



Programming Your GPU with OpenMP*

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

Includes material from the Bristol HPC research group: Matt Martineau, Andrei Poenaru, Patrick Atkinson, Tom Deakin and James Price

Where to find these slides

http://uob-hpc.github.io/assets/OMPUG_GPU_2019.pdf

Preliminaries

- 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
- 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



EPSRC


CRAY
THE SUPERCOMPUTER COMPANY

ARM

Plan

Module	Concepts	Exercises
OpenMP overview	<ul style="list-style-type: none">• OpenMP recap• Hardcore jargon of OpenMP	<ul style="list-style-type: none">• None ... we'll use demo mode so we can move fast
The device model in OpenMP	<ul style="list-style-type: none">• Intro to the Target directive with default data movement	<ul style="list-style-type: none">• Vadd program
Understanding execution	<ul style="list-style-type: none">• Intro to nvprof	<ul style="list-style-type: none">• nvprof ./vadd
Working with the target directive	<ul style="list-style-type: none">• CPU and GPU execution models and the fundamental combined directive	<ul style="list-style-type: none">• Vadd with combined directive using nvprof to understand execution
Basic memory movement	<ul style="list-style-type: none">• The map clause	<ul style="list-style-type: none">• Jacobi solver
Optimizing memory movement	<ul style="list-style-type: none">• Target data regions	<ul style="list-style-type: none">• Jacobi with explicit data movement
Optimizing GPU code	<ul style="list-style-type: none">• Common GPU optimizations	<ul style="list-style-type: none">• 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:

```
C$OMP FLUSH
```

```
#pragma omp critical
```

```
C$OMP THREADPRIVATE (/ABC/)
```

```
CALL OMP SET NUM THREADS (10)
```

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

```
C$OMP PARALLEL COPYIN (/blk/)
```

```
C$OMP DO lastprivate (XX)
```


```
Nthrds = OMP_GET_NUM_PROCS ()
```

```
omp_set_lock (lck)
```

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

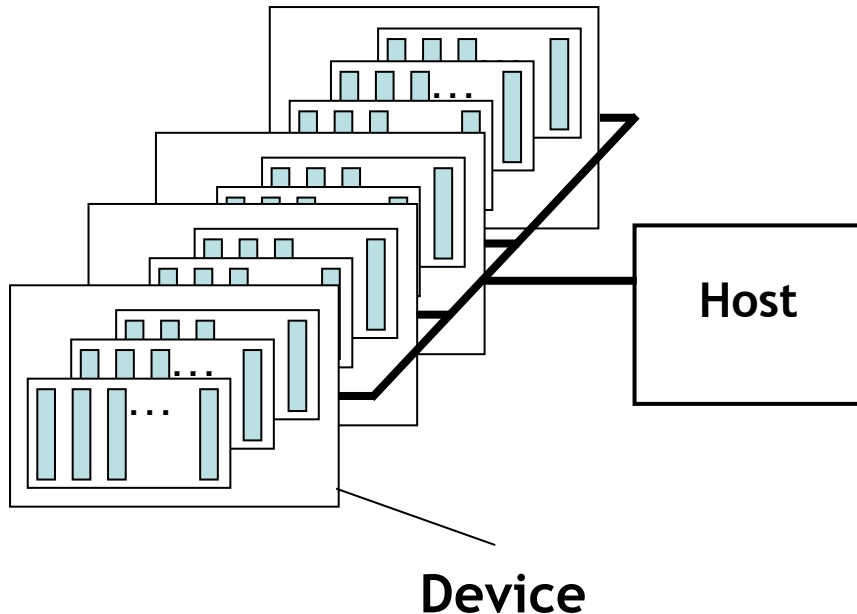
OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread 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.

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

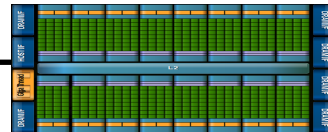
3. Initial task begins execution

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

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.

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[0:N])

- map-type may be *to*, *from*, *tofrom*, or *alloc*. The clause defines how the variables in the 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.

The target data environment

Scalars and statically allocated arrays that are referenced in the target region are moved onto the device implicitly before execution

Host thread

Generating Task

```
float A[N], B[N];  
#pragma omp target
```

Target task

A, B and N
mapped to the
device

Initial task

```
{  
    target region,  
    can use A, B and N
```

Device Initial
thread

Host thread
waits for the
task region to
complete

the arrays
A and B
mapped back to
the host

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

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 openmpJUN19
ssh br-trainXX@isambard.gw4.ac.uk
```

```
# Log in to Isambard Phase 1
ssh phase1
```

```
# Change to directory containing exercises
cd openmp-tutorial
```

```
# List files
ls
```

Job submission scripts

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

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 = Queued

R = Running

C = Complete

(job will disappear shortly after completion)

Job ID	Username	Queue	Jobname	SessID	NDS	TSK	Req'd Memory	Req'd Time	S	Elap Time
9376.master.gw4	br-traï	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
```

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
```


Exercise - vadd

Running on the GPU in **serial**

```
#pragma omp target
```


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

Running on the CPU in **parallel**

```
#pragma omp parallel for
```

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

Agenda

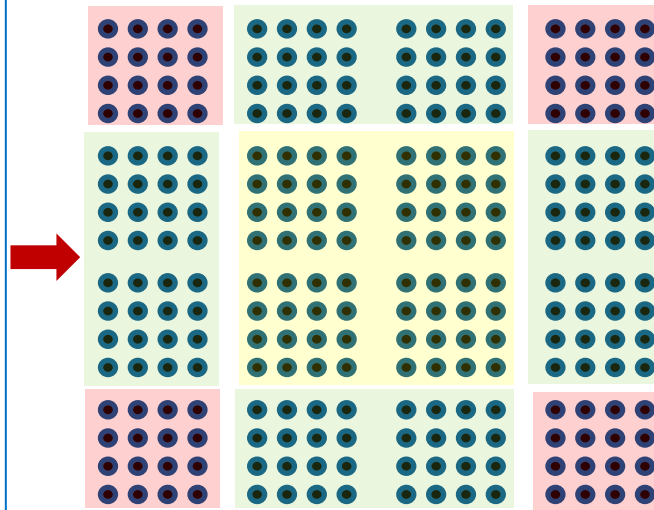
- OpenMP overview
- The device model in OpenMP
-  • Understanding execution on the GPU with nvprof
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

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

1. Turn source code into a scalar work-item
2. Map work-items onto an N dim index space.
4. Run on hardware designed around the same SIMT execution model

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

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



3. Map data structures onto the same index space



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

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

```
extern void reduce( __local float*, __global float*);

__kernel void pi( const int niters, float step_size,
                 __local float* l_sums, __global float* p_sums)
{
    int n_wrk_items = get_local_size(0);
    int loc_id      = get_local_id(0);
    int grp_id      = get_group_id(0);
    float x, accum = 0.0f;  int i,istart,iend;

    istart = (grp_id * n_wrk_items + loc_id) * niters;
    iend   = istart+niters;

    for(i= istart; i<iend; i++){
        x = (i+0.5f)*step_size;  accum += 4.0f/(1.0f+x*x); }

