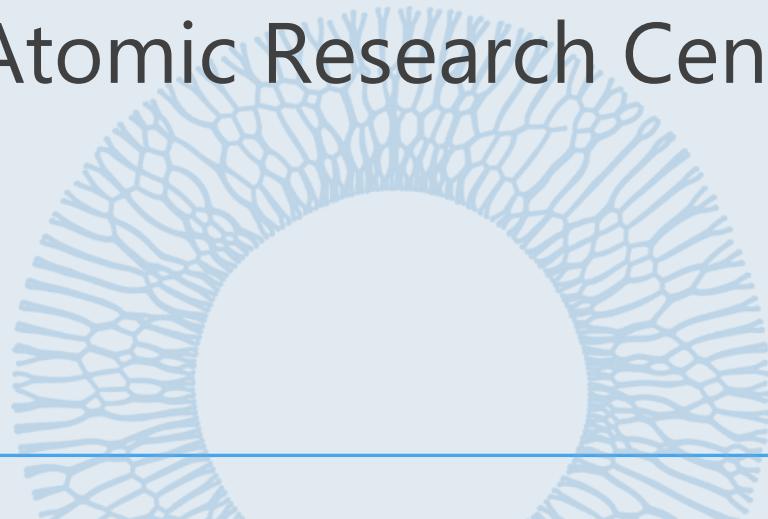
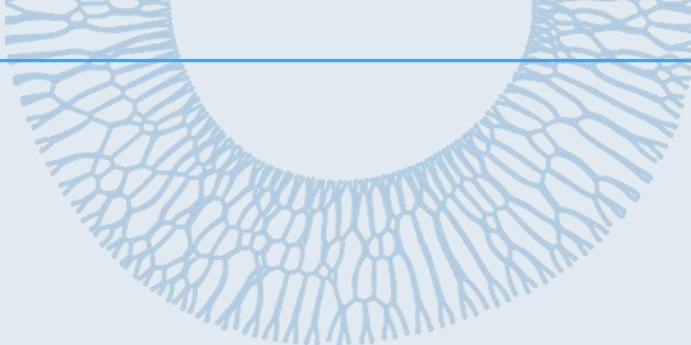


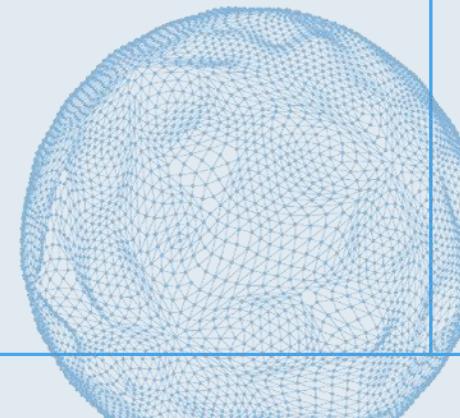
OpenMP GPU

Computer Division
Bhabha Atomic Research Centre



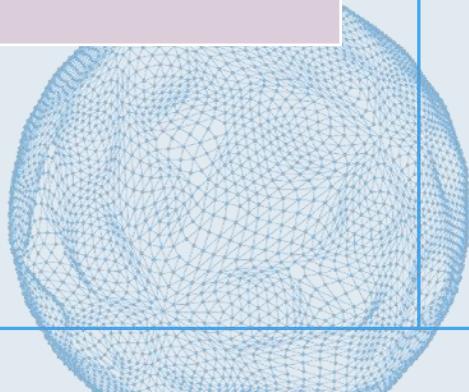


TOPICS

- Computing with Accelerators
 - Heterogeneous Computing
 - OpenMP GPU
 - Development Cycle
 - Directives
 - Data Mapping
 - NVIDIA HPC SDK
 - Vector Addition Example
- 

COMPUTING WITH ACCELERATORS

CPU	OpenMP	MPI		OpenACC	OpenCL
GPU	OpenMP		CUDA	OpenACC	OpenCL
DSP				OpenACC	OpenCL
FPGA				OpenACC	OpenCL



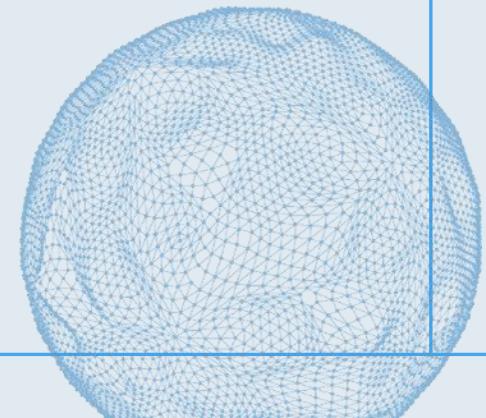
Software for Heterogenous Computing

- Low Level APIs
 - CUDA: Specific to Nvidia GPUs
 - OpenCL: All Accelerators supported, even FPGA
 - Both are openly available, but one needs to learn new syntax for writing and calling kernel functions
- High Level APIs (requires compiler support)
 - OpenACC: Compiler directive based, user friendly, similar to OpenMP
 - OpenMP: Ver 4.0 onwards has support for heterogenous archs
- Compilers
 - NVCC and CRAY compilers provide support for OpenACC
 - Intel Compilers provide support through OpenMP



HETEROGENEOUS COMPUTING

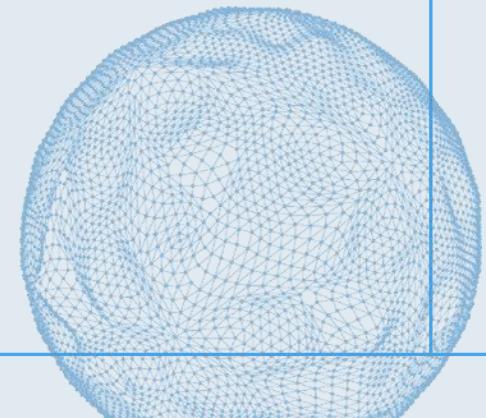
- Libraries
 - cuBLAS, cuFFT, CUDA Math
- Directive based
 - OpenACC, OpenMP
- Programming languages
 - CUDA - NVIDIA
 - OpenCL - NVIDIA, ATI, FPGA

