

Tutorial:

OpenMP Accelerator Model

(OpenMP for Heterogeneous Computing)

Eric Stotzer



Topics and Schedule

- 14:00 – 15:30
 - Background
 - Device and execution model
 - Control the execution on host and devices
- 15:30 – 16:00
 - Coffee Break
- 16:00 – 17:30
 - Accelerated workshare
 - Examples
 - Future Directions

What is OpenMP?



- De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran
- Consists of Compiler Directives, Runtime routines and Environment variables
- Specification maintained by the OpenMP Architecture Review Board (<http://www.openmp.org>)
- New ARB mission statement:
“The OpenMP ARB mission is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.”
- **Version 4.0 has been released July 2013**

OpenMP is widely supported by the industry, as well as the academic community

The screenshot shows the OpenMP.org website. The browser address bar displays 'openmp.org/wp/about-openmp/'. The page layout includes a search bar on the left with the text 'Search OpenMP.org' and a 'Google™ Custom Search' input field. Below the search bar is a 'Search' button. To the right of the search bar is a list of 'Archives' with dates from June 2014 down to November 2011. The main content area is titled 'Members' and lists 'Permanent Members of the ARB:' and 'Auxiliary Members of the ARB:'. The permanent members list includes AMD (Dibyendu Das), Convey Computer (Kirby Collins), Cray (James Beyer/Luiz DeRose), Fujitsu (Eiji Yamanaka), HP (Sujoy Saraswati), IBM (Kelvin Li), Intel (Xinmin Tian), NEC (Kazuhiro Kusano), NVIDIA (Jeff Larkin), Oracle Corporation (Nawal Copt), Red Hat (Matt Newsome), ST Microelectronics (Christian Bertin), and Texas Instruments (Andy Fritsch). The auxiliary members list includes ANL (Kalyan Kumaran), ASC/LLNL (Bronis R. de Supinski), BSC (Xavier Martorell), cOMPunity (Barbara Chapman), EPCC (Mark Bull), LANL (David Montoya), NASA (Henry Jin), ORNL (Oscar Hernandez), RWTH Aachen University (Dieter an Mey), SNL-Sandia National Lab (Steven Oliver), Texas Advanced Computing Center (Kent Milfeld), and University of Houston (Yonghong Yan/Barbara Chapman).

openmp.org/wp/about-openmp/

Search OpenMP.org
Google™ Custom Search
Search

Archives

- June 2014
- April 2014
- March 2014
- February 2014
- January 2014
- December 2013
- November 2013
- September 2013
- July 2013
- May 2013
- April 2013
- March 2013
- February 2013
- January 2013
- December 2012
- November 2012
- October 2012
- September 2012
- July 2012
- June 2012
- May 2012
- April 2012
- March 2012
- February 2012
- January 2012
- November 2011

Members

Permanent Members of the ARB:

- AMD (Dibyendu Das)
- Convey Computer (Kirby Collins)
- Cray (James Beyer/Luiz DeRose)
- Fujitsu (Eiji Yamanaka)
- HP (Sujoy Saraswati)
- IBM (Kelvin Li)
- Intel (Xinmin Tian)
- NEC (Kazuhiro Kusano)
- NVIDIA (Jeff Larkin)
- Oracle Corporation (Nawal Copt)
- Red Hat (Matt Newsome)
- ST Microelectronics (Christian Bertin)
- Texas Instruments (Andy Fritsch)

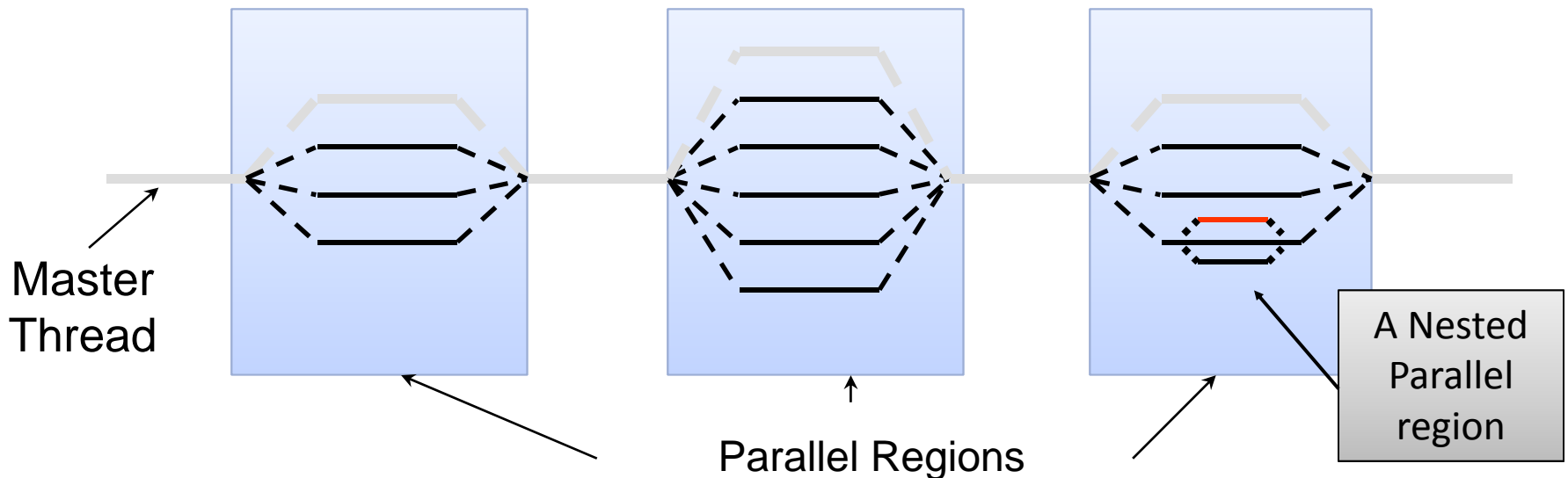
Auxiliary Members of the ARB:

- ANL (Kalyan Kumaran)
- ASC/LLNL (Bronis R. de Supinski)
- BSC (Xavier Martorell)
- cOMPunity (Barbara Chapman)
- EPCC (Mark Bull)
- LANL (David Montoya)
- NASA (Henry Jin)
- ORNL (Oscar Hernandez)
- RWTH Aachen University (Dieter an Mey)
- SNL-Sandia National Lab (Steven Oliver)
- Texas Advanced Computing Center (Kent Milfeld)
- University of Houston (Yonghong Yan/Barbara Chapman)

OpenMP Execution Model

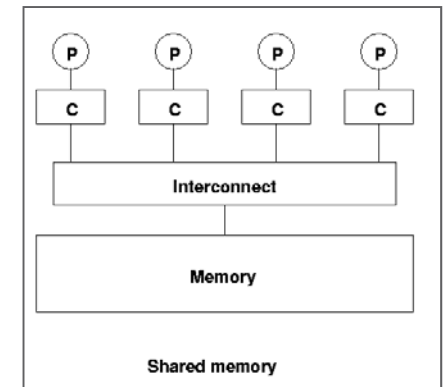
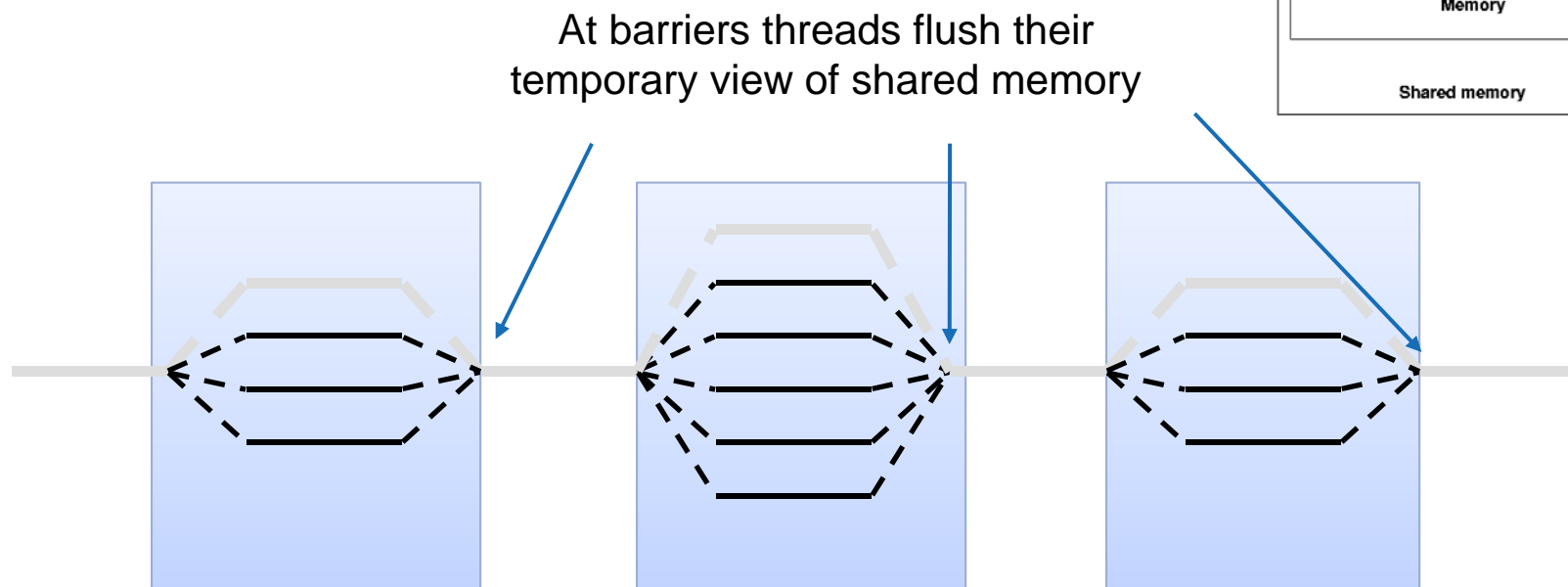


- **Master thread** spawns a **team of threads** as needed.
- Parallelism is added incrementally until desired performance is achieved: i.e. the sequential program evolves into a parallel program.