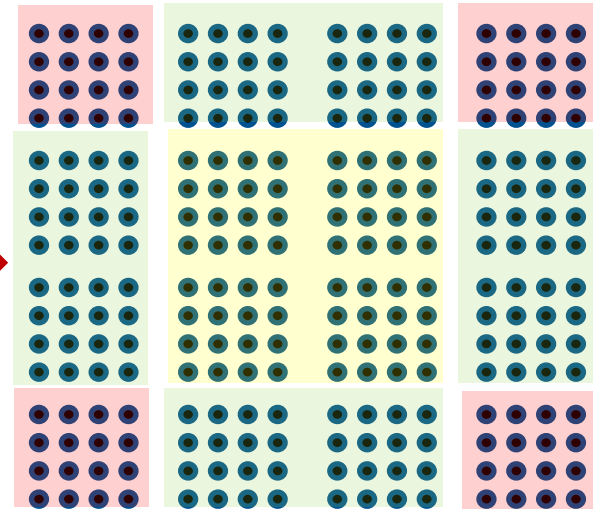
    l_sums[loc_id] = accum;
    barrier(CLK_LOCAL_MEM_FENCE);
    reduce(l_sums, p_sums);
}
```

This code defines a **kernel**

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



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

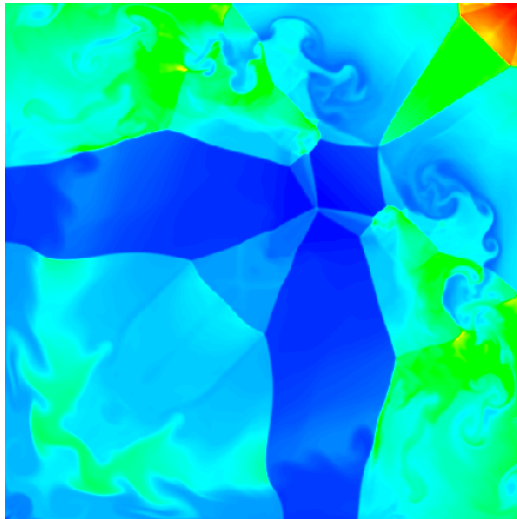


OpenCL index space is
called an **NDRange**. CUDA
calls this a **Grid**

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 special configuration



We will demonstrate using an OpenMP 4.5 version of **flow**

Hydrodynamics mini-app
solving Euler's compressible
equations

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
55.51%	205.74ms	53	3.8818ms	896ns	12.821ms
28.69%	106.32ms	14	7.5942ms	576ns	55.648ms
5.31%	19.682ms	2	9.8411ms	3.8686ms	15.814ms
1.52%	5.6321ms	2	2.8160ms	2.8121ms	2.8199ms
1.05%	3.9072ms	32	122.10us	1.2160us	217.21us
0.80%	2.9801ms	1	2.9801ms	2.9801ms	2.9801ms
0.73%	2.7061ms	1	2.7061ms	2.7061ms	2.7061ms

Profiling data

Time to copy
data onto GPU

Time to copy
data back
from GPU

Name
[CUDA memcpy HtoD]
[CUDA memcpy DtoH]
set_problem_2d\$ck_L240_28
set_timestep\$ck_L92_5
allocate_data\$ck_L30_1
artificial_viscosity\$ck_L198_16
pressure_acceleration\$ck_L128_9

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

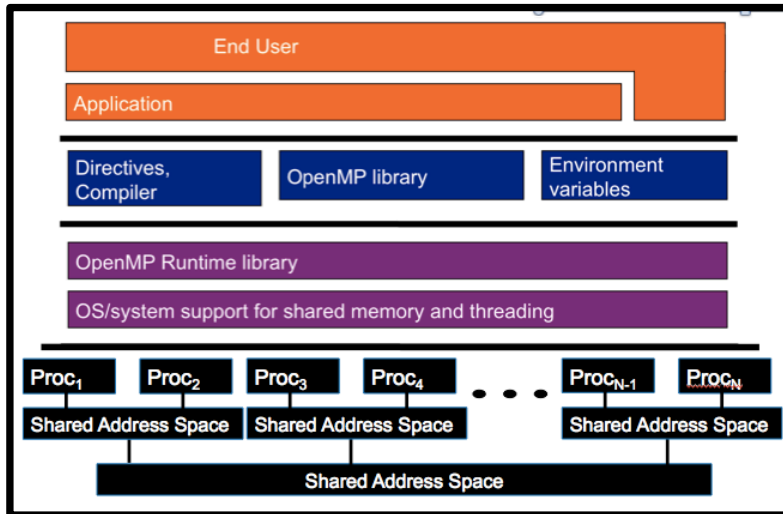
- Use the vadd program from the previous exercise and explore how it executes on a GPU using nvprof.
