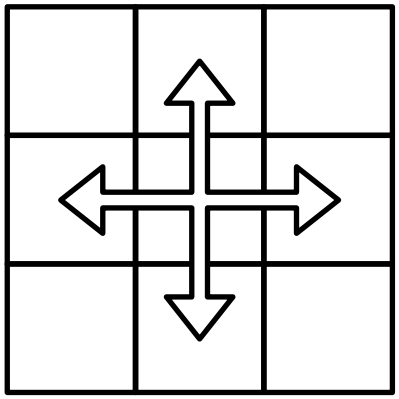
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

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

  - 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, 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.

Method of Manufactured Solution

- Stencil codes are notoriously difficult to **know** if the answer is “correct”.

- Analytic solutions hard to come by:
  - It’s why you’re using a computer to solve the equation approximately after all!

- Method of Manufactured Solution (MMS) is a way to help determine if the code does the correct thing.
- An approach often used to find errors in CFD codes and check convergence properties.

Method of Manufactured Solution

- **Choose** a function for $u(t, x, y)$, substitute into the equation and work through the algebra.

- Ideally like the equation to evaluate to zero so don't need to consider a right-hand side to the equation.

- $u(0, x, y)$ gives the initial conditions.
- Can evaluate boundary conditions, e.g. bottom boundary $u(0,0, y)$

- Because u is **known** for all timesteps (it was chosen!), the exact solution is **known**.

- Compare the **computed** solution to the known u to compute an error.
- Any differences come from approximations in the method, or a bug in your code.

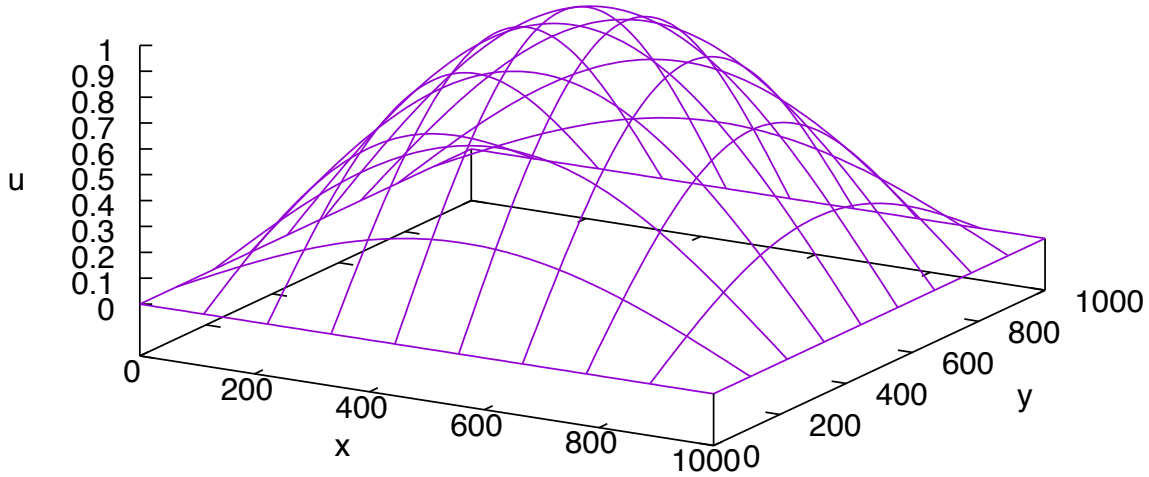
Method of Manufactured Solution

- For the problem of length l, choose u:

$$u(t,x,y) = e^{\frac{-2\alpha\pi^2 t}{l^2}} \sin\frac{\pi x}{l} \sin\frac{\pi y}{l}$$

- Boundary conditions: u is always zero on the boundaries

- Initial value of grid is then $u(0,x,y) = \sin\frac{\pi x}{l} \sin\frac{\pi y}{l}$



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 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);
    }
  }
}
```

Exercise: heat program

- Start from the provided heat program. Verify that you can run it serially.
  - A good problem to start with is ./heat 1000 10
    - 1000x1000 cells, 10 timesteps.
  - If you try other problems, make sure r < 0.5. The code prints a warning if it isn't.

- Parallelize for a CPU using the *parallel for* construct on the major loops.

- Use *target* directives to run on a GPU.

Solution (1/2): moving data to/from the device

```
// Allocate two nxn grids
  double *u     = malloc(sizeof(double)*n*n);
  double *u_tmp = malloc(sizeof(double)*n*n);
  double *tmp;

  // Set the initial value of the grid under the MMS scheme
  initial_value(n, dx, length, u);
  zero(n, u_tmp);

  // Copy data to the device
#pragma omp target enter data map(to: u[0:n*n], u_tmp[0:n*n])

// … timestep code omitted …

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

Solution (2/2): target directives on 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
#pragma omp target teams distribute parallel for simd collapse(2)
  for (int j = 0; j < n; ++j) {
    for (int i = 0; i < n; ++i) {

