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

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

1

Disclaimer & Optimization Notice

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

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	• Vadd program
Understanding execution	• Intro to nvprof	• nvprof ./vadd
Working with the target directive	• CPU and GPU execution models and the fundamental combined directive	• Vadd with combined directive using nvprof to understand execution
Basic memory movement	• The map clause	• Jacobi solver
Optimizing memory movement	• Target data regions	• Jacobi with explicit data movement
Optimizing GPU code	• Common GPU optimizations	• Optimized Jacobi solver

Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

OpenMP* overview:

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP_SET_NUM_THREADS(10)

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

OpenMP: An API for Writing Multithreaded Applications

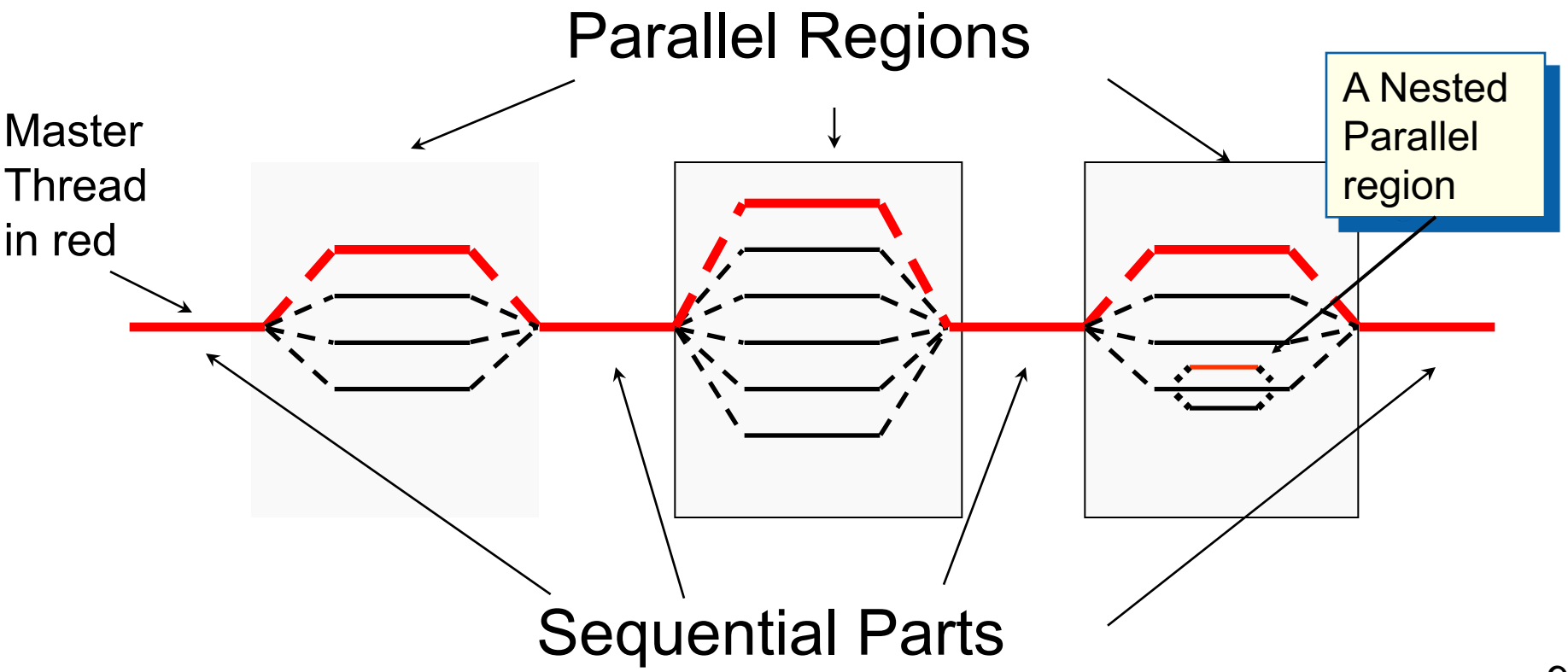
- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

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

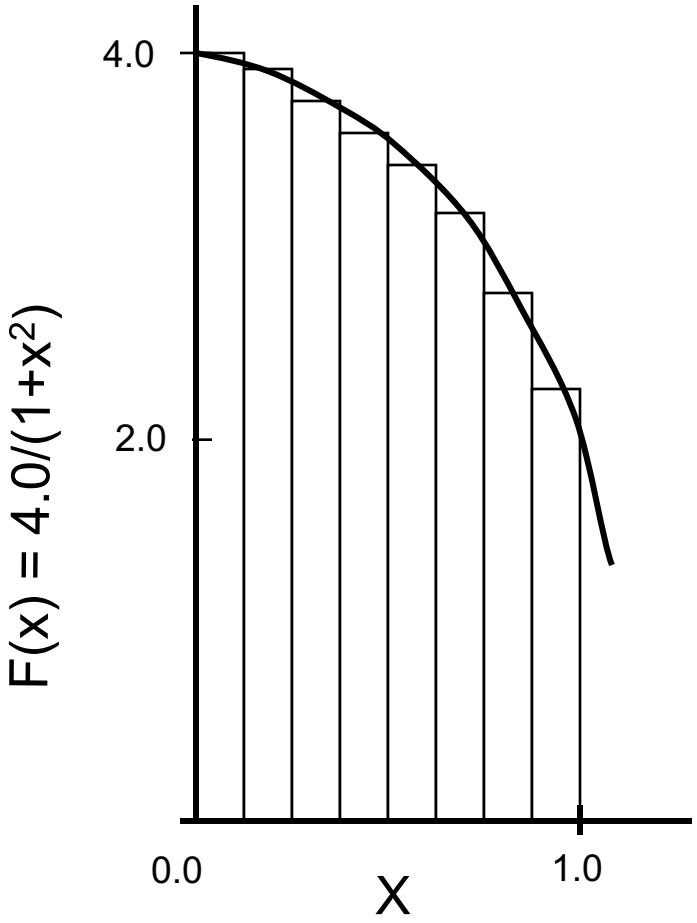
OpenMP programming model:

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.



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 sc17-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.

Results*: pi with a loop and a reduction

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: Pi with a loop and a reduction