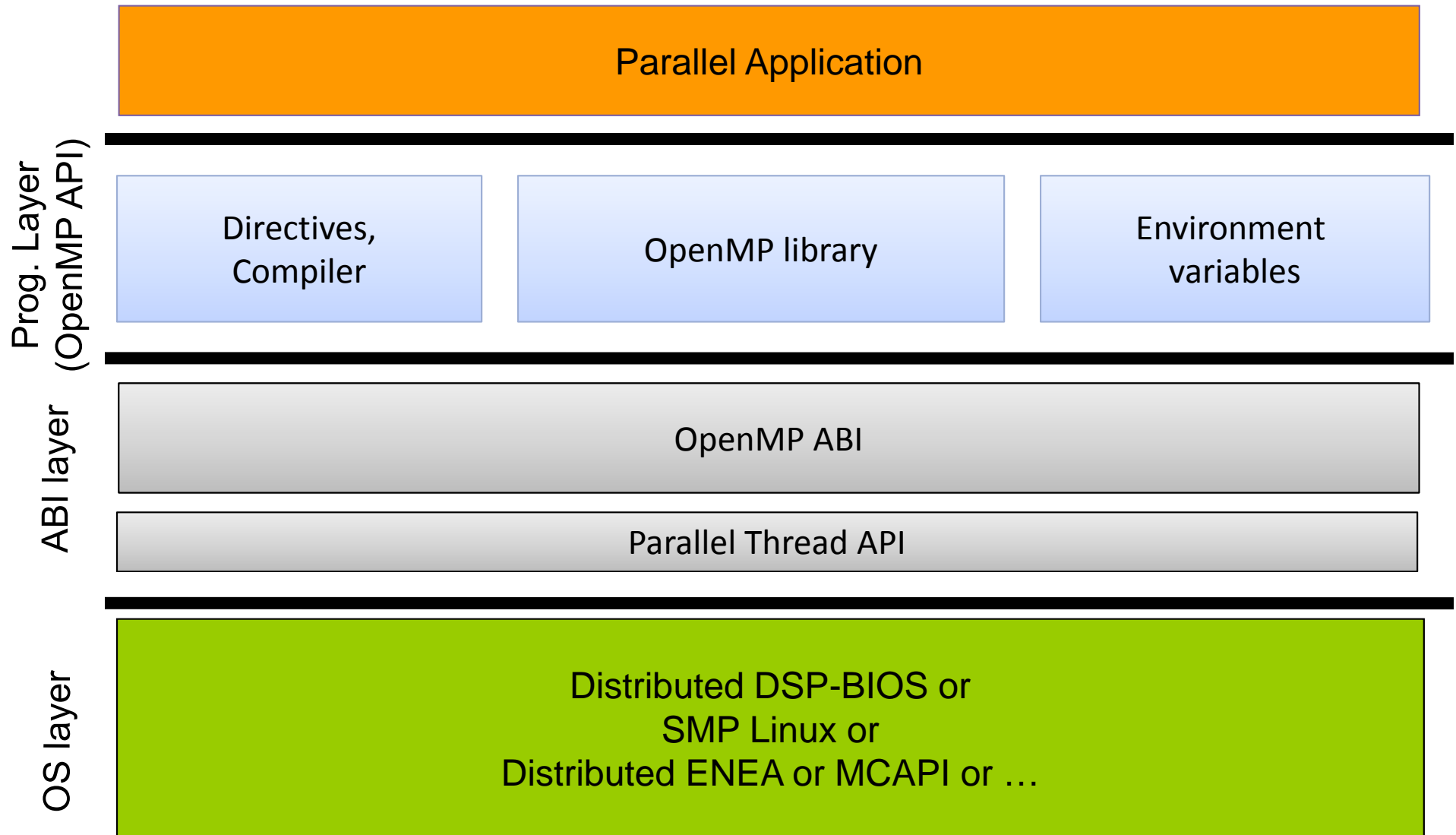


OpenMP Memory Model

- Threads have access to a *shared* memory
 - for **shared** data
 - each thread can have a temporary view of the shared memory (e.g. registers, cache, etc.) between synchronization barriers.
- Threads have *private* memory
 - for **private** data
 - Each thread has a stack for data local to each task it executes



OpenMP Parallel Computing Solution Stack



OpenMP Features



Provides the means to:

- create and destroy threads
- assign / distribute work (tasks) to threads
- specify which data is shared and which is private
- coordinate thread access to shared data

Syntax and Usage:

- Directives in OpenMP are compiler pragmas applied to a statement:
`#pragma omp construct [clause [clause]...] statement;`
- An Include file:
`#include <omp.h>`
- A runtime library:
`-l libomp.lib`

OpenMP: Parallel Regions

- You create threads in OpenMP with the “omp parallel” pragma.
- For example, To create a 4-thread Parallel region:

Each thread redundantly executes the code within the structured block

```
float A[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int id = omp_get_thread_num();  
    lots_of_work(id,A);  
}
```

- Each thread calls lots_of_work(id,A) for `id = 0` to `3`

OpenMP Data Attributes: Private Clause



- Private(var) creates a local copy of var for each thread.
 - The value is uninitialized
 - Private copy is *not* storage associated with the original

```
void wrong(){  
    int IS = 0;  
    #pragma parallel for private(IS)  
    for(int J=1;J<1000;J++)  
        IS = IS + J;  
    printf("%i", IS);  
}
```

OpenMP Parallel API



Compiler Directives

- Parallelization
 - parallel
- Worksharing
 - for ,sections, parallel for, task...
- Synchronization
 - barrier, critical, atomic, flush, ...
- Data-sharing attributes
 - shared, private, firstprivate, threadprivate, reduction, ...

Library Functions

- Thread Control
 - omp_get_thread_num(),
omp_get_num_threads(),
omp_set_num_threads(), ...
- Locks
 - omp_set_lock(),
omp_unset_lock(), ...

Environment Variables

- **OMP_NUM_THREADS,**
OMP_SCHEDULE, ...

Summary

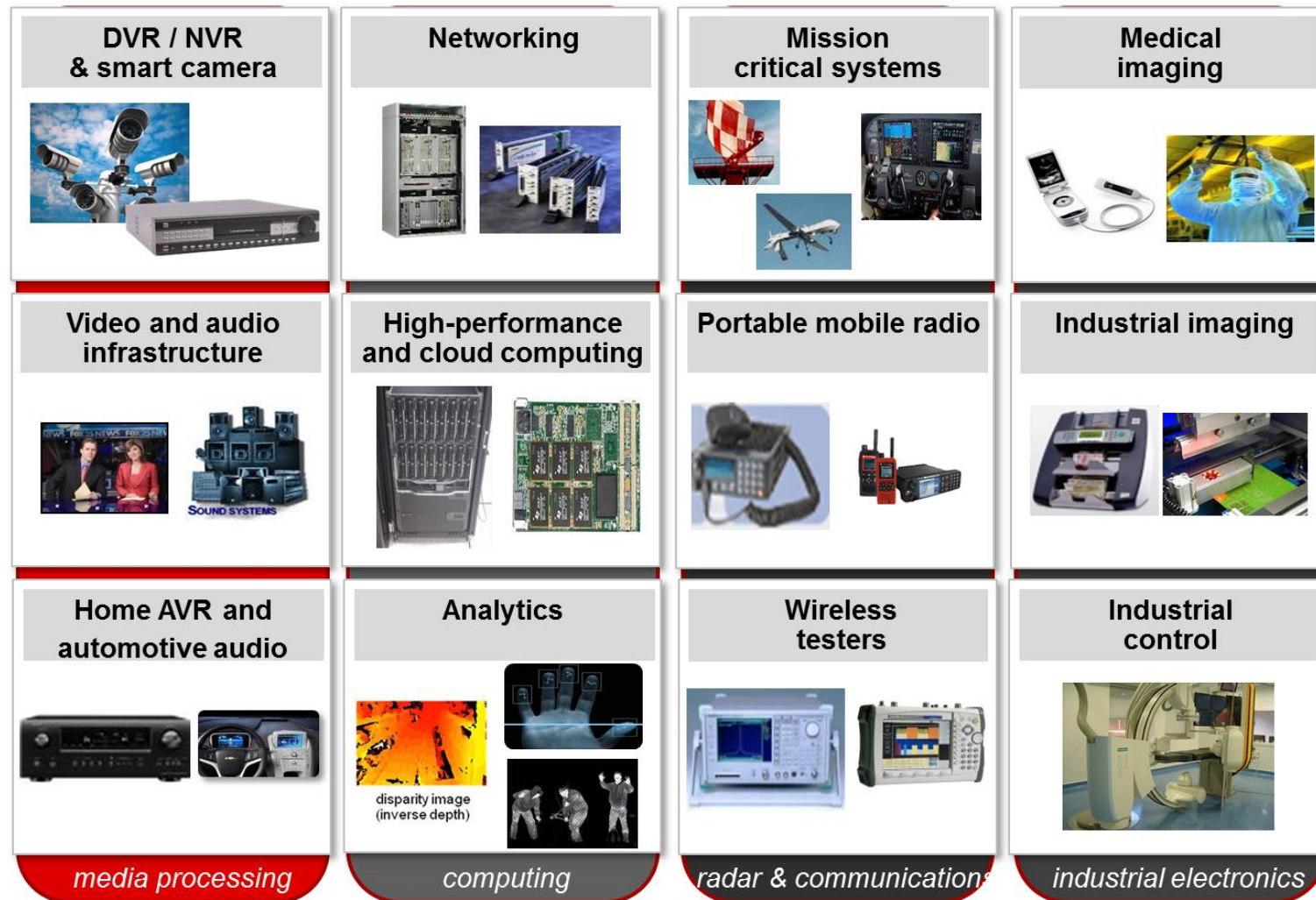


- Parallel programming model
 - Data parallelism (omp parallel for)
 - Task parallelism (omp task)
 - Productivity and flexibility
- Runtime requirements
 - Thread create/destroy on multiple cores
 - Barriers
 - locks (semaphores, atomics, mutex, ...)
 - Shared/private memory management
 - Memory consistency model

Why Texas Instruments?

