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

James Reinders
self employed,
retired from Intel Corp.
sc16@jamesreinders.com

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

1

Disclaimer & Optimization Notice

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 <u>not</u> speaking for our employers.
    - We are <u>not</u> 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!
  - <u>Don't cheat</u>: Do Not look at the solutions before you complete an exercise … even if you get really frustrated.

Plan

Module	Concepts	Exercises
OpenMP overview	• Intro to OpenMP • Creating threads • Parallel loops	• pi
The device model in OpenMP	• Intro to the Target directive	
Working with the target directive	• Target directive details	• Jacobi solver
Controlling memory movement	• Target data environment	• Jacobi solver with explicit data movement
Optimizing GPU code	• Common GPU optimizations	• Optimized Jacobi solver
CPU/GPU portability	• Mapping between CPU and GPU concepts	

Cray is Awesome!!!

They are providing hardware and software for this tutorial

Visit us at booth #

1731

CRAY®

CRAY CUSTOMERS ARE AMAZING.
And we're showcasing some of them at SC16.

Cray customers are fighting famine, predicting weather and improving drug discovery success rates. These are just a few of their stories, which we'll be highlighting at SC16. Stop by booth 1731 for more details.

Marketing Partner Network Agreement

https://partners.cray.com/marketing-partner-network-agreement.pdf

The Cray XC40 system

- Logging on:
  - `ssh <user>@swan.cray.com`

- Get the code:
  - git clone https://github.com/UoB-HPC/sc16-tutorial

- Load some modules:
  - `module swap craype-{broadwell,ivybridge}`
  - `module load craype-accel-nvidia35`

- Build the code:
  - `make` # builds everything
  - `make <binary>` # builds the specific binary

- Running jobs (batch jobs via the queue)
  - We have provided one submission script per binary:
  - `qsub submit_<exe>`

Agenda

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

OpenMP* overview:

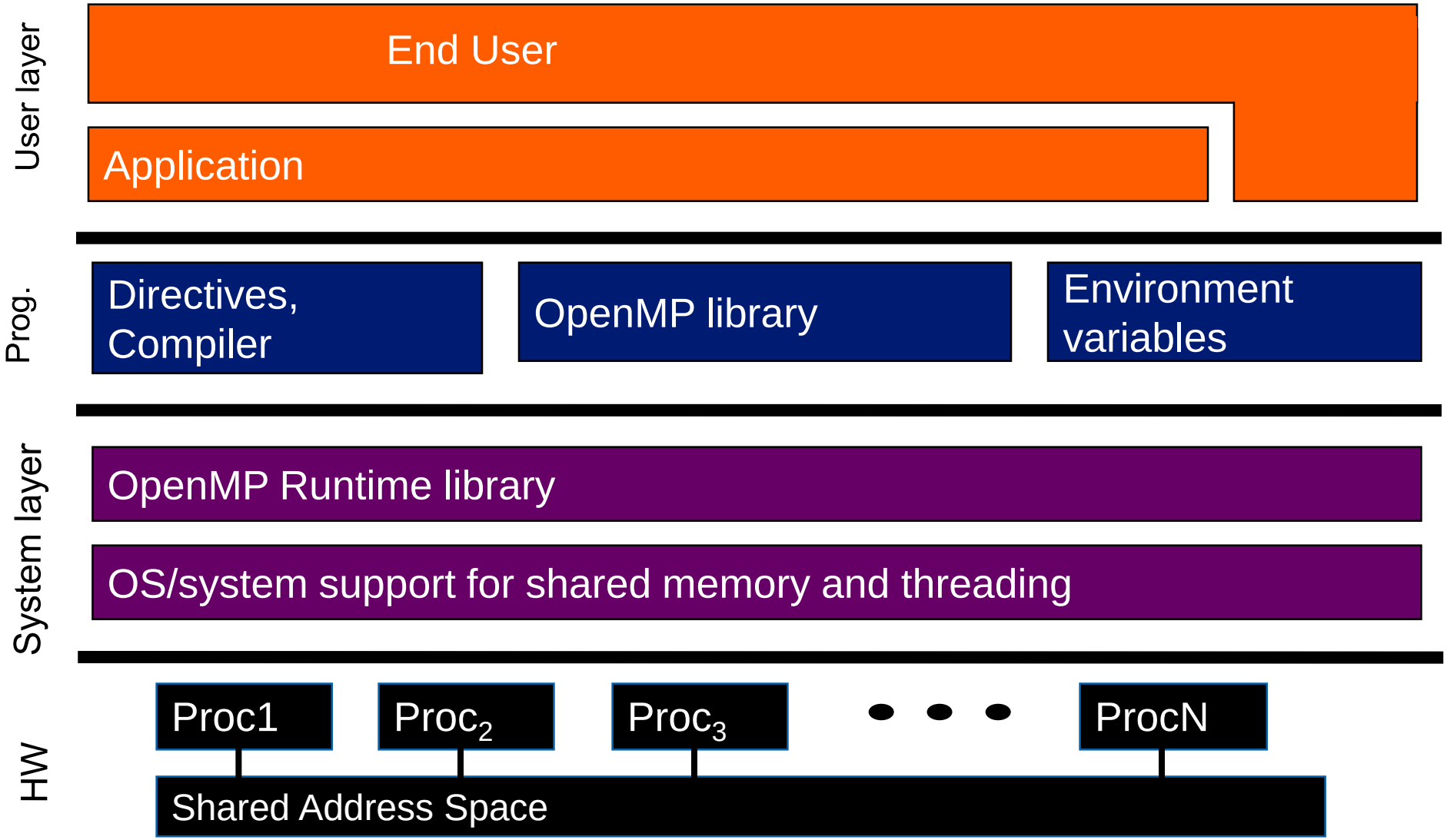
```
C$OMP FLUSH
#pragma omp critical
C$OMP THREADPRIVATE(/ABC/)
CALL OMP_SET_NUM_THREADS(10)
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

OpenMP: An API for Writing Multithreaded Applications

- A set of **compiler directives, library routines,** and **environment variables**, 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.

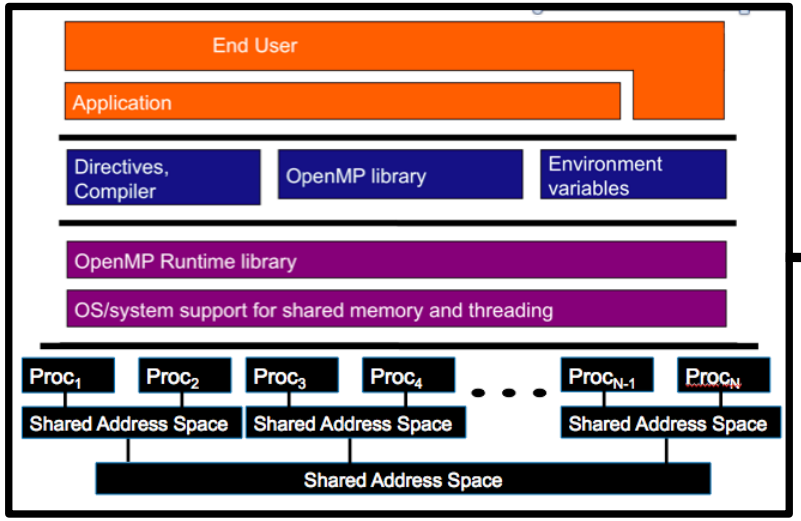
OpenMP basic definitions: Basic Solution stack
Versions 1.0 to 3.1



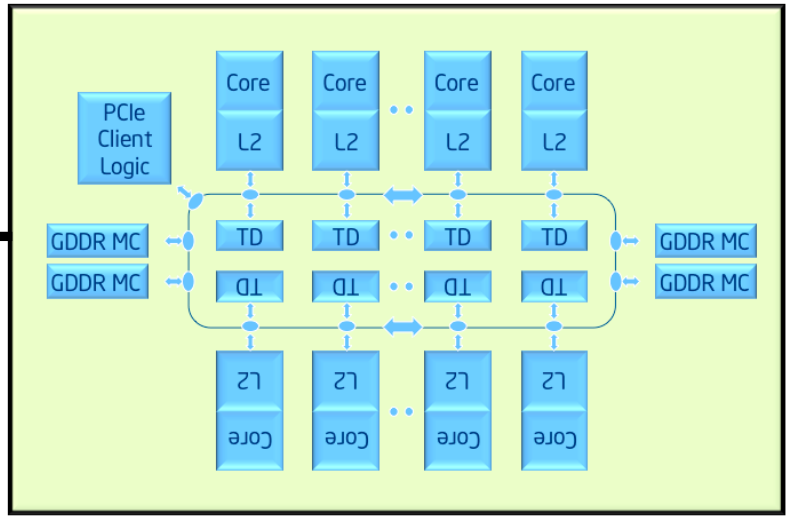
OpenMP basic definitions: Target solution stack

Version 4.0-4.5

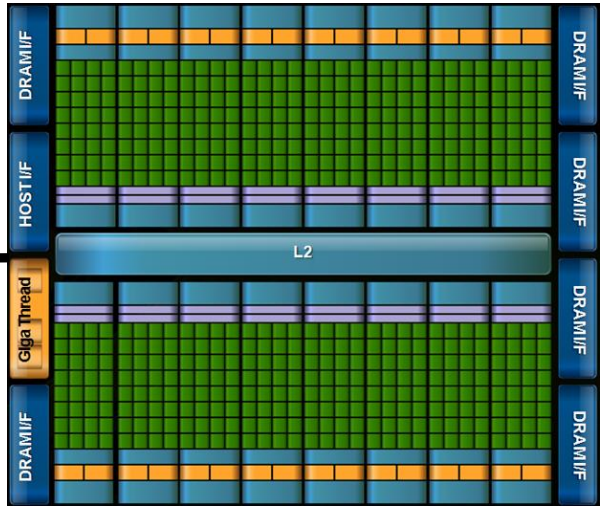
Supported (since OpenMP 4.0) with target, teams, distribute, and other constructs



Host



Target Device: Intel® Xeon Phi™ coprocessor



Target Device: GPU

OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp construct [clause [clause]…]

  – Example

  #pragma omp parallel num_threads(4)

- Function prototypes and types in the file:

  #include <omp.h>

- Most OpenMP* constructs apply to a “structured block”.
  – Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  – It’s OK to have an exit() within the structured block.

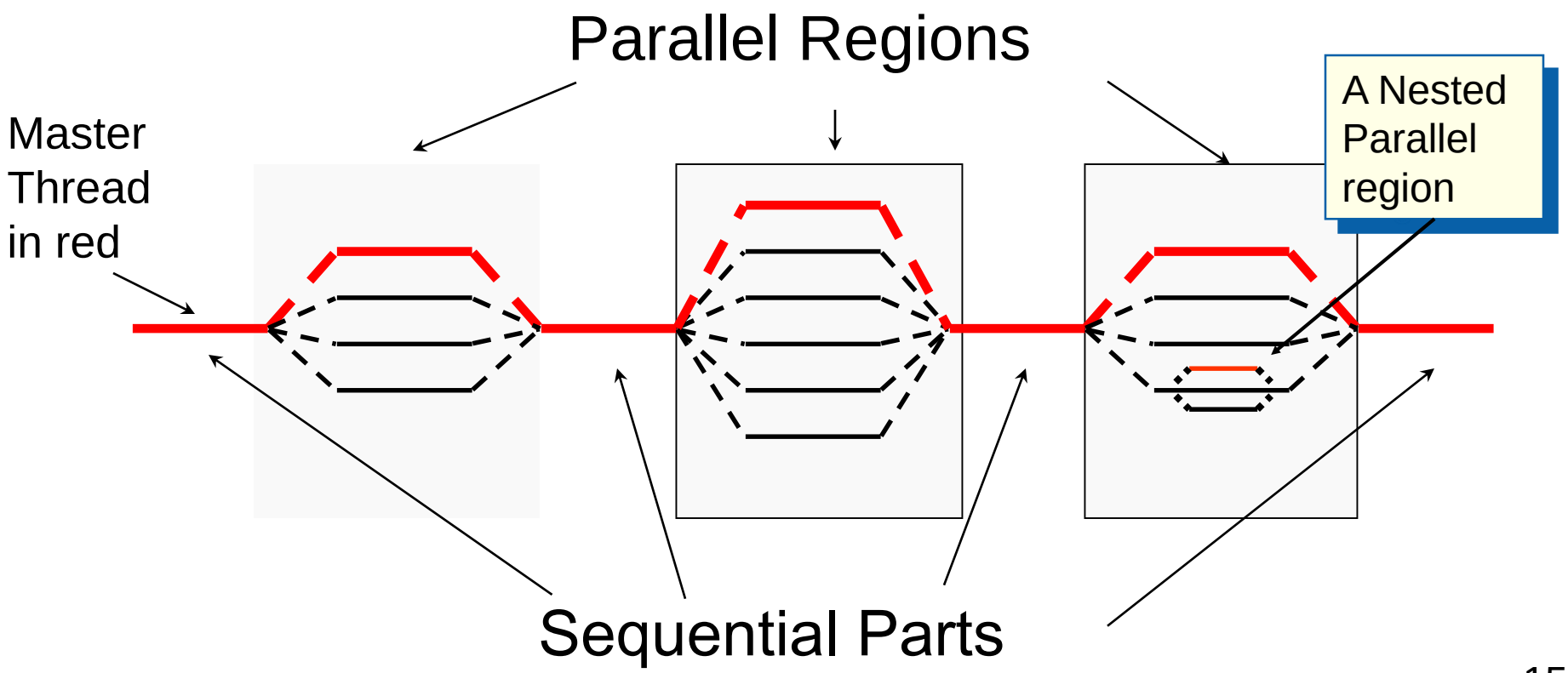
OpenMP overview:
How do threads interact?

- OpenMP is a multi-threading, shared address model
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - Race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization

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.



Loop worksharing constructs

Sequential code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel
#pragma omp for
        for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

Loop worksharing constructs

Sequential code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP combined parallel worksharing construct