- You will need to **edit the job submission files** to submit the job to the queue with nvprof. For example:
 - Replace the line
 - `./vadd`
 - with the line (for example)
 - `nvprof --print-gpu-trace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

Agenda

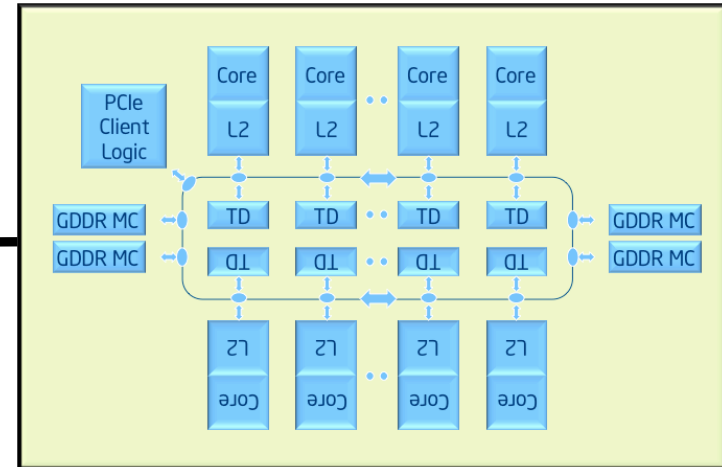
- OpenMP overview
- The device model in OpenMP
- Understanding execution on the GPU with nvprof
-  • Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

OpenMP device model: Examples

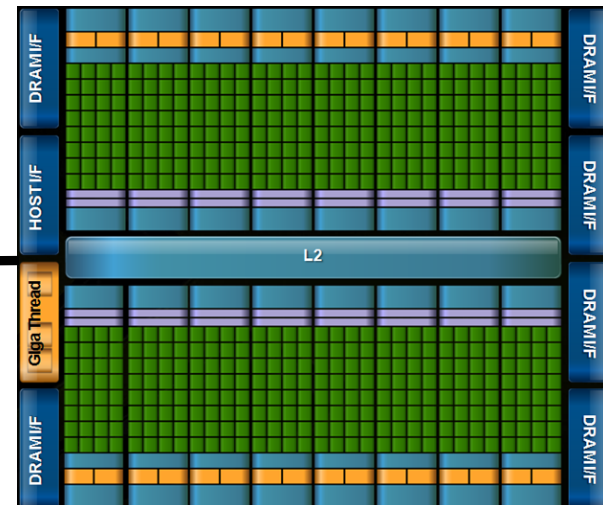
A couple of key devices considered when designing the device model in OpenMP



Host

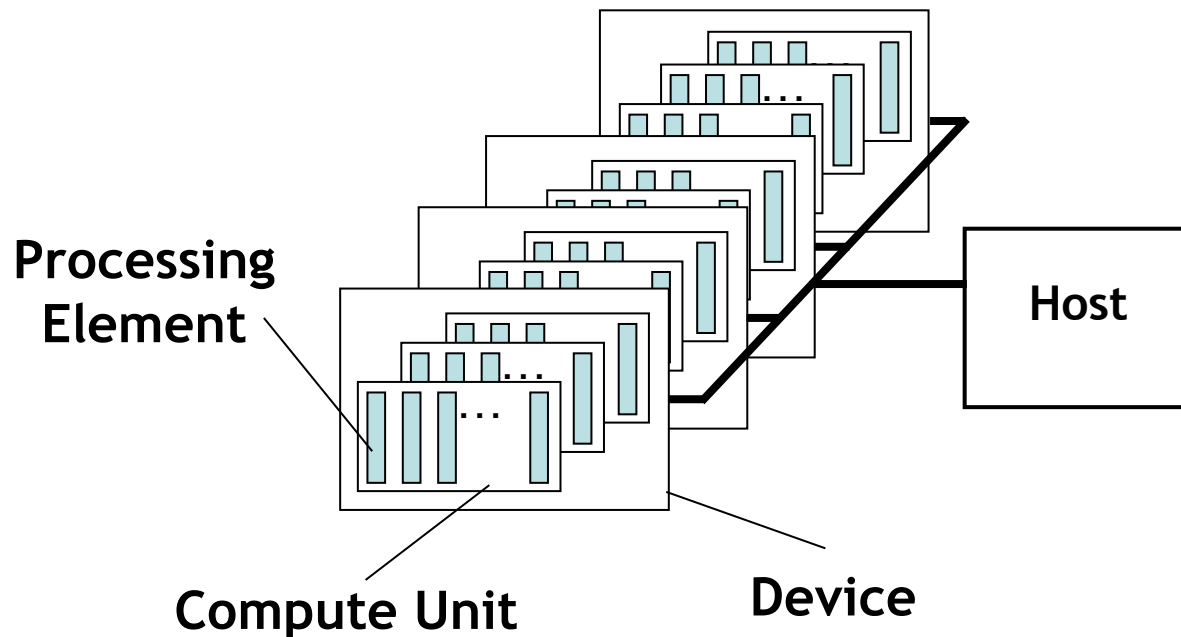


Target Device: Intel® Xeon Phi™ processor



Target Device: GPU

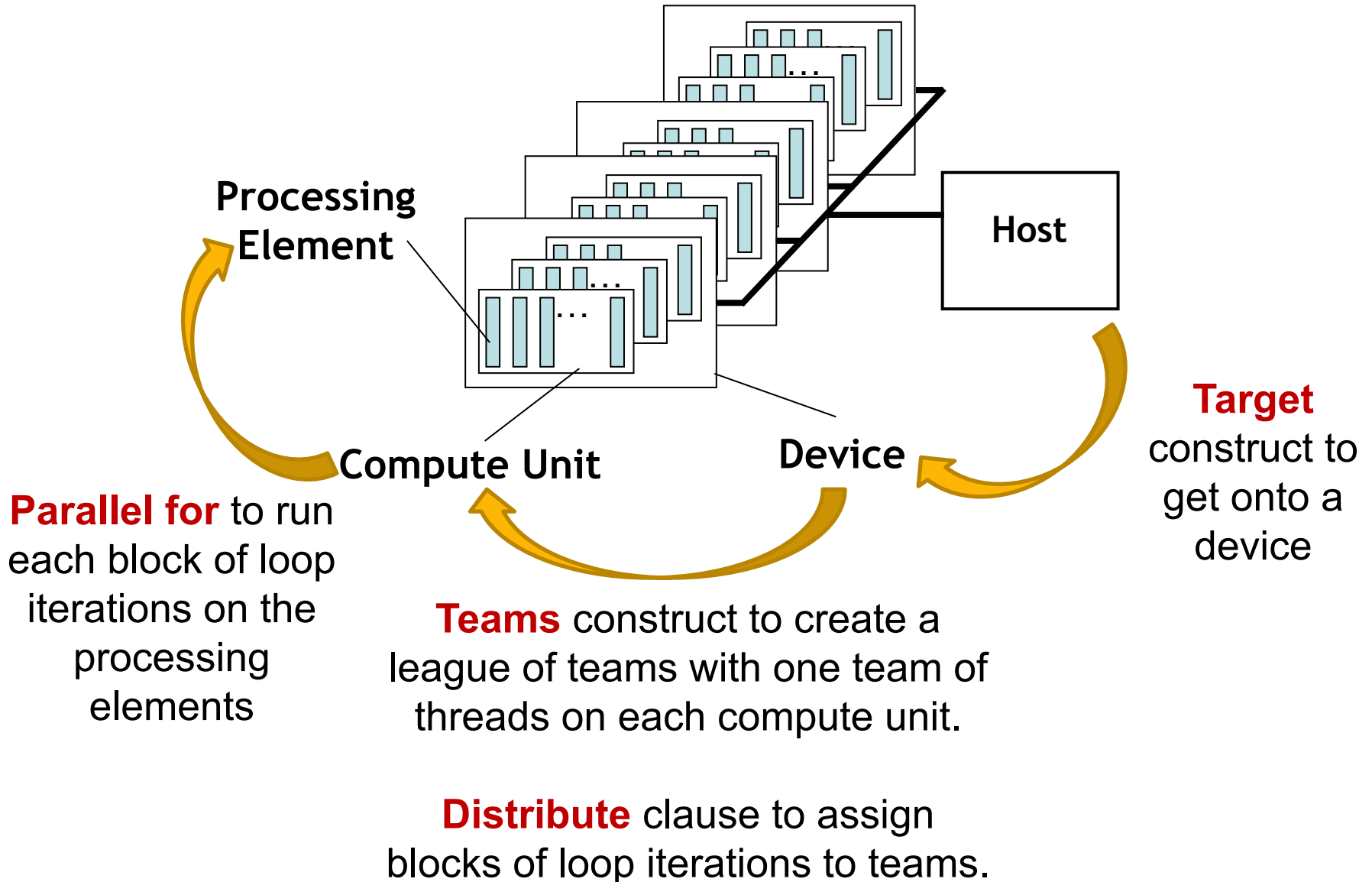
A Generic Host/Device Platform Model



- 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**

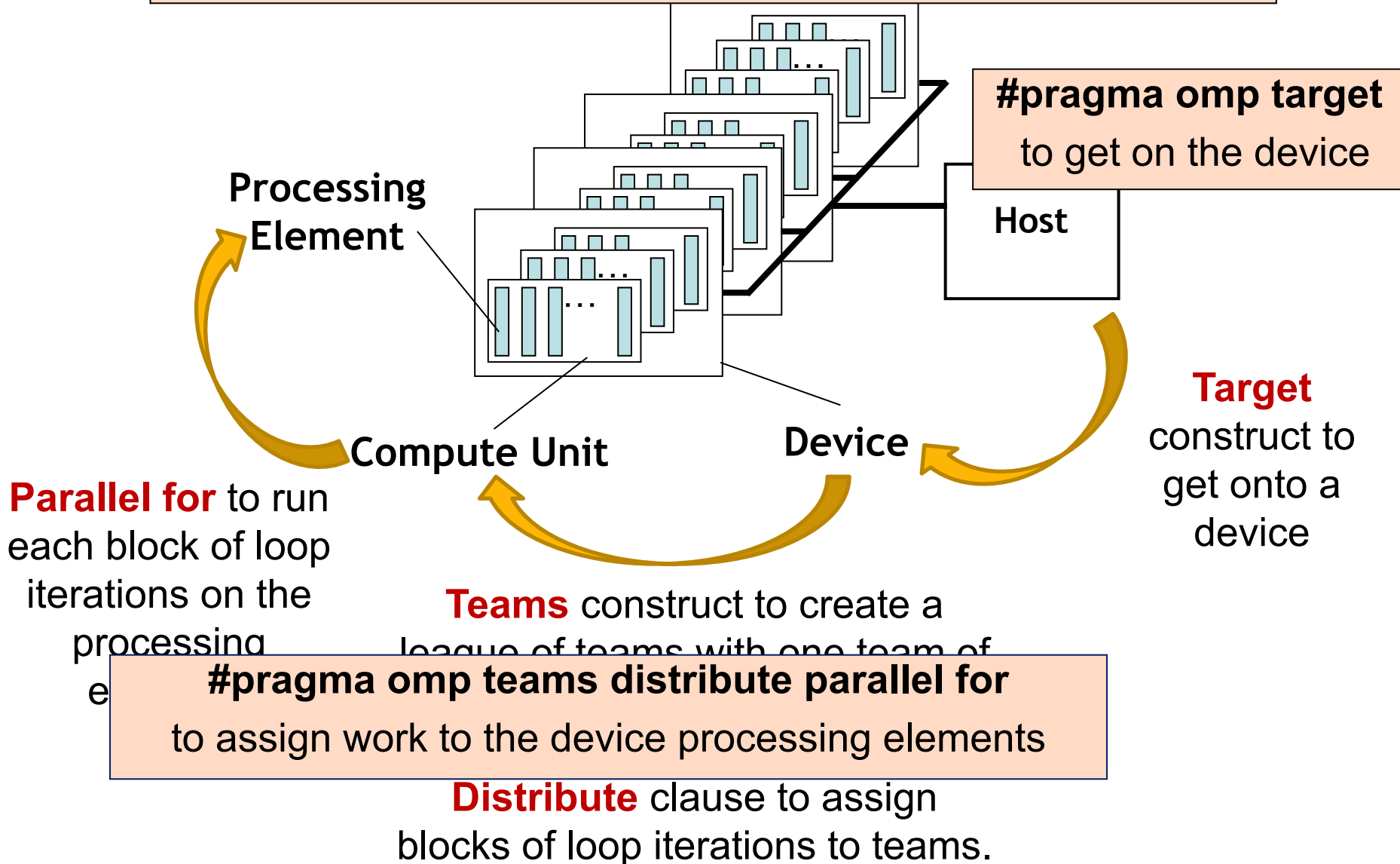
Third party names are the property of their owners.

Our host/device Platform Model and OpenMP



Our host/device Platform Model and OpenMP

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



Consider the familiar VADD example

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

    // initialize a, b and c ...

    for(int 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];

    // initialize a, b and c ...
```

