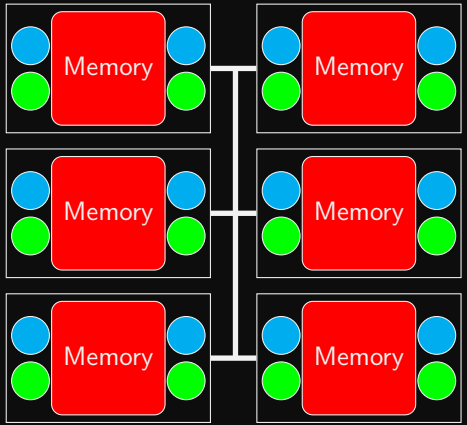
# GPU Computing with Directives

Ramses van Zon

PHY1610 Winter 2024

# Hybrid architectures with accelerators

- Multicore nodes linked together with an (high-speed) interconnect.
- Nodes also contain one or more accelerators, usually GPUs.
- These are specialized, super-threaded (500-2000+) processors.
- GPUs have their own, limited, shared memory.
- Specialized programming languages, CUDA, OpenCL, OpenACC, OpenMP.
- Can be mixed with MPI, OpenMP.



# Heterogeneous Computing

## What is it?

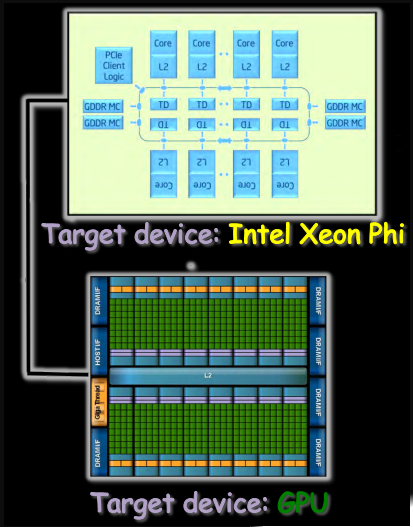
- Use different compute device(s) concurrently in the same computation.
- Example: Leverage CPUs for generatl computing components and GPUs for data parallel and floating point intensive components.
- Pros: GPUs are faster and cheaper ($/FLOP/Watt) for compute
- Cons: More complicated to program, only benefits certain applications.

## Terminology

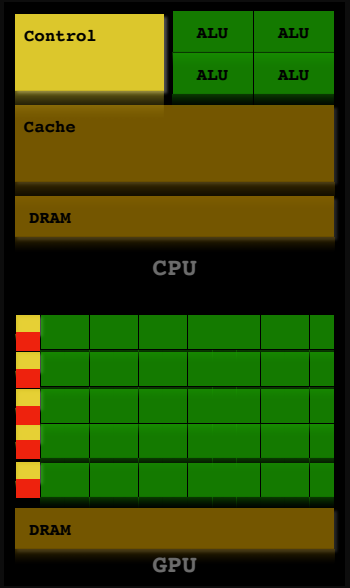
- GPGPU Programming: General Purpose Graphics Processing Unit Programming
- HOST: CPU and its memory
- DEVICE: Accelerator (GPU) and its memory

# 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, OpenMP offloading, and OpenCL.
- Needs to be combine with at least some 'host' code: **heterogeous computing**.



# Accelerators: CPUs vs GPUs




## CPU

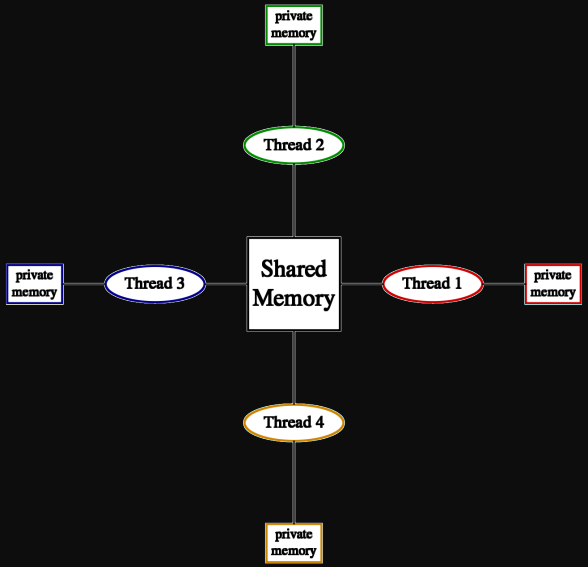
- general purpose
- task parallelism (diverse tasks)
- maximize serial performance
- large cache
- multi-threaded (4-16)
- some Single-Instruction-Multiple-Data (SIMD)