```
#pragma omp parallel for
        for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP data environment - motivation

When operating in parallel – proper sharing, or NOT sharing is essential to correctness and performance.

```
#pragma omp parallel for
{
        for(i=0; i<n; i++){
                tmp= 2.0*a[i];
                a[i] = tmp;
                b[i] = c[i]/tmp;
        }
}
```

```
#pragma omp parallel for private(temp)
{
        for(i=0; i<n; i++){
                tmp= 2.0*a[i];
                a[i] = tmp;
                b[i] = c[i]/tmp;
        }
}
```

By default, all threads share a common address space. Therefore, all threads will be modifying *tmp* simultaneously in the code on the LEFT.

On the RIGHT –**private** clause directs that each thread will have an (uninitialized) private copy.

Initialization is possible with "**firstprivate**" and grabbing the last value is possible with "**lastprivate**." **Reduction**s are important enough to have a special clause, and **default**s can be set (including to "none.")

OpenMP data environment - summary

1. Variables are shared by default.
2. Global variables are shared by default.
3. Automatic variables within subroutines called from within a parallel region are private (reside on a stack private to each thread), unless scoped otherwise.
4. Default scoping rule can be changed with default clause.

Data scope attribute clause description
private clause: declares the variables in the list to be private (not shared) to each thread.
firstprivate clause: declares variables in the list to be **private** *plus* the private variables are initialized to the value of the variable when the construct is encountered (”entered”).
lastprivate clause: declares variables in the list to be **private** *plus* the value of from the sequentially last iteration of the associated loops, or the lexically last section construct, is assigned to the original list item(s) after the end of the construct.
shared clause: declares the variables in the list to be shared among all the threads in a team. All threads within a team access the same storage area for shared variables. Synchronization is generally advised if variables are updated.
reduction clause: performs a reduction on the scalar variables that appear in the list, with a specified operator.
default clause: allows the user to affect the data-sharing attribute of the variables appeared in the parallel construct.

Reduction

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
 double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
  for (i=0;i< MAX; i++) {
        ave + = A[i];
  }
  ave = ave/MAX;
```

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

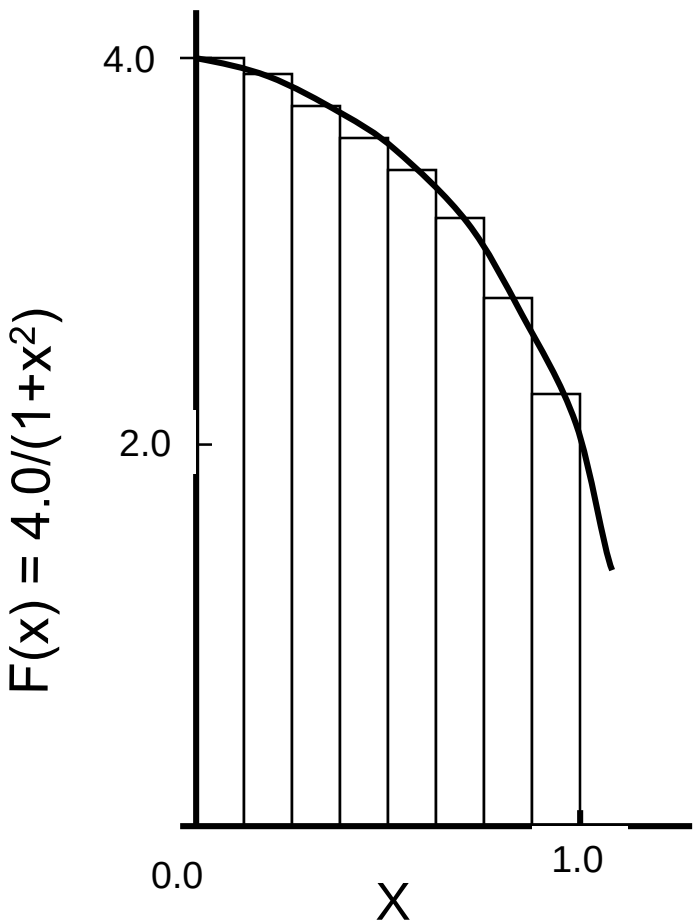
Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

OpenMP 4.0 added user defined reductions (discussed later).

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 OMP_exercises/pi.c

Exercise: Pi with loops

- Parallelize the pi program with a loop construct
- Try to minimize the number of changes made to the serial program.