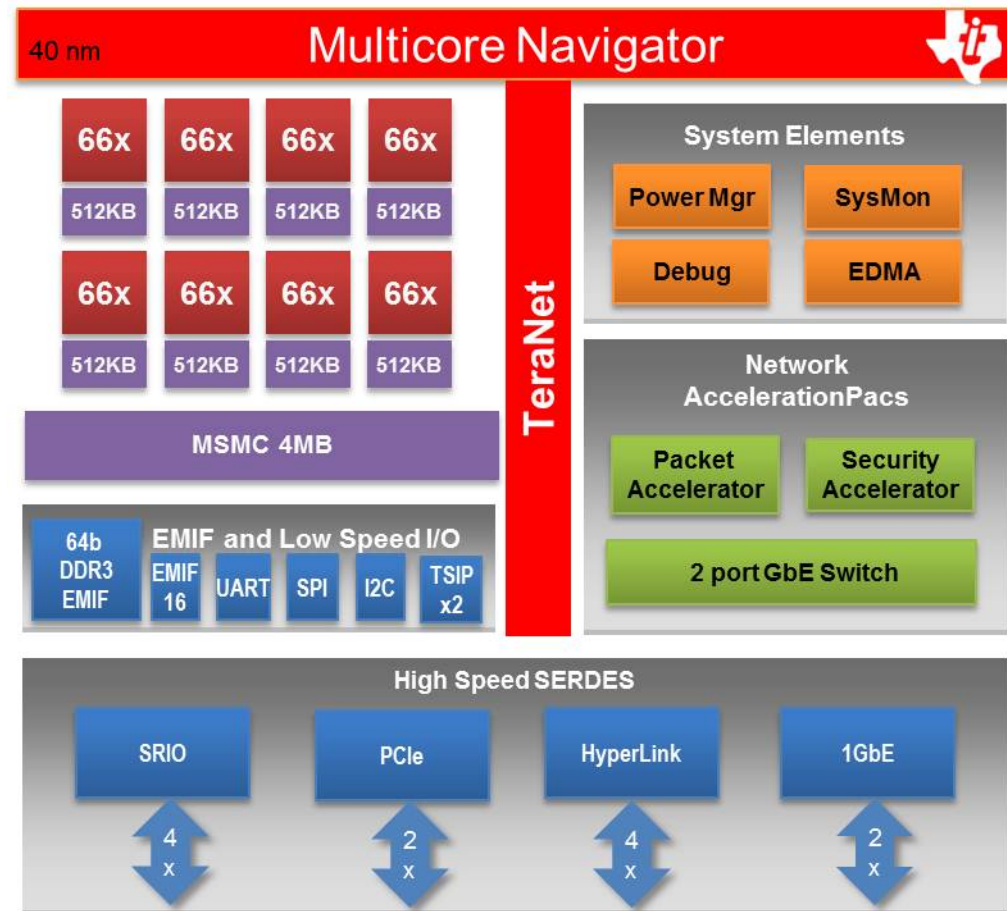
WHY ME?

High Performance Embedded Computing



Keystone I: C6678 SoC

- Eight 8 C66x cores
- Each with 32k L1P, 32k L1D, 512k L2
- 1 to 1.25 GHz
- 320 GMACS
- 160 SP GFLOPS
- 512 KB/Core of local L2
- 4MB Multicore Shared Memory (MSMC)
- Multicore Navigator (8k HW queues) and TeraNet
- Serial-RapidIO, PCIe-II, Ethernet, 1xHyperlink

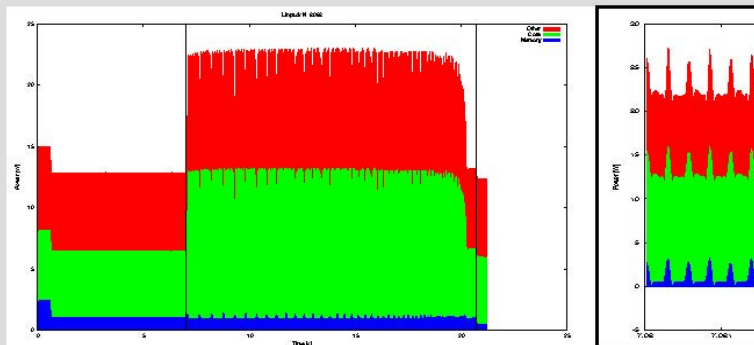


24mm x 24mm package

LINPACK running on C6678 achieves 25.6 Gflops, ~2.1 Gflops/W

PRACE First Implementation Project, Grant RI-261557, Final Report on Prototypes Evaluation. Lennart Johnsson, Gilbert Netzer, SNIC/KTH, 3/29/2013.

Linpack Power Profile



The plot shows the power consumption over time during a single execution of the Linpack benchmark code. Blue shows memory power, green is added power fed to the DSP and red other module consumers stacked atop. The vertical lines denote the timed section of the code. Distinct phases of execution can be seen, for instance the serial back-substitution at the end of the run. A zoom in also reveals the power peaks caused by DMA block copies to and from the main memory.

DSP Linpack Energy Efficiency

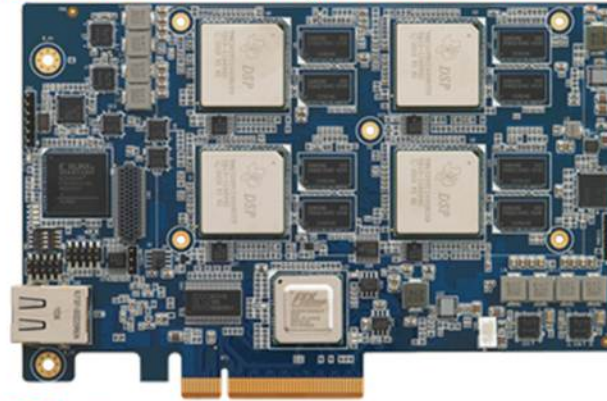
Size	Perf. GF/s	Eff. %	Core W	Mem. W	Other W	Total W	Core + Mem MF/J	Tot MF/J
127	1.3	4	5.95	1.26	6.87	14.08	176	90
255	2.8	9	4.78	0.99	5.17	10.95	493	260
511	6.0	19	6.40	1.12	6.58	14.09	796	425
1023	11.3	35	8.02	1.19	7.65	16.86	1230	672
2047	16.9	53	9.16	1.10	8.13	18.40	1649	920
4095	22.0	69	10.30	1.03	8.70	20.03	1939	1097
8063	25.6	80	11.20	0.99	9.20	21.39	2097	1195

The table shows the power and energy consumption of the major components of the C6678 DSP EVM. Core refers to the C6678 DSP SoC excluding I/O power. Mem is the DDR3 memory subsystem. The 5-9 watts of "Other" power is to a large part, except for about 1.5 W DC converter losses, consumed by debugging and unused hardware features that would not be present in an HPC server node. Therefore the "Core + Mem" power is a good estimate for the energy efficiency of an HPC server node. The values for small problem sizes show deviations due to various parts of the benchmark not being executed. The outmost loop step size is 128 columns, another breakpoint occurs at 1024.

High Density COTS boards



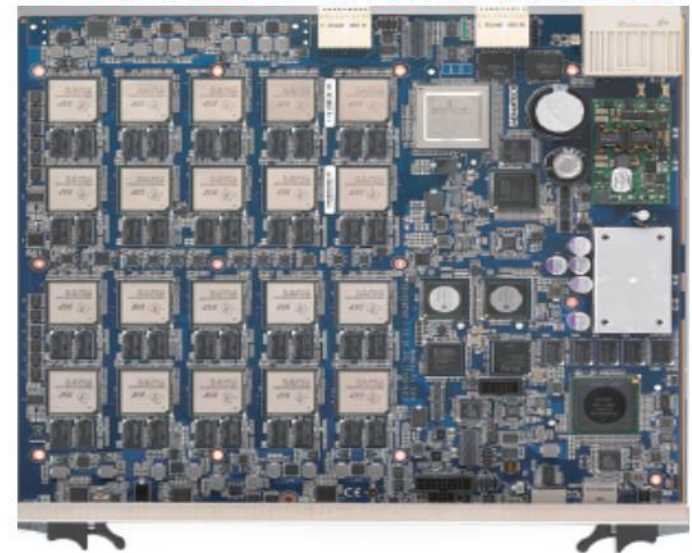
DSPC-8681 1/2 length PCIe card - 54Watts



DSPC-8682 PCIe Full-Length Card - 110Watts



DSPC-8682 ATCA blade 350Watts



Keystone II: 66AK2H12/06 SoC

C66x Fixed or Floating Point DSP

- 4x/8x 66x DSP cores up to 1.4GHz
- 2x/4x Cortex ARM A15
- 1MB of local L2 cache RAM per C66 DSP core
- 4MB shared across all ARM

Large on chip and off chip memory

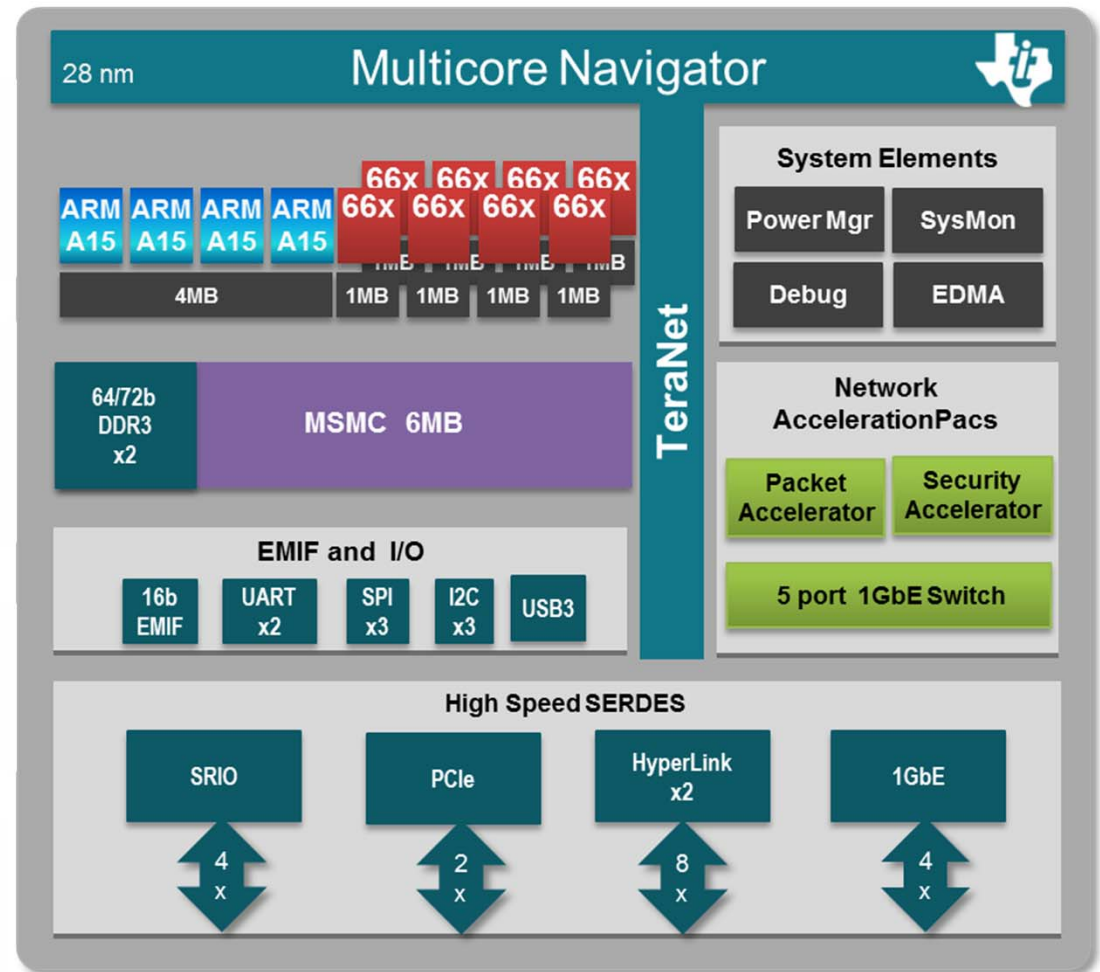
- Multicore Shared Memory Controller provides low latency & high bandwidth memory access
- 6MB Shared L2 on-chip
- 2 x 72 bit DDR3, 72-bit (with ECC), 10GB total addressable, DIMM support (4 ranks total)