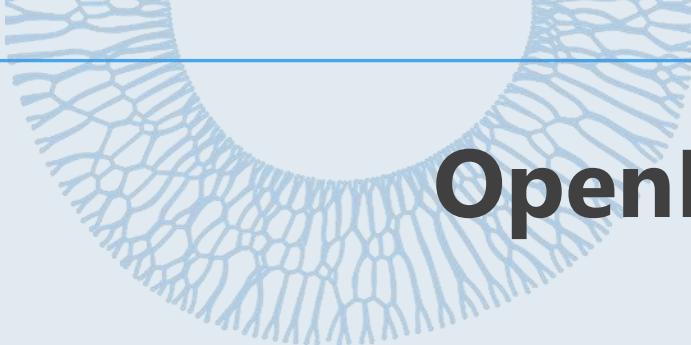




OpenMP GPU – INTRODUCTION

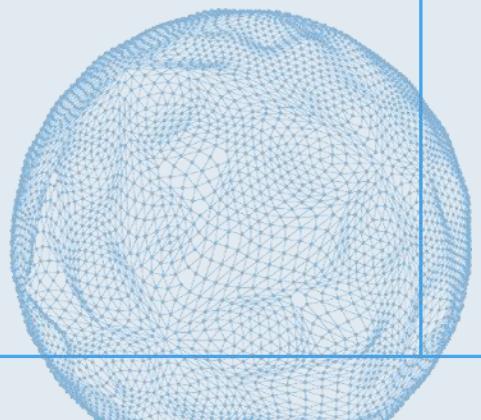
- Supported by
 - GCC, and NVIDIA
- Incremental
- Single Source
- Low Learning Curve





OpenMP GPU – INCREMENTAL

- Maintain existing sequential code
- Add annotations to expose parallelism
- Verify correctness
- Annotate more code



OpenMP GPU – INCREMENTAL

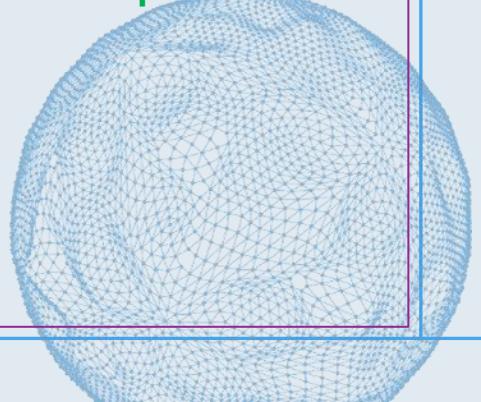
- Working sequential code
- Parallelization with OpenMP GPU
- Verify correctness
- Compute performance

// Sequential code

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

// OpenMP GPU code

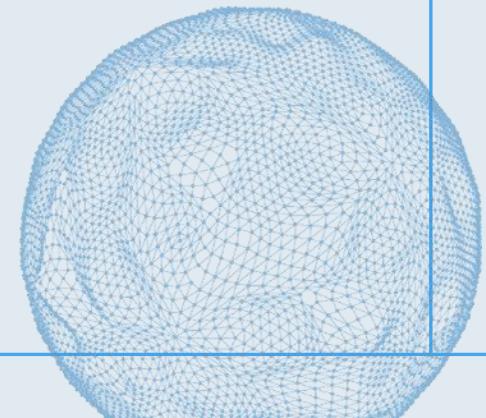
```
#pragma omp target teams distribute parallel for  
for( i=0; i <N; i++)  
{  
    c[i] = a[i] + b[i]  
}
```





OpenMP GPU – SINGLE SOURCE

- Rebuild same code - Multiple architectures
- Compiler based parallelization - Desired machine
- Maintained sequential code



OpenMP GPU – SINGLE SOURCE

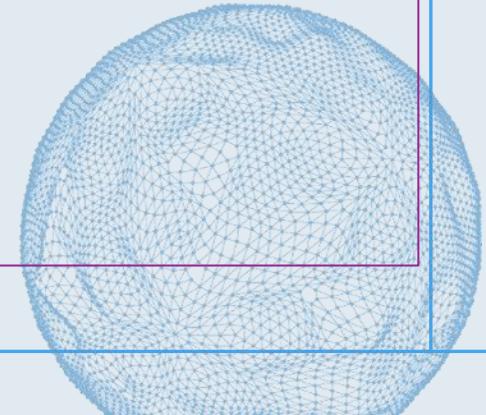
- Supported platforms
 - NVIDIA GPU, and AMD GPU
- Optional OpenMP GPU code additions
- Same code for parallel or sequential execution

// Sequential code

```
#pragma omp target teams distribute parallel for
for( i=0; i <N; i++)
{
    c[i] = a[i] + b[i]
}
// nvc++ -mp=multicore main.c
```

// OpenMP GPU code

```
#pragma omp target teams distribute parallel for
for( i=0; i <N; i++)
{
    c[i] = a[i] + b[i]
}
// nvc++ -mp=gpu main.c
```



OpenMP GPU – LOW LEARNING CURVE

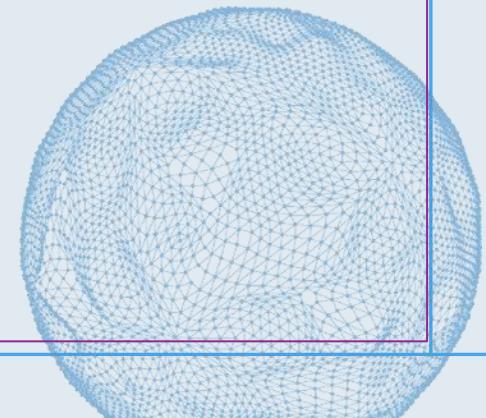
- Easy to use
- Easy to learn
- Programming languages - C, C++, or Fortran
- No need to learn low level hardware details