```
#pragma omp parallel
#pragma omp for …   with clause such as reduction(op: list), private(list)
int omp_set_num_threads();
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
```

Goal: To verify that you can use our local environment to run a basic OpenMP program. This is also the opportunity to make sure everyone understands enough OpenMP to complete the rest of this tutorial.

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
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.

OpenMP Directives – Most Used

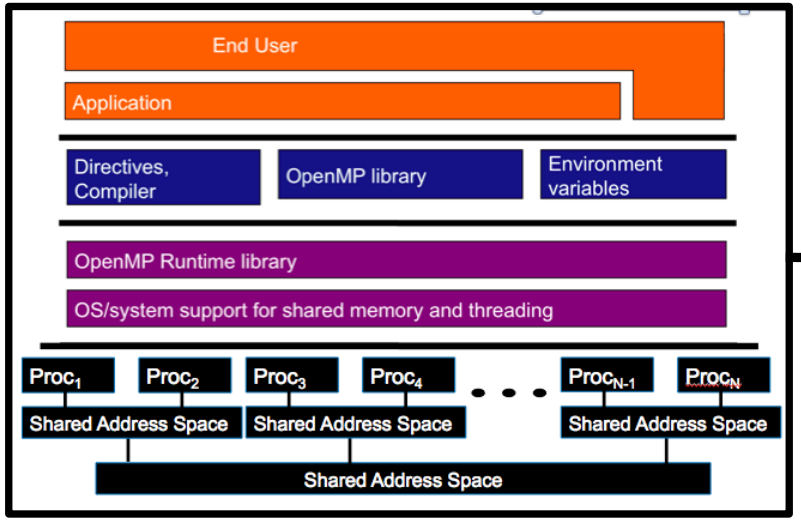
Directive name (put after #pragma omp)	**Description**
parallel	Defines a parallel region where all threads are executing everything, not just the “master” thread.
for	Split up loop iterations among a team of threads
parallel for	A shortcut for a PARALLEL region that contains a single for directive.
master	Defines a serial region where only the master thread is executing.
critical	Defines a serial region where only one thread can run at a time.
threadprivate	Makes the named COMMON blocks or variables private to a thread. The list argument consists of a comma-separated list of COMMON blocks or variables.
target	Offload support to “target” devices (e.g., GPUs).

Agenda

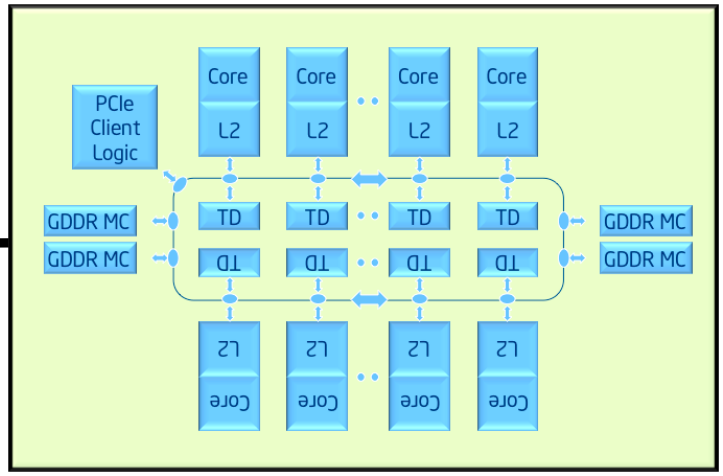
- OpenMP overview
- The device model in OpenMP
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

OpenMP basic definitions: Target solution stack

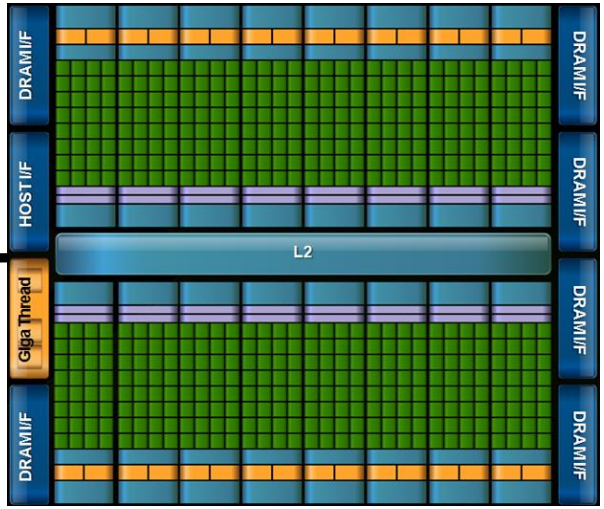
Supported (since OpenMP 4.0) with target, teams, distribute, and other constructs



Host



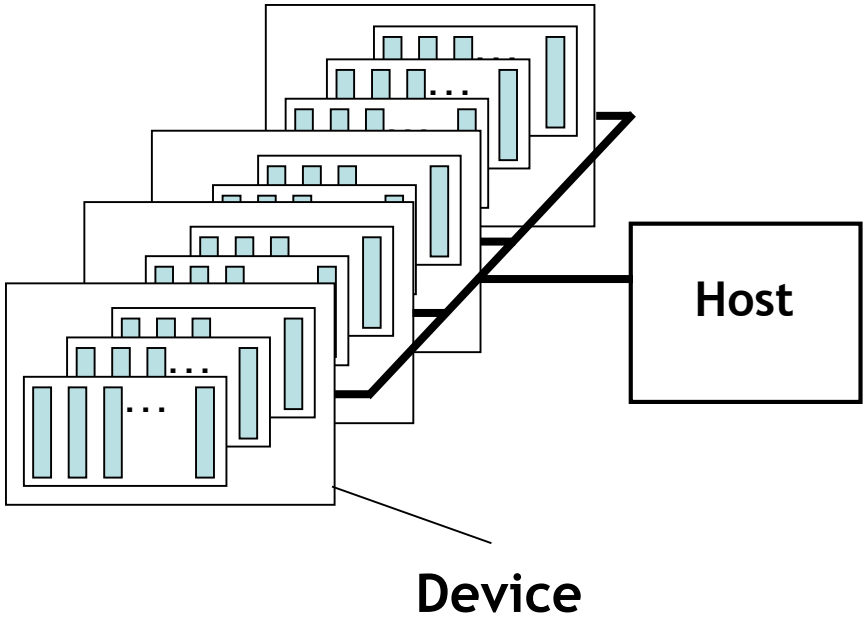
Target Device: Intel® Xeon Phi™ processor



Target Device: GPU

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());
}
```

Target directive

- The target construct offloads a code region to a device.

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

- An initial thread running on the device executes the code in the code block.

```
#pragma omp target
{
    #pragma omp parallel for
        {do lots of stuf}
}
```

Target directive

- The target construct offloads a code region to a device.

```
#pragma omp target device(1)
{….}  // a structured block of code
```

Optional clause to select some device other than the default device.

- An initial thread running on the device executes the code in the code block.

```
#pragma omp target
{
    #pragma omp parallel for
        {do lots of stuff}
}
```

The target data environment

- The target clause creates a data environment on the device:

```
int i, a[N], b[N], c[N];
#pragma omp target
```

Original variables on the host: N, i, a, b, c …

```
#pragma omp parallel for private(i)
        for(i=0;i<N;i++){
            c[i]+=a[i]+b[i];
        }
```

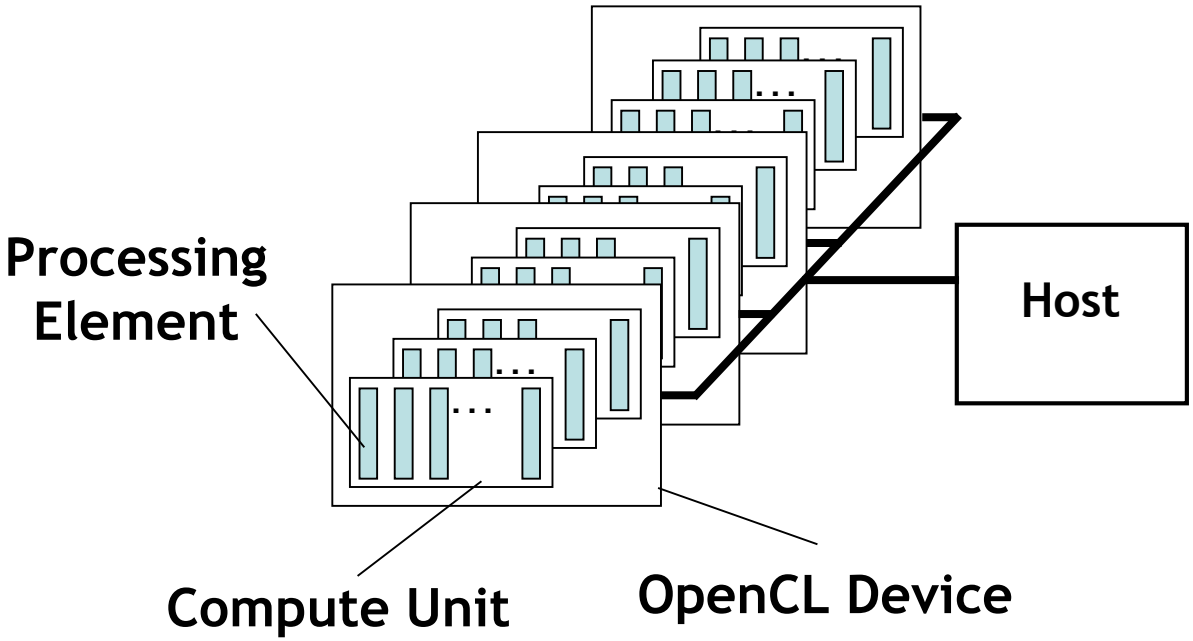
Are mapped onto the corresponding variables on the device: N, i, a, b, c …

- Originals variables copied into corresponding variables before the initial thread begins execution on the device.
- Corresponding variables copied into original variables when the target code region completes

Agenda

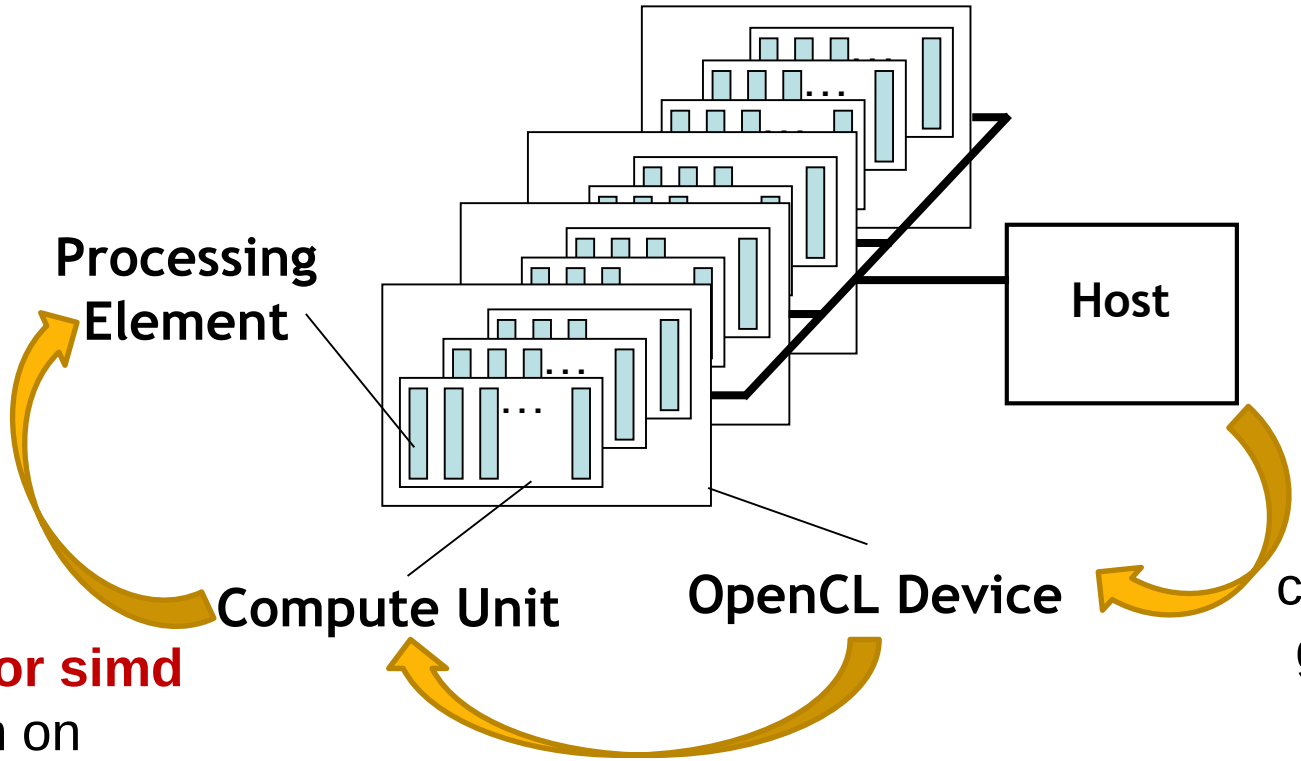
- OpenMP overview
- The device model in OpenMP
- Working with the target directive
- Controlling memory movement
- Optimizing GPU code
- CPU/GPU portability