KeyStone multicore architecture and acceleration

- Multicore Navigator, TeraNet, HyperLink
- 1GbE Network coprocessor (IPv4/IPv6)
- Crypto Engine (IPSec, SRTP)

Peripherals

- 4 Port 1G Layer 2 Ethernet Switch
- 2x PCIe, 1x4 SRIO 2.1, EMIF 16, USB 3.0 UARTx2, SPI, I²C
- 15-25W depending upon DSP cores, speed, temp & other factors

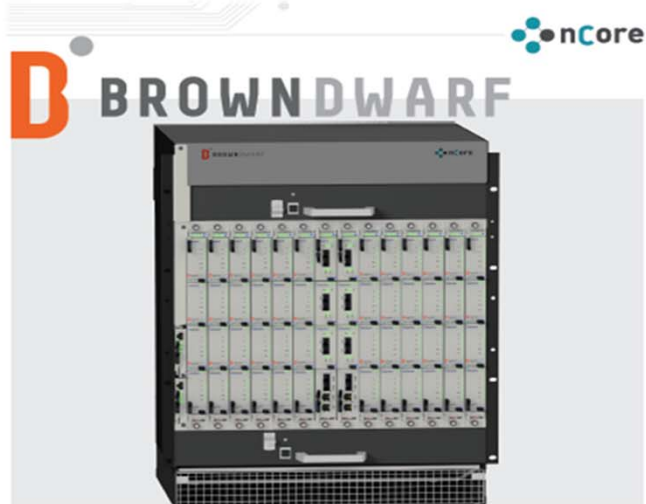


40mm x 40mm package

Available HPC Platforms



nCore BrownDwarf



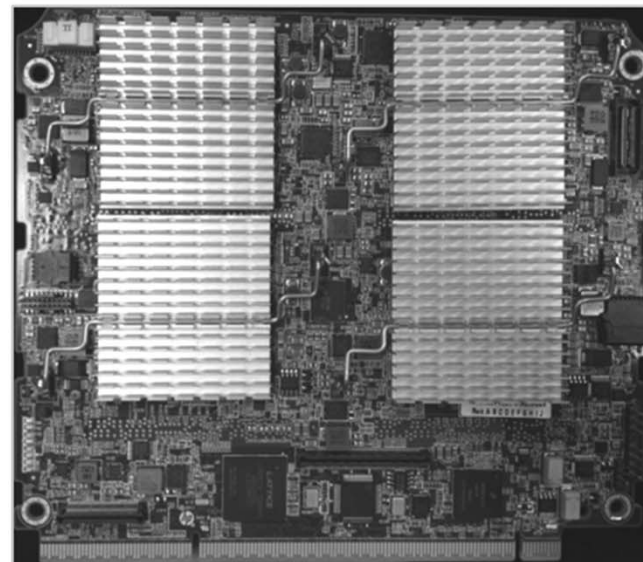
BrownDwarf Y-Class System
ARM-DSP Supercomputer

The nCore Y-Class supercomputer combines unprecedented low power computational performance with a

"The BrownDwarf Y-Class system is an incredibly important milestone in HPC system development. -Working in close collaboration with TI, IDT and our hardware partner Prodrive, we have successfully established a new class of energy efficient supercomputers designed to fulfill the demands of a wide range of scientific, technical and commercial applications. We are very excited to be launching the most capable energy efficient supercomputer available. The innovative design of the BrownDwarf Y-Class system has resulted in a network fabric that far exceeds the latency and power efficiencies of traditional supercomputing systems based on x86 and Infiniband or Ethernet systems. - By utilizing existing programming models and toolsets, the BrownDwarf Y-Class supercomputer is a disruptive force in HPC as it leapfrogs a number of the supercomputing incumbents."

--- Ian Lintault, Managing Director, nCore HPC

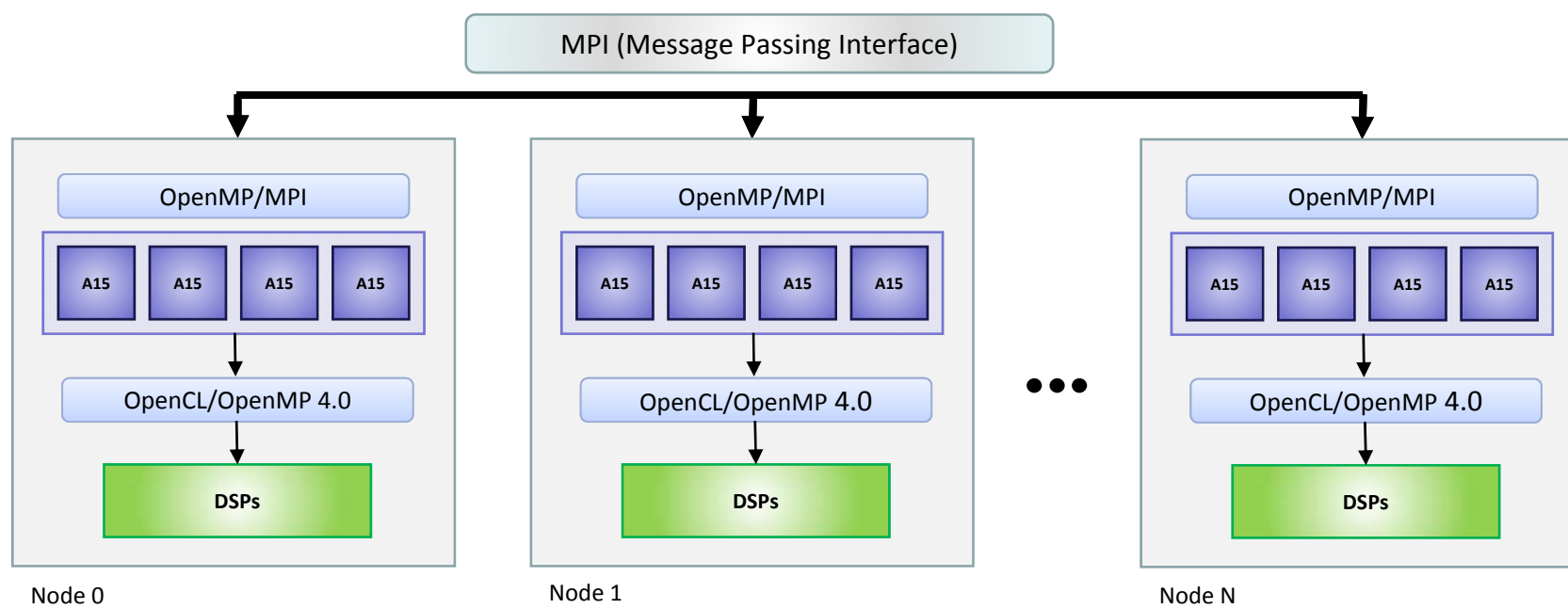
HP Moonshot



"As a partner in HP's Moonshot ecosystem dedicated to the rapid development of new Moonshot servers, we believe TI's KeyStone design will provide new capabilities across multiple disciplines to accelerate the pace of telecommunication innovations and geological exploration."

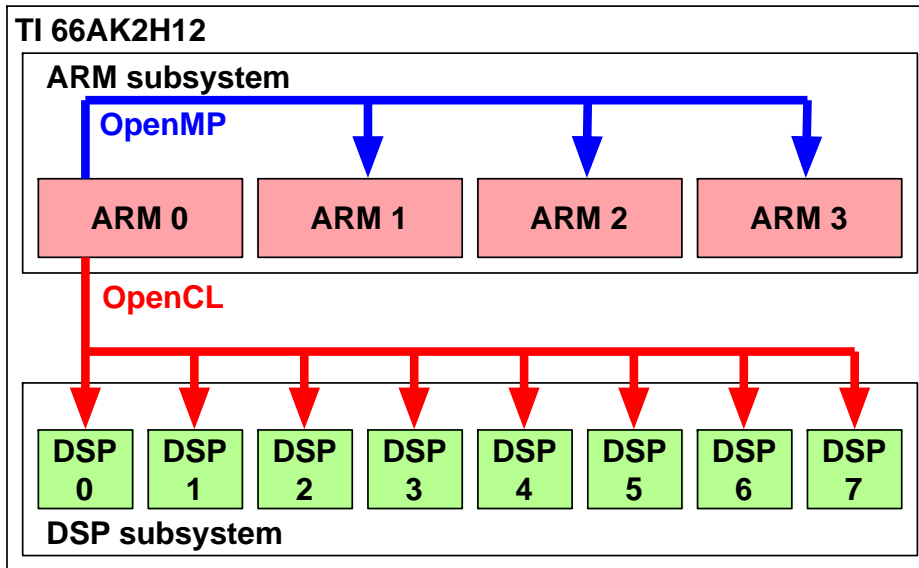
--- Paul Santeler, vice president and general manager, Hyperscale Business, HP

Heterogeneous Multicore Programming



- Within a node, OpenCL™ or OpenMP® 4.0 can be used to program heterogeneous compute cores
- Across nodes, MPI is used to partition the application and manage program execution, data transfer and synchronization