OpenMP GPU – LOW LEARNING CURVE

- Hints to compiler
- Parallelization by compiler

```
int main(int argc, char ** argv)
{
    // Sequential code

#pragma omp target teams distribute parallel for
    for( i=0; i < N; i++)
    {
        // Parallel code
    }
}
```



OpenMP GPU – DEVELOPMENT CYCLE

Analyze

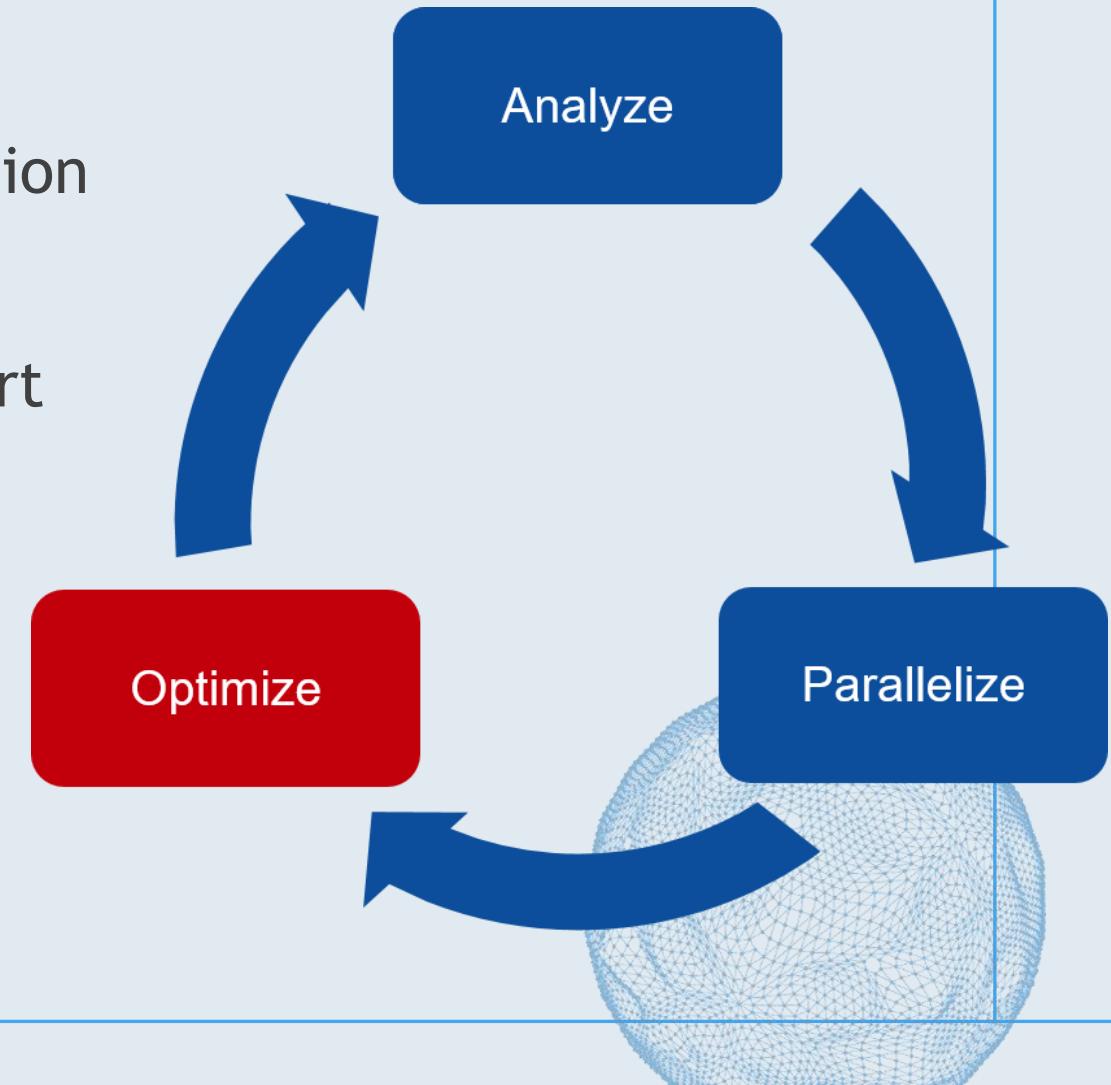
- Identify code region for parallelization

Parallelize

- Parallelize most time consuming part
- Check for correctness

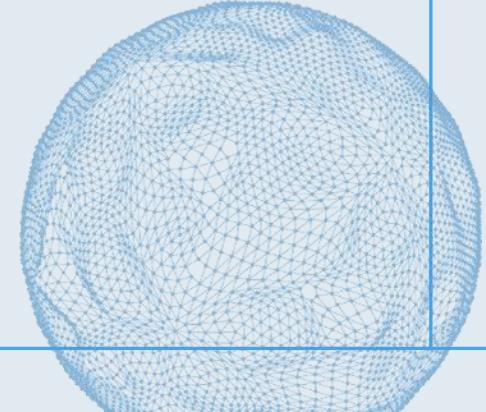
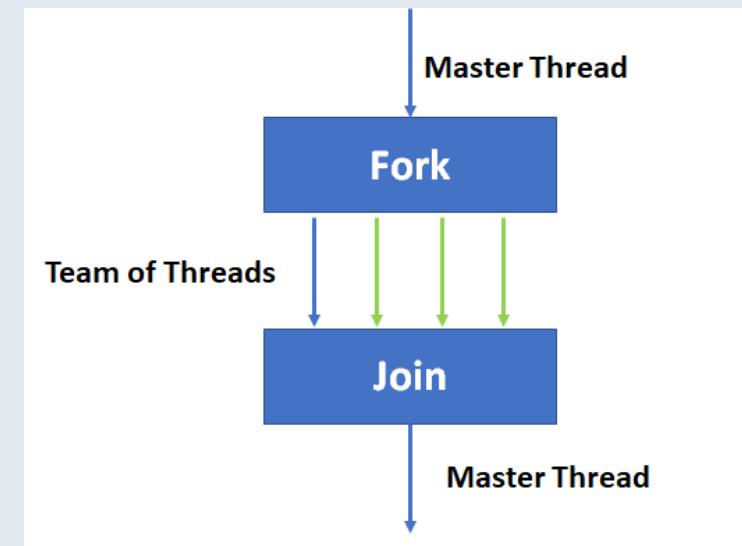
Optimize

