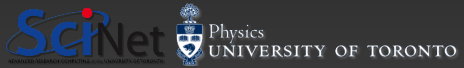
# PHY1610H - Scientific Computing: Heterogeneous Computing with OpenMP

Ramses van Zon and Marcelo Ponce

*SciNet HPC Consortium/Physics Department*
*University of Toronto*

April 2021



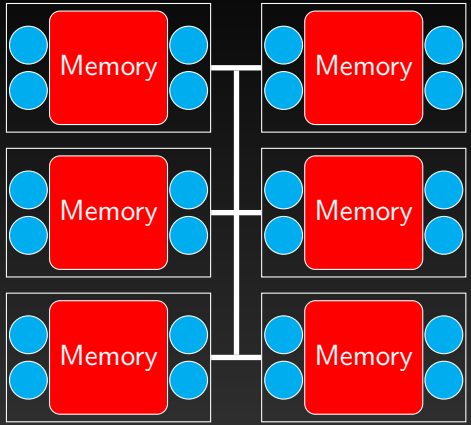
# Lecture 24: Today's class

Today we will discuss the following topics:

- Accelerators: GPGPU and co-processors.
- Heterogeneous Computing.
- OpenMP.
- Other approaches/languages...

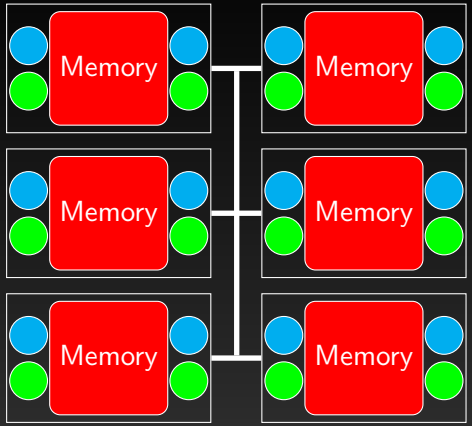
# Hybrid architectures

- Multicore nodes linked together with an (high-speed) interconnect.
- Many cores have modest vector capabilities.
- MPI, OpenMP or OpenMP + MPI can be used in this scenario.



# Hybrid architectures: accelerators

- Multicore nodes linked together with an (high-speed) interconnect.
- Nodes also contain one or more accelerators, GPGPUs (General Purpose Graphics Processing Units) or Xeon Phis.
- These are specialized, super-threaded (500-2000+) processors.
- Machines with GPU: GPU is multi-core, but the amount of shared memory is limited.
- Specialized programming languages, CUDA and OpenCL, are used to program these devices.
- MPI and OpenMP can also be used with the accelerator.




- OpenMP alone can be used within node, to offload computations to accelerators/GPUs.

SciNet Physics UNIVERSITY OF TORONTO

# Standard Parallel Techniques

We have looked at two common ways to parallelize programs in research computing:

- OpenMP: for shared memory systems using compiler directives, and
- MPI: for distributed sytsems using explicit message passing library.

But there are other options!

# Standard Parallel Techniques

We have looked at two common ways to parallelize programs in research computing:

- OpenMP: for shared memory systems using compiler directives, and
- MPI: for distributed sytsems using explicit message passing library.

But there are other options!

## Other Parallel Techniques

- For accelerators (eg. GPUs, Xeon Phi, FPGAs, etc.): CUDA, OpenACC, OpenCL, **OpenMP ($\geq 4$)**
- For alternative shared memory programming: Pthreads, C++11 threads, Cilk++
- For programming distributed memory systems more as similarly as if they were shared memory systems. UPC, MPI3, Coarray (Fortran)
- Hybrid techniques: Combining MPI+OpenMP or any of the above.

# Heterogeneous Computing

## What is it?

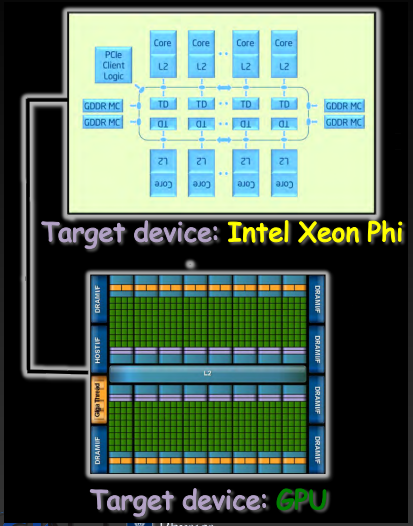
- Use different compute device(s) concurrently in the same computation.
- Example: Leverage CPUs for general computing components and use GPU's for data parallel / FLOP intensive components.
- Pros: Faster and cheaper ($/FLOP/Watt) computation
- Cons: More complicated to program

## Terminology

- GPGPU: General Purpose Grap[hics Processing Unit
- HOST: CPU and its memory
- DEVICE: Accelerator (GPU) and its memory

# Accelerators

- Systems with accelerators are machines which contain an "off-host" accelerator, such as a GPU or Xeon Phi.
- These accelerator devices are very fast and good at massively parallel processing (having 500-2000+ cores).
- Complicated to program.
- Programming model: CUDA, OpenACC, and OpenCL.
- Needs to be combine with at least some 'host' code: **heterogeous computing**.