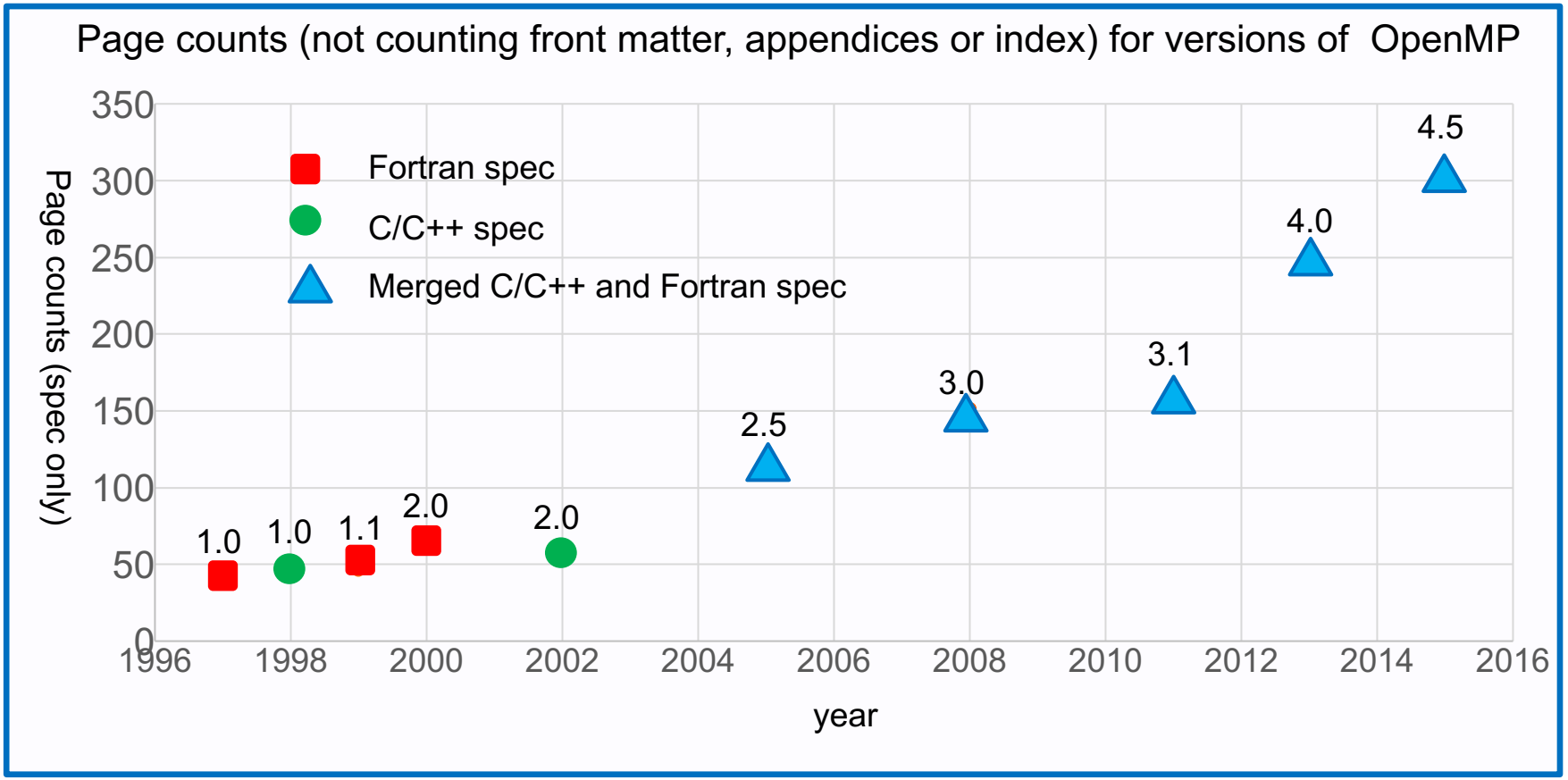
```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{   int i;          double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;
      #pragma omp for reduction(+:sum)
          for (i=0;i< num_steps; i++){
       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

threads	PI Loop (s)
1	1.91
2	1.02
3	0.80
4	0.68

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

The growth of complexity in OpenMP

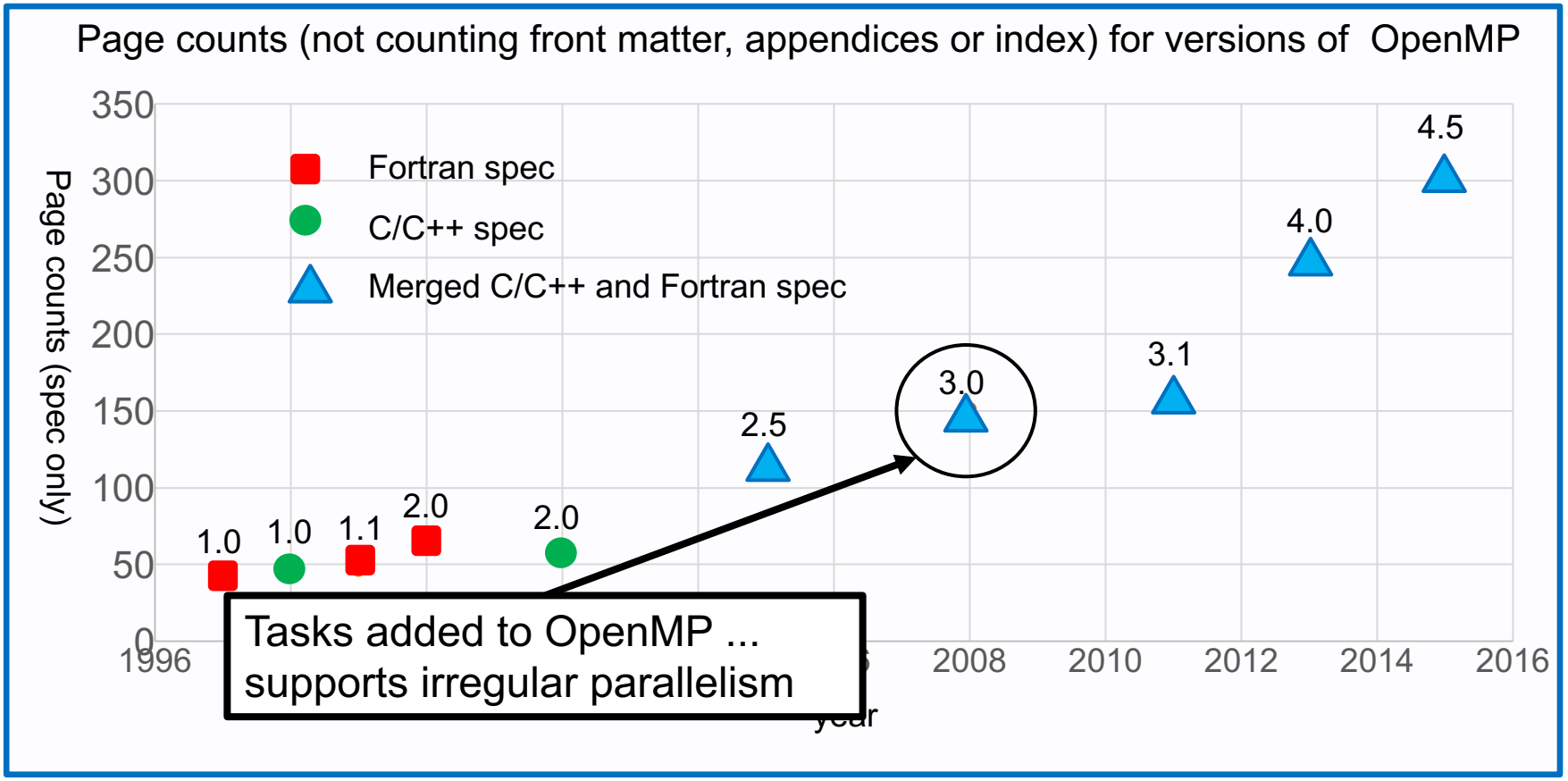
- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.
- The complexity has grown considerably over the years!



The complexity of the full spec is overwhelming, so we focus on the 16 constructs most OpenMP programmers restrict themselves to … the so called “OpenMP Common Core”

The growth of complexity in OpenMP

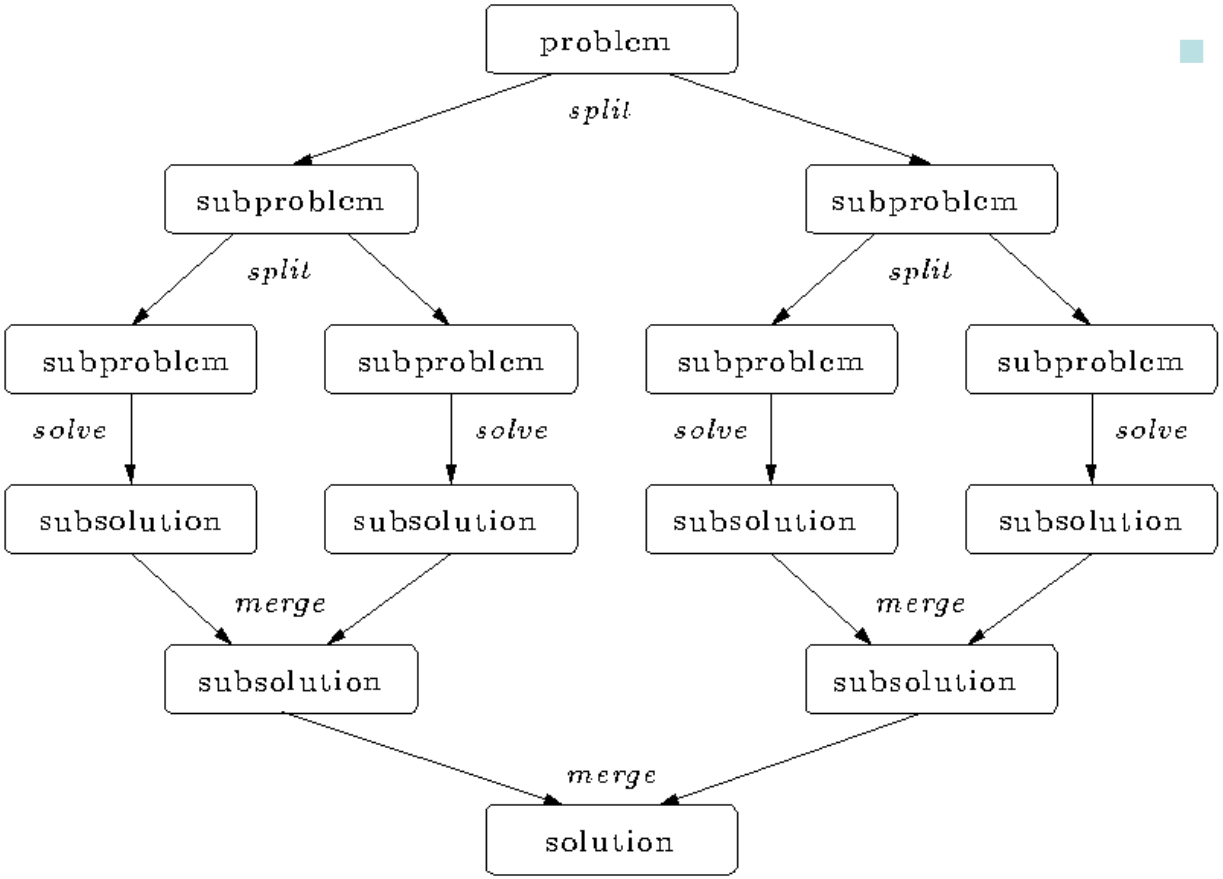
- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.
- The complexity has grown considerably over the years!



The complexity of the full spec is overwhelming, so we focus on the 19 constructs most OpenMP programmers restrict themselves to … the so called “OpenMP Common Core”

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: OpenMP tasks

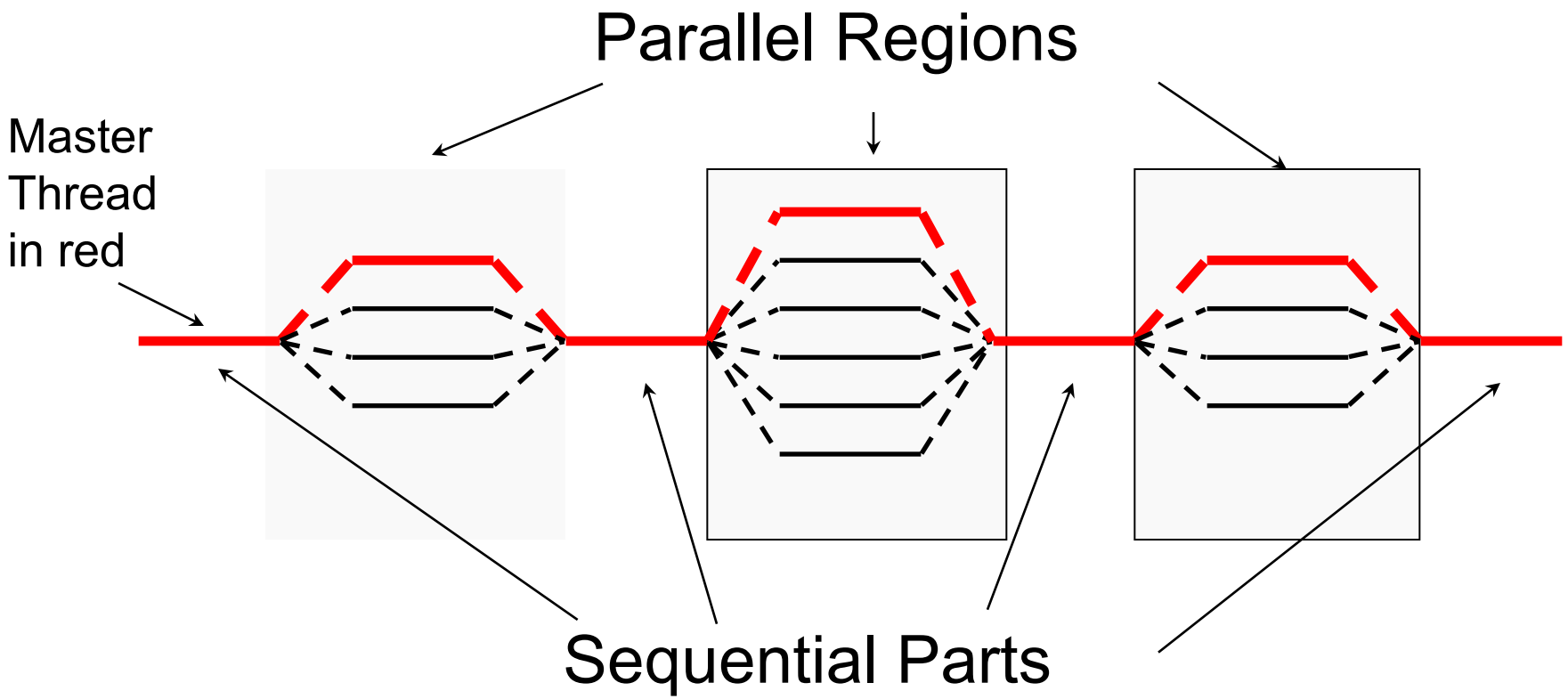
```
#include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
   double x, sum = 0.0,sum1, sum2;
   if (Nfinish-Nstart < MIN_BLK){
      for (i=Nstart;i< Nfinish; i++){
         x = (i+0.5)*step;
         sum = sum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

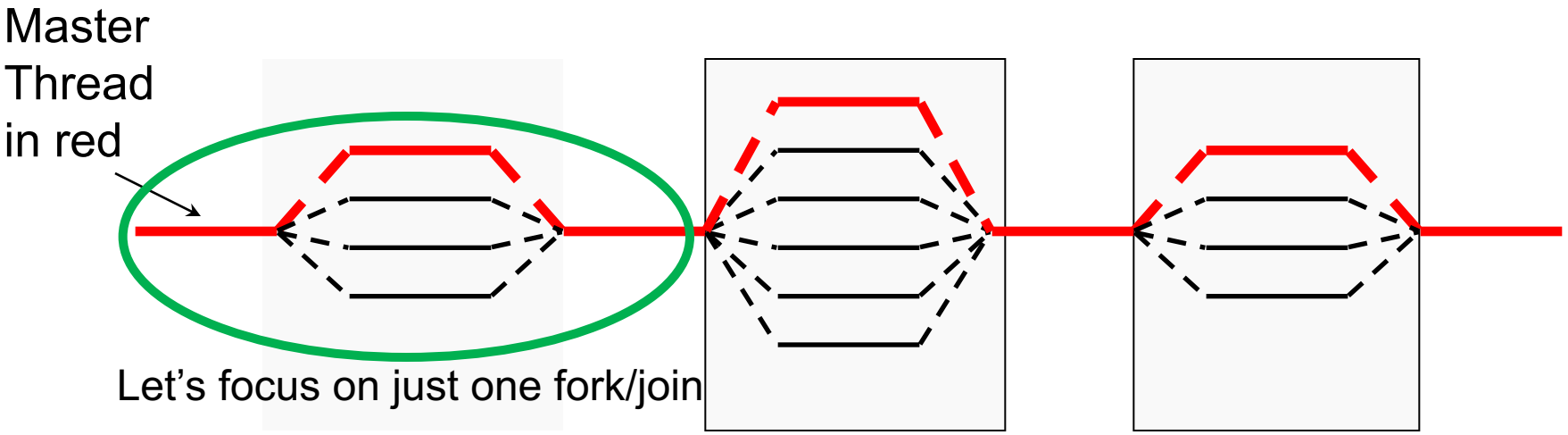
- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



Adding tasks to OpenMP required major changes to the specification

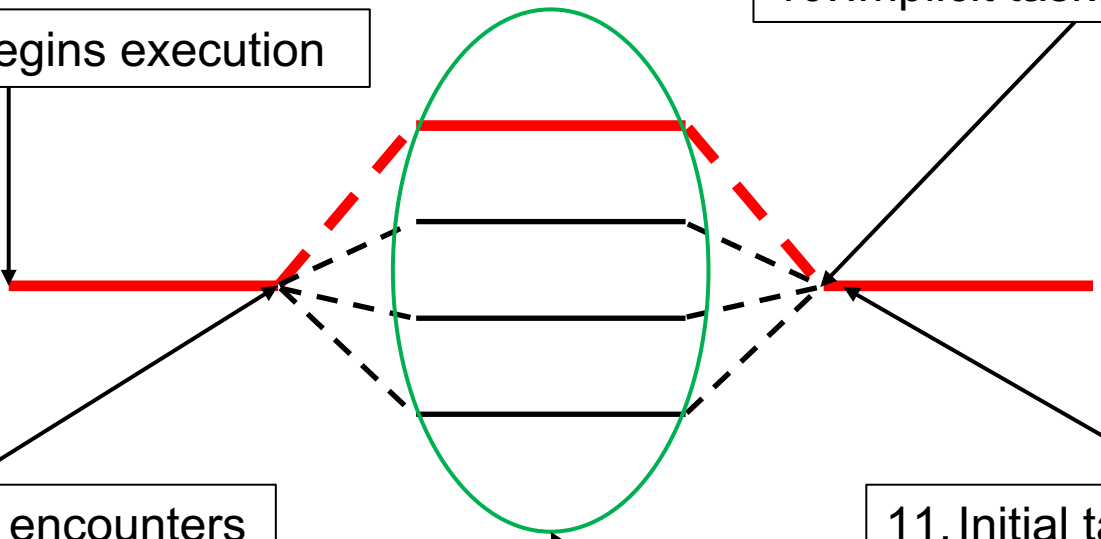
Fork-Join Parallelism:

- 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

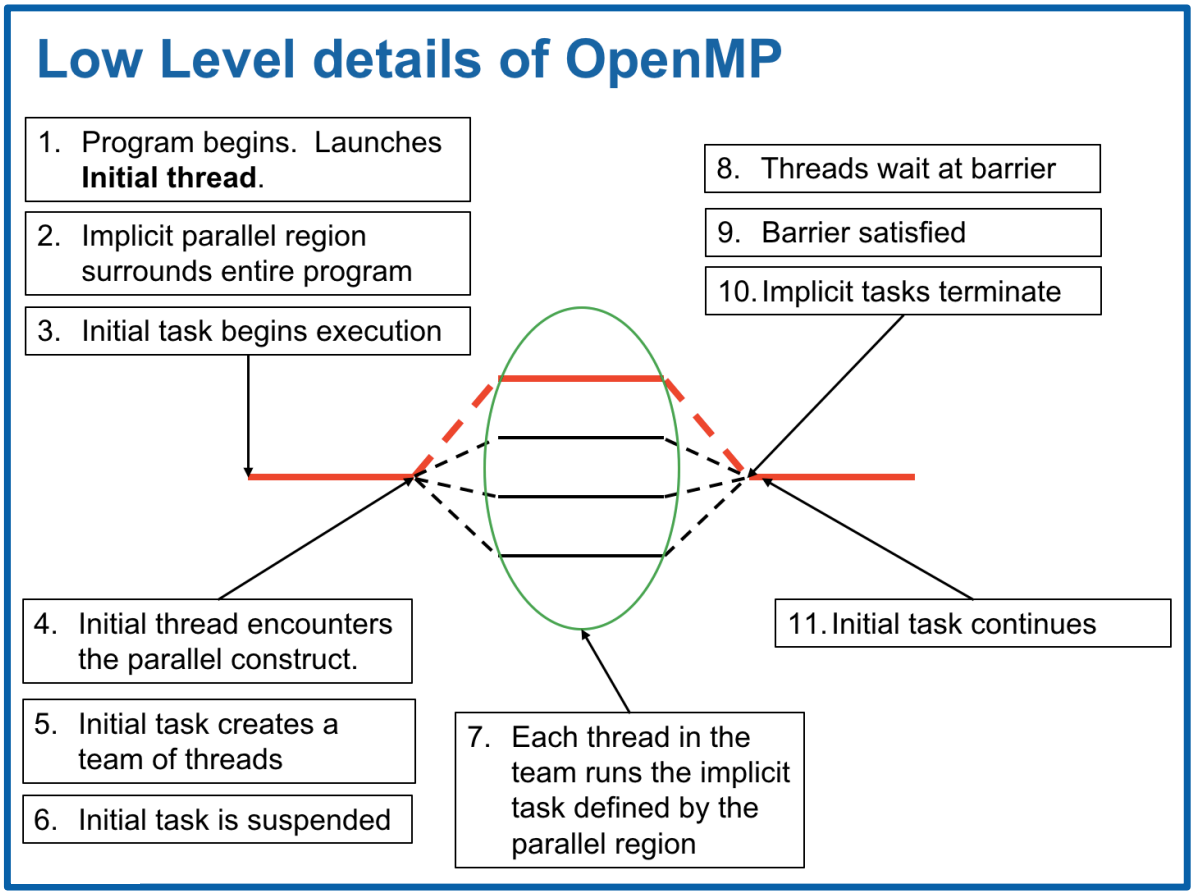
1. Program begins. Launches **Initial thread**.
2. Implicit parallel region surrounds entire program
3. Initial task begins execution
4. Initial thread encounters the parallel construct.
5. Initial task creates a team of threads
6. Initial task is suspended
7. Each thread in the team runs the implicit 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.



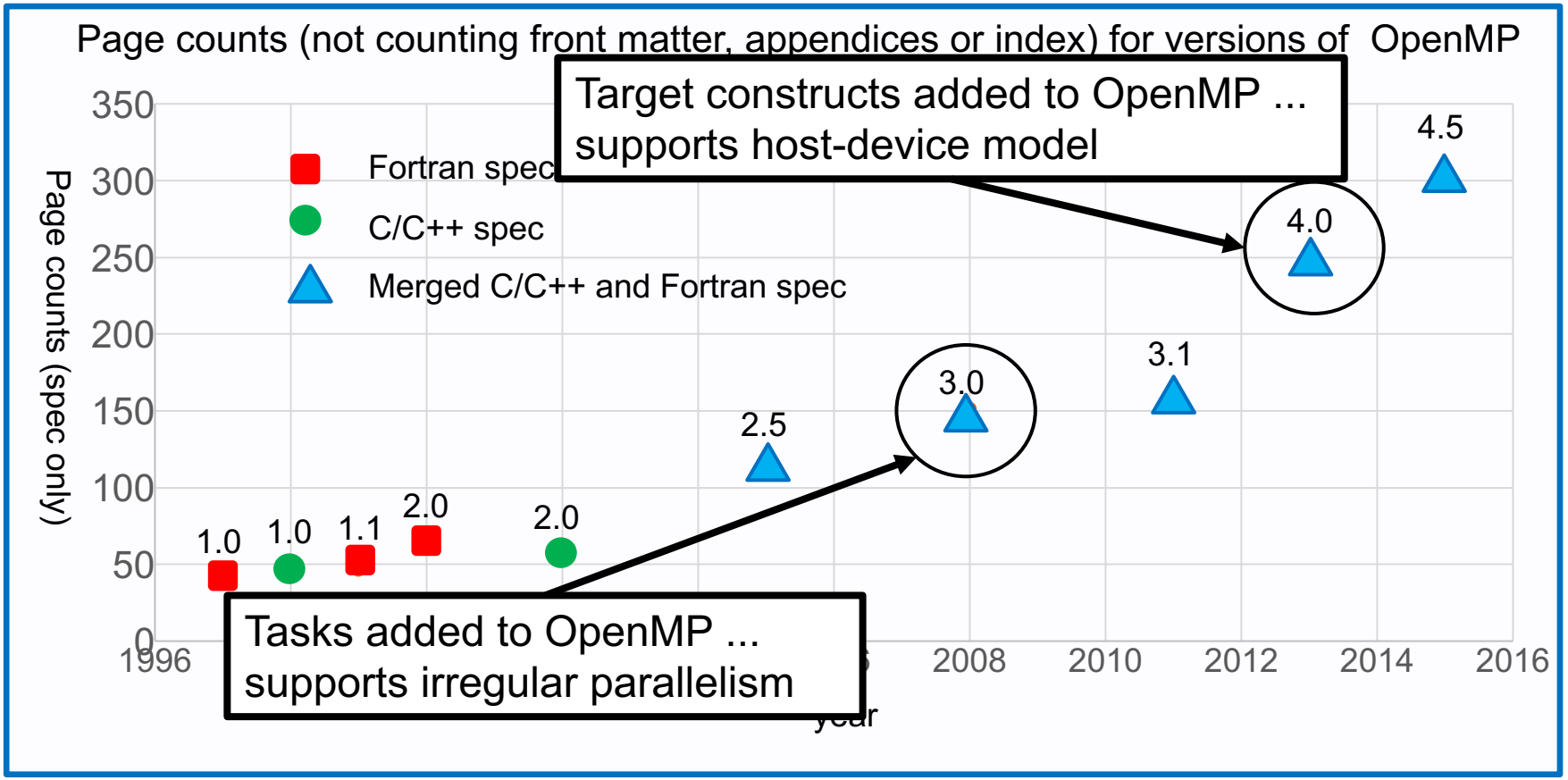
While all these initial threads, implicit tasks, and such are confusing to the programmer, they actually make life easier for people who implement OpenMP.

The OpenMP Common Core: Most OpenMP programs only use these 19 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers. The flush concept (but not the concept)
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for 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!



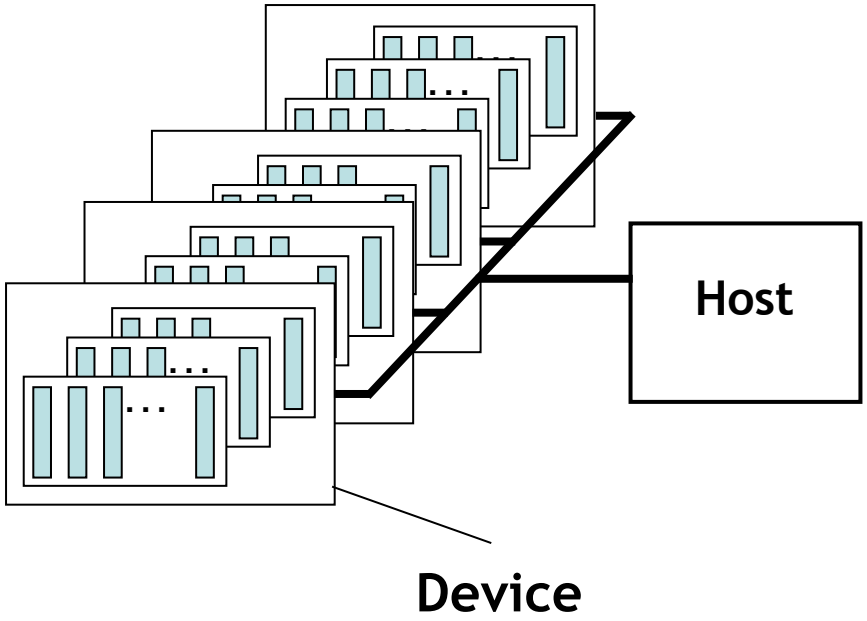
The complexity of the full spec is overwhelming, so we focus on the 19 constructs most OpenMP programmers restrict themselves to … the so called “OpenMP Common Core”

Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

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



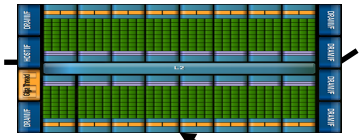

```
#include <omp.h>
#include <stdio.h>
int main()
{
    printf("There are %d devices\n",
            omp_get_num_devices());
}
```