- Improve speed-up



OpenMP GPU – FORK JOIN MODEL

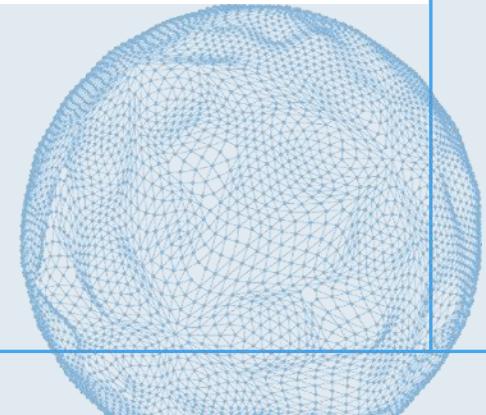
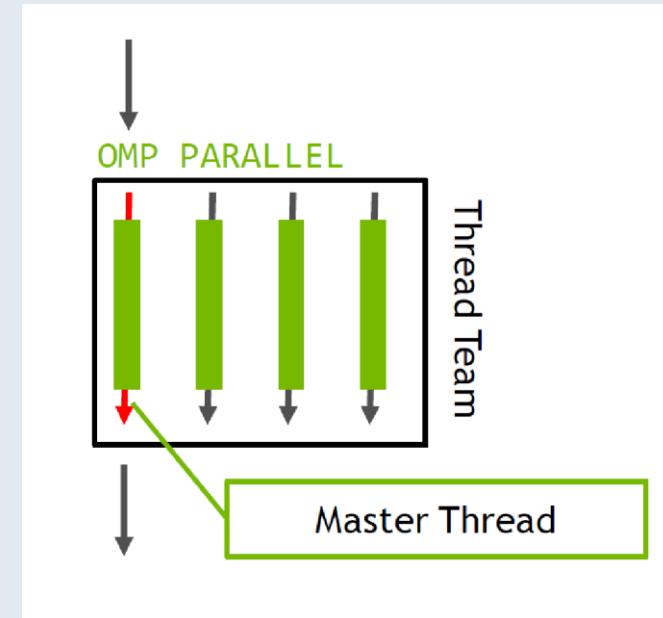
- Fork Join Model
 - Master thread
 - Team of threads
 - Team of parallel threads
 - Execute in parallel
 - Join threads
 - Continue master thread



OpenMP GPU DIRECTIVES

#pragma omp parallel

- Master thread
- Create team of threads
- Execute in parallel
- Join threads
- Continue master thread

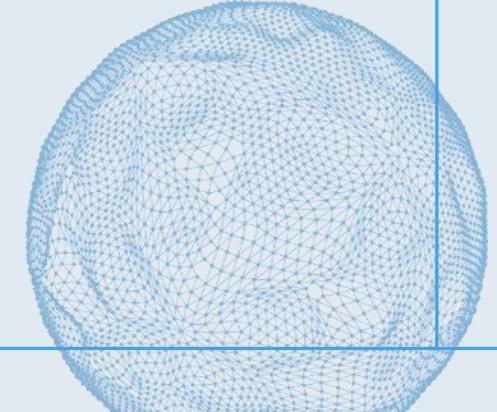
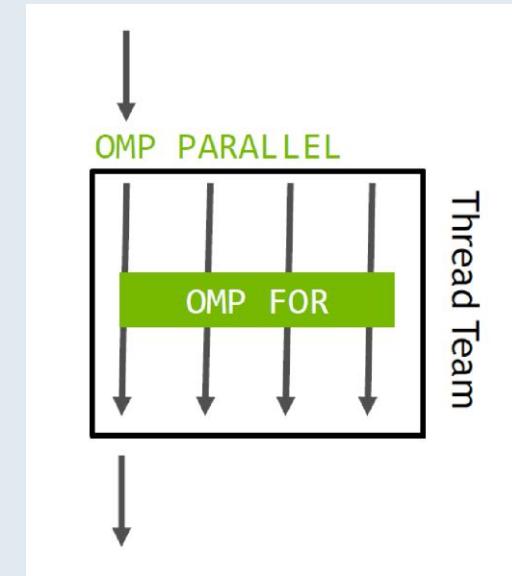


OpenMP GPU DIRECTIVES

#pragma omp for

- Divide workshare
- Divide workshare across thread teams

```
// Create team of threads
#pragma omp parallel
{
    // Workshare loop across threads
    #pragma omp for
    for (i = 0; i < N; i++) {
        c[i] = a[i] * b[i];
    }
}
```