      // Update the 5-point stencil, using boundary conditions on the edges of the domain.
      // Boundaries are zero because the MMS solution is zero there.
      u_tmp[i+j*n] =  r2 * u[i+j*n] +
      r * ((i < n-1) ? u[i+1+j*n] : 0.0) +
      r * ((i > 0)   ? u[i-1+j*n] : 0.0) +
      r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
      r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
    }
  }
}
```

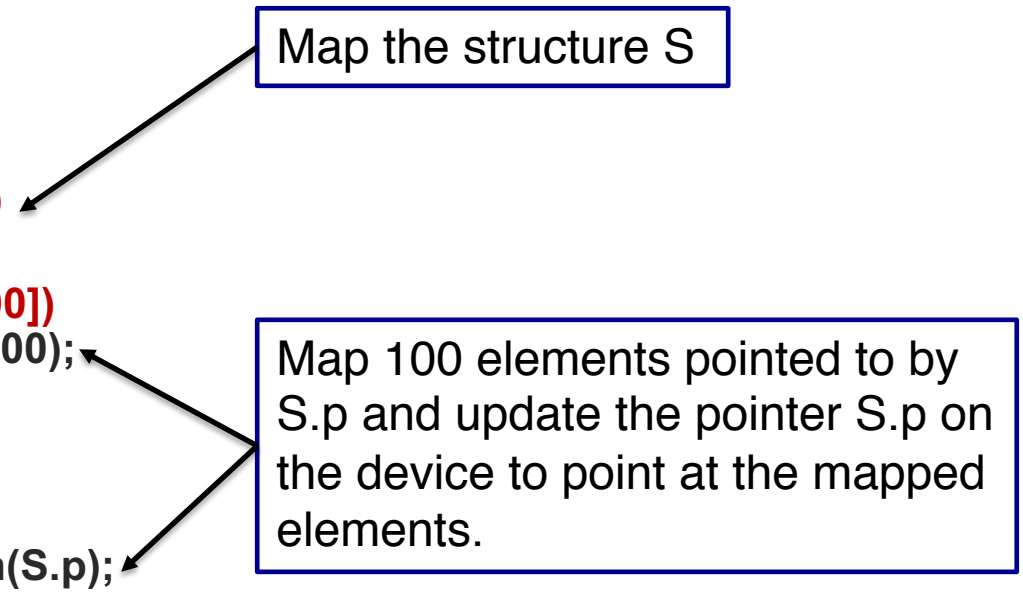
Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Moving memory to/from the device
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability
- 5-point stencil
- OpenMP 5.0

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);
}
```

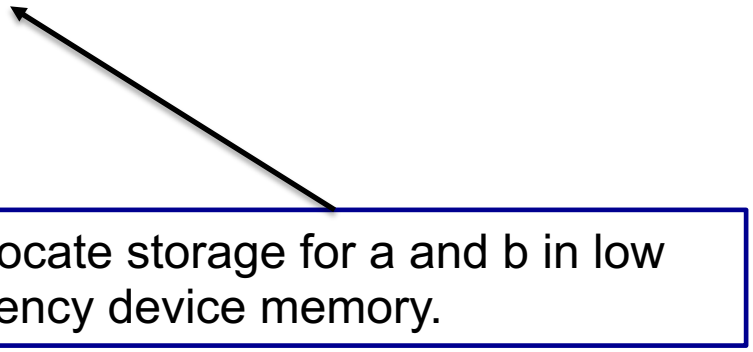
This code assumes that the host and device share memory.

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

OpenMP 5.0: memory allocators

- Memory spaces – storage resources where variables can be stored and retrieved.
  - E.G. omp_default_mem_space(default), omp_high_bw_mem_space (high bandwidth), omp_low_lat_mem_space (low latency), …
- Memory Allocators –Methods for making allocation requests from memory spaces.

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

#pragma omp target allocate(omp_low_lat_mem_alloc) \
                    map(a) firstprivate(b, c)
{
  for (int i=0; i<N; i++)
    a[i] = F(b[i], c[i]);
}
```



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 fastFUNC(double, double);

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

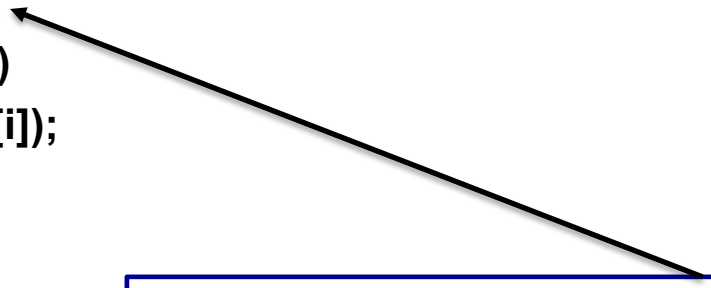
Declare fastFUNC as a variant for FUNC when executing in a target region.

Call fastFUNC here instead of FUNC

OpenMP 5.0: loop construct

- Assert that the iterations in a loop nest may execute in any order, including concurrently
  - Let the compiler figure our how to best utilize parallel resources

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

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

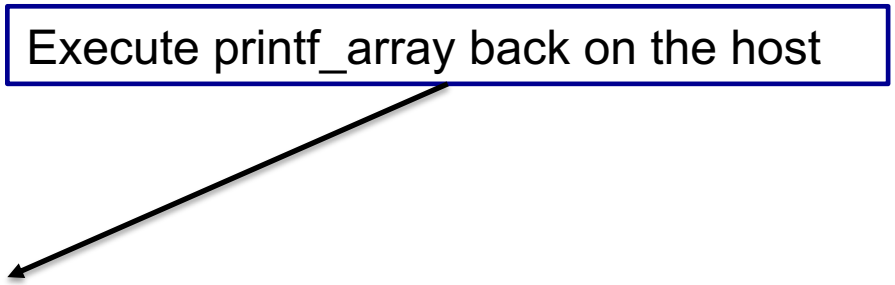


Iterations can execute in any order. Rely on the compiler to schedule iterations across teams, threads, simd, …

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 misc.

- 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…

Please tell our SC tutorial overlords how amazingly GREAT this tutorial is!!!!

Online feedback forms available through the following URL or QR code

Evaluation site URL: http://bit.ly/sc18-eval

Conclusion

- You can program your GPU with OpenMP
  - There is no reason to use proprietary “standards”.
- Implementations of OpenMP supporting target devices are evolving rapidly … expect to see great improvements in quality and the range of devices supported.

To learn more about programming your GPU with OpenMP, get a copy of this awesome book!