## GPU

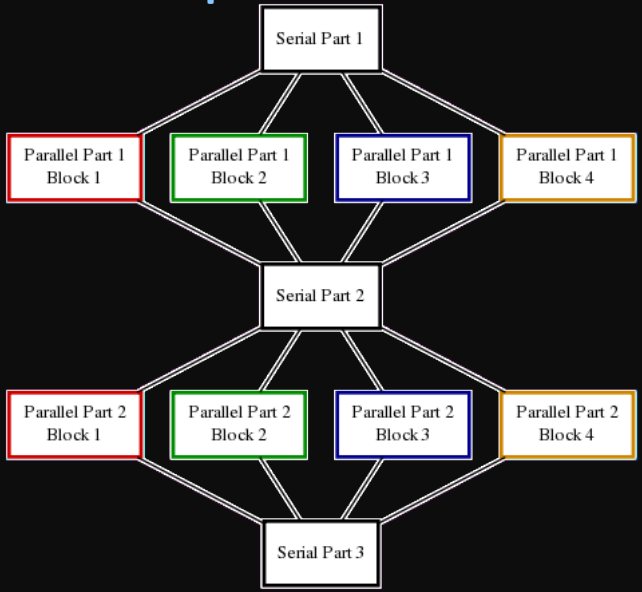
- data parallelism (single task)
- maximize throughput
- small cache
- super-threaded (500-2000+)
- "streaming multiprocessors" (SMs)
- almost all SIMD

# Programming Accelerators with OpenMP

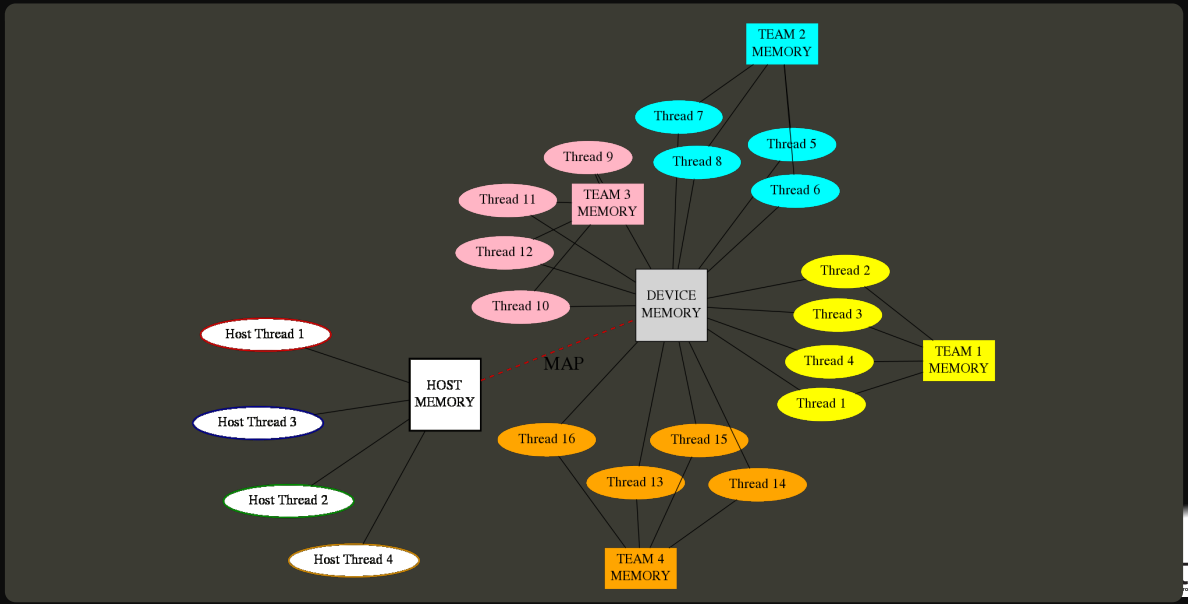
# Memory Model in OpenMP (3.1)



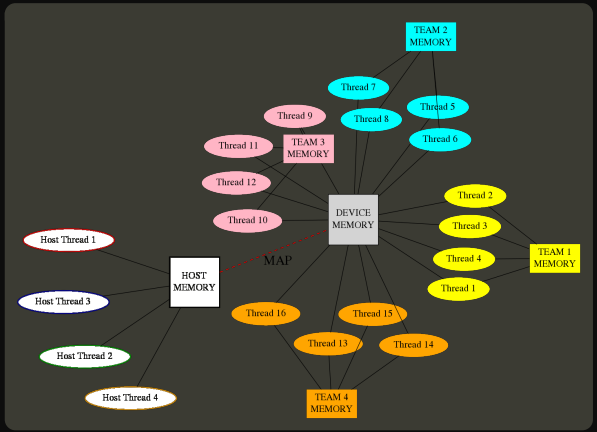
# Execution Model in OpenMP



# Memory Model in OpenMP 4+



# Memory Model in OpenMP 4+




- Device has its own data environment
- And its own shared memory
- Threads can be bundled in a teams of threads
- These threads can have memory shared among threads of the same team
- Whether this is beneficial depends on the memory architecture of the device.