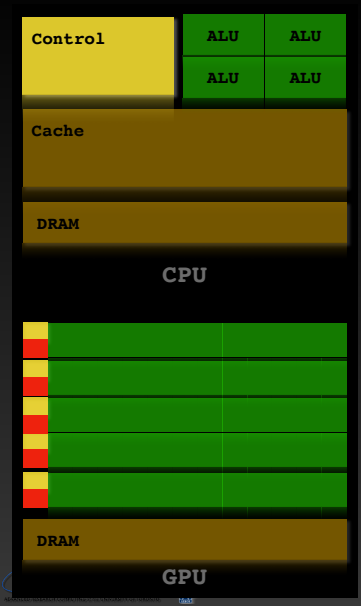
# Accelerators: CPUs vs GPUs

## CPU

- general purpose
- task parallelism (diverse tasks)
- maximize serial performance
- large cache
- multi-threaded (4-16)
- some SIMD (SSE, AVX)

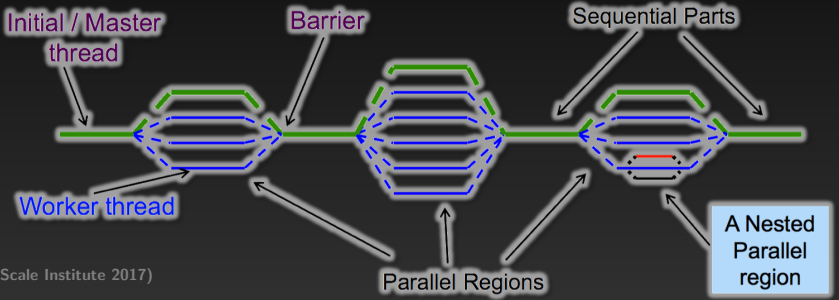
## GPU

- data parallelism (single task)
- maximize throughput
- small cache
- super-threaded (500-2000+)
- "streaming multiprocessors" (SMs)
- almost all SIMD



# OpenMP Execution Model

- Execution starts with single thread (the initial / master thread)
- Master thread spawns multiple worker threads as needed, together they form a team
  **team = master + workers**
- Parallel region is a block of code executed by all threads in a team simultaneously

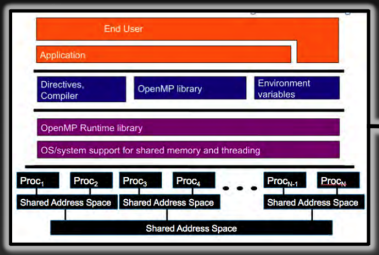



(Credit PetaScale Institute 2017)

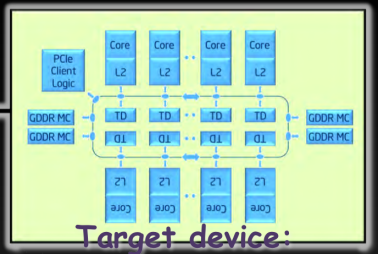
- Number of threads in a team may be dynamically adjusted

# OpenMP 4.5

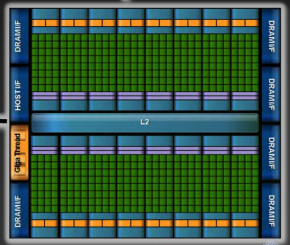
Supported (OpenMP 4.0) with target, teams, distribute, and other constructs




host




Target device:
Intel Xeon Phi




Target device:
GPU

OpenMP 4.5

(Credit PetaScale Institute 2017)