OpenMP GPU DIRECTIVES

#pragma omp target

- Transfer execution to device
- Transfer data to/from device

Host execution



Device execution

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

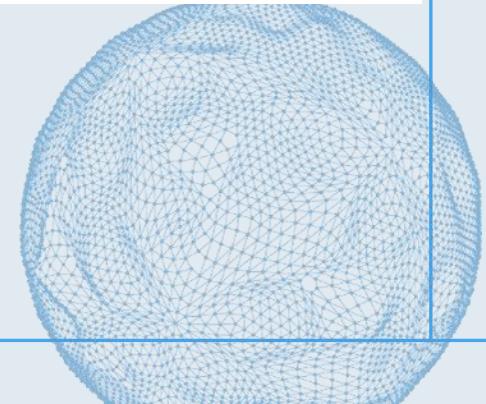
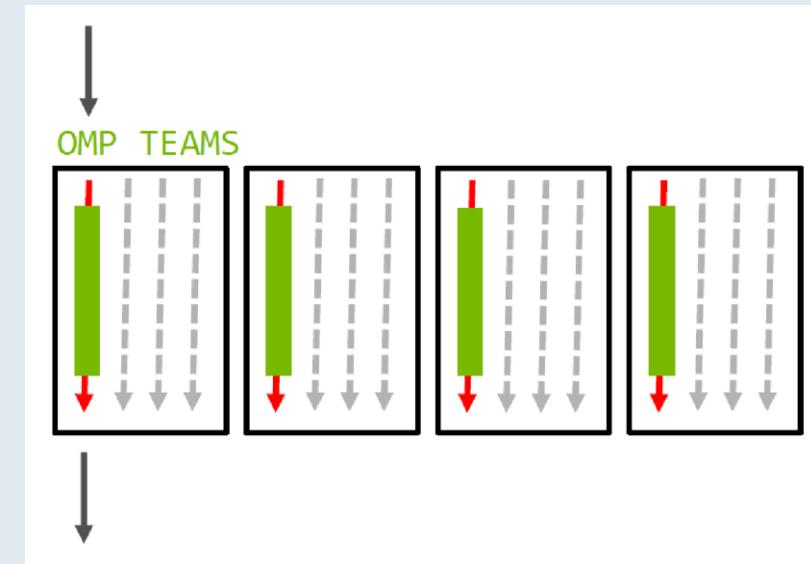
Host execution



OpenMP GPU DIRECTIVES

#pragma omp teams

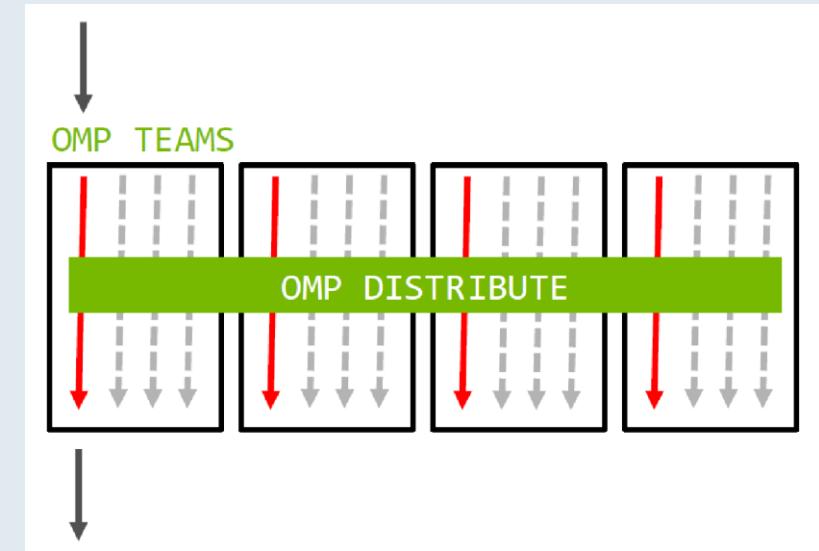
- Utilize GPU resources
- Create one or more thread teams
- Continue on team master threads
- No synchronization between teams



OpenMP GPU DIRECTIVES

#pragma omp distribute

- Distribute iterations to master threads of teams
- Static distribution
- No guaranteed execution order
- Teams may not execute simultaneously

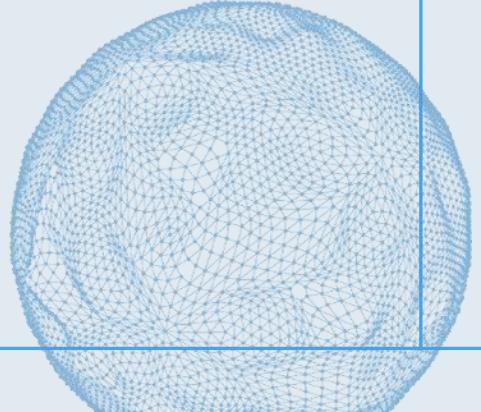


OpenMP GPU DIRECTIVES

#pragma omp reduction(op:var)

- Support map reductions
- Team of threads perform local reductions
- Master threads perform final reduction
- Reduction operations are done on GPU

```
#pragma omp \
target teams distribute \
parallel for \
reduction(+:total)
for( i=0; i <N; i++)
{
    total += array[i]
}
```





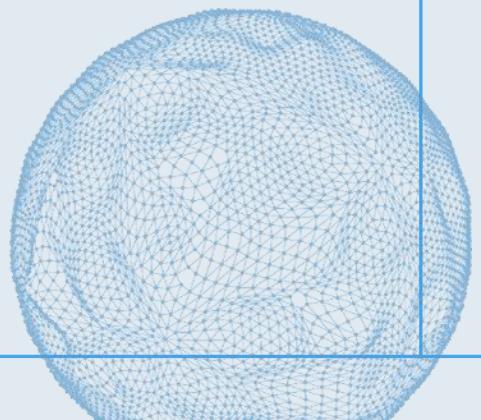
OpenMP GPU – DATA MAPPING

Data mapping controls:

- How data is created on GPU ?
- When data moves between CPU and GPU ?
- What data is transferred ?
- Direction of data movement
- Lifetime of GPU data

Applied with:

- `#pragma omp target directive`



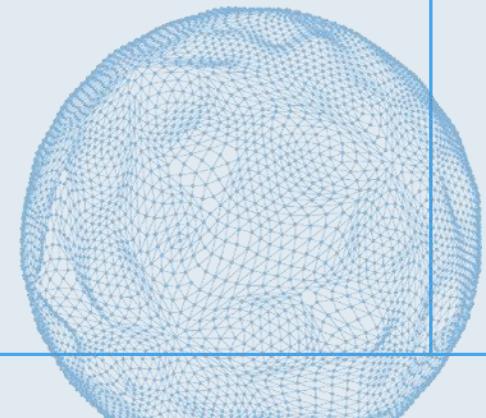
OpenMP GPU – DATA MAPPING

map(tofrom: var)