OpenCL Platform Model



- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL 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***

OpenCL Platform Model and OpenMP



Target construct to get onto a device

Parallel for simd to run on processing elements

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

Distribute clause to assign work-groups 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 …

}
```

We will explore how to map this code onto Many-core processors (GPU and CPU) using the OpenMP constructs:

- target
- teams
- distribute

2 Constructs to control devices

- **teams** construct creates a league of thread teams:

  #pragma omp teams
- Supports the clauses:
  - num_teams(int) … the number of teams in the league
  - thread_limit(int) … max number of threads per team
  - Plus private(), firstprivate() and reduction()

- **distribute** construct distributes iterations of following loops to the master thread of each team in a league:

  #pragma omp distribute

  //immediately following for loop(s)
- Supports the clauses:
  - dist_schedule(static [, chunk] … the number of teams in the league.
  - collapse(int) … combine n closely nested loop into one before distributing.
  - Plus private(), firstprivate() and reduction()

Vadd: with OpenMP

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

  #pragma omp teams num_teams(NCU) simdlen(NPE)

  #pragma omp distribute
   for (ib=0;ib<N; ib=ib+wrk_grp_sz)

     #pragma omp parallel for simd
     for (i=ib;  i<ib+wrk_grp_sz; i++)
       c[i] += a[i] + b[i];
```

Offload to a device.

Describe a device … NCU compute units & NPE proc. elements per compute unit

Distribute work-groups to compute units

The body of this loop are the Individual work-items in a work-group

Vadd: with OpenMP

```
 int blksz=32, ib, Nblk;
 Nblk = N/blksz;
#pragma omp target map(to:a,b) map(tofrom:c)
   #pragma omp teams num_teams(NCU) thread_limit(NPE)

   #pragma omp distribute
    for (ib=0;ib<Nblk;ib++){
      int ibeg=ib*blksz;
      int iend=(ib+1)*blksz;
      if(ib==(Nblk-1))iend=N;

      #pragma omp parallel for simd
      for (i=ibeg; i<iend; i++)
        c[i] += a[i] + b[i];
   }
```

You can include any work-group wide code you want .. For example to explicitly control how iterations map onto work items in a work-group.

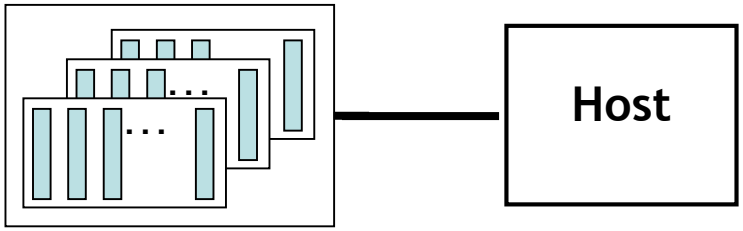
Vadd: with OpenMP

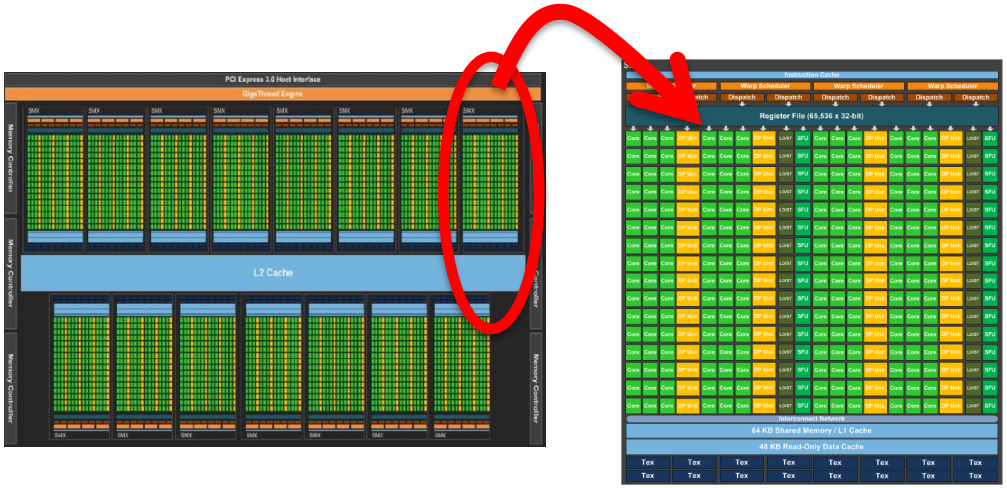
```
 // A more compact way to write the VADD code, letting the runtime
//  worry about work-group details

#pragma omp target map(to:a,b) map(tofrom:c)
 #pragma omp teams distribute parallel for simd
     for (i=0; i<N; i++)
             c[i] += a[i] + b[i];
```

In many cases, you might be better off to just distribute the parallel loops to the league of teams and leave it to the runtime system to manage the details. This would be more portable code as well.

OpenMP Platform Model: GPU

- Let's consider one host and one Device.





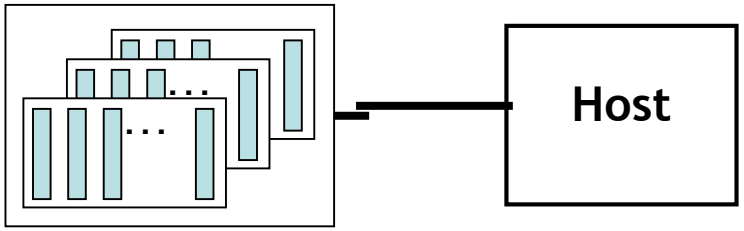
NVIDIA Tesla K20X (Kepler) GPU with 14 streaming multiprocessor cores*.

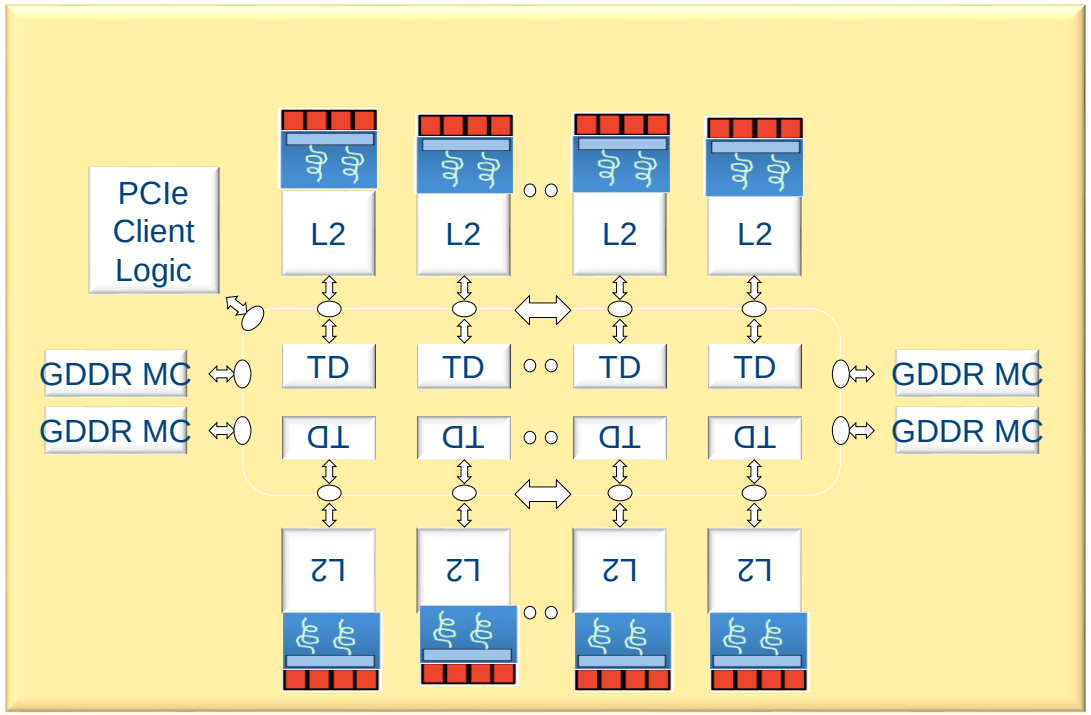
- Number of compute units: 14
- Number of PEs: 192
- Ideal work-group size: multiple of 32

*Source: http://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-

OpenMP Platform Model: Intel® Xeon Phi™ processor

- Let's consider one host and one Device.