OpenMP with target devices

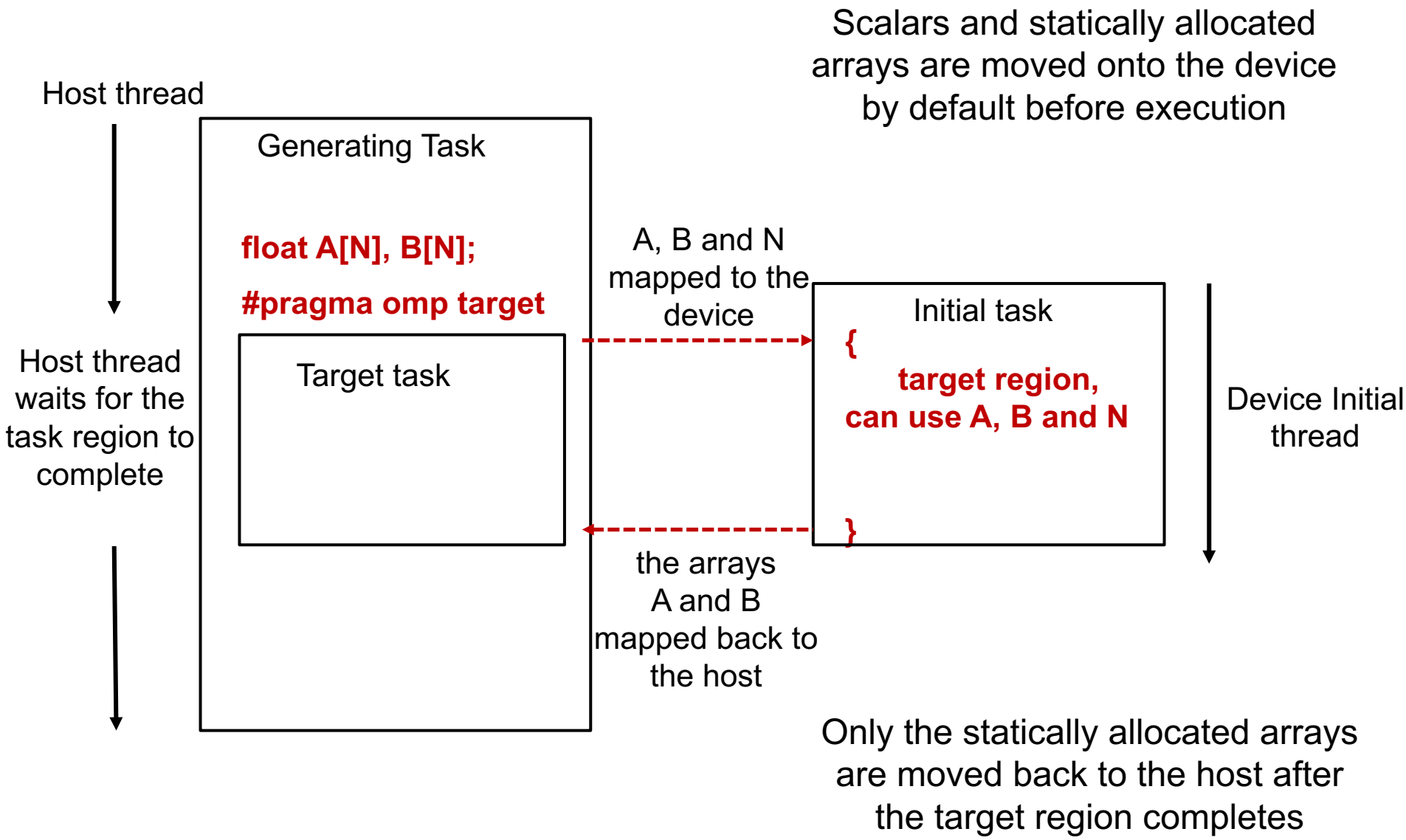
- The target construct offloads execution to a device.

```
#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



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

Exercise

- Use the provided serial vadd program which adds two vectors and tests the results.
- Use the target construct to run the code on a GPU.
- Experiment with the other OpenMP constructs if time.

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

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 openmpSC17
ssh br-trainXX@isambard.gw4.ac.uk

# Log in to Isambard Phase 1
ssh phase1

# Change to directory containing exercises
cd sc17-tutorial

# List files
ls
```

```
[br-train01@login-01 sc17-tutorial]$ ls
jac_solv.c      makefile      mm_utils.h    submit_jac_solv   vadd.c
make.def        makefile      pi.c          submit_pi
Make_def_files  mm_utils.c    Solutions     submit_vadd
```

Job submission scripts

Jacobi exercise starting code

Pi exercise starting code

vadd exercise starting code

Using Isambard (2/2)

```
# Build exercises
make

# Submit job
qsub submit_vadd

# 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  vadd       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 vadd.oXXXX
```

OpenMP target directive: the nowait clause

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

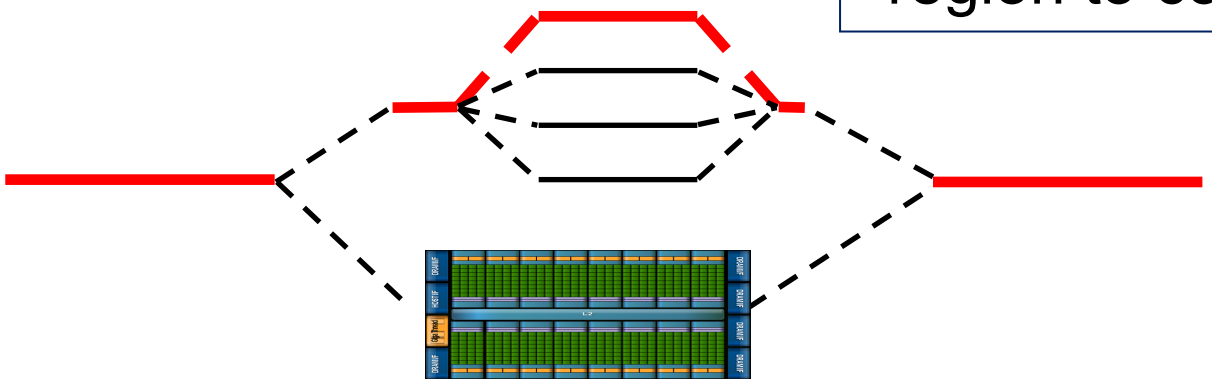
- Make the target task (running on the host) execute as a deferred task

```
#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.



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 original 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.

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

Agenda

- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

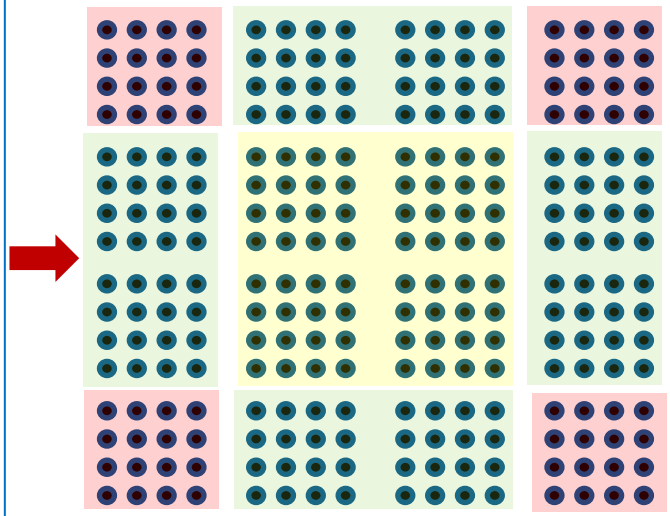
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

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

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

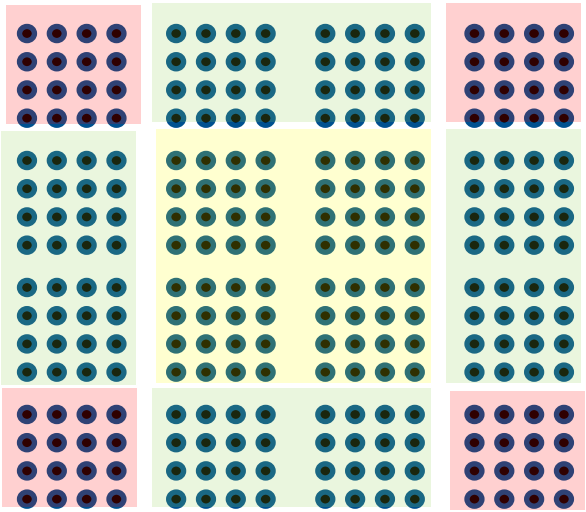
Turn source code into a scalar **work-item** (a CUDA **thread**)

```
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**

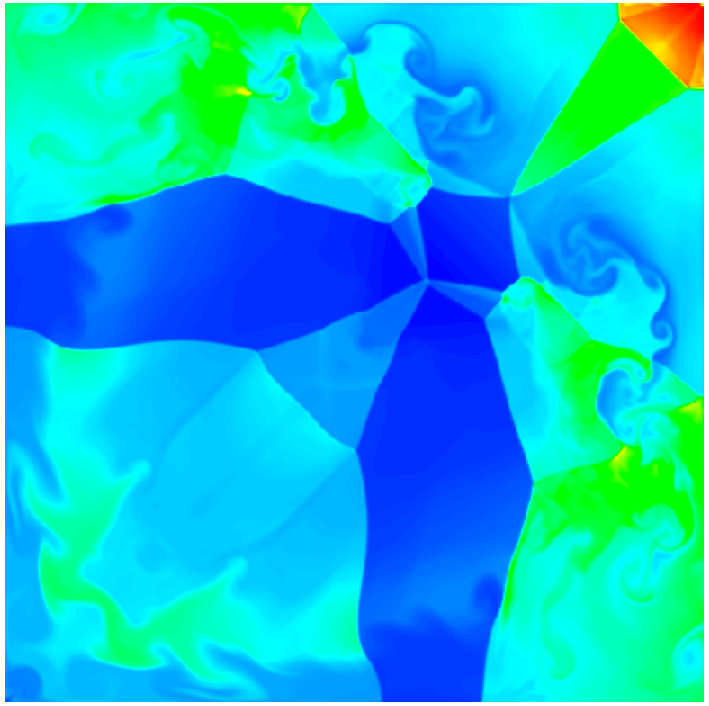


OpenCL index space is called an **NDRange.** CUDA calls this a **Grid**

It's called SIMT, but GPUs are really vector-architectures with a block of work-items called a **warp** executing together

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

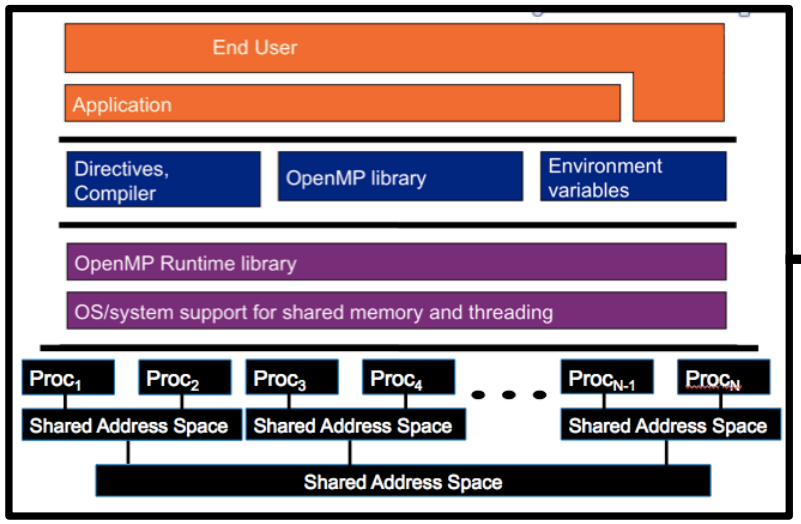
- Use the vadd program from the previous exercise and explore how it executes on a GPU using nvprof.
  - nvprof ./vadd
  - nvprof --print-gpu-trace ./vadd
- Change the problem size and get a feel for how the grid and block-size change.
- You will need to edit the job submission files to submit the job to the queue with nvprof. For example:
  - Replace the line
    - ./vadd
  - with the line (for example)
    - nvprof --print-gpu-trace ./vadd

Agenda

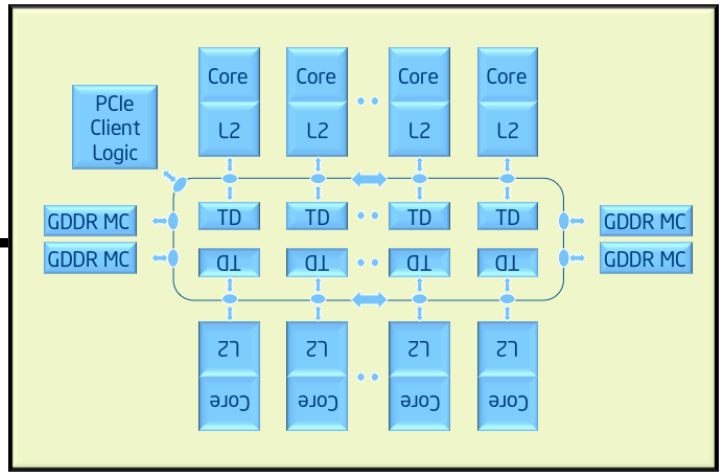
- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

OpenMP device model: Examples

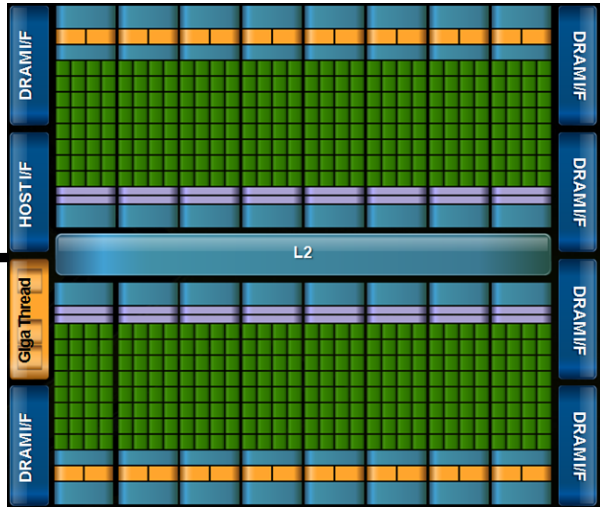
A couple key devices we considered when designing the device model in OpenMP