ARM + OpenCL DSP Acceleration

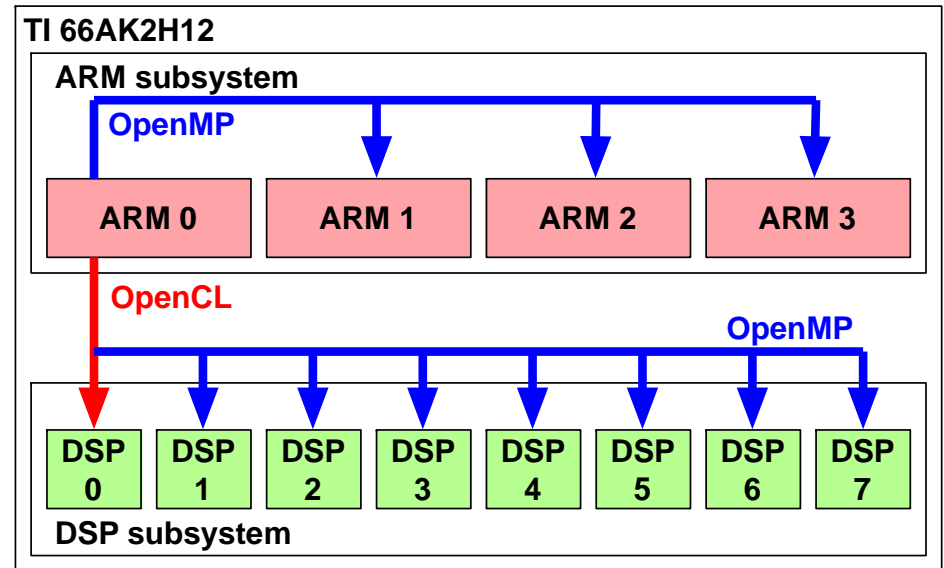


Data parallel

- A kernel is enqueued
- OpenCL divides into N workgroups
- Each workgroup is assigned a core
- After all workgroups finish a new kernel can be dispatched

Task parallel

- A task is enqueued
- OpenCL dispatches tasks to cores
- OpenCL can accept and dispatch more tasks asynchronously



OpenCL + OpenMP regions

- A task is enqueued
- OpenCL dispatches the task to DSP 0
- Tasks can use additional DSP cores by entering OpenMP regions
- A task completes before another task is dispatched
- Note: This is a TI extension

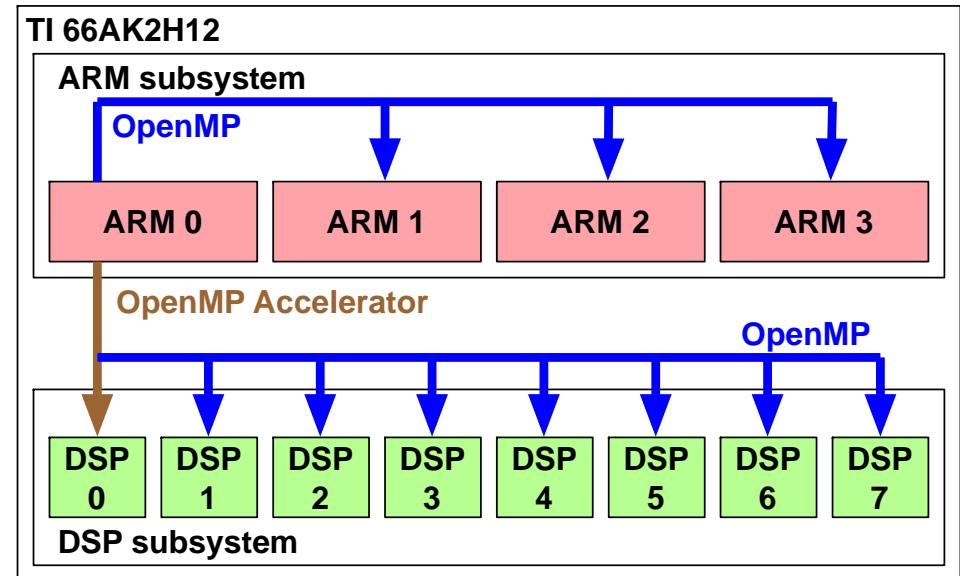
Example use

- Want to call existing OpenMP based DSP code from the ARM

ARM + OpenMP 4.0



```
// OpenMP Accelerator vector add
// OpenMP for loop parallelization
void ompVectorAdd(int    N,
                  float *a,
                  float *b,
                  float *c)
{
    #pragma omp target
    map(to:  N, a[0:N], b[0:N]) \
    map(from: c[0:N])
    {
        int i;
        #pragma omp parallel for
        for (i = 0; i < N; i++)
            c[i] = a[i] + b[i];
    }
}
```



Data movement

- **to** copies variables from the ARM memory to the DSP memory
- **from** copies variables from the DSP memory to the ARM memory
- TI provides special **alloc** and **free** functions to allocate DSP memory such that copies are not needed

Calling existing DSP code from the ARM

- Wrapping existing DSP functions with OpenMP Accelerator code is straightforward

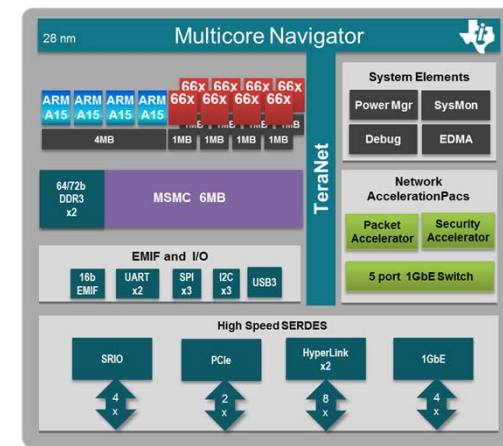
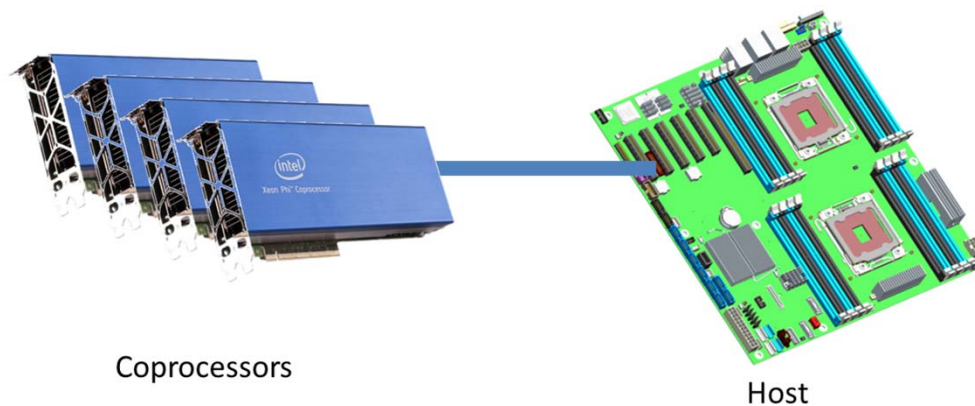
New in OpenMP 4.0



- Support for accelerators (or heterogeneous devices)
- Thread affinity support
- SIMD support for vectorization
- Thread cancellation
- Fortran 2003 support
- Extended support for
 - Tasking (groups, dependencies, abort)
 - Reductions (i.e. User Defined Reductions)
 - Atomics (sequential consistency)

Device Model

- OpenMP 4.0 supports accelerators/coprocessors
- Device model:
 - One host
 - Multiple accelerators/coprocessors of the same kind



Heterogeneous SoC

Terminology

- **Device:**
an implementation-defined (logical) execution unit
- **Device data environment:**
The initial *data environment* associated with a device.

The execution model is host-centric such that the host device offloads **target** regions to target devices.

OpenMP 4.0 Device Constructs

- Execute code on a target device
 - **omp target** [*clause*[[*,*] *clause*],...] *structured-block*
 - **omp declare target** [*function-definitions-or-declarations*]
- Map variables to a target device
 - **map** ([*map-type*:] *list*) // *map clause*
map-type := **alloc** | **tofrom** | **to** | **from**
 - **omp target data** [*clause*[[*,*] *clause*],...] *structured-block*
 - **omp target update** [*clause*[[*,*] *clause*],...]
 - **omp declare target** [*variable-definitions-or-declarations*]
- Workshare for acceleration
 - **omp teams** [*clause*[[*,*] *clause*],...] *structured-block*
 - **omp distribute** [*clause*[[*,*] *clause*],...] *for-loops*

Device Runtime Support

- Runtime support routines
 - `void omp_set_default_device(int dev_num)`
 - `int omp_get_default_device(void)`
 - `int omp_get_num_devices(void);`
 - `int omp_get_num_teams(void)`
 - `int omp_get_team_num(void);`
 - `Int omp_is_initial_device(void);`
- Environment variable
 - Control default device through `OMP_DEFAULT_DEVICE`
 - Accepts a non-negative integer value

Offloading Computation

■ Use target construct to

→ Transfer control from the host to the device

→ Map variables to/from the device data environment

■ Host thread waits until offloaded region completed

→ Use other OpenMP constructs for asynchronicity

```
#pragma omp target map(to:b[0:count]) map(to:c,d) map(from:a[0:count])
{
#pragma omp parallel for
    for (i=0; i<count; i++) {
        a[i] = b[i] * c + d;
    }
}
```

host
target
host

target Construct

- Transfer control from the host to the device

- Syntax (C/C++)

```
#pragma omp target [clause[[,] clause],...]  
structured-block
```

- Syntax (Fortran)

```
!$omp target [clause[[,] clause],...]  
structured-block  
!$omp end target
```

- Clauses

```
device(scalar-integer-expression)  
map(alloc | to | from | tofrom: list)  
if(scalar-expr)
```

map Clause

- Map a variable or an array section to a device data environment

- Syntax:

```
map(alloc | to | from | tofrom: list)
```

- Map-types

- `alloc`: allocate storage for corresponding variable
- `to`: alloc and assign value of original variable to corresponding variable on entry
- `from`: alloc and assign value of corresponding variable to original variable on exit
- `tofrom`: default, both to and from

map Clause



```
extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);

    #pragma omp target map(to:v1[0:N],v2[:N]) \\\
                        map(from:p[0:N])
    #pragma omp parallel for
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];

    output(p, N);
}
```

- The target construct updates the *device data environment* and explicitly **maps** the array sections `v1[0:N]`, `v2[:N]` and `p[0:N]` to the new device data environment.
- The variable `N` is implicitly mapped into the new device data environment from the encountering task's data environment.