SciNet

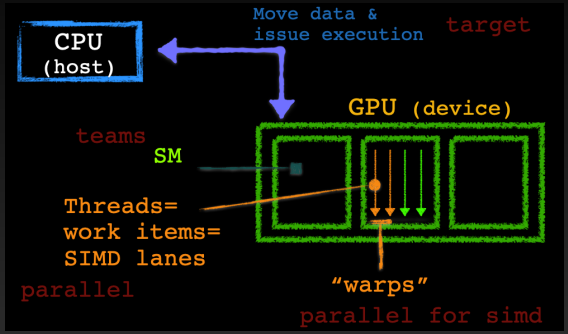
Physics
UNIVERSITY OF TORONTO

# Relevant features in OpenMP

- Target directives (support for accelerators)
  - structured and unstructured target data regions
  - Asynchronous execution (nowait) and data dependences (depend)
- Tasking directives (support asynchronous programming)
  - Load balancing computation
  - Orchestrate work between multicores and accelerators
  - Multi-level parallelism
  - Taskloops
- Loop directives for Worksharing (to support multi-cores and accelerators)
- SIMD directives (to support SIMD parallelism)
- Thread affinity (better control of thread/core bindings)
  - Per parallel region (including nested parallelism)
- Extended runtime APIs
  - Device Memory Routines
- Full list of OpenMP compilers is maintained here:
  `https://www.openmp.org/resources/openmp-compilers-tools`

# Modern OpenMP - Execution Mapping

- The **target** construct offloads the enclosed code to the accelerator: single thread on a device (GPU)
- The **teams** construct creates a league of teams: one thread each, concurrent (not parallel) execution (on SMs)
- The **parallel** construct creates a new team of threads: parallel execution (by hardware threads/by *sub-"warps"*)
- The **simd** construct indicates SIMD execution is allowed: SIMD execution (among *sub-"warps"*)



# OpenMP Target

- Device
  - An implementation-defined (logical) execution unit (or accelerator)
- Device data environment
  - Storage associated with the device
- The execution model is host-centric (or initial device)
  - Host creates/destroys data environment on device(s)
  - Host maps data to the device(s) data environment
  - Host offloads OpenMP target regions to target device(s)
  - Host updates the data between the host and device(s)
  - OpenMP target can be offloaded to an initial device (CPU)

# OpenMP Target

- Device
  - An implementation-defined (logical) execution unit (or accelerator)
- Device data environment
  - Storage associated with the device
- The execution model is host-centric (or initial device)
  - Host creates/destroys data environment on device(s)
  - Host maps data to the device(s) data environment
  - Host offloads OpenMP target regions to target device(s)
  - Host updates the data between the host and device(s)
  - OpenMP target can be offloaded to an initial device (CPU)

## Target construct

- Transfer control from the host to the device
- Syntax (C/C++)
  - `#pragma omp target [clause [[,] clause], ...]`
    structured-block
- Clauses
  - device(scalar-integer-expression)
  - map(alloc | to | from | tofrom: list)
  - if(scalar-expr)

# OpenMP Target

- Device
  - An implementation-defined (logical) execution unit (or accelerator)
- Device data environment
  - Storage associated with the device
- The execution model is host-centric (or initial device)
  - Host creates/destroys data environment on device(s)
  - Host maps data to the device(s) data environment
  - Host offloads OpenMP target regions to target device(s)
  - Host updates the data between the host and device(s)
  - OpenMP target can be offloaded to an initial device (CPU)

## Target construct

- Transfer control from the host to the device
- Syntax (C/C++)
  - `#pragma omp target [clause [[,] clause], ...]`
    structured-block
- Clauses
  - device(scalar-integer-expression)
  - map(alloc | to | from | tofrom: list)
  - if(scalar-expr)

Use target construct to:

- Transfer control from the host to the target device
- Map variables to/from the device data env.

Host thread waits until target region completes (use `nowait` for asynchronous execution)

# OpenMP Target Data Regions

- The map clauses determine how an original (initial device) variable in a data environment is mapped to a corresponding variable in a device data environment
  - Mapped variable:
    - ⋆ An original variable in a (host) data environment has a corresponding variable in a device data environment
- Mapped type:
  - A type that is amenable for mapped variables (e.g. to, from, tofrom, etc)
  - Bitwise copy-able plus additional restrictions
- `map` is not necessarily a copy: copy on multiple cache lines that need to synchronize

# OpenMP Target Data Regions

- The map clauses determine how an original (initial device) variable in a data environment is mapped to a corresponding variable in a device data environment
  - Mapped variable:
    - ⋆ An original variable in a (host) data environment has a corresponding variable in a device data environment
- Mapped type:
  - A type that is amenable for mapped variables (e.g. to, from, tofrom, etc)
  - Bitwise copy-able plus additional restrictions
- `map` is not necessarily a copy: copy on multiple cache lines that need to synchronize