Host

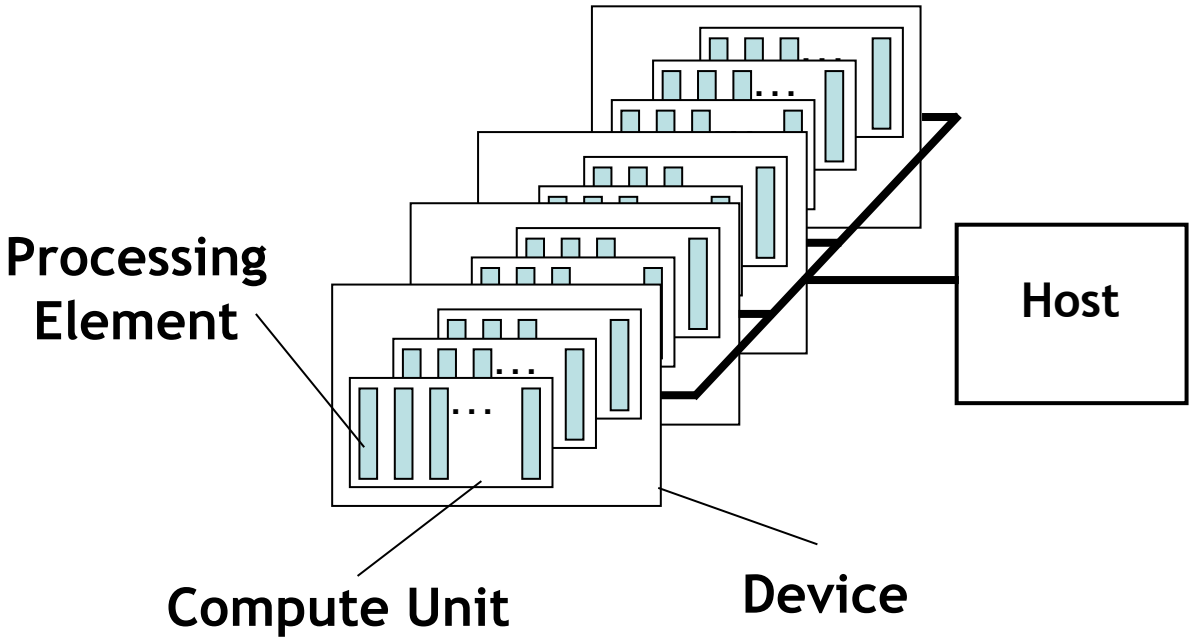


Target Device: Intel® Xeon Phi™ processor



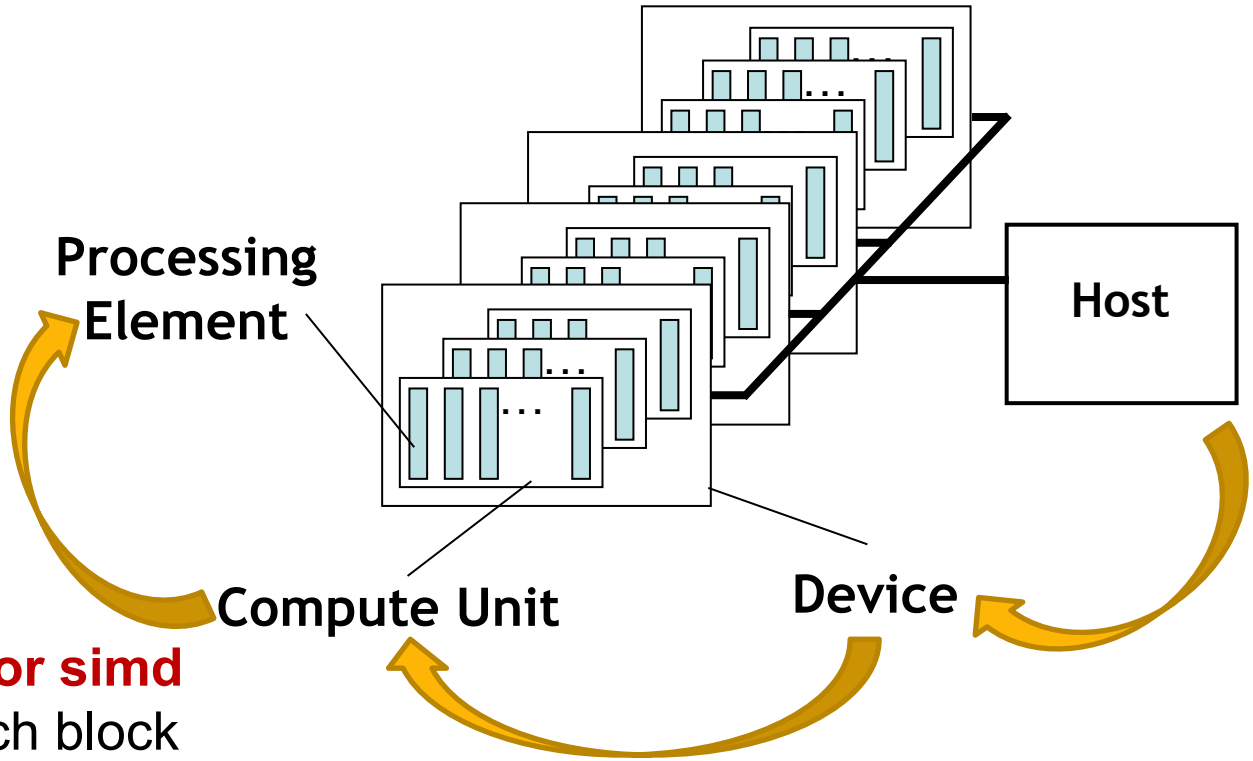
Target Device: GPU

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***

Our host/device Platform Model and OpenMP



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

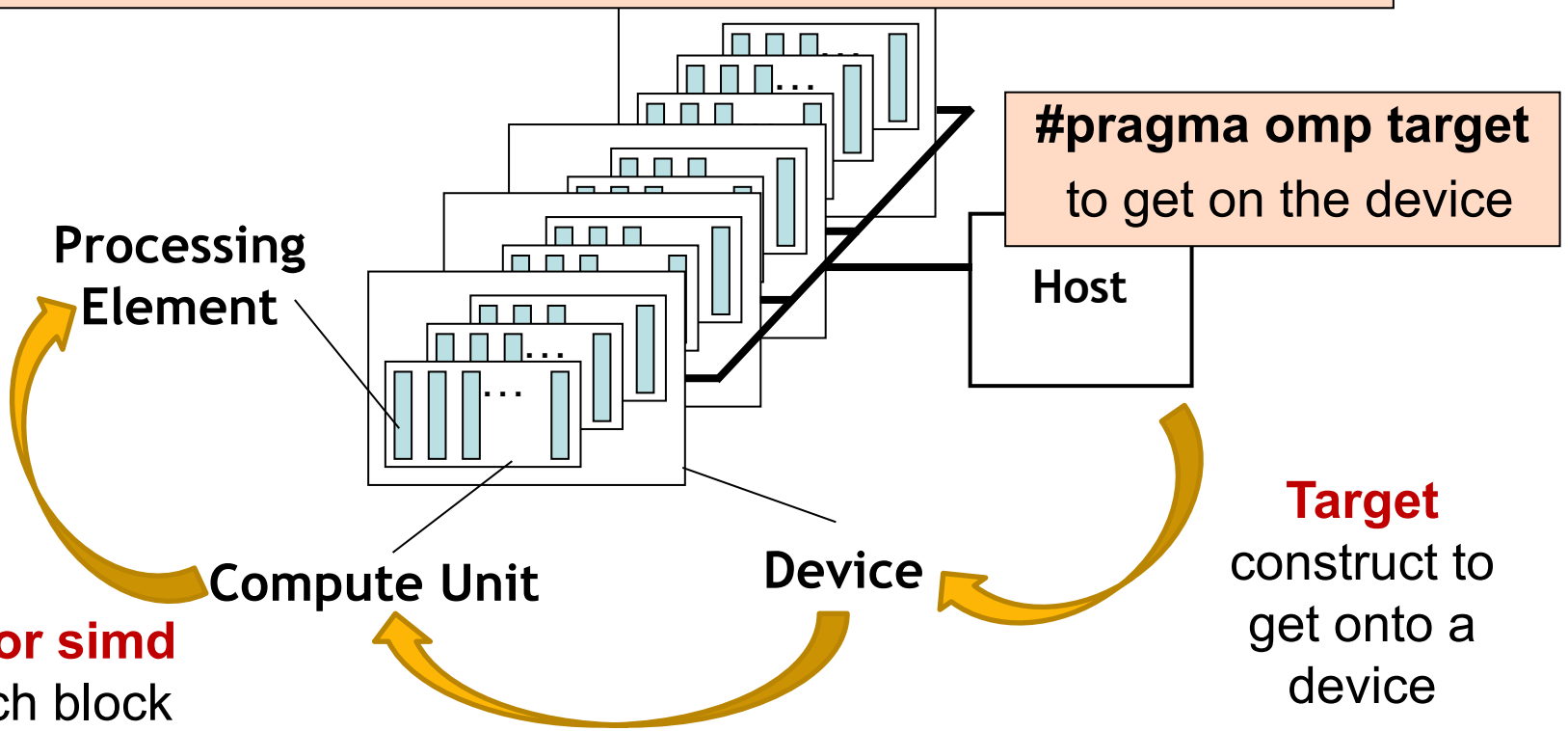
Teams construct to create a league of teams with one team of threads on each compute unit.

Target construct to get onto a device

Distribute clause 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:



Teams construct to create a

#pragma omp teams distribute parallel for simd

to assign work to the device processing elements

Distribute clause to assign blocks of loop iterations to teams.

Consider the familiar VADD example

```
#include<omp.h>
#include<stdio.h>
#define N  1024
int main()
{
   float a[N], b[N], c[N];
   int i;

// initialize a, b and c ….

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

// Test results, report results …

}
```

Consider the familiar VADD example

```
#include<omp.h>
#include<stdio.h>
#define N  1024
int main()
{
   float a[N], b[N], c[N];
   int i;

// initialize a, b and c ….
#pragma omp target
#pragma omp teams distribute parallel for simd
   for(i=0;i<N;i++)
         c[i] += a[i] + b[i];

// Test results, report results …

}
```

original variables **i, a, b, c** are copied to the device at beginning of construct

original variables **a, b, c** are copied to the host at the end of construct

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).

Controlling data movement

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

Data movement can be explicitly controlled with the map clause

- The various forms of the map clause
  - **map(to:list)**: *read-only* data on the device. Variables in the list are initialized on the device using the original values from the host.
  - **map(from:list)**: *write-only* data on the device: initial value of the variable is not initialized. At the end of the target region, the values from variables in the list are copied into the original variables.
  - **map(tofrom:list)**: the effect of both a map-to and a map-from
  - **map(alloc:list)**: data is allocated and uninitialized on the device.
  - **map(list)**: equivalent to map(tofrom:list).
- For pointers you must use array notation ..
  - **map(to:a[0:N])**

Default Data Mapping: Scalar variables
(i.e. mapping scalars between host and target regions)

- Default mapping behavior for scalar variables
  - OpenMP 4.0 implicitly mapped all scalar variables as **tofrom**
  - OpenMP 4.5 implicitly maps scalar variables as **firstprivate**
- Key difference between the two options:
  - firstprivate:
    - a new value is created per work-item and initialized with the original value. The variable is not copied back to the host
  - map(tofrom):
    - A new value is created in the target region and initialized with the original value, but it is shared between work-items on the device. Its value is copied back to the host at the end of the target region.
- Why is default firstprivate for scalars a better choice?
  - OpenMP target regions for GPUs execute with CUDA or OpenCL. 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 Sharing

WARNING: Make sure not to confuse the implicit mapping of pointer variables with the data that they point to

```
int main(void) {
  int A = 0;
  int* B = malloc(sizeof(int)*N);
  #pragma omp target
  #pragma omp teams distribute parallel for simd
  for(int ii = 0; ii < N; ++ii) {
    // A, B, N and ii all exist here
    // B is a pointer variable! The data that B points to DOES NOT exist here!
  }
}
```

If you want to access the data that is pointed to by **B**, you will need to perform an explicit mapping using the **map** clause

Default Data Sharing

WARNING: Make sure not to confuse the implicit mapping of pointer variables with the data that they point to

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

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)
  - #pragma omp target teams distribute parallel for simd

Jacobi Solver (serial 1/2)

```
<<< allocate and initialize the matrix A and >>>
<<< vectors x1, x2 and b                     >>>
while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
    xnew = iters % s ? x2 : x1;
    xold   = iters % s ? x1 : x2;

    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 (serial 2/2)

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

} // end while loop
```

Jacobi Solver (Par Targ, 1/2)

```
while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
    xnew = iters % 2 ? x2 : x1;
    xold   = iters % 2 ? x1 : x2;
#pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                    map(to:A[0:Ndim*Ndim], b[0:Ndim] ,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(to: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
```

Jacobi Solver (Par Targ, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
#pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                    map(to: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.