Terminology

- **Mapped variable:**

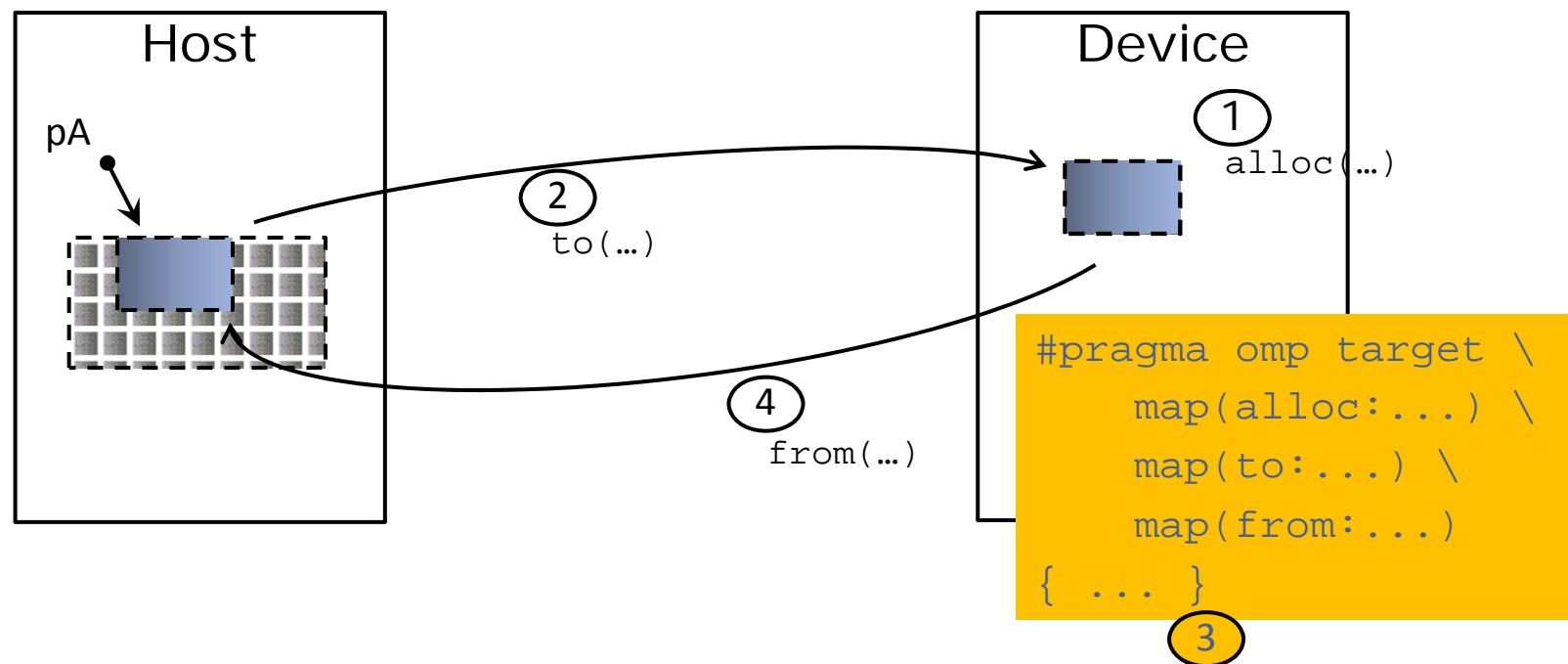
An *original variable* in a (host) data environment with a *corresponding variable* in a device data environment

- **Mappable type:**

A type that is amenable for mapped variables.
(Bitwise copyable plus additional restrictions.)

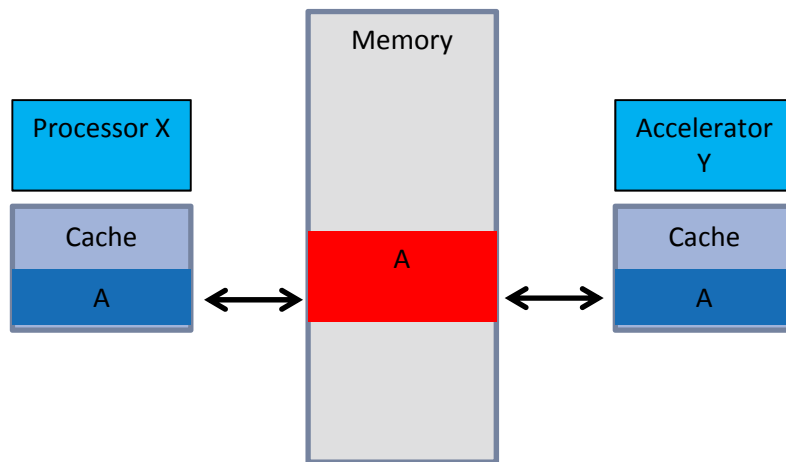
Device Data Environment

- The **Map** clauses determine how an *original variable* in a data environment is mapped to a *corresponding variable* in a device data environment.



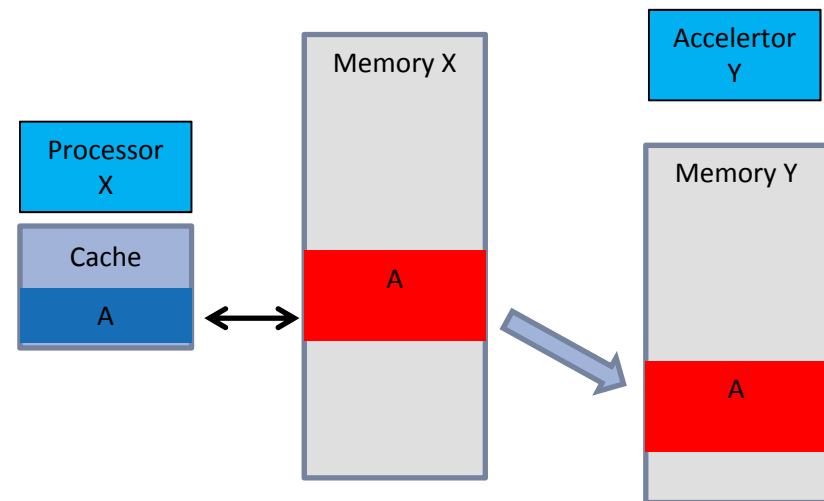
MAP is not necessarily a copy

Shared memory



- The corresponding variable in the device data environment *may* share storage with the original variable.
- Writes to the corresponding variable may alter the value of the original variable.

Distributed memory



Map variables across multiple target regions



- Optimize data transfers
- The `target data` construct creates a scoped device data environment
 - The `map` clauses control direction of data flow
 - The variables remain in the device data environment during the target data region
- Use `target update` to request data transfers from within a target data region

Optimize data transfers

- Avoid frequent transfers or overlap computation/comm.
- Pre-allocate temporary fields

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
{
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
        tmp[i] = some_computation(input[i], i);

    do_some_other_stuff_on_host();

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
        res += final_computation(tmp[i], i)
}
```

host target host target host

target data Construct

- Map variables to a device data environment for the extent of the region.

- Syntax (C/C++)

```
#pragma omp target data [clause[[,] clause],...]  
structured-block
```

- Syntax (Fortran)

```
!$omp target data [clause[[,] clause],...]  
structured-block  
!$omp end target data
```

- Clauses

```
device(scalar-integer-expression)  
map(alloc | to | from | tofrom: list)  
if(scalar-expr)
```

target data Construct Example



```
extern void init(float*, float*, int);
extern void init_again(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;

    init(v1, v2, N);

    #pragma omp target data map(from: p[0:N])
    {
        #pragma omp target map(to: v1[:N], v2[:N])
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = v1[i] * v2[i];

        init_again(v1, v2, N);

        #pragma omp target map(to: v1[:N], v2[:N])
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = p[i] + (v1[i] * v2[i]);
    }

    output(p, N);
}
```

- The target data construct maps variables to the *device data environment* and encloses target regions, which have their own device data environments.
- The target data construct is used to create variables that will persist throughout the target data region.
- v1 and v2 are mapped at each target construct.
- Instead of mapping the variable p twice, once at each target construct, p is mapped once by the target data construct.

Synchronize mapped variables



- Synchronize the value of an original variable in a host data environment with a corresponding variable in a device data environment

```
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
{
#pragma omp target device(0)
#pragma omp parallel for
    for (i=0; i<N; i++)
        tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);

#pragma omp target update device(0) to(input[:N])

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
    for (i=0; i<N; i++)
        res += final_computation(input[i], tmp[i], i)
}
```

host target host target host

target update Construct



- Issue data transfers between host and devices

- Syntax (C/C++)

```
#pragma omp target update [clause[[,] clause],...]
```

- Syntax (Fortran)

```
!$omp target update [clause[[,] clause],...]
```

- Clauses

```
device(scalar-integer-expression)
```

```
to(list)
```

```
from(list)
```

```
if(scalar-expr)
```


Map a variable for the whole program

```
define N 1000
#pragma omp declare target
float p[N], v1[N], v2[N];
#pragma omp end declare target

extern void init(float *, float *, int);
extern void output(float *, int);

void vec_mult()
{
    int i;
    init(v1, v2, N);
    #pragma omp target update to(v1, v2)
    #pragma omp target
    #pragma omp parallel for
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];

    #pragma omp target update from(p)
    output(p, N);
}
```

- Indicate that global variables are mapped to a device data environment for the whole program
- Use `target update` to maintain consistency between host and device

target declare Construct



- Map variables to a device data environment for the whole program

- Syntax (C/C++):

```
#pragma omp declare target  
    [variable-definitions-or-declarations]  
#pragma omp end declare target
```

- Syntax (Fortran):

```
!$omp declare target (list)
```

Prepare Functions for a Device



- The tagged functions will be compiled for
 - Host execution (as usual)
 - Target execution (to be invoked from offloaded code)