- Allocate GPU memory
- Copy data from CPU to GPU at start
- Copy data from GPU to CPU at end
- Release GPU memory

map(to: var)

- Allocate GPU memory
- Copy data from CPU to GPU at start
- Release GPU memory
- **No copy** back at end



OpenMP GPU – DATA MAPPING

map(from: var)

- Allocate GPU memory
- No initial copy
- Copy data from GPU to CPU at end
- Release GPU memory

OpenMP GPU – DATA MAPPING

Create space for **a**, **b**, **c**

Copy **a**, **b**, **c** data at beginning

Copy back **a**, **b**, **c** data at end

Release space for **a**, **b**, **c**

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

OpenMP GPU – DATA MAPPING

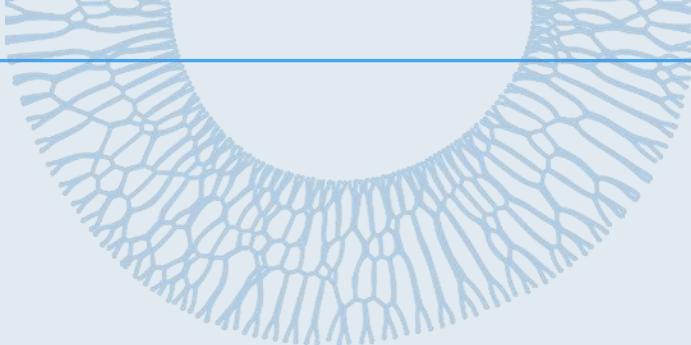
Create space for **a**, **b**, **c**

Copy **a**, **b** data at beginning

Copy back **c** data at end

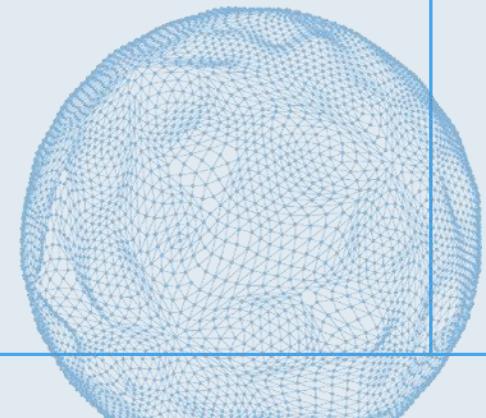
Release space for **a**, **b**, **c**

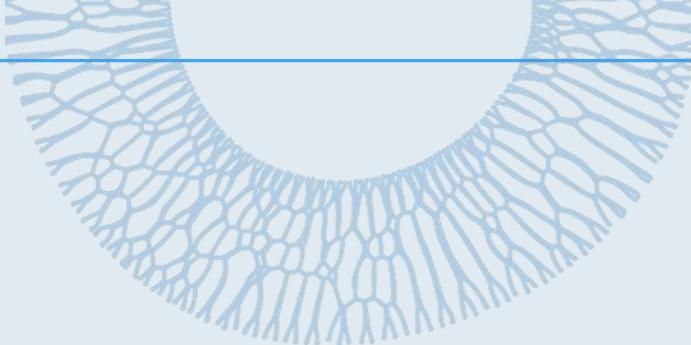
```
#pragma omp target teams distribute parallel for \
    map(to: a[0:N], b[0:N]) \
    map(from: c[0:N])
{
    for (i = 0; i < N; i++) {
        c[i] = a[i] * b[i];
    }
}
```



NVIDIA HPC SDK

- Unified suite for GPU accelerated HPC development
- Includes compilers, libraries, and developer tools
- Supports CUDA, OpenACC, and OpenMP
- NVIDIA HPC compiler
 - OpenMP target offload onto GPU
 - nvc - C code
 - nvc++ - C++ code
 - nvfortran - Fortran code



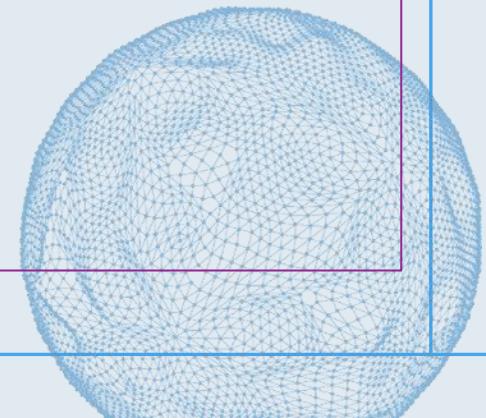


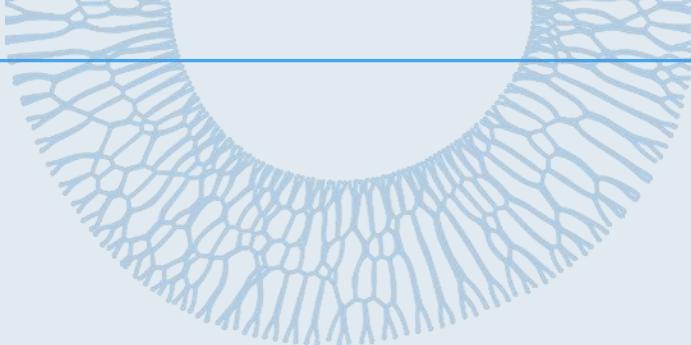
NVIDIA HPC SDK

- -mp
 - Compiler switch to enable OpenMP directives and pragmas
- gpu
 - OpenMP directives are compiled for GPU execution and multicore CPU as fallback
- multicore
 - OpenMP directives are compiled for multicore CPU execution only

```
/// OpenMP multicore CPU code
#pragma omp target teams distribute parallel for
for( i=0; i <N; i++)
{
    c[i] = a[i] + b[i]
}
/// nvc++ -mp=multicore main.c
```