Agenda

- OpenMP overview
- The device model in OpenMP
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

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] ,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(to: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

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 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}
}
```

Copy A on the device to A on the host.

Copy A on the host to A on the device.

Target update details

- #pragma omp target update clause[[[,]clause]...]new-line
- creates a target task to handle data movement between the host and a devce

- 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)
- This directive generates a target task.
- nowait and depend apply to the target task running on the host.

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 data
  - #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 data map(tofrom:x1[0:Ndim],x2[0:Ndim]) \
                        map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
     // alternate x vectors.
    xnew  = iters % 2 ? x2 : x1;
    xold   =  iters % 2 ? x1 : x2;

#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);
} // end while loop
```

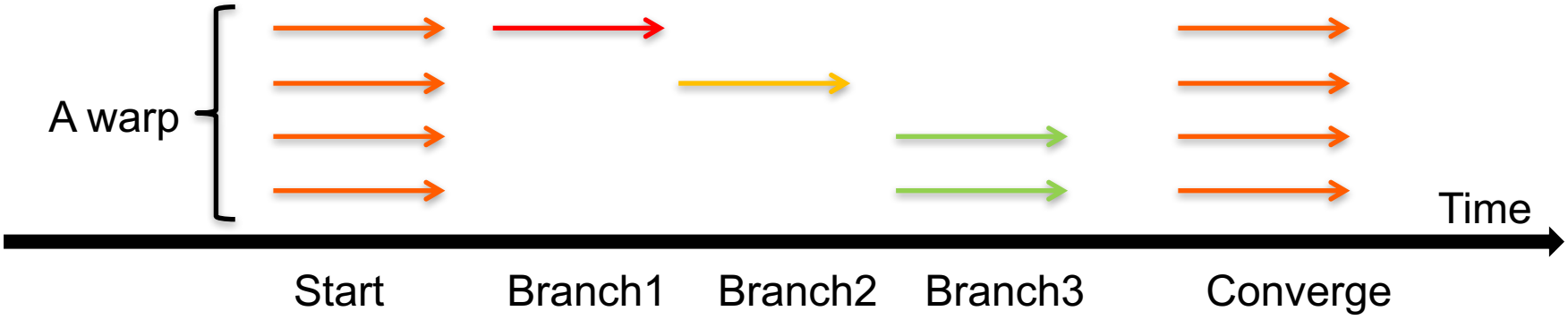
System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs

Agenda

- OpenMP overview
- The device model in OpenMP
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

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);
```

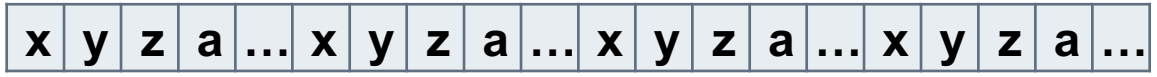
Coalesced Access

- ***Coalesced memory accesses*** are key for high performance code
- 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;
```

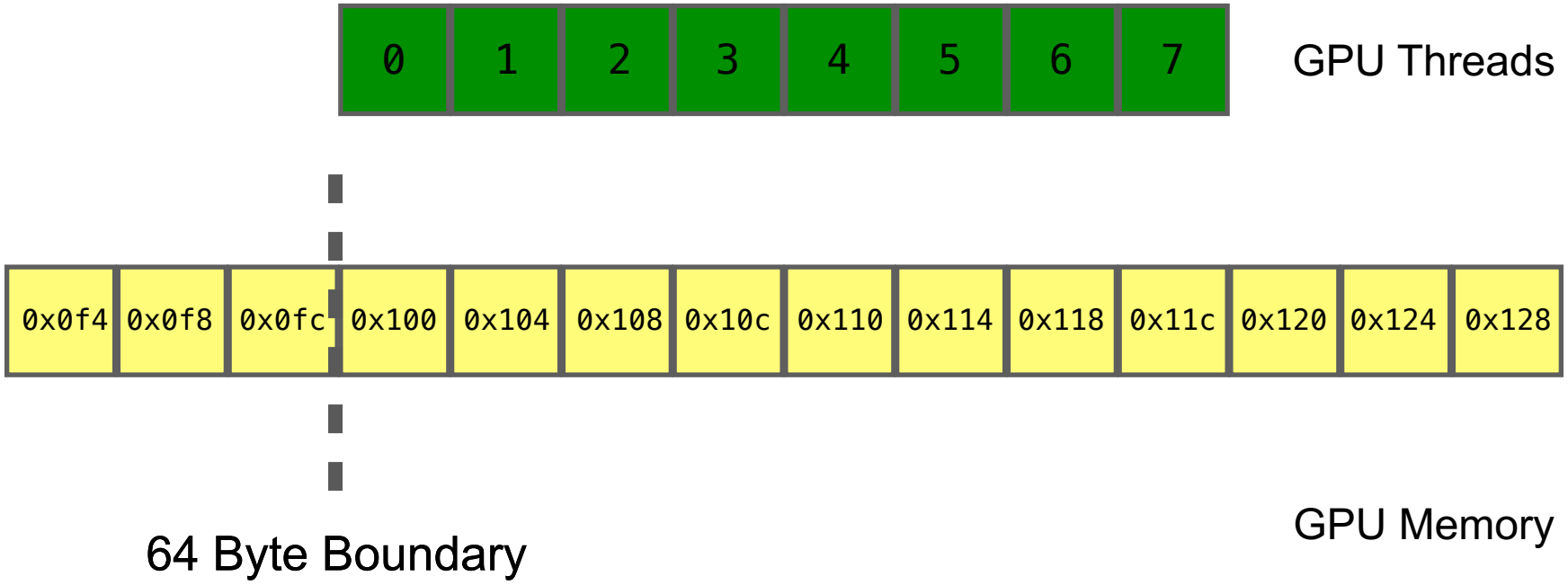
Adjacent work-items/vector-lanes like to access adjacent memory locations

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…

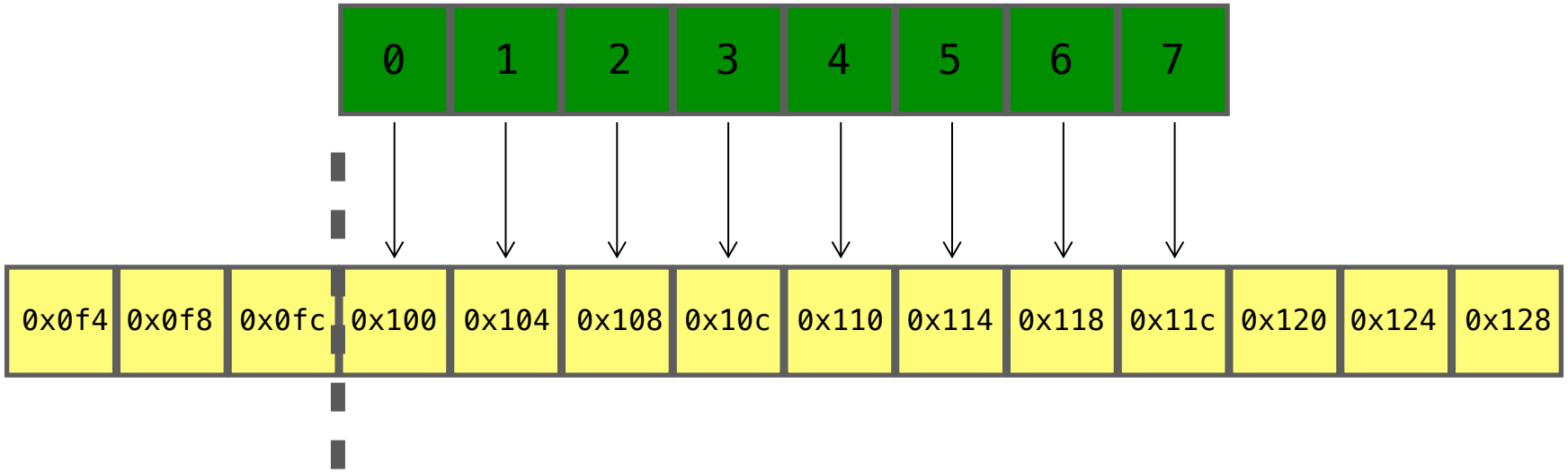
```
for (int id = 0; id < size; id++)
{
  // ideal
     float val1 = memA[id];

  // 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

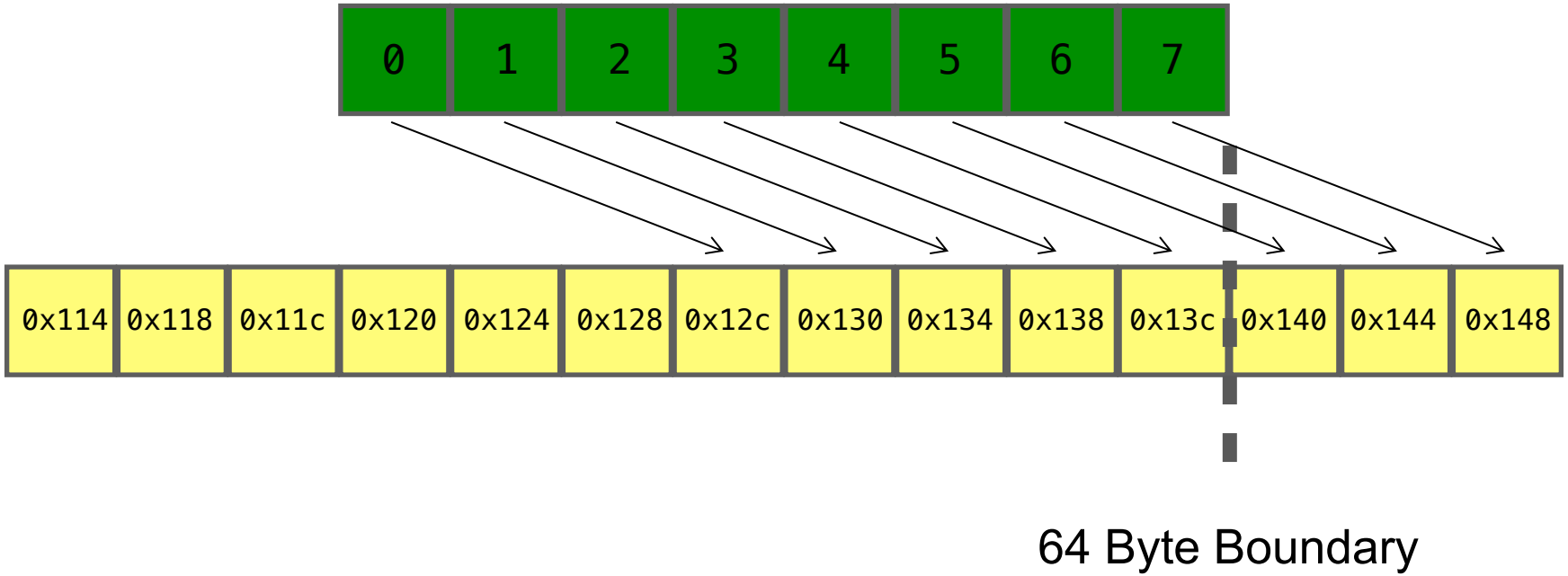
```
float val1 = memA[id];
```



64 Byte Boundary

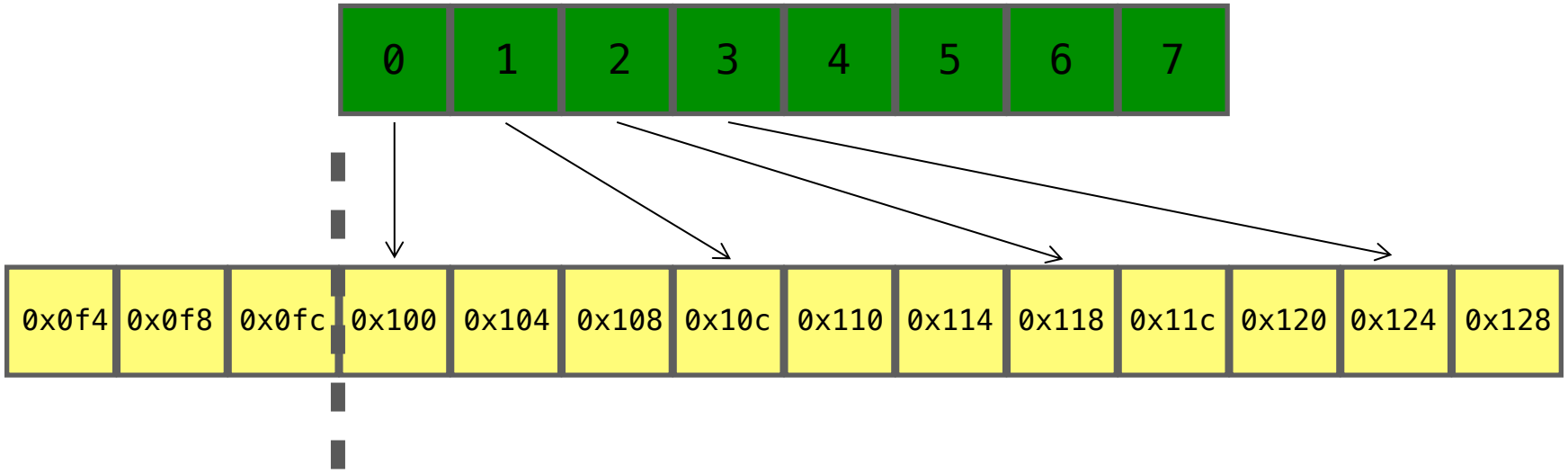
Memory access patterns

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



Memory access patterns

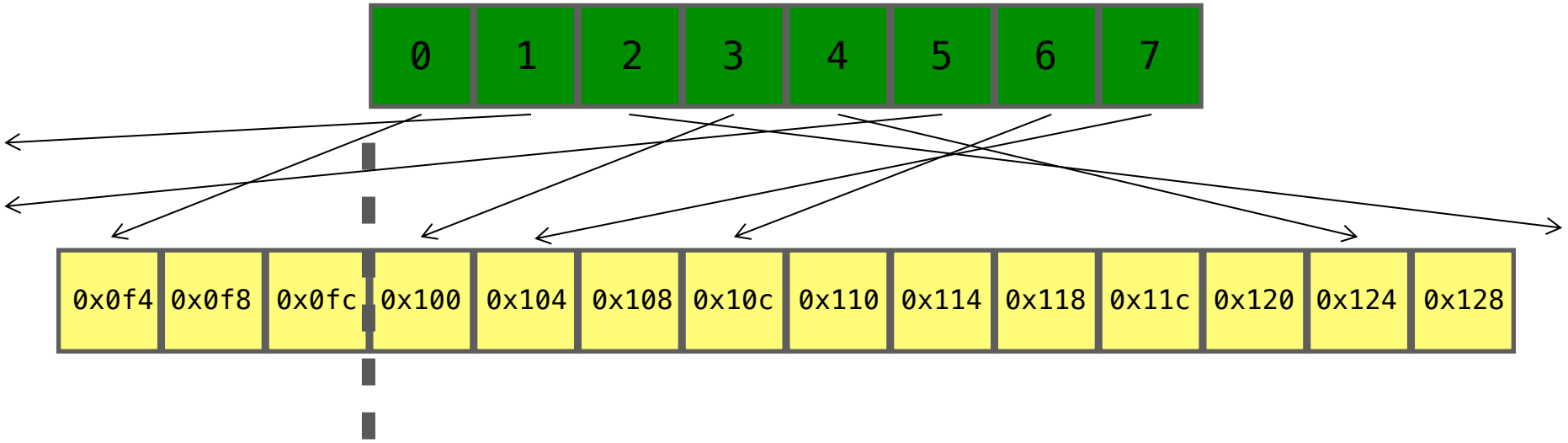
```
float val3 = memA[3*id];
```



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 data map(tofrom:x1[0:Ndim],x2[0:Ndim]) \
                        map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
     // alternate x vectors.
    xnew  = iters % 2 ? x2 : x1;
    xold   =  iters % 2 ? x1 : x2;

#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);

} // 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

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.

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