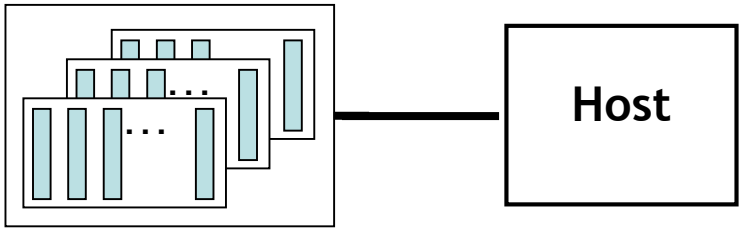
Intel® Xeon Phi™ processor: 60 cores, with 2 HW threads per core and a 512 bit wide vector unit.

- Number of compute units: 60
- Number of PEs: 2*vector width
- Ideal work-group size: multiple of vector width

Where "vector width" depends floating point type: 512/4*8 for float, 512/8*8 for double.

OpenMP Platform Model: summary

- Let's consider one host and one Device.



Device: GPU

NVIDIA Tesla K20X (Kepler) GPU with 14 streaming multiprocessor cores*.

- Number of compute units: 14
- Number of PEs: 192
- Ideal work-group size: multiple of 32

Device: Many Core CPU

Intel® Xeon Phi™ processor: 60 cores, with 2 HW threads per core and a 512 bit wide vector unit.

- Number of compute units: 60
- Number of PEs: 2*vector width
- Ideal work-group size: multiple of vector width

Where "vector width" depends floating point type: 512/4*8 for float, 512/8*8 for double.

Which pragma should I use?

- Different compilers associate teams, threads and vectors with physical hardware units.

- The catch all pragma is:

#pragma omp target teams distribute parallel for simd

- There are two common ways to describe a GPU
  - CUDA
    - Thread-blocks contain a number of threads.
    - Each thread-block executes on an SMX, with each cuda-thread running on a CUDA core.
  - OpenCL
    - Work-groups contain a number of work-items.
    - Each work-group executes on a compute unit, with each work-item running on a processing element

Which pragma should I use?

- The Cray compiler:
  - #pragma omp target teams distribute simd
  - teams: thread-blocks (CUDA) or work-groups (OpenCL)
  - distribute simd: threads (CUDA) or work-items (OpenCL)
    - If the compiler can auto-vectorize the loop, you can drop the simd
  - parallel for: the compiler ignores this with a warning. We often include it just to make the code more portable.

- The Clang compiler:
  - #pragma omp target teams distribute parallel for
  - teams: thread-blocks (CUDA) or work-groups (OpenCL)
  - distribute: work-share the loop between thread-blocks or work-groups
  - parallel for: work-share the divided loop between “threads in the thread-block” (CUDA) or work-items in a work-group (OpenCL)
  - Including simd often causes the compiler to fail

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 Sharing

- **Scalar** variables have *implicit* mapping behaviour
- OpenMP 4.0 implicitly mapped all scalar variables as **tofrom**
- OpenMP 4.5 implicitly maps scalar variables as **firstprivate**

#pragma omp target teams distribute parallel for ***firstprivate(a)***

If **a** is a scalar, this is equivalent to:

#pragma omp target teams distribute parallel for

This generally means explicitly mapping **scalar** variables is unnecessary

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 teams distribute parallel for
  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

Exercise: Jacobi solver, parallel for and target

- 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 % 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];
    }
```

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 stuf 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 from the device onto the host.

Copy A on the host to A on the device.

The pointer swap

- The code swaps the xold and xnew pointers every iteration
- Doing the swap with xtmp confuses the compiler and it doesn’t actually result in a swap on the target device!

- So, map the original x1, x2 vectors, instead of xold and xnew

```
#pragma omp target data map(tofrom:x1[0:Ndim],x2[0:Ndim])
```

- And assign xnew and xold inside the target region appropriately

```
xnew = iters % 2 ? x2 : x1;
xold = iters % 2 ? x1 : x2;
```

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
```

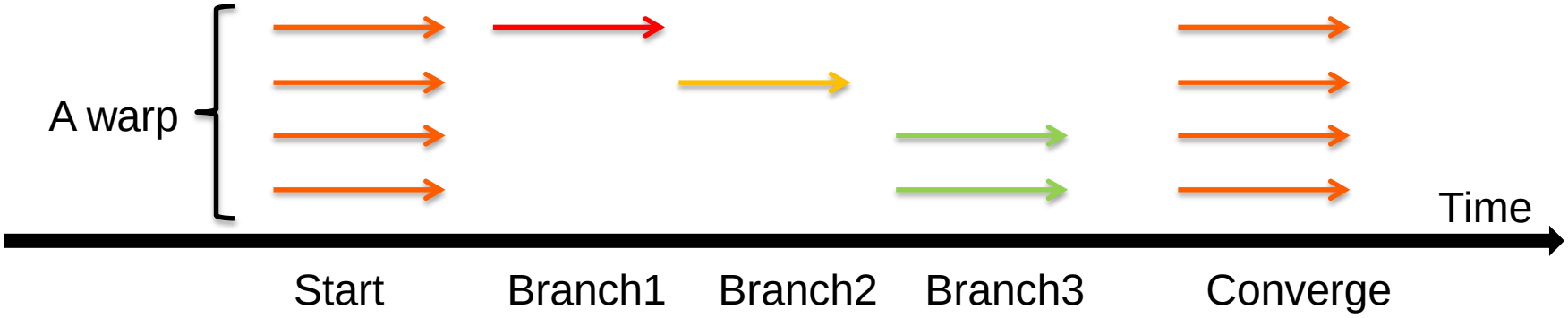
System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs

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 threads of a warp start together at the same program address
- Each thread has its own instruction address counter and register state
  - Each thread is free to branch and execute independently
  - Provide the MIMD abstraction
- Branch behavior
  - Each branch will be executed serially
  - Threads 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)*
- These are even worse: 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);
```

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.

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 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;
```

x	y	z	a	…	x	y	z	a	…	x	y	z	a	…	x	y	z	a	…

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

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

x	x	x	x	…	y	y	y	y	…	z	z	z	z	…	a	a	a	a	…

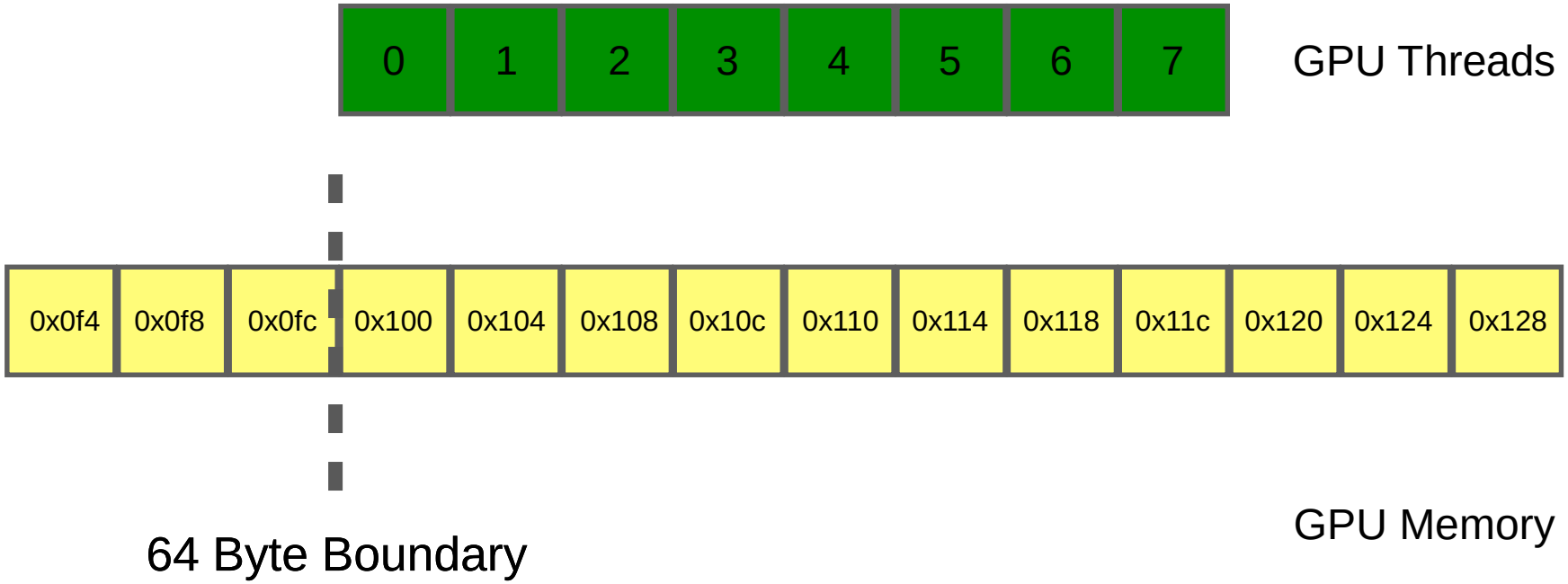
Adjacent work-items/vector-lanes like to access adjacent memory locations