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

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

// Test results, report results ...
}
```

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

Commonly used clauses on teams distribute parallel for

- The basic construct is:

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

- The most commonly used clauses are:
 - **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
 - **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

Data movement
can be explicitly
controlled with
the map clause

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

- 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 on the host.
 - **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 behaviour 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 = (int*)malloc(sizeof(int)*N);  
    #pragma omp target  
    #pragma omp teams distribute parallel for  
    for(int i = 0; i < N; ++i) {  
        // A, B, N and i all exist here  
        // B is a pointer – the data 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 = (int*)malloc(sizeof(int)*N);  
    #pragma omp target map(tofrom: B[0:N])  
    #pragma omp teams distribute parallel for  
    for(int i = 0; i < N; ++i) {  
        // A, B, N and i all exist here  
        // The data that B points to DOES exist here!  
    }  
}
```

Exercise: Jacobi solver

- Start from the provided `jac_solv` 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[0:N])
map(from: list[0:N]) map(tofrom: list[0:N])`
 - `#pragma omp target teams distribute parallel for`

Jacobi Solver (serial 1/2)

```
<<< allocate and initialize the matrix A and >>>
<<< vectors xold, xnew and b >>>
while((conv > TOL) && (iters<MAX_ITERS))
{
    iters++;
    for (int i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (int 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)

```
conv = 0.0;
for (int i=0; i<Ndim; i++){
    TYPE 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, 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
        for (int i=0; i<Ndim; i++){
            xnew[i] = (TYPE) 0.0;
            for (int 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)

```
conv = 0.0;
#pragma omp target map(to: xnew[0:Ndim], xold[0:Ndim]) \
                    map(tofrom:conv)
#pragma omp teams distribute parallel for reduction(+:conv)
    for (int i=0; i<Ndim; i++){
        TYPE 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)


```
conv = 0.0;  
#pragma omp target map(to:xnew[0:Ndim], xold[0:Ndim]) map(tofrom:conv)  
#pragma omp teams distribute parallel for reduction(+:conv)  
for (int i=0; i<Ndim; i++){  
    TYPE 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
NVIDIA® 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++;
```

Typically over 4000 iterations!

```
#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  
for (int i=0; i<Ndim; i++){  
    xnew[i] = (TYPE) 0.0;  
    for (int 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];  
}  
conv = 0.0;
```

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

```
#pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \  
                    map(tofrom:conv)  
#pragma omp teams distribute parallel for reduction(+:conv)  
for (int i=0; i<Ndim; i++){  
    tmp = xnew[i]-xold[i];  
    conv += tmp*tmp;  
}  
conv = sqrt((double)conv);  
// Pointer swap...
```

Each iteration, **copy**
 $2*Ndim*sizeof(TYPE)$
bytes **from** device

Each iteration, **copy**
 $2*Ndim*sizeof(TYPE)$
bytes **to** device

Target data directive

- The **target enter data** and **target exit data** provide explicit control of the device data environment using map clauses

```
#pragma omp target enter data map(to: ...)  
#pragma omp target exit data map(from: ...)
```

- Data can be allocated and/or copied to the device on a **target enter data**, or copied back and/or destroyed with a **target exit data**

```
#pragma omp target enter data map(to: A[0:N],B[0:N],C[0:N])
```

```
#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}
```

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

Target update directive

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

```
#pragma omp target enter data map(to: A[0:N],B[0:N],C[0:N])
```

```
#pragma omp target
```

```
{do lots of stuff with A, B and C}
```

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

Copy A on the device to A on the host.

```
{do something on the host}
```

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

Copy A on the host to A on the device.