# Data mapping

- Host memory and device memory usually district.
- OpenMP 4+ allows host and device memory to be shared (e.g. on Mist).
- To accommodate both, the relation between variables on host and memory gets expressed as a *mapping*:

  Different types:

  - `to`: existing host variables copied to a corresponding variable in the target before
  - `from`: target variables copied back to a corresponding variable in the host after
  - `tofrom`: Both `from` and `to`
  - `alloc`: Neither `from` nor `to`, but ensure the variable exists on the target but no relation to host variable.

  Note: arrays and array sections are supported.

# OpenMP Device Constructs – Core Functionality

## Execute code on a target device

- `omp target`
- `omp declare target`

## Parallelism and Workshare for devices

- `omp teams`
- `omp distribute`

## Manage the device data environment

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

## Device Runtime Routines

- `omp_get_...`

## Environment variables

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

# Example

```
#include <rarray>
double sumarray(rarray<double,1> a) {
    double sum=0.0;
    double* data = a.data();
    int n = a.size();
    #pragma omp target map(data[0:n]) map(tofrom:sum)
    #pragma omp teams distribute parallel for reduction(+:sum)
    for (int i = 0; i < n; i++)
        sum += data[i];
    return sum;
}
```

```
int main() {
    int n = 50'000'000;
    int i = 0;
    rarray<double,1> a(n);
    for (double& x: a)
       x = (++i)/(0.5*n*(n+1));
    double sum = sumarray(a);
    std::cout <<  "Sum is: " << sum << "\n";
    std::cout << "This should be 1.0 (up to epsilon)\n";
    std::cout <<  "Sum - 1.0 is: " << sum - 1.0 << "\n";
}
```

- Sums elements in array on the GPU
- Specify data needed on device
- Does not work with `rarray`, `std::vector`, etc. (OpenMP 5.2 -> custom mappers)
- Instead: use pointers and sizes in `map`
- Multiple levels of parallelization

# Compilation

E.g. Mist or Graham, you can use the NVIDIA compilers

```
$ module load nvhpc
$ nvc++ -std=c++17 -mp=gpu foo.cpp -o foo
```

If your version of gcc supports gpu offloading and you have an NVIDIA GPU:

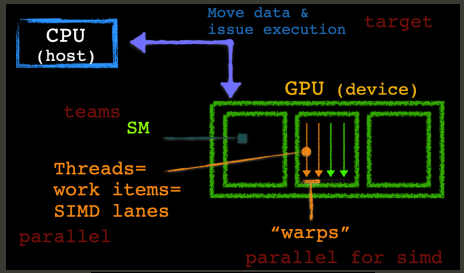
```
$ g++ -std=c++17 -fopenmp -foffload=nvptx-none foo.cpp -o foo
```

Run as usual:

```
$ ./foo
Sum is: 1.0
This should be 1.0 (up to epsilon)
Sum - 1.0 is: 1.66e-16
```

# 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 execution (on SMs)
- The **parallel** construct creates a new team of threads: parallel execution
- The **simd** construct indicates SIMD execution is allowed: SIMD execution




# 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
  - Host creates/destroys data on device(s)
  - Host maps data to the device(s)
  - Host offloads OpenMP target regions to target device(s)
  - Host updates the data between host and device(s)

**Target construct**

Transfer control from the host to the device

- `pragma omp target [clause, ...]`
- 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 (or use `nowait`)

# 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, $a = b + scalar * c$

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

But you can delay data transfer:

```
// 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])

#pragma omp target exit data
```

# OpenMP Implicit Data Offload

**target offload program**

```
int main() {
  #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 gpu
  //   "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 Explicit Data Management

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

# 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];
    }
}
```

Only when the hardware supports it!

# Conclusion GPU with OpenMP

- Incremental parallel programming
- Single source code for sequential and parallel programs
  - Use compiler flag to enable or disable
  - No major rrwrite of the serial code
    (But mapping requires rewriting code if not using pointers for arrays, or defining mappers)
- Works for both CPU and GPU/accelarators
- On GPUs, must worry about data movement for performance.
- Simpler programming model than lower level programming models
- Alternatives: OpenACC, CUDA/HIP

**References**


- "Introduction to Directive Based Programming on GPU", Helen He (Feb'20)
- Using OpenMP with GPUs (pt 1)

# Course Conclusion

# Course Recap PHY1610 (2023)

## Best Practices in Scientific Computing

- version control (git)
- commenting
- modular programming
- testing
- debugging

## Reusing Existing Solutions

- using libraries
- rarray, STL, FFTW, BLAS, LAPACK, GSL
- calling C functions in C++

## Performance

- profiling
- file IO: NetCDF
- performance metrics (speedup, efficiency, throughput)
- using clusters and schedulers
- shared memory programming (OpenMP)
- parallel programming (MPI)
- heterogeneous computing (OpenMP)

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

***Thank you!***