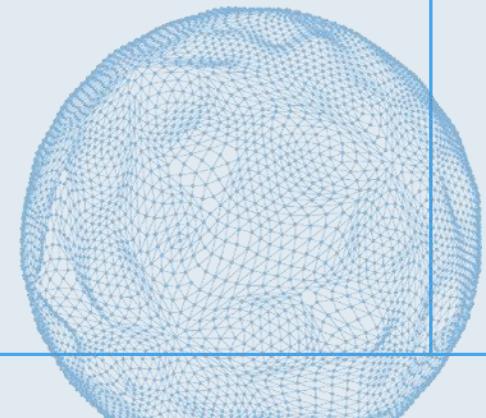
```
/// OpenMP GPU code
#pragma omp target teams distribute parallel for
for( i=0; i <N; i++)
{
    c[i] = a[i] + b[i]
}
/// nvc++ -mp=gpu main.c
```





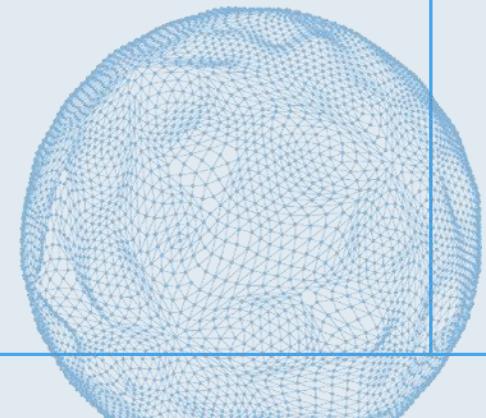
OpenMP GPU – VECTOR ADDITION

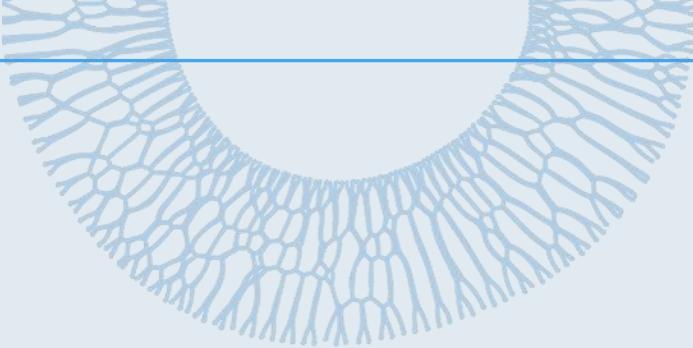
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
```



OpenMP GPU – VECTOR ADDITION

```
int main( int argc, char* argv[] )  
{  
    // Size of vectors  
    int n = 10000;  
    // Input vectors  
    double * a;  
    double * b;  
    // Output vector  
    double * c;
```





OpenMP GPU – VECTOR ADDITION

```
// Size, in bytes, of each vector
```

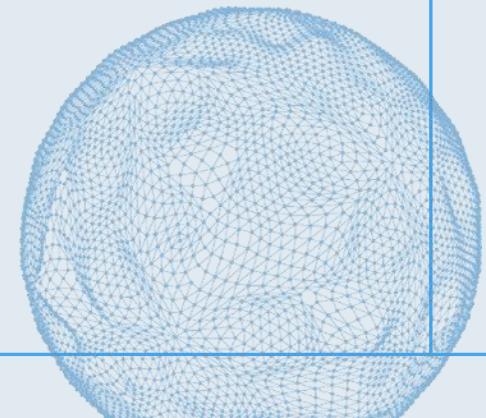
```
size_t bytes = n*sizeof(double);
```