```
#pragma omp target
```

```
{do lots of stuff with A, B, and C}
```

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

Exercise

- Modify your parallel `jac_solv` from the last exercise.
- Use the **target enter data** construct to create a data region. Manage data movement with map clauses to minimize data movement.
 - **#pragma omp target enter data map(to: ...)**
 - **#pragma omp target exit data map(from: ...)**
 - Array section syntax: **array-name[offset : length]**, e.g. `A[0:N]`

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
```

```
    for (int i=0; i<Ndim; i++){  
        xnew[i] = (TYPE) 0.0;  
        for (int 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)

```
conv = 0.0;
#pragma omp target map(tofrom: conv)
#pragma omp teams distribute parallel for reduction(+:conv)
    for (int i=0; i<Ndim; i++){
        TYPE tmp = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
conv = sqrt((double)conv);
// Pointer swap
} // end while loop
```

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

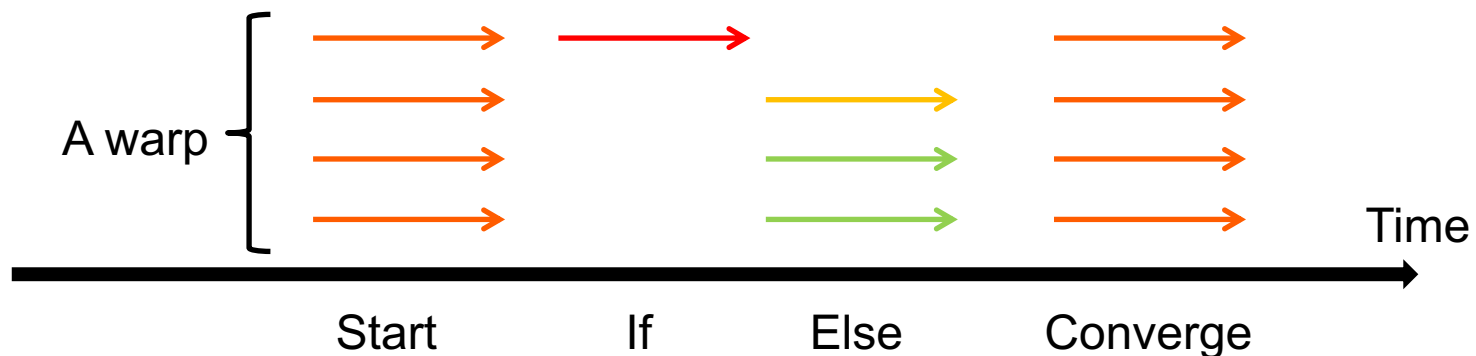
System	Implementation	Ndim = 4096
NVIDIA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target enter data	18.37 secs

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
 - Provide the MIMD abstraction
- Branch behavior
 - Each branch path will be executed serially (e.g. if/else)
 - Threads not following the current path will be 'disabled'



Branching

- GPUs tend not to support speculative execution, which means that branch instructions have high latency.
- Often important to avoid branching on SIMD CPUs too.
- This latency can be hidden by switching to alternative work-items/work-groups, but avoiding branches where possible is usually 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**).
- 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.
- Modern HPC GPUs can cope with branchy code increasingly well.

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];

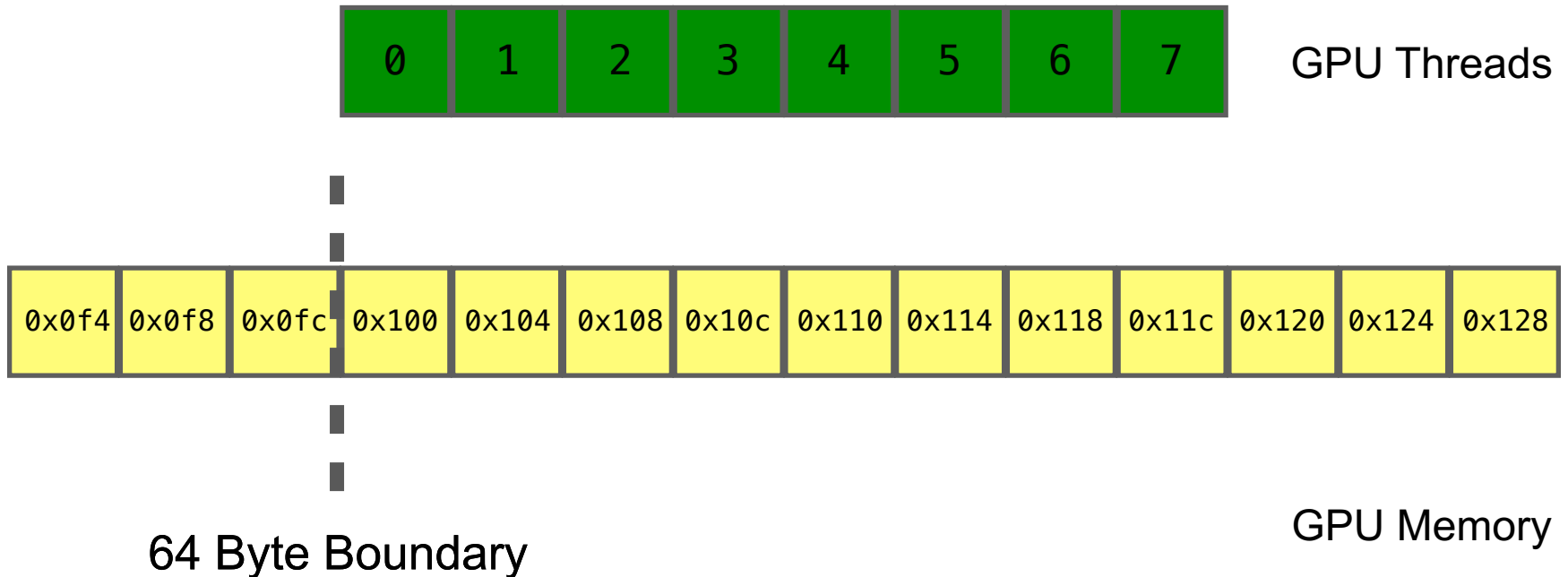
    // still pretty good
    const int c = 3;
    float val2 = memA[id + c];

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

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

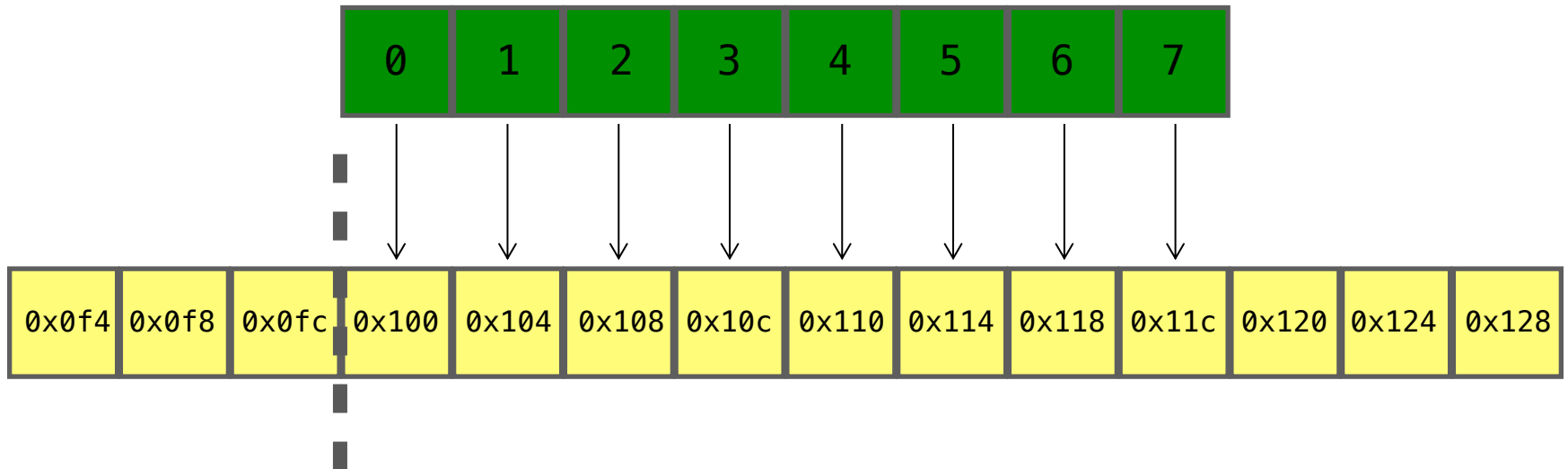
    float val4 = memA[loc];
}
```