## OpenMP Map-types to target data regions

```
#pragma omp target data map(to:u)
map(from:uold)
```

- `alloc` - allocates data on the device
- `to` - allocates data and moves data to the device
- `from` – allocates data and moves data from the device (target exit data – only transfers)
- `tofrom` – allocates data and moves data to and from the device
- `delete` – deletes the data from the device and sets the ref.count to 0
- `release` – decrements the reference count of a variable

# OpenMP - Execution Example, from CPU to device...

Ex: Multiplies one vector by a scalar and then adds it to another, $a = b + scalar * c$

## CPU implementation

```
#pragma omp parallel for
for (j=0; j<N; j++)
     a[j] = b[j] + scalar*c[j];


// depending on the compiler/hardware combination
// an optimization may result from the simd construct
#pragma omp parallel for simd
for (j=0; j<N; j++)
     a[j] = b[j] + scalar*c[j];
```

# OpenMP - Execution Example, from CPU to device...

Ex: Multiplies one vector by a scalar and then adds it to another, $a = b + scalar * c$

## CPU implementation

```
#pragma omp parallel for
for (j=0; j<N; j++)
     a[j] = b[j] + scalar*c[j];


// depending on the compiler/hardware combination
// an optimization may result from the simd construct
#pragma omp parallel for simd
for (j=0; j<N; j++)
     a[j] = b[j] + scalar*c[j];
```

## target & teams device-offload program

```
#pragma omp target teams distribute parallel for [simd]
for (j=0; j<N; j++)
     a[j] = b[j] + scalar*c[j];
```

# OpenMP - Execution Example, from CPU to device...

Ex: Multiplies one vector by a scalar and then adds it to another, $a = b + scalar * c$

## target & teams device-offload program

```
#pragma omp target teams distribute parallel for [simd]
for (j=0; j<N; j++)
     a[j] = b[j] + scalar*c[j];
```

# OpenMP - Execution Example, from CPU to device...

Ex: Multiplies one vector by a scalar and then adds it to another, $a = b + scalar * c$

## target & teams device-offload program

```
#pragma omp target teams distribute parallel for [simd]
for (j=0; j<N; j++)
     a[j] = b[j] + scalar*c[j];
```

## in general,

```
// data transfer
#pragma omp target enter data map(to:a[0:N])
#pragma omp target enter data map(to:b[0:N])

 . . .

#pragma omp target teams distribute parallel for [simd]
for (j=0; j<N; j++)
     a[j] = b[j] + scalar*c[j];

 . . .

// data transfer
#pragma omp target update from(a[0:N])
```

# OpenMP - Execution Example: implicit data offload

## target offload program

```
#define N 128
double x[N*N];
int i, j, k;
for (k=0; k<N*N; ++k) x[k] = k;

#pragma omp target
// OpenMP implicitly moves data btn
    host and device
//   "x" mapped to and from
//   Scalars are made firstprivate

// Distribute for-loop its btn teams
#pragma omp teams distribute
for (i=0; i<N; ++i) {
// Distribute for-loop its btn threads
     #pragma omp parallel for
     for (j=0; j<N; ++j) {
          x[j+N*i] *= 2.0;
     }
}
```

- The `target` construct offloads the enclosed code to the accelerator
- The `teams` construct creates a league of teams
- The `distribute` construct distributes the outer loop iterations between the league of teams
- The `parallel for` combined construct creates a thread team for each team and distributes the inner loop iterations to threads

# OpenMP - Execution Example: Explicit data managament

```
#define N 100
double *p = malloc(N * sizeof(*p));

#pragma omp parallel for
for (int i=0; i<N; ++i) p[i] = 2.0;

#pragma omp target map(tofrom:p[0:N])
#pragma omp teams distribute parallel for
for (int i=0; i<N; ++i) p[i] *= 2.0;
```

- Data management must be explicit when using pointer variables
- Same pointer name used in host and device environments
- Programmer responsibility to keep the values consistent as needed
- Data directives move data between host and device address spaces

# OpenMP - Execution Example: Explicit data managament

```
#define N 100
double *p = malloc(N * sizeof(*p));

#pragma omp parallel for
for (int i=0; i<N; ++i) p[i] = 2.0;

#pragma omp target map(tofrom:p[0:N])
#pragma omp teams distribute parallel for
for (int i=0; i<N; ++i) p[i] *= 2.0;
```

- Data management must be explicit when using pointer variables
- Same pointer name used in host and device environments
- Programmer responsibility to keep the values consistent as needed
- Data directives move data between host and device address spaces