```
// Allocate memory for each vector
```

```
a = (double*)malloc(bytes);
```

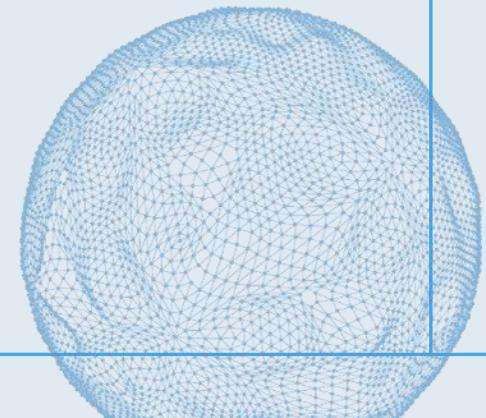
```
b = (double*)malloc(bytes);
```

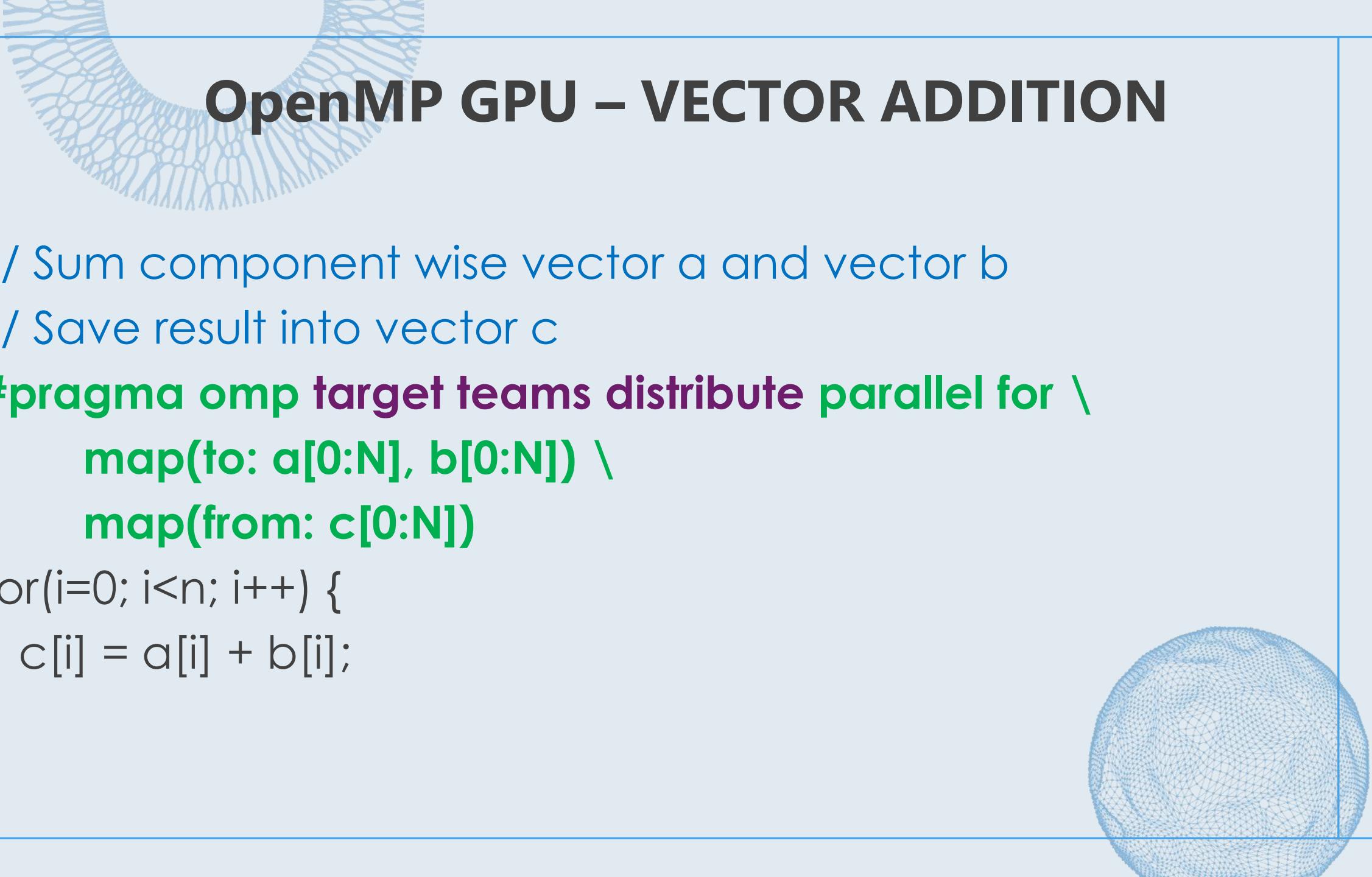
```
c = (double*)malloc(bytes);
```



OpenMP GPU – VECTOR ADDITION

```
// Initialize content of input vectors,  
// vector a[i] = sin(i)^2  
// vector b[i] = cos(i)^2  
  
int i;  
  
for(i=0; i<n; i++) {  
    a[i] = sin(i)*sin(i);  
    b[i] = cos(i)*cos(i);  
}
```



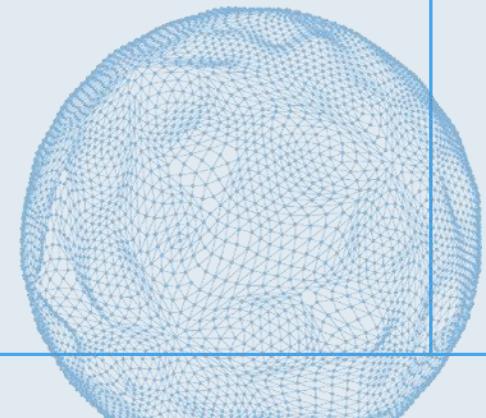


OpenMP GPU – VECTOR ADDITION

```
// Sum component wise vector a and vector b  
// Save result into vector c  
#pragma omp target teams distribute parallel for \  
    map(to: a[0:N], b[0:N]) \  
    map(from: c[0:N])  
for(i=0; i<n; i++) {  
    c[i] = a[i] + b[i];  
}
```

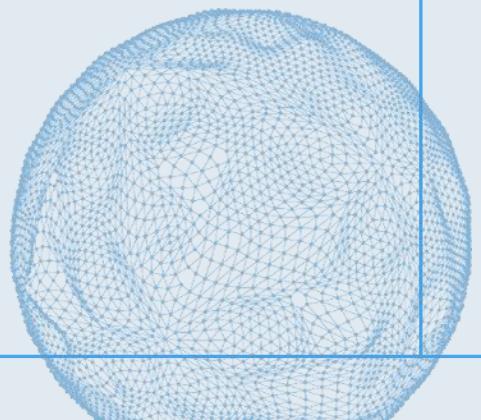
OpenMP GPU – VECTOR ADDITION

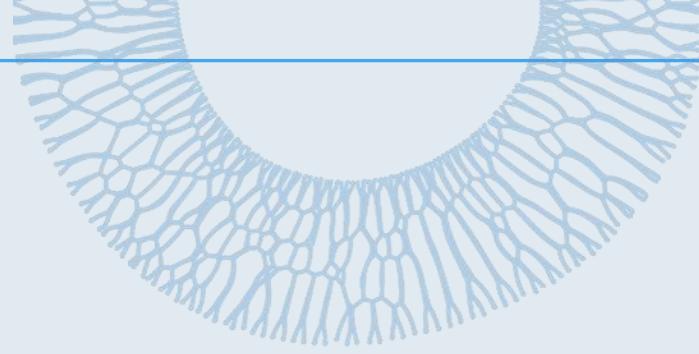
```
// Sum up vector c  
// Print result divided by n  
// This should equal 1 within error  
  
double sum = 0.0;  
  
for(i=0; i<n; i++) {  
    sum += c[i];  
}  
  
sum = sum/n;  
  
printf("final result: %f\n", sum);
```



OpenMP GPU – VECTOR ADDITION

```
// Release memory  
free(a);  
free(b);  
free(c);  
  
return(0);  
}
```





QUESTIONS?

Thank you