```
#pragma omp target data map(tofrom:x1[0:Ndim],x2[0:Ndim]) \
                        map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
     // alternate x vectors.
    xnew  = iters % 2 ? x2 : x1;
    xold   =  iters % 2 ? x1 : x2;
#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);

} // 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

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.

Agenda

- OpenMP overview
- The device model in OpenMP
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

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.

- **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/

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.

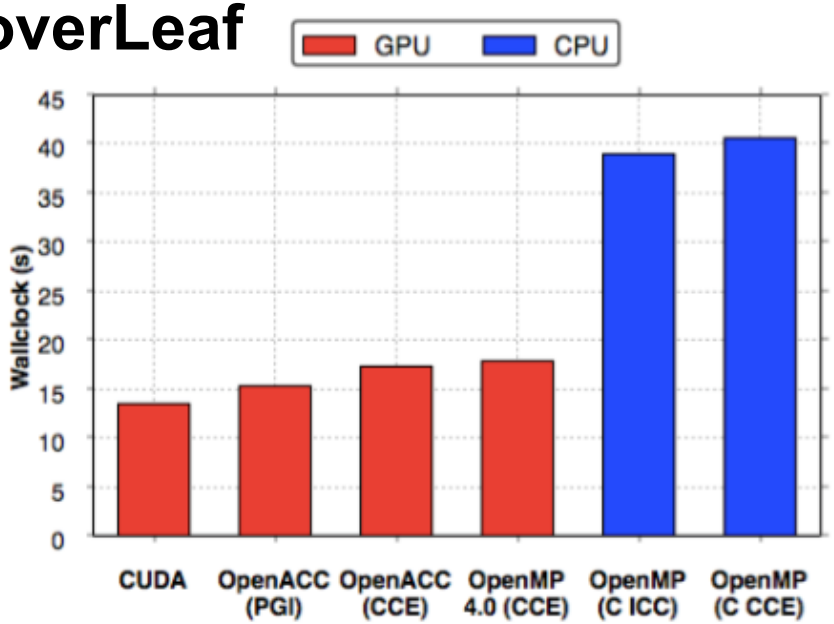
Performance?

* 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)

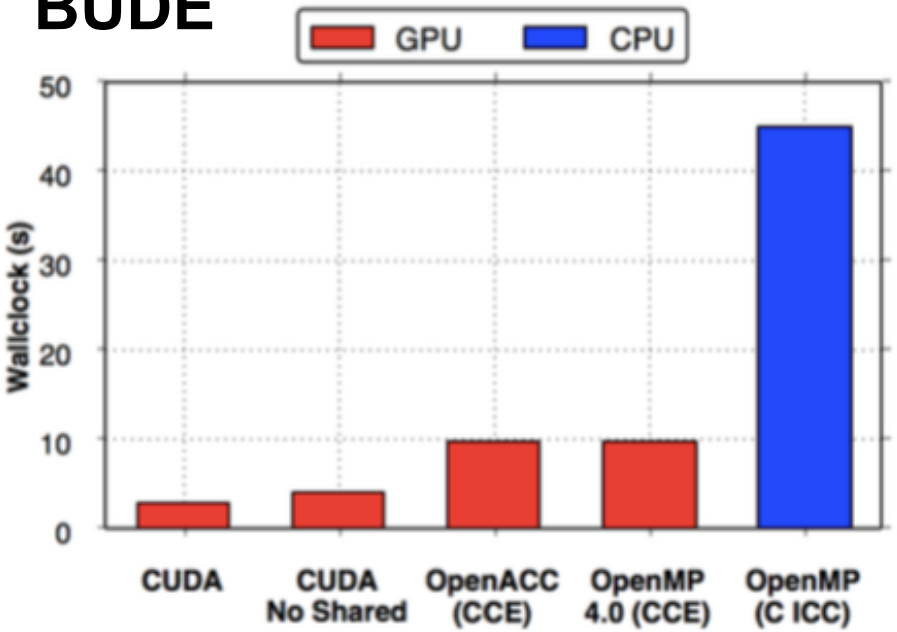
Immediately we see impressive performance compared to CUDA

Clearly the Cray compiler leverages the existing OpenACC backend

CloverLeaf



BUDE



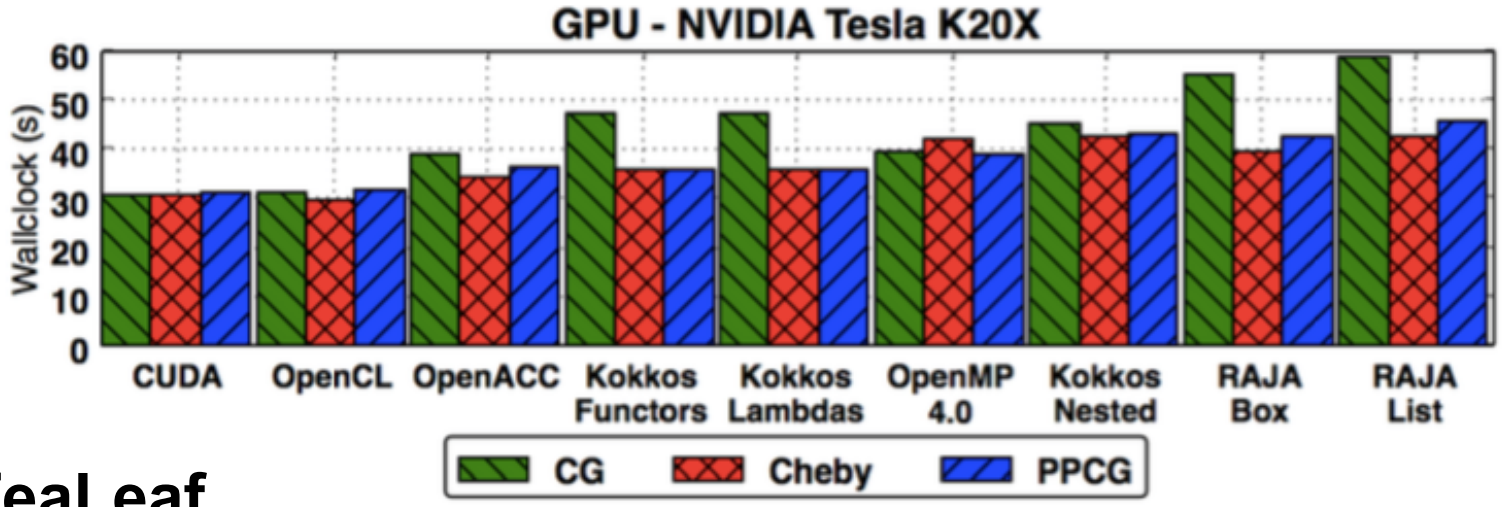
Even with OpenMP 4.5 there is still no way of targeting shared memory directly.

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

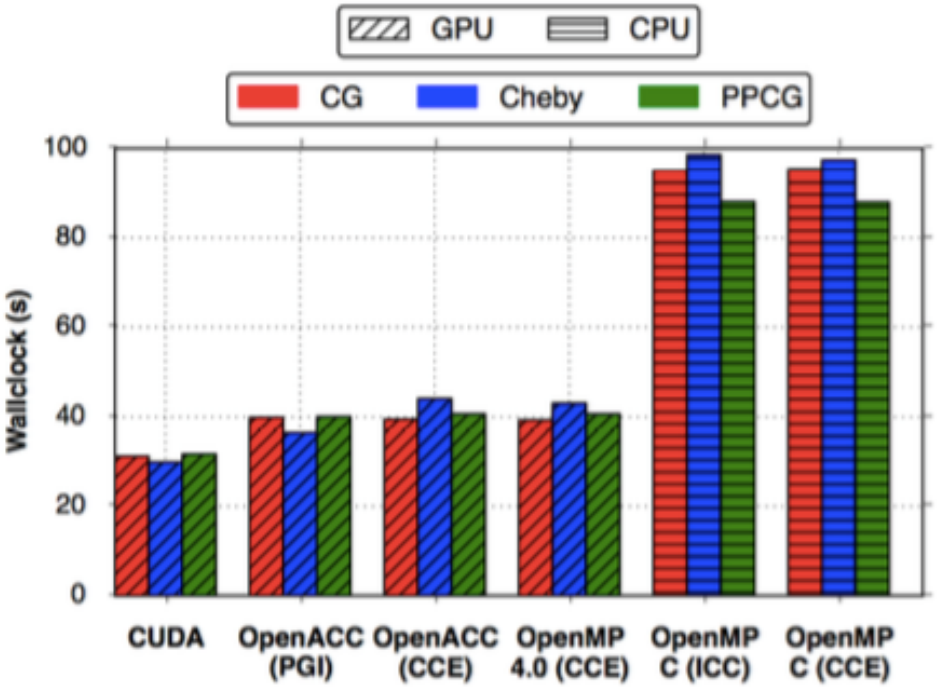
Martineau, M., McIntosh-Smith, S. Gaudin, W., *Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model, 2016, HIPS'16*

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

It's likely that these figures have improved even more with maturity of the portable models

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

How do you get good performance?