Memory access patterns



Memory access patterns

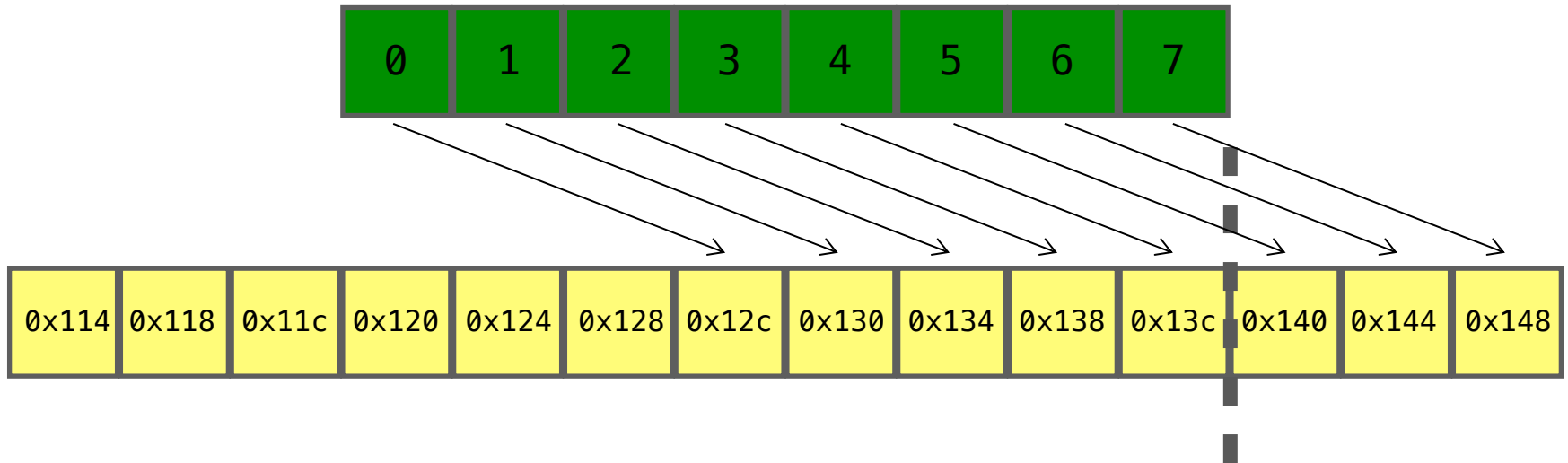
```
float val1 = memA[id];
```



64 Byte Boundary

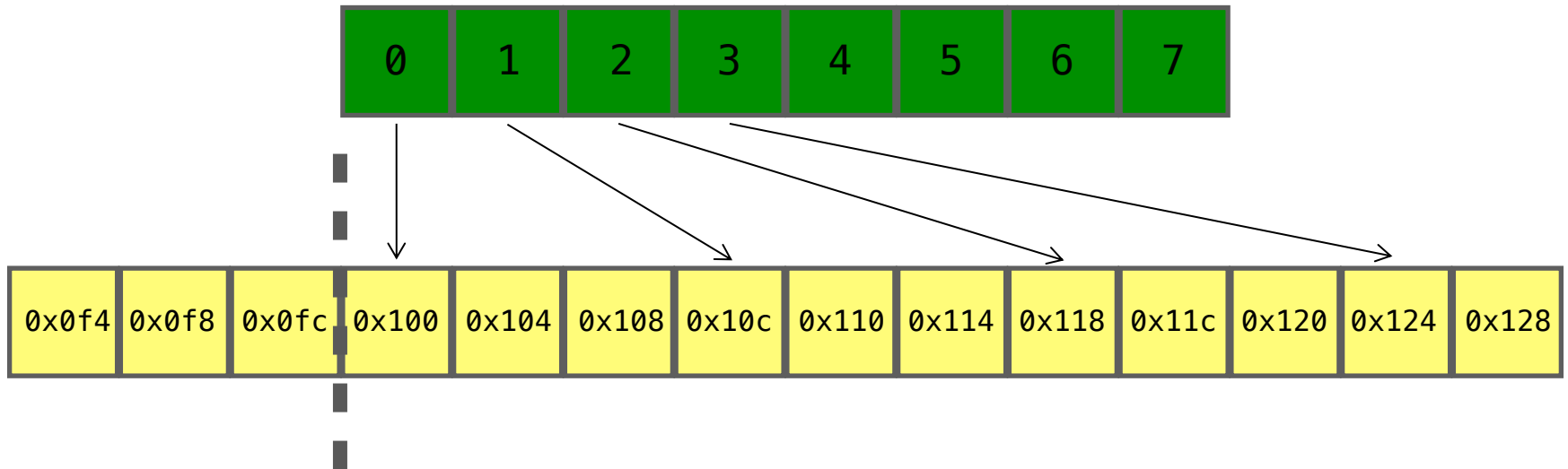
Memory access patterns

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



Memory access patterns

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

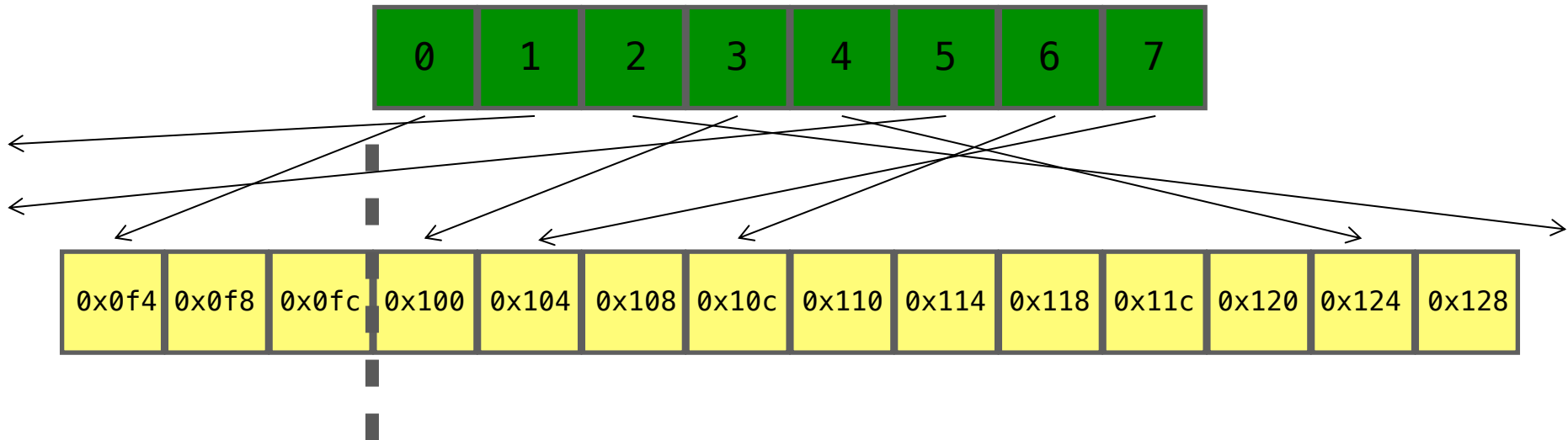


64 Byte Boundary

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

Memory access patterns

```
const int loc =  
    some_strange_func(id);  
  
float val4 = memA[loc];
```