```
#pragma omp declare target
float some_computation(float fl, int in) {
    // ... code ...
}

float final_computation(float fl, int in) {
    // ... code ...
}

#pragma omp end declare target
```

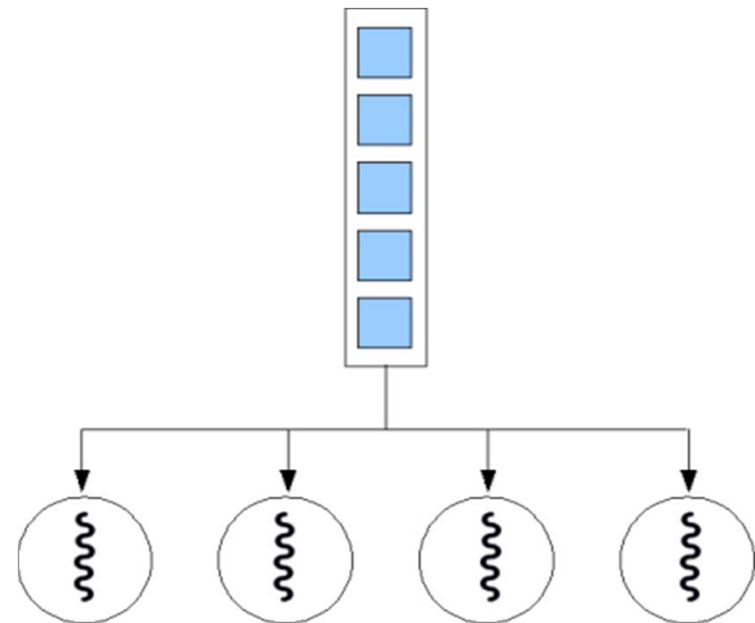
Review: OpenMP Task Construct



- Task Model
 - Support irregular data dependent parallelism
 - Tasks are assigned to a queue
 - Threads execute tasks that they remove from a task queue

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

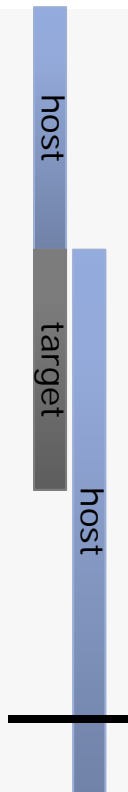
            p = next(p);
        }
    }
}
```



Asynchronous Offloading

- Use existing OpenMP features to implement asynchronous offloads.

```
#pragma omp parallel sections
{
  #pragma omp task
  {
    #pragma omp target map(to:input[:N]) map(from:result[:N])
    #pragma omp parallel for
    for (i=0; i<N; i++) {
      result[i] = some_computation(input[i], i);
    }
  }
  #pragma omp task
  {
    do_something_important_on_host();
  }
  #pragma omp taskwait
}
```



Review: Work-Sharing Constructs

- The “for” Work-Sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
#pragma omp for
for (I=0;I<N;I++){
    NEAT_STUFF(I);
}
```

By default, there is a barrier at the end of the “omp for”. Use the “nowait” clause to turn off the barrier.

Work Sharing Constructs

Sequential code

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

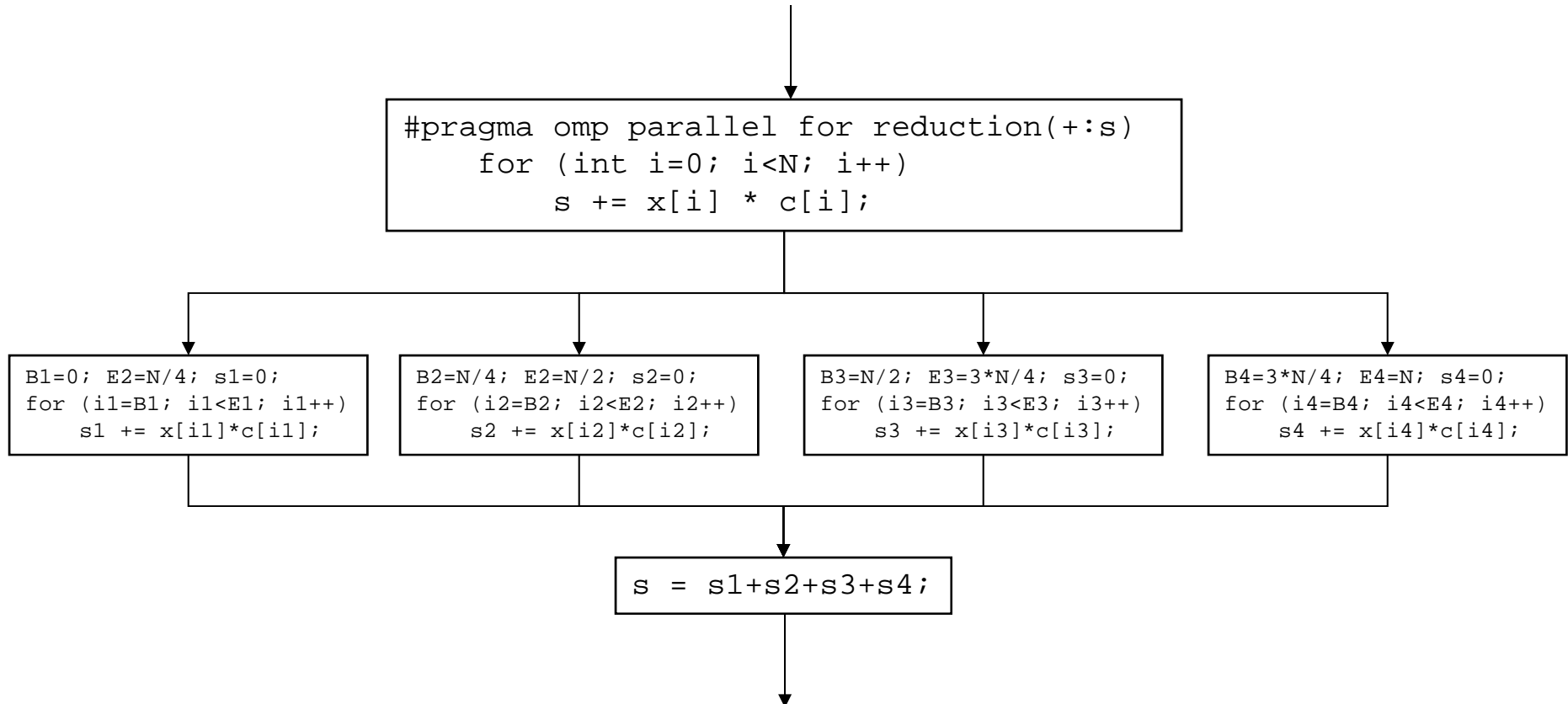
OpenMP Parallel
Region

```
#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;
    for(i=istart;I<iend;i++) {a[i]=a[i]+b[i];}
}
```

OpenMP Parallel
Region and a work-
sharing for
construct

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

Work-sharing construct



- A single copy of `x[]` and `c[]` is shared by all the threads

Terminology

- **League:**
the set of threads teams created by a teams construct
- **Contention group:**
threads of a team in a league and their descendant threads

teams Construct

■ Syntax (C/C++):

```
#pragma omp teams [clause[[,] clause],...]  
structured-block
```

■ Syntax (Fortran):

```
!$omp teams [clause[[,] clause],...]  
structured-block
```

■ Clauses

```
num_teams(integer-expression)  
thread_limit(integer-expression)  
default(shared | none)  
private(list), firstprivate(list)  
shared(list), reduction(operator : list)
```

teams Construct – Restrictions



■ Creates a league of thread teams

- The master thread of each team executes the `teams` region
- Number of teams is specified with `num_teams ()`
- Each team executes with `thread_limit ()` threads

teams Construct – Restrictions



- A `teams` constructs must be “perfectly” nested in a `target` construct:
 - No statements or directives outside the `teams` construct
 - Teams cannot synchronize
- Only special OpenMP constructs can be nested inside a `teams` construct:
 - `distribute` (see next slides)
 - `parallel`
 - `parallel for` (C/C++), `parallel do` (Fortran)
 - `parallel sections`

SAXPY: host

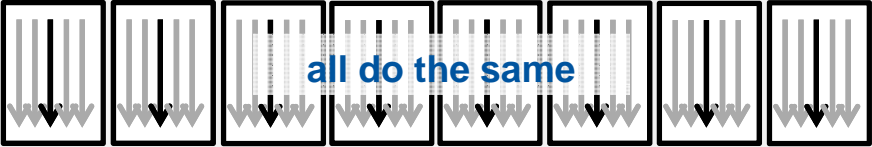
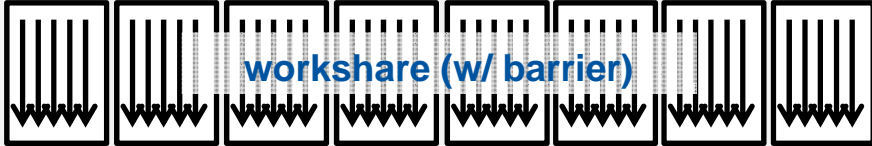
```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
    #pragma omp parallel for
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
}
```

SAXPY: on device

```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
    #pragma omp target map(to:x[0:n], n, a) map(y[0:n])
    {
        #pragma omp parallel for
        for (int i = 0; i < n; ++i){
            y[i] = a*x[i] + y[i];
        }
    }
}
```

SAXPY: Accelerated

```

void saxpy(float * restrict y, float * restrict x, float a, int n)
{
    int num_blocks = Fb(n);
    int nthreads = Ft(n);
    #pragma omp target data map(to:x[0:n], n, a) map(y[0:n])
    #pragma omp teams num_teams(num_blocks) thread_limit(nthreads)
    {
        
        #pragma omp parallel for
        for (int i = 0; i < n; i += num_blocks){
            
            for (int j = i; j < i + num_blocks; j++) {
                y[j] = a*x[j] + y[j];
            }
        }
    }
}

```

all do the same

workshare (w/ barrier)

distribute Construct

■ Syntax (C/C++):

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

■ Syntax (Fortran):

```
!$omp teams [clause[[,] clause],...]  
do-loops
```

■ Clauses

```
private(list)
```

```
firstprivate(list)
```

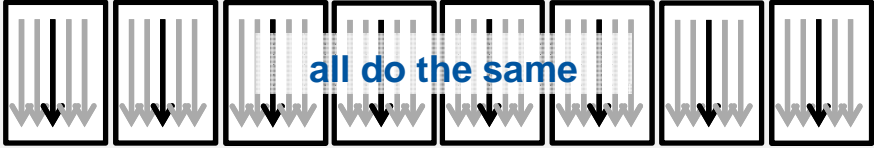

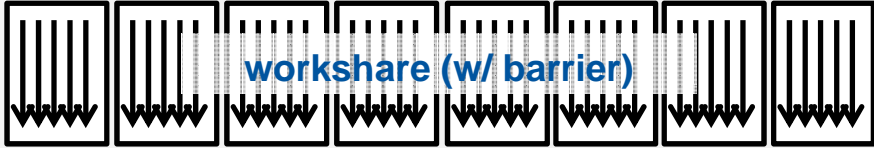
```
collapse(n)
```

```
dist_schedule(kind[, chunk_size])
```


distribute Construct

- New kind of worksharing construct
 - Distribute the iterations of the associated loops across the master threads of a `teams` construct
 - No implicit barrier at the end of the construct
- `dist_schedule(kind[, chunk_size])`
 - If specified scheduling kind must be static
 - Chunks are distributed in round-robin fashion of chunks with size `chunk_size`
 - If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

SAXPY: Accelerated