- Our finding so far: *You can achieve good performance 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.

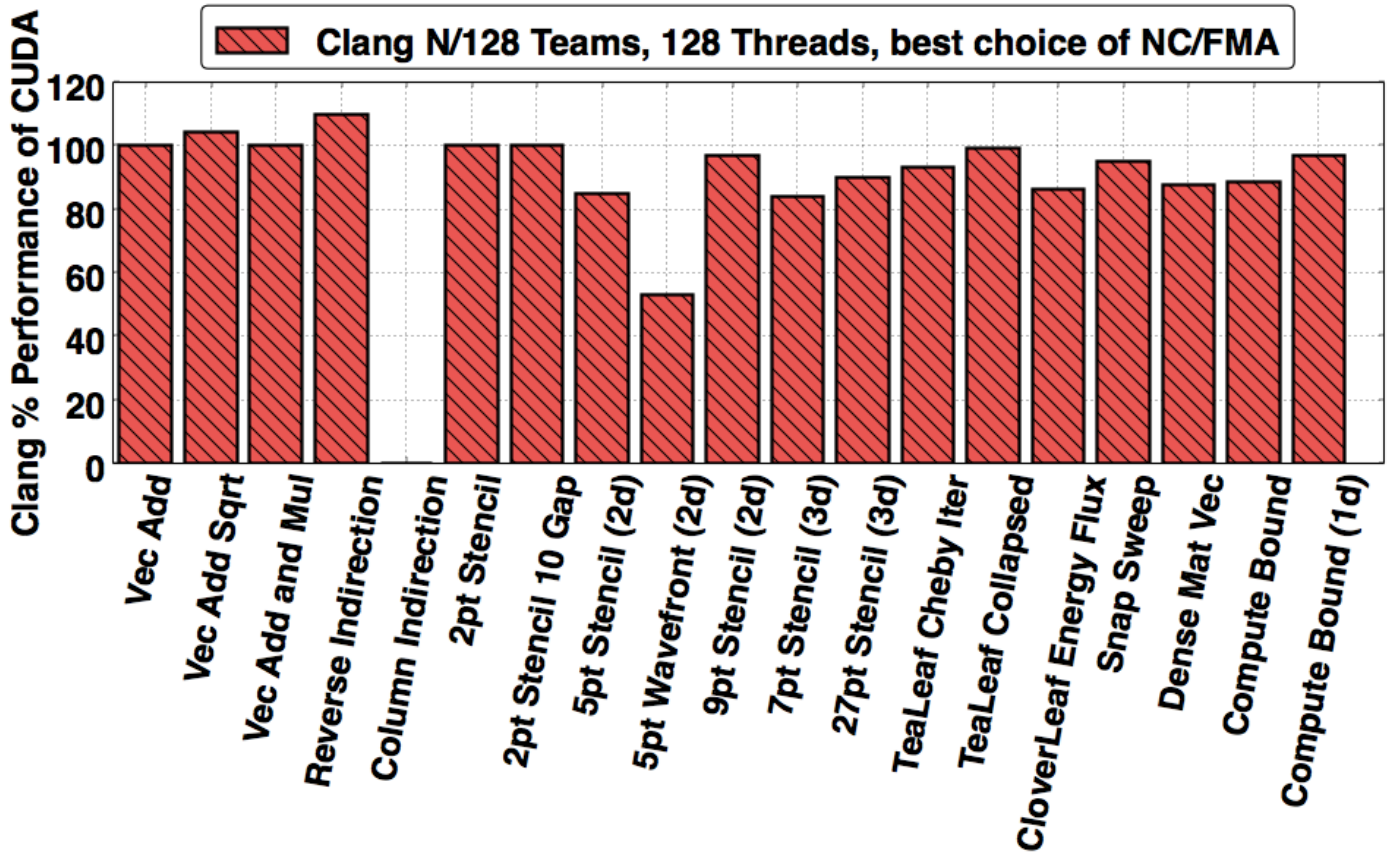
  - Using the **simd** directive to vectorize inner loops.

  - 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*

Good. Performance… and Portability?

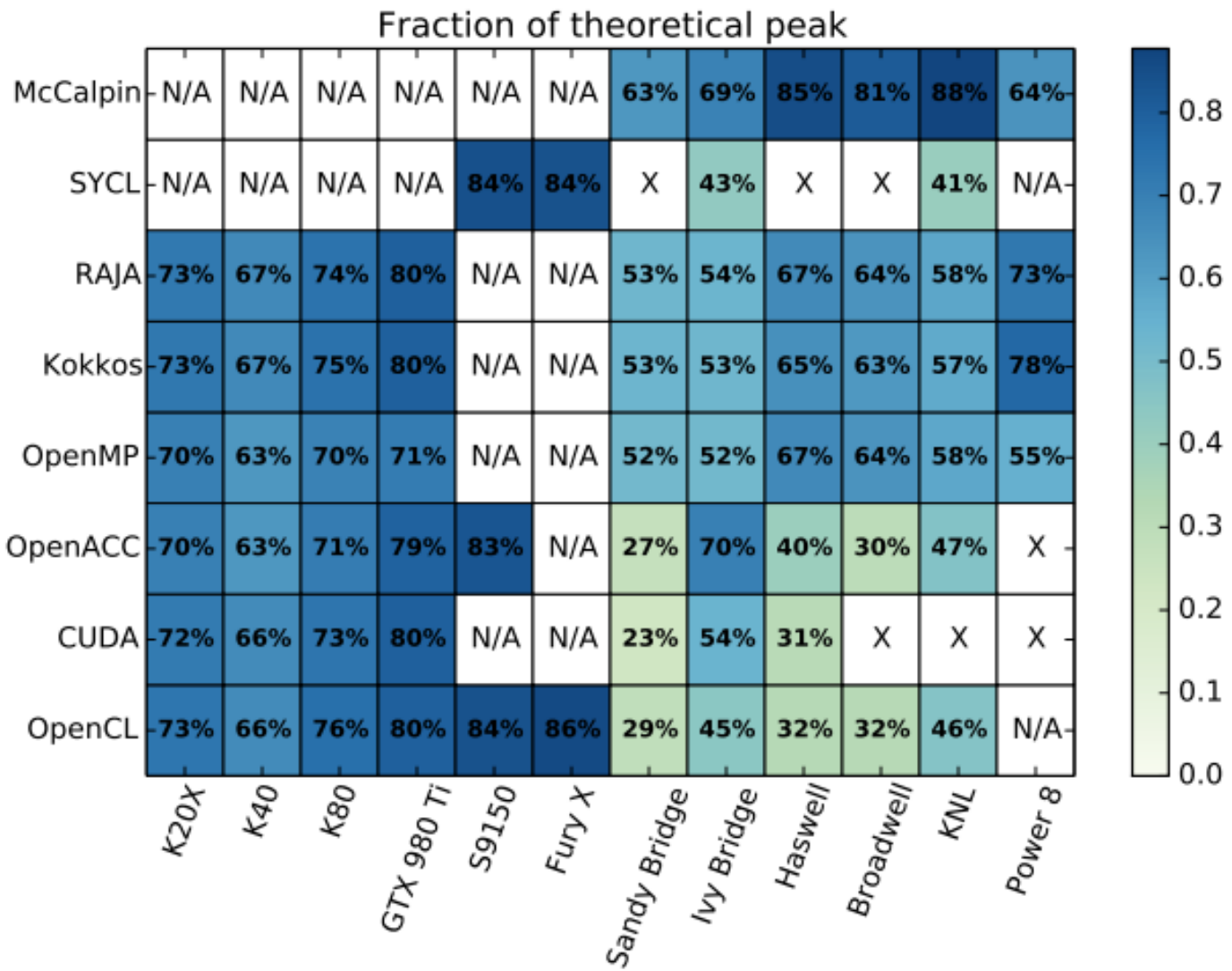
- Up until this point we had implicitly proven a good level of portability as we had successfully run OpenMP 4.x on many devices (Intel® CPU, Intel Xeon® Phi™ processors, NVIDIA® GPUs).

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

- We keep tracking this improvement over time.

OpenMP in The Matrix.

System Details on the following slide



On those supported target architectures, OpenMP achieves good performance

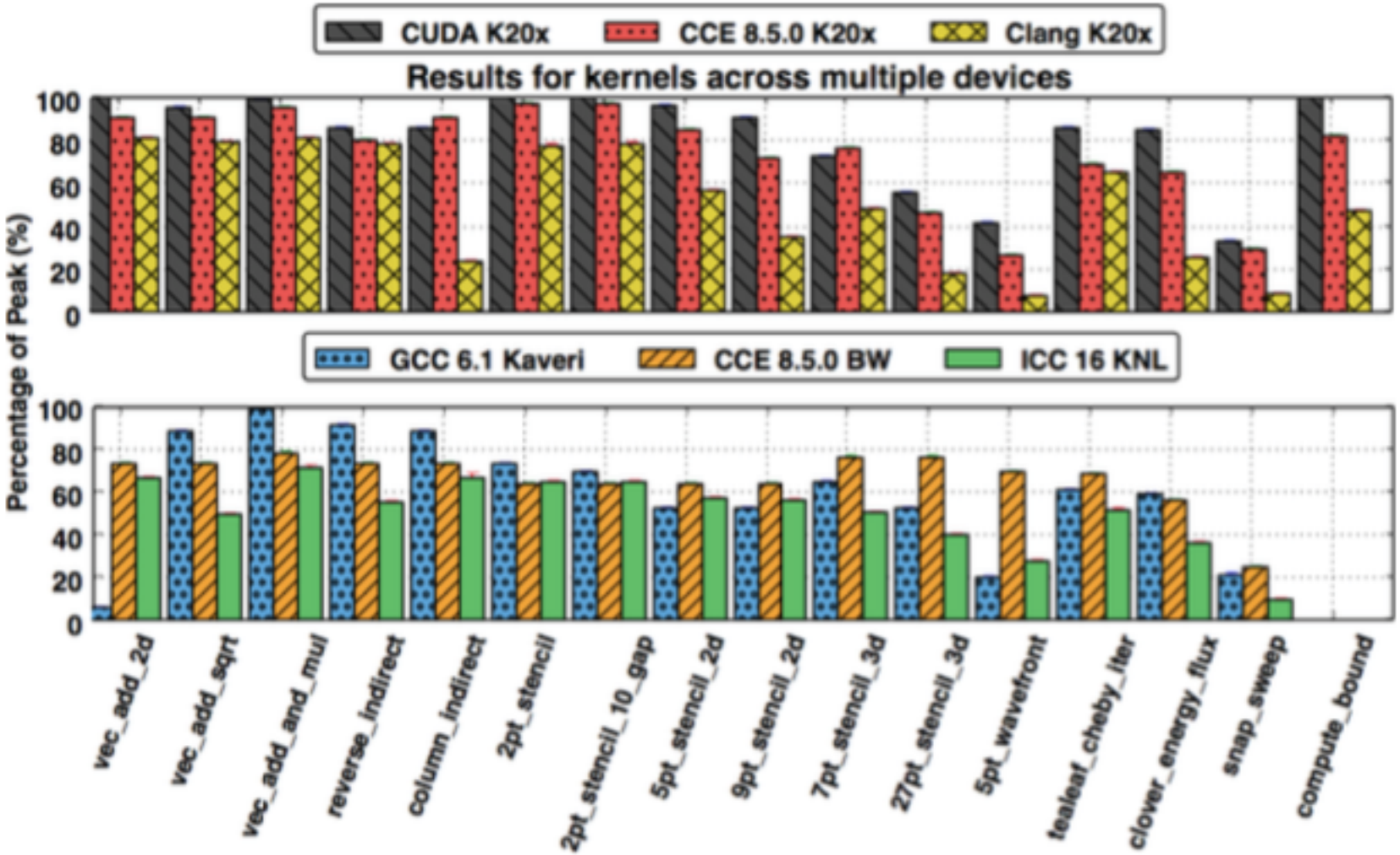
Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S., *GPU-STREAM v2.0 Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models, ISC'16*

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*

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 architecture.

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.

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/sc17-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 diversity.