64 Byte Boundary

Exercise

- Modify your parallel jacobi_solver from the last exercise.
- Experiment with the optimizations we've discussed.
- 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
    for (int i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (int 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)

```
conv = 0.0;
#pragma omp target map(tofrom: conv)
#pragma omp teams distribute parallel for reduction(+:conv)
  for (int i=0; i<Ndim; i++){
    TYPE tmp = xnew[i]-xold[i];
    conv += tmp*tmp;
  }
conv = sqrt((double)conv);
// Pointer swap
} // end while loop
#pragma omp target exit data map(from:xold[0:Ndim], xnew[0:Ndim])
```

System	Implementation	Ndim = 4096
NVIDIA® K20X™ GPU	Target dir per loop	131.94 secs
	Above + target data region	18.37 secs
	Above + reduce branching	13.74 secs

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

```
#pragma omp enter target data map(from: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
    for (int i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (int 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)

```
conv = 0.0;
#pragma omp target map(tofrom: conv)
#pragma omp teams distribute parallel for reduction(+:conv)
for (int i=0; i<Ndim; i++){
    TYPE tmp  = xnew[i]-xold[i];
    conv += tmp*tmp;
}
conv = sqrt((double)conv);
// Pointer swap
} // end while loop
#pragma omp target exit data map(from:xold[0:Ndim], xnew[0:Ndim])
```

System	Implementation	Ndim = 4096
NVIDIA® 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

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/>

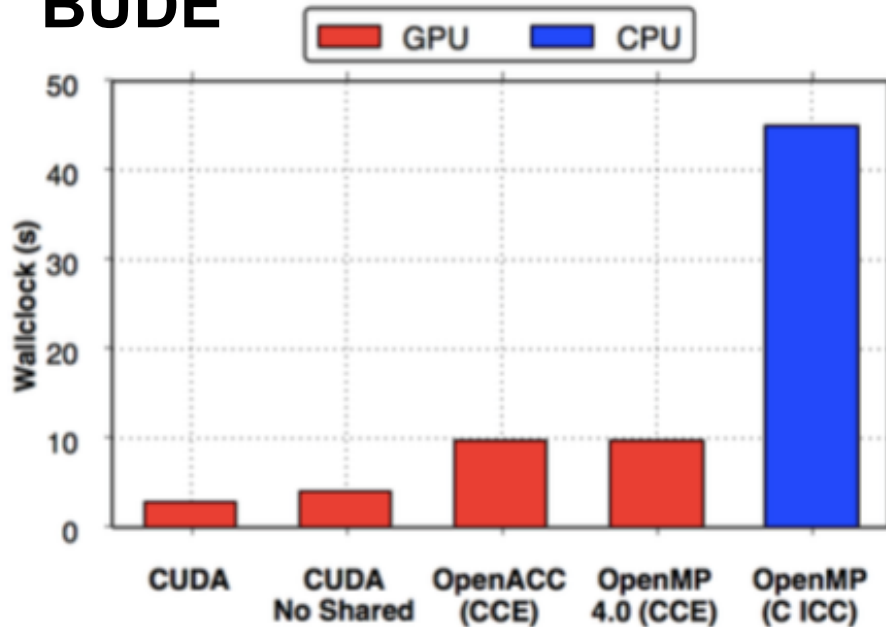
Performance?

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