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…

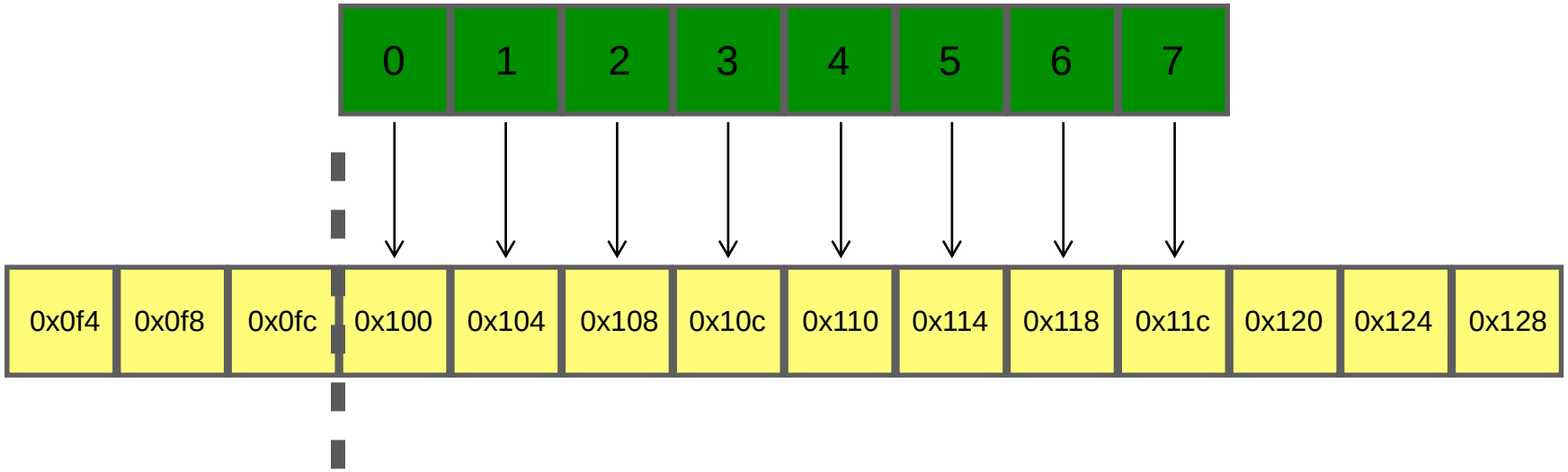
```
for (int id = 0; id < size; id++)
{
        // ideal
        float val1 = memA[id];


        // 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

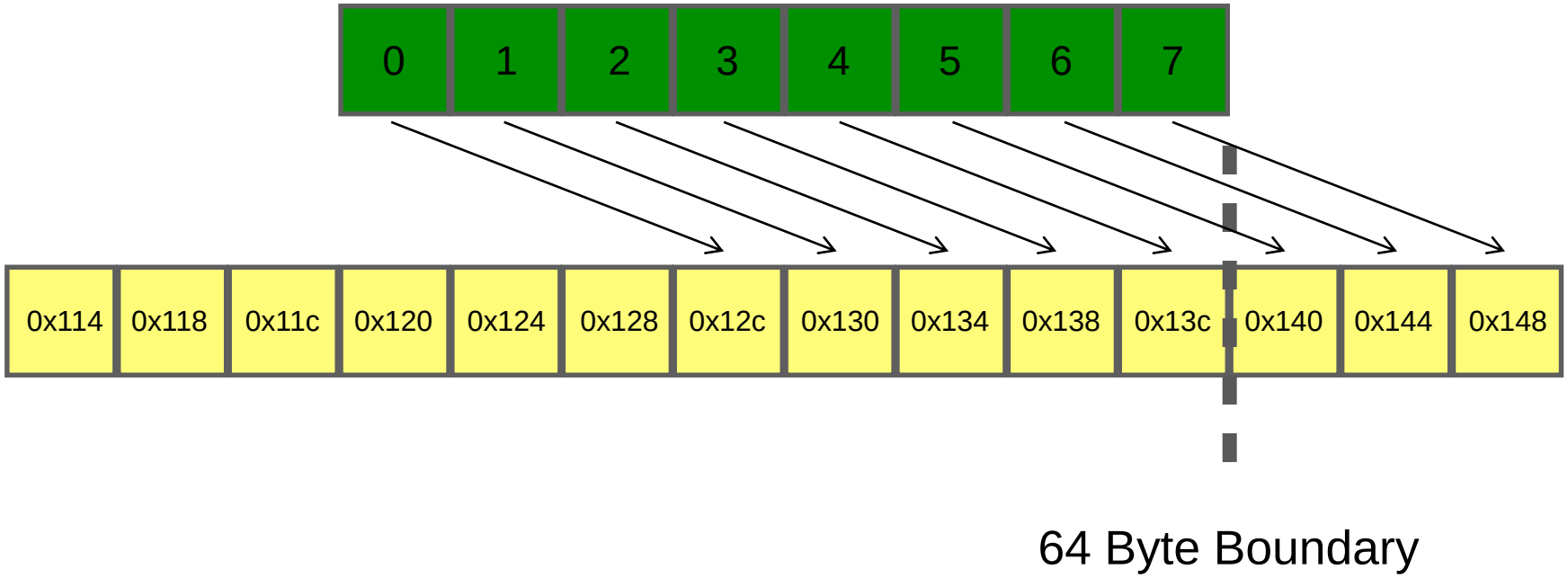
```
float val1 = memA[id];
```



64 Byte Boundary

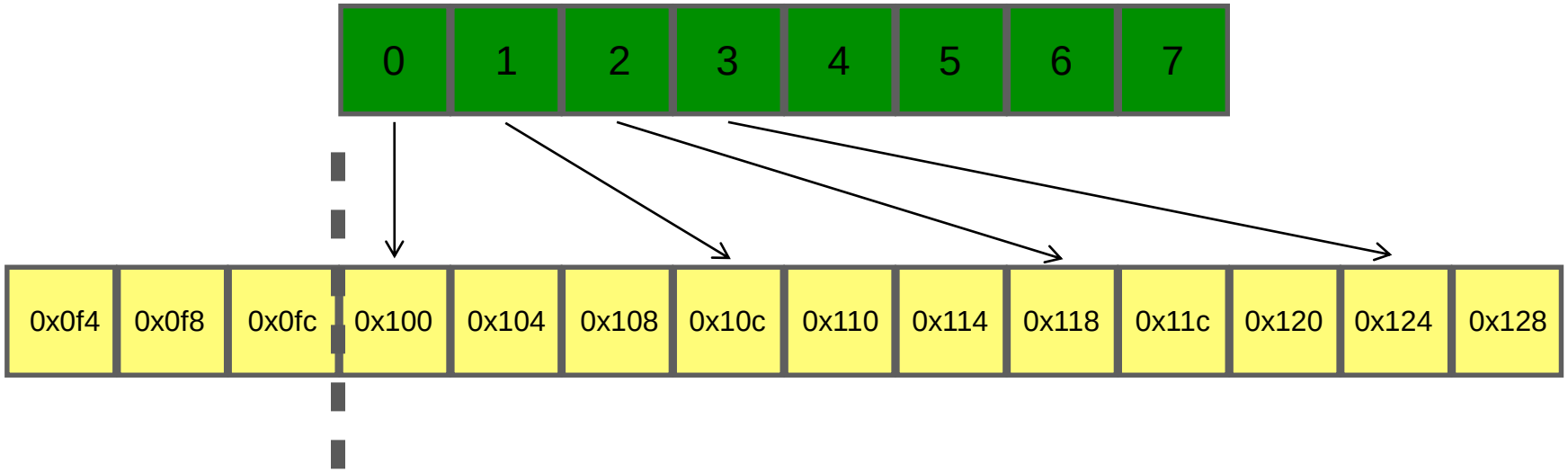
Memory access patterns

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



Memory access patterns

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

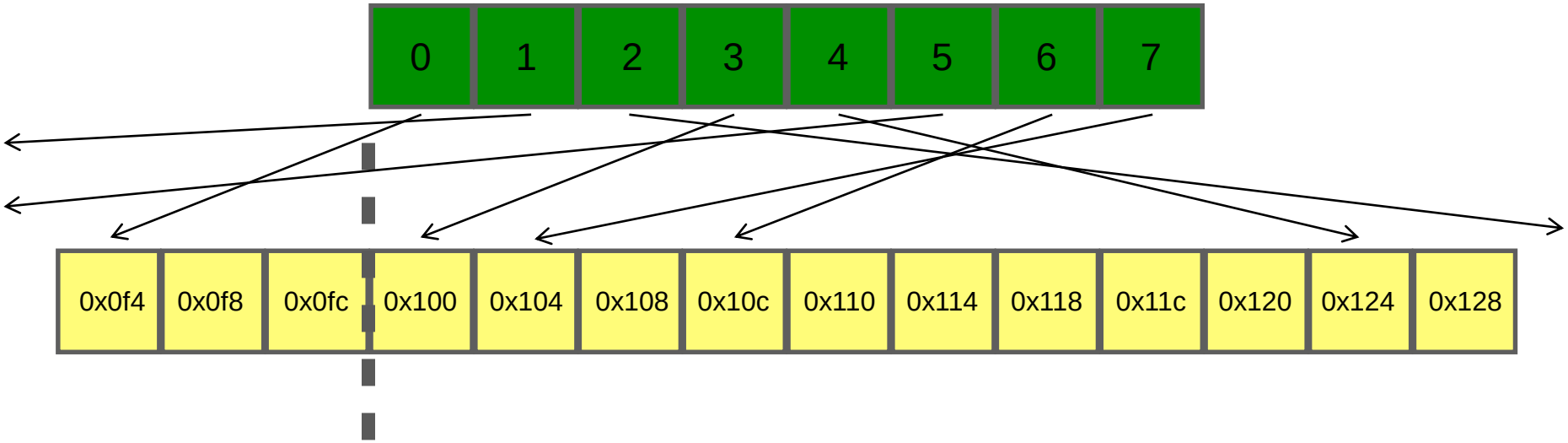


64 Byte Boundary

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

Jacobi solver code

```
for (i=0; i<Ndim; i++)
{
   size_t i = get_global_id(0);

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

Swap indices on A to match column major layout – was **A[i*Ndim+j]**

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.

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)

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.

- **Cray's** provided the first vendor support implementation targeting NVIDIA GPUs late 2015. Now supports OpenMP 4.0, and a *subset* of OpenMP 4.5.

- **IBM** has recently completed a compiler implementation using Clang, that fully supports OpenMP 4.5. This is being slowly introduced into the Clang main trunk.

- **GCC 6.1** introduced full support for OpenMP 4.5, and can target Intel Xeon Phi, or HSA enabled AMD GPUs.

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.