## Target update construct

Can be used to specify data transfers between host and devices

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

# Unified Virtual Memory Support (OpenMP $\geq$ 5.0)

- Single address space over CPU and GPU memories
- Data migrated between CPU and GPU memories transparently to the application - no need to explicitly copy data

```
#pragma omp requires unified_shared_memory
for (k=0; k < NTIMES; k++)
{
    // No data directive needed for pointers a, b, c
    #pragma omp target teams distribute parallel for
    for (j=0; j<N; j++) {
        a[j] = b[j] + scalar*c[j];
    }
}
```

# OpenMP - Use of Unified Memory: OpenMP+CUDA

```
// combining OpenMP-4.5 and CUDA to use unified memory

cudaMallocManaged((void**)&a, sizeof(double) * N);
cudaMallocManaged((void**)&b, sizeof(double) * N);
cudaMallocManaged((void**)&c, sizeof(double) * N);

#pragma omp target teams distribute \
 parallel for is_device_ptr(a, b, c)
 for (j=0; j<N; j++)
      a[j] = b[j] + scalar*c[j];
```

# OpenMP Device Constructs – Core Functionality

## Execute code on a target device

- `omp target`
- `omp declare target`

## Manage the device data environment

- `map`
- `omp target data`
- `omp target enter/exit data`
- `omp target update`
- `omp declare target`

## Parallelism and Workshare for devices

- `omp teams`
- `omp distribute`

## Device Runtime Routines

- `omp_get_...`

## Environemnt Variables

- `OMP_DEFAULT_DEVICE`
- `OMP_THREAD_LIMIT`
- `OMP_TARGET_OFFLOAD`
- ...

SciNet Physics UNIVERSITY OF TORONTO

# OpenMP - OpenACC

## OpenMP Platform Model

target/teams

#pragma omp teams distribute parallel for simd

```
#define N 128
double x[N*N];
int i, j, k;

// initialization
for (k=0; k<N*N; ++k) x[k] = k;

#pragma omp target
#pragma omp teams distribute
for (i=0; i<N; ++i) {
     #pragma omp parallel for simd
     for (j=0; j<N; ++j) {
          x[j+N*i] *= 2.0;
     }
}
```

## OpenACC Platform Model

gangs/worker vector

#pragma acc parallel #pragma acc loop

```
#define N 128
double x[N*N];
int i, j, k;

// initialization
for (k=0; k<N*N; ++k) x[k] = k;

#pragma acc parallel
#pragma acc gang worker
for (i=0; i<N; ++i) {
     #pragma acc vector
     for (j=0; j<N; ++j) {
          x[j+N*i] *= 2.0;
     }
}
```

# OpenACC - OpenMP Conversion

| | |
|---|---|
| acc parallel | omp [target] teams |
| acc loop independent | omp loop |
| acc loop gang | omp distribute order(concurrent) |
| acc loop worker | omp parallel for order(concurrent) |
| acc loop vector | omp simd order(concurrent) |
| acc parallel loop | omp [target] teams loop |
| acc copyin(), copyout(), copy() | omp map(to:), map(from:), map(tofrom:) |
| acc data, acc end data | omp target data, omp end target data |
| acc enter data,acc exit data | omp target enter data, omp target exit data |
| acc update host(), acc update device() | omp target update to(), omp target update from() |

# Summary

- Several advantages of directive-based parallelism
- Incremental parallel programming
- Single source code for sequential and parallel programs
  - Use compiler flag to enable or disable
  - No major overwrite of the serial code
- Works for both CPU and GPU/accelarators
- Low learning curve, familiar C/C++/Fortran program environment
  - Do not need to worry about lower level hardware details
- Simple programming model than lower level programming models
- Portable implementation:
  - Different architectures, different compilers handle the hardware differences
  - Performance strongly dependes on compiler/hardware and constructs, must experiment!

## References


- "Introduction to Directive Based Programming on GPU", Helen He (Feb'20)
- "OpenMP 5.0/5.1 Tutorial", EPC (2020)

# Course Recap – PHY1610 (2021)

## Best Practices in Scientific Computing

- Version Control (git)
- Modular Programming
- Testing
- Debugging
- File IO: NetCDF

## Reusing existing solutions

- Using Libraries
- RARRAY, STL, FFTW, BLAS, LAPACK, GSL, BOOST

## Performance

- Profiling
- Performance metrics (speedup, efficiency, throughput)
- Using clusters and schedulers
- Shared memory programming (OpenMP)
- Parallel programming (MPI)
- Heterogeneous Computing (OpenMP)

If you haven't yet, please take some minutes to complete the course evaluation
Thank you!