```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
    int num_blocks = Fb(n);
    int nthreads = Ft(n);
    #pragma omp target data map(to:x[0:n], n, a) map(y[0:n])
    #pragma omp teams num_teams(num_blocks) thread_limit(nthreads)
    {
        
        #pragma omp distribute
        for (int i = 0; i < n; i += num_blocks){
            
            #pragma omp parallel for
            for (int j = i; j < i + num_blocks; j++) {
                
                y[j] = a*x[j] + y[j];
            }
        }
    }
}
```

Combined Constructs

- The distribution patterns can be cumbersome
- OpenMP 4.0 defines combined constructs for typical code patterns
 - `distribute simd`
 - `distribute parallel for` (C/C++)
 - `distribute parallel for simd` (C/C++)
 - `distribute parallel do` (Fortran)
 - `distribute parallel do simd` (Fortran)
 - ... plus additional combinations for `teams` and `target`
- Avoids the need to do manual loop blocking

SAXPY: Combined Constructs



```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
    int num_blocks = Fb(n);
    int nthreads = Ft(n);

    #pragma omp target data map(to:x[0:n], n, a) map(y[0:n])
    #pragma omp teams num_teams(num_blocks) thread_limit(bsize)
    #pragma omp distribute parallel for
        for (int i = 0; i < n; ++i)
            y[i] = a*x[i] + y[i];
}
```

SAXPY: Combined Constructs



```
void saxpy(float * restrict y, float * restrict x, float a, int n)
{
    int num_blocks = Fb(n);
    int nthreads = Ft(n);

    #pragma omp target map(to:x[0:n], n, a) map(y[0:n])
    #pragma omp teams distribute parallel for \
        num_teams(num_blocks) thread_limit(bsize)
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

Comparing OpenMP with OpenACC



► OpenMP 4.0 – Accelerated workshare

```
#pragma omp target map(B[0:N])
#pragma omp teams distribute parallel for \
    num_teams(numblocks)
for (i=0; i<N; ++i) {
    B[i] += sin(B[i]);
}
```

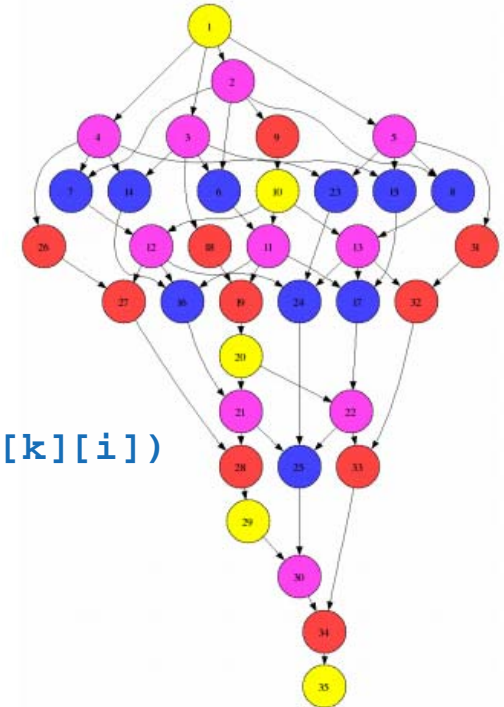
► OpenACC – accelerated workshare

```
#pragma acc parallel copy(B[0:N]) num_gangs(numblocks)\
    vector_length(bsize)
#pragma acc loop gang vector
    for (i=0; i<N; ++i) {
        B[i] += sin(B[i]);
    }
```

Task Dependencies



```
void blocked_cholesky( int NB, float A[NB][NB] ) {
    int i, j, k;
    for (k=0; k<NB; k++) {
        #pragma omp task depend(inout:A[k][k])
        spotrf (A[k][k]) ;
        for (i=k+1; i<NT; i++)
            #pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
            strsm (A[k][k], A[k][i]);
        // update trailing submatrix
        for (i=k+1; i<NT; i++) {
            for (j=k+1; j<i; j++)
                #pragma omp task depend(in:A[k][i],A[k][j])
                depend(inout:A[j][i])
                sgemm( A[k][i], A[k][j], A[j][i]);
            #pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
            ssyrk (A[k][i], A[i][i]);
        }
    }
}
```



* image from BSC

OpenMP and OpenACC async



```
!$acc parallel loop async(stream1)
<Kernel A>
```

```
!$acc parallel loop async(stream1)
<Kernel B>
```

```
!$acc parallel loop async(stream2)
<Kernel C>
```

```
!$acc parallel loop async(stream3) &
!$acc      wait(stream1,stream2)
<Kernel D>
```

```
!$acc parallel loop async(stream4) &
!$acc      wait(stream3)
<Kernel E>
```

```
!$acc parallel loop async(stream5) &
!$acc      wait(stream3)
<Kernel F>
```

```
!$acc parallel loop async(stream5)
<Kernel G>
```

```
!$acc wait ! ensures all completed
```

```
!$omp task target depend(stream1)
<Kernel A>
```

```
!$omp task target depend(stream1)
<Kernel B>
```

```
!$omp task target depend(stream2)
<Kernel C>
```

```
!$omp task target depend(stream3) &
!$omp      depend(in:stream1,stream2)
<Kernel D>
```

```
!$omp task target depend(stream4) &
!$omp      depend(in:stream3)
<Kernel E>
```

```
!$omp task target depend(stream5) &
!$omp      depend(in:stream3)
<Kernel F>
```

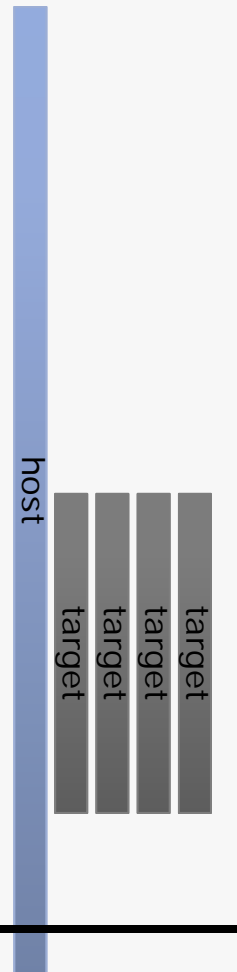
```
!$omp task target depend(stream5)
<Kernel G>
```

```
!$omp task depnd(in:stream4,stream5)&
<host Kernel> ! ensures all completed
```


Multi-device Example



```
int num_dev = omp_get_num_devices();
int chunksz = length / num_dev;
assert((length % num_dev) == 0);
#pragma omp parallel sections firstprivate(chunksz,num_dev)
{
    for (int dev = 0; dev < NUM_DEVICES; dev++) {
#pragma omp task firstprivate(dev)
        {
            int lb = dev * chunksz;
            int ub = (dev+1) * chunksz;
#pragma omp target device(dev) map(in:y[lb:chunksz]) map(out:x[lb:chunksz])
                {
#pragma omp parallel for
                    for (int i = lb; i < ub; i++) {
                        x[i] = a * y[i];
                    }
                }
        }
    }
}
```



if Clause Example

```
#define THRESHOLD1 1000000
#define THRESHOLD2 1000

extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);

    #pragma omp target if(N>THRESHOLD1) \
        map(to: v1[0:N], v2[:N]) map(from: p[0:N])
    #pragma omp parallel for if(N>THRESHOLD2)
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}
```

- The `if` clause on the `target` construct indicates that if the variable `N` is smaller than a given threshold, then the `target` region will be executed by the host device.
- The `if` clause on the `parallel` construct indicates that if the variable `N` is smaller than a second threshold then the `parallel` region is inactive.

Array Sections Example

- If the type of the variable appearing in an array section is pointer, then the variable is implicitly treated as if it had appeared in a **map** clause with a *map-type* of **alloc**
- If any part of the original storage of a list item has corresponding storage in the enclosing device data environment, all of the original storage must have corresponding storage in the enclosing device data environment.

```
void foo (int *A, int N)
{
    int *p;

    #pragma omp target data map( A[:N])
    {
        // implicit map(tofrom: A) for the pointer
        // A = storage allocated for array section on device
        p = &A[0];
        #pragma omp target map( p[0:N/2] )
        {
            A[N-1] = 0;
            p[0] = 0;
        }
    }
}
```

67

Tasks and target Example



```
#pragma omp declare target
#include <stdlib.h>
#include <omp.h>
extern void init(float *, float *, int);
extern void output(float *, int);
#pragma omp end declare target

void vec_mult(float *p, float *v1, float *v2, int
N, int dev)
{
    int i;
    init(p, N);

    #pragma omp task depend(out: v1, v2)
    #pragma omp target device(dev) map(v1, v2)
    {
        // check whether on device dev
        if (omp_is_initial_device())
            abort();
        v1 = malloc(N*sizeof(float));
        v2 = malloc(N*sizeof(float));
        init(v1,v2);
    }

    foo(); // execute asynchronously

    #pragma omp task depend(in: v1, v2)
    #pragma omp target device(dev) \
        map(to: v1, v2) \
        map(from: p[0:N])
    {
        // check whether on device dev
        if (omp_is_initial_device())
            abort();

        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = v1[i] * v2[i];

        output(p, N);
        free(v1);
        free(v2);
    }

    #pragma taskwait
} // end vec_mult()
```

OpenMP 4.0 Capabilities



Feature	OpenACC	OpenMP 4.0
Support for C and C++, Fortran	✓	✓
Support single code base of hetero-machine	✓	✓
Overlap communication and computation	✓	✓
Interoperate with MPI	✓	✓
Interoperate with OpenMP		✓
Offload to GPU	✓	✓
Offload to Intel Xeon Phi Coprocessor		✓
Ability to support all accelerators		✓
Ability to support all GPUs		✓
Ability to support all co-processors		✓
Support for nested parallelism		✓
User-managed memory consistency	✓	✓
Multiple vendor support	✓	✓
Support for dynamic dispatch		✓
Parallel on/off separate from offload		✓
Intel compiler support		2013
Broad standards body approval		✓

OpenMP 4.1 device constructs

- Improve Asynchronous execution of target regions
 - `#pragma target nowait depend(dep-type: list)`
 - `#pragma target update nowait depend(dep-type: list)`
- Unstructured data mapping
 - `#pragma target enter data nowait depend(dep-type: list)`
 - `#pragma target exit data nowait depend(dep-type: list)`
- Device/host memory mgmt API routines
- Miscellaneous refinements