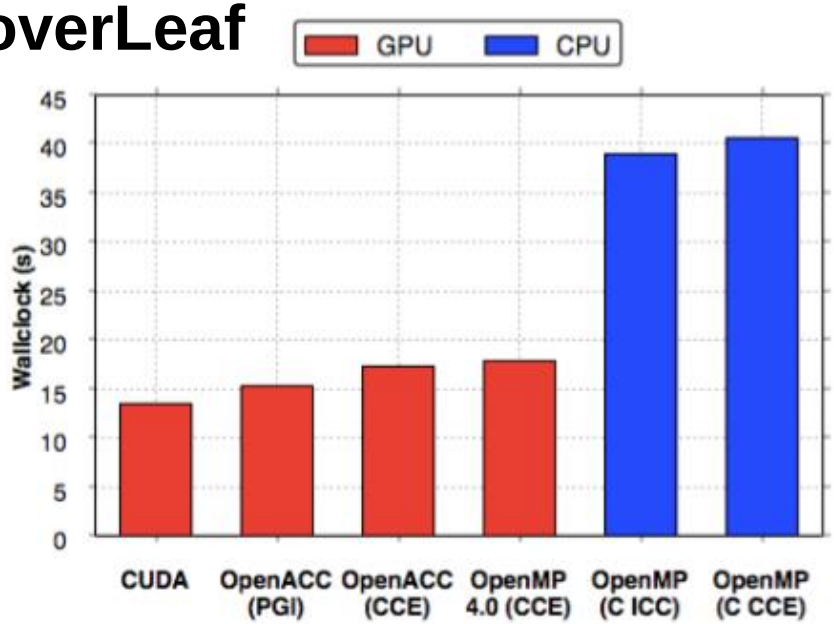
Performance?

* 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)

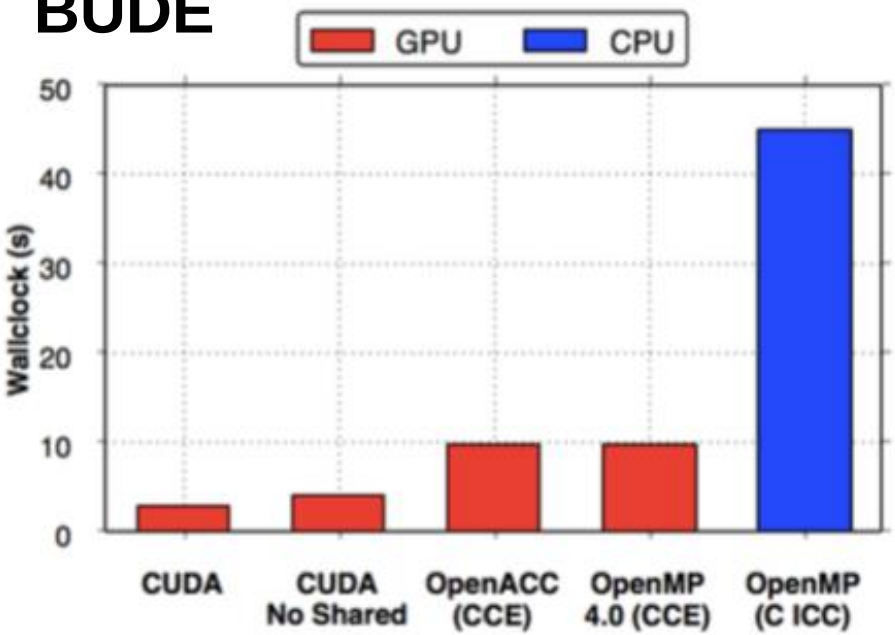
Immediately we see impressive performance compared to CUDA

Clearly the Cray compiler leverages the existing OpenACC backend

CloverLeaf



BUDE



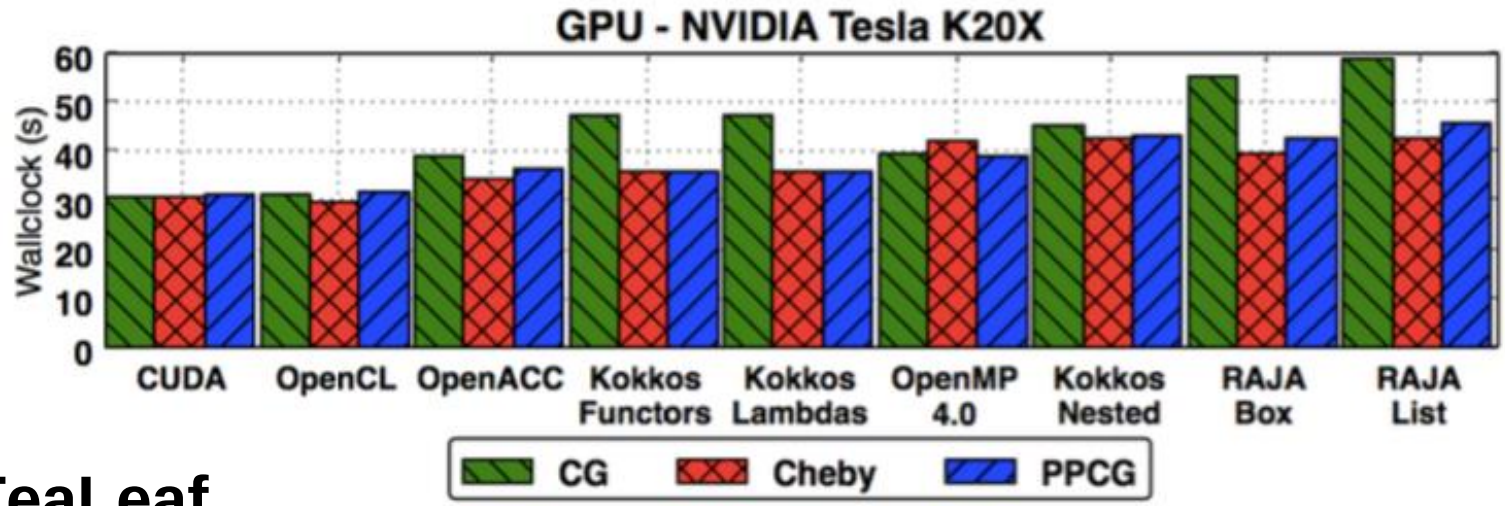
Even with OpenMP 4.5 there is still no way of targeting shared memory directly.

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

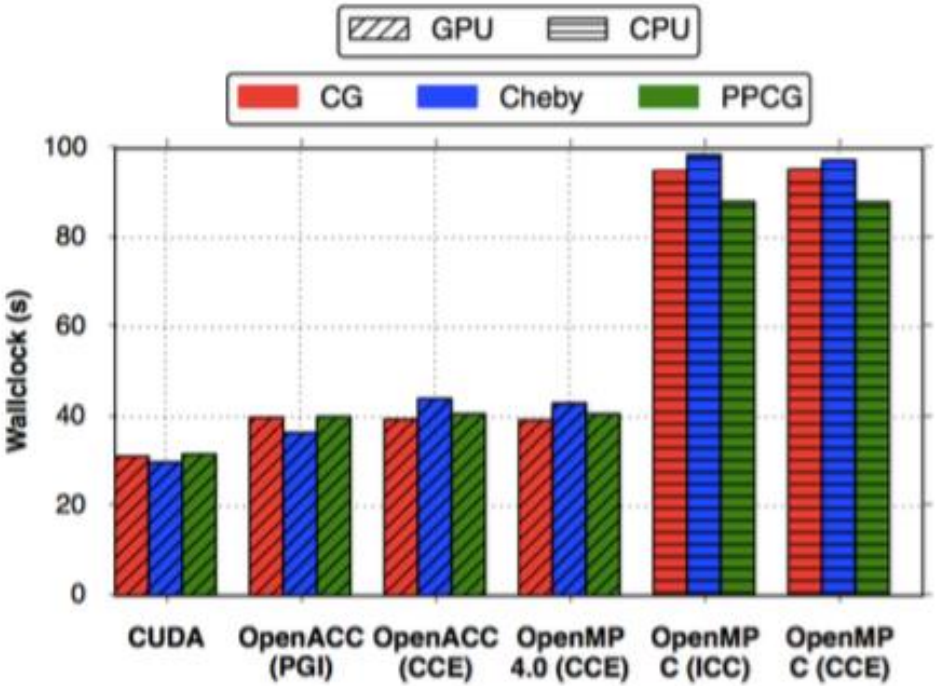
Martineau, M., McIntosh-Smith, S. Gaudin, W., *Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model, 2016, HIPS'16*

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*

Third party names are the property of their owners.

How do you get good performance?

- Our finding so far: *You <u>can</u> 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.