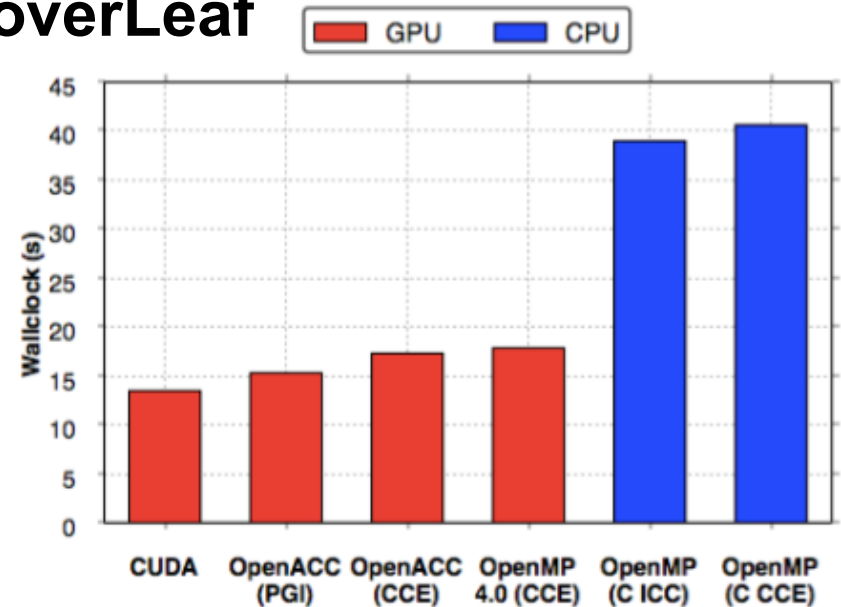
Immediately we see impressive performance compared to CUDA

Clearly the Cray compiler leverages the existing OpenACC backend

BUDE



CloverLeaf



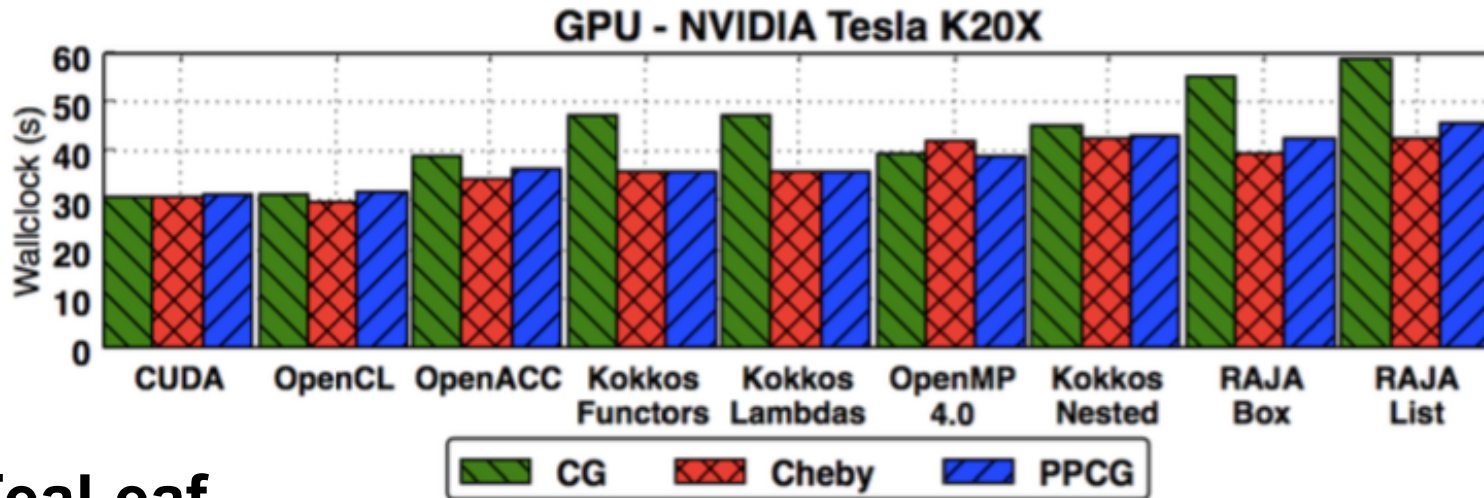
Even with OpenMP 4.5 there is still no way of targeting shared memory directly.

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

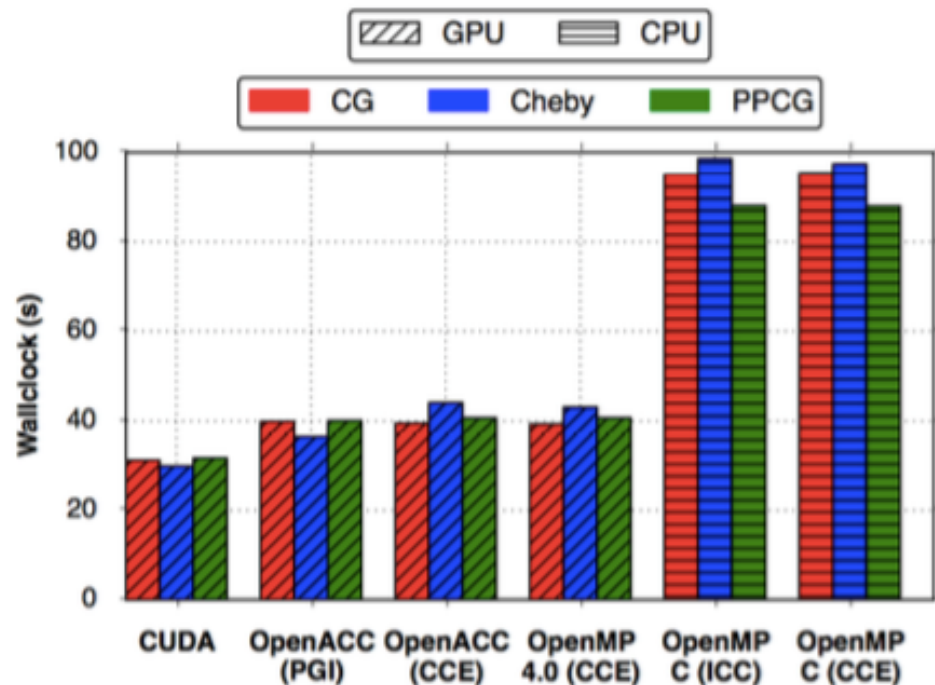
Martineau, M., McIntosh-Smith, S. Gaudin, W., *Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model*, 2016, HIPS'16

Performance?

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



TeaLeaf



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

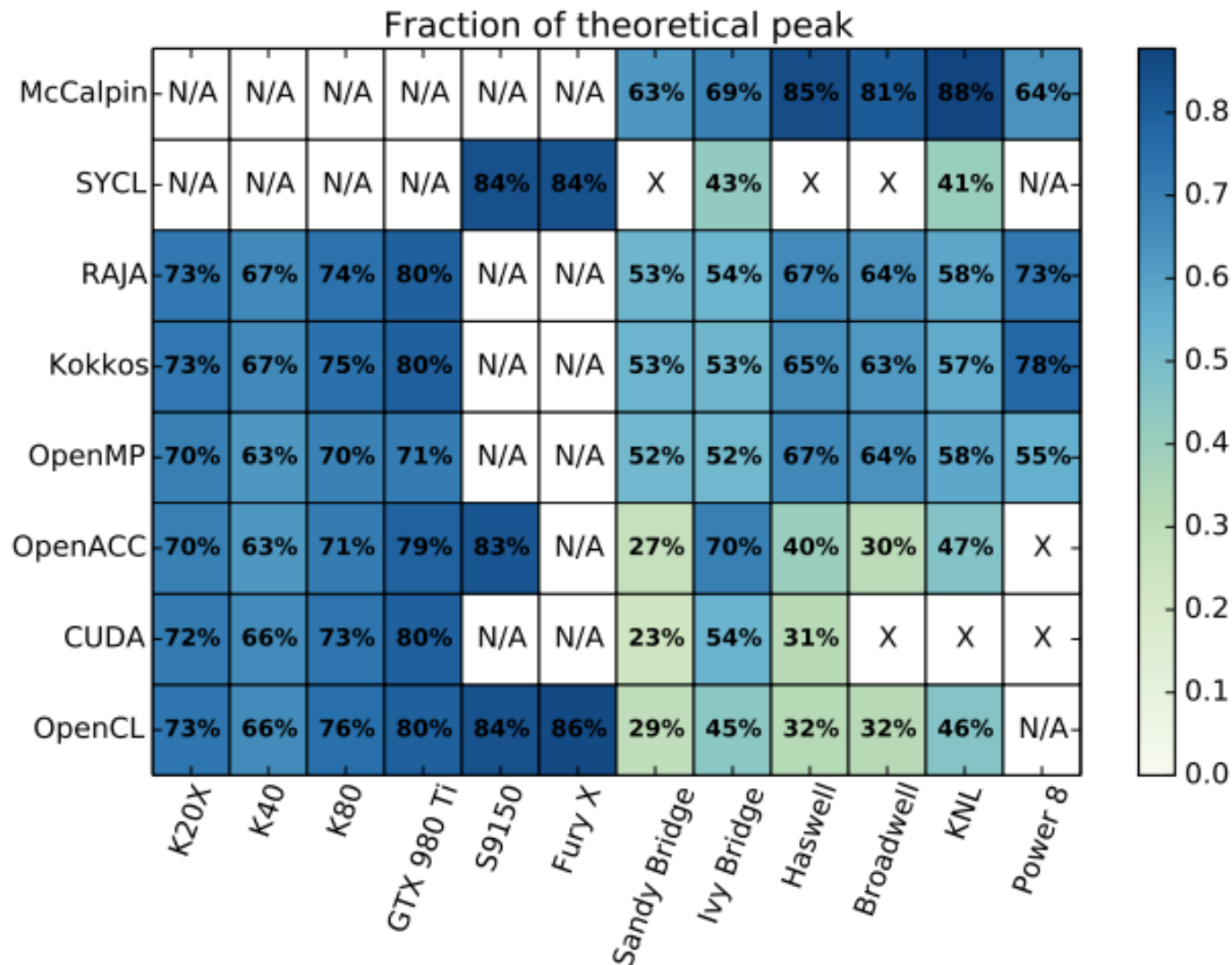
Martineau, M., McIntosh-Smith, S. Gaudin, W., *Assessing the Performance Portability of Modern Parallel Programming Models using TeaLeaf*, 2016, CC-PE

How do you get good performance?

- Our finding so far:
 - You can achieve good performance with OpenMP 4.5.
- 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 *nvprof* to carefully optimise each kernel.

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*

Combined Construct

- For performance, prefer to use the full combined construct:

`#pragma omp target teams distribute parallel for`

- The construct makes a lot of guarantees to the compiler, enabling optimisations.
- Real applications will have algorithms that are structured such that they can't immediately use the combined construct – *you have to port your code for the GPU.*

Conclusion

- You can program your GPU with OpenMP
- Implementations of OpenMP supporting target devices are evolving rapidly ... expect to see great improvements in quality and diversity.