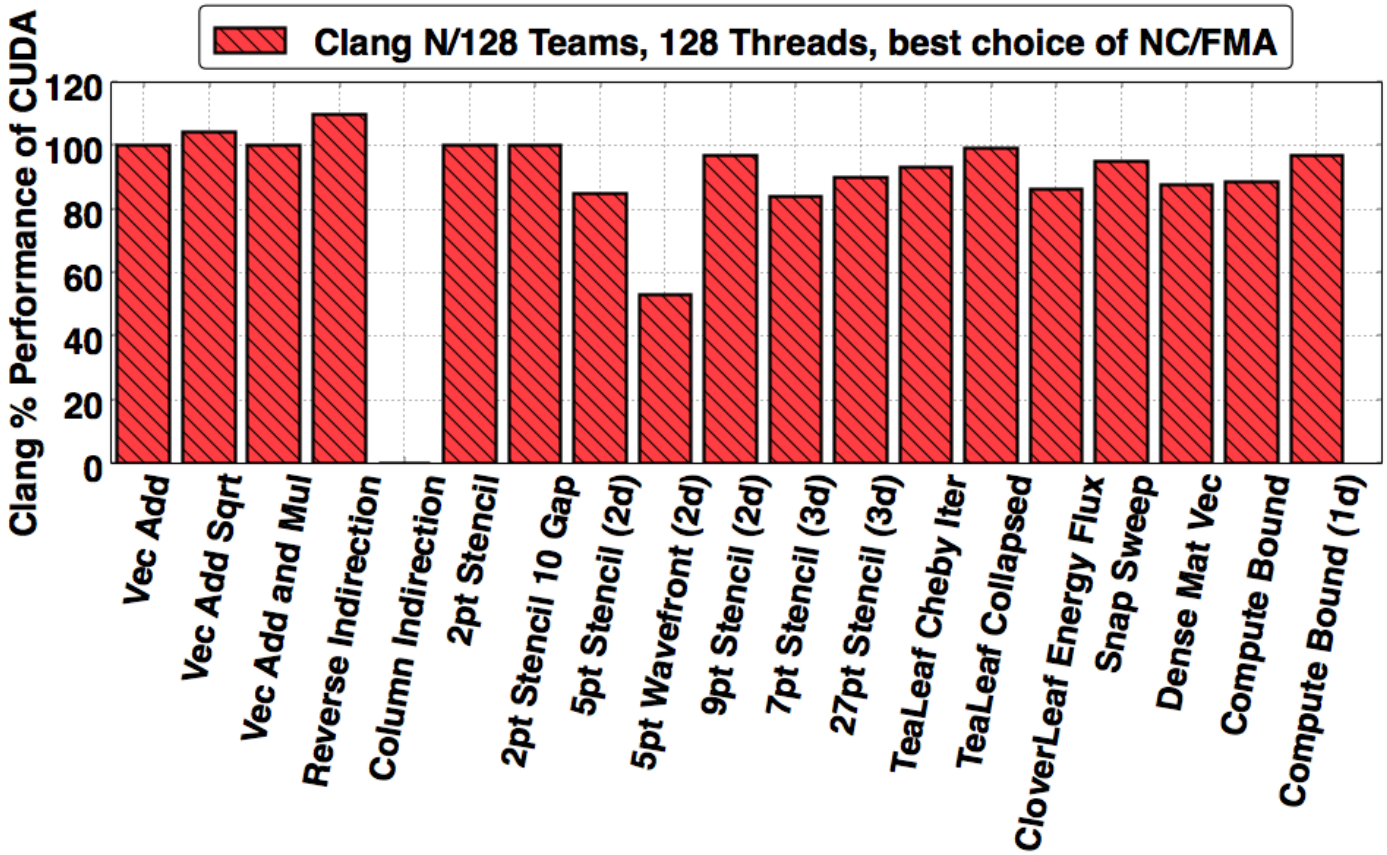
  - Using the **simd** directive to vectorize inner loops.

  - 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?

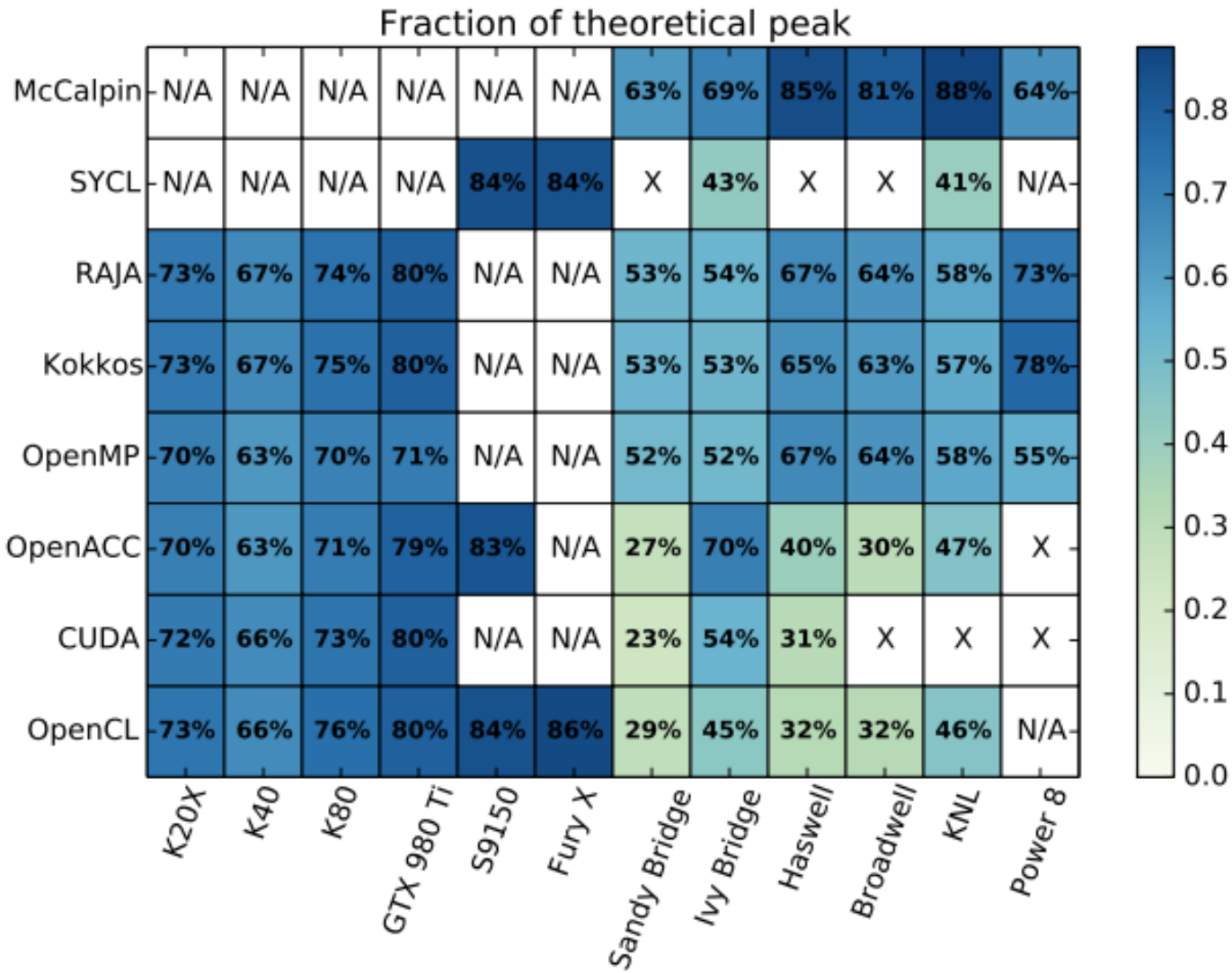
- Up until this point we had implicitly proven a good level of portability as we had successfully run OpenMP 4.x on many devices (Intel® CPU, Intel Xeon® Phi™ processors, NVIDIA® GPUs).

- 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

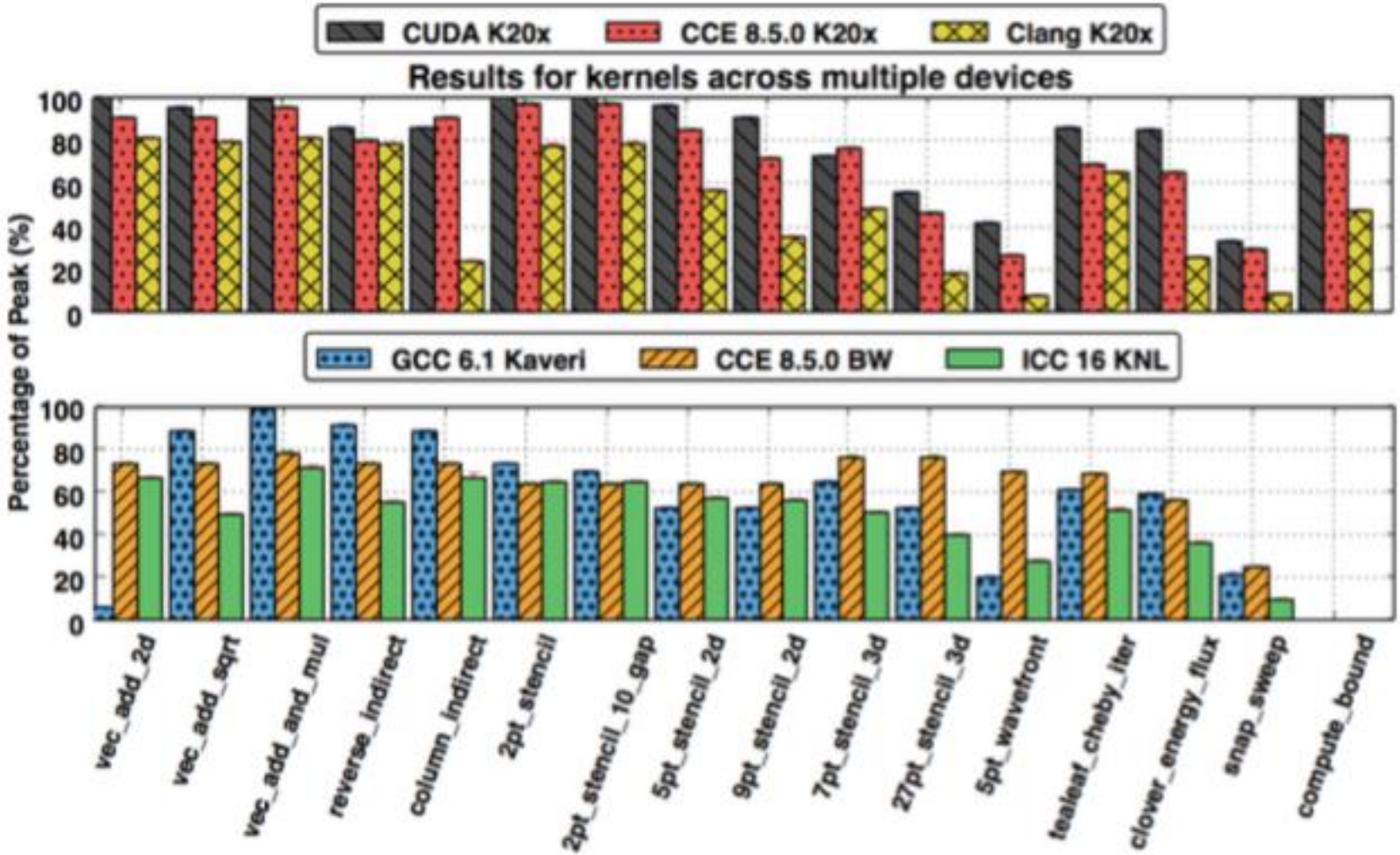
Deakin, T., Price, J., Martineau, M., McIntosh-Smith, S., *GPU-STREAM v2.0 Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models, ISC'16*

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 <u>not</u> 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.

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.

- On-line evaluation form … Please fill this out

http://bit.ly/sc16-eval