



INAF

ISTITUTO NAZIONALE
DI ASTROFISICA

GPU Programming

Lecture 2

UniTS Advance HPC Course 2024/2025

Agenda

- Brief introduction to CUDA
- OpenMP for GPUs
- Examples and Exercises

But not everything today....

15 minutes of CUDA...

The Kernel concept

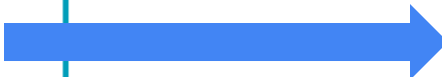
- A function which runs on a GPU is called “**kernel**”
 - when a kernel is launched on a GPU thousands of threads will execute its code
 - programmer chooses the number of threads to run
 - **each thread acts on a different data element independently**
 - the GPU parallelism is very close to the SPMD paradigm

```
void vecAddCPU (int N, const float *A,  
               const float *B, float *C)  
{  
    for ( int i = 0; i < N; i++ )  
        c[i] = a[i] + b[i];  
}  
...  
// call vecAddCPU on N elements  
vecAddCPU ( N, a, b, c );
```

```
void vecAddGPU (int N, const float *A,  
               const float *B, float *C)  
{  
    int i = threadIdx.x;  
    if ( i < N) c[i] = a[i] + b[i];  
}  
...  
// call vecAddGPU on N elements  
vecAddGPU<<<1, N>>>>( N, a, b, c );
```

Example: Vector Sum

```
int main(int argc, char *argv[]) {  
    int i;  
    const int N = 1000000;  
    double u[N], v[N], z[N];  
  
    initVector (u, N, 1.0);  
    initVector (v, N, 2.0);  
    initVector (z, N, 0.0);  
  
    printVector (u, N);  
    printVector (v, N);  
  
    // z = u + v  
    for (i=0; i<N; i++)  
        z[i] = u[i] + v[i];  
  
    printVector (z, N);  
  
    return 0;  
}
```



This is the **computing intensive** part that must run on the accelerator:
a **kernel** can be created

Example: Vector Sum on GPU

```
__global__ void gpuVectAdd (int N, const double *u, const double *v, double *z)
{
    // index is a unique identifier of each GPU thread
    int index = blockIdx.x * blockDim.x + threadIdx.x ;
    if (index < N)
        z[index] = u[index] + v[index];
}
```

The `__global__` qualifier
declares this function to be a CUDA kernel

CUDA kernels are special C functions:

- can be called from host only
- must be called using the *execution configuration* syntax
- the return type must be *void*
- they are **asynchronous**: control is returned immediately to the host code
- an explicit synchronization is needed in order to be sure that a CUDA kernel has completed the execution

CUDA Kernels launch syntax

Triple chevron launch syntax <<< >>> contains
“**kernel launch parameters**”

vecAddGPU<<< **1, 1024** >>>(N, a, b, c)

1° parameter defines the
number of **blocks** to use

2° parameter defines the
number of **threads per block**

```
void vecAddGPU (int N, const float *A,  
               const float *B, float *C)  
{  
    int i = threadIdx.x;  
    if ( i < N) c[i] = a[i] + b[i];  
}  
...  
// call vecAddGPU on N elements  
vecAddGPU<<<1, N>>>( N, a, b, c );
```

CUDA Kernels launch syntax

Insert calls to CUDA kernels using the execution configuration syntax:

```
kernelCUDA<<<numBlocks,numThreads>>>( ... )
```

specifying the thread/block hierarchy you want to apply:

- **numBlocks**: specify grid size in terms of thread blocks along each dimension
- **numThreads**: specify the block size in terms of threads along each dimension

```
dim3 numThreads(32);  
dim3 numBlocks( ( N + numThreads.x - 1 ) / numThreads.x );  
gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev );
```

```
type(dim3) :: numBlocks, numThreads  
numThreads = dim3( 32, 1, 1 )  
numBlocks = dim3( (N + numThreads%x - 1) / numThreads%x, 1, 1 )  
call gpuVectAdd<<<numBlocks,numThreads>>>( N, u_dev, v_dev, z_dev )
```


CUDA Kernels launch syntax

If we want to use a 2D set of threads, then

`blockDim.x`, `blockDim.y` give the dimensions, and
`threadIdx.x`, `threadIdx.y` give the thread indices

and to launch the kernel we would use something like

```
dim3 nthreads(16,4);  
my_new_kernel<<<nblocks,nthreads>>>(d_x);
```

where `dim3` is a special CUDA datatype with 3 components
`.x`, `.y`, `.z` each initialised to 1.

1D thread ID defined by

```
threadIdx.x +  
threadIdx.y * blockDim.x +  
threadIdx.z * blockDim.x * blockDim.y
```

CUDA Kernels launch syntax

```
void vecAddGPU (int N, const float *A,  
               const float *B, float *C)  
{  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if ( i < N) c[i] = a[i] + b[i];  
}  
  
...  
// call vecAddGPU on N elements  
dim3 threads(32);  
dim3 blocks ( (N + threads.x - 1) / threads.x ); // with extra block for reminders  
vecAddGPU<<< blocks, threads >>>( N, a, b, c);
```

```
for blockIdx.x = 0  
    i = 0 * 32 + threadIdx.x = { 0, 1, 2, ... , 31 }  
for blockIdx.x = 1  
    i = 1 * 32 + threadIdx.x = { 32, 33, 34, ... , 63 }  
for blockIdx.x = 2  
    i = 2 * 32 + threadIdx.x = { 64, 65, 66, ... , 95 }
```

Memory management: Step 1 ALLOCATE

```
double *u_dev, *v_dev, *z_dev;  
cudaMalloc(&u_dev, N * sizeof(double));  
cudaMalloc(&v_dev, N * sizeof(double));  
cudaMalloc(&z_dev, N * sizeof(double));
```

```
cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);  
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);
```

```
dim3 numThreads( 256); // 128-512 are good choices  
dim3 numBlocks( (N + numThreads.x - 1) / numThreads.x );  
gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev );  
cudaMemcpy(z, z_dev, N * sizeof(double), cudaMemcpyDeviceToHost);
```

Memory management: Step 2 COPY TO

```
double *u_dev, *v_dev, *z_dev;  
cudaMalloc(&u_dev, N * sizeof(double));  
cudaMalloc(&v_dev, N * sizeof(double));  
cudaMalloc(&z_dev, N * sizeof(double));
```

```
cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);  
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);
```

```
dim3 numThreads( 256); // 128-512 are good choices  
dim3 numBlocks( (N + numThreads.x - 1) / numThreads.x );  
gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev );  
cudaMemcpy(z, z_dev, N * sizeof(double), cudaMemcpyDeviceToHost);
```

Memory management: Step 2 COPY BACK

```
double *u_dev, *v_dev, *z_dev;  
cudaMalloc(&u_dev, N * sizeof(double));  
cudaMalloc(&v_dev, N * sizeof(double));  
cudaMalloc(&z_dev, N * sizeof(double));  
  
cudaMemcpy(u_dev, u, sizeof(u), cudaMemcpyHostToDevice);  
cudaMemcpy(v_dev, v, sizeof(v), cudaMemcpyHostToDevice);  
  
dim3 numThreads( 256); // 128-512 are good choices  
dim3 numBlocks( (N + numThreads.x - 1) / numThreads.x );  
gpuVectAdd<<<numBlocks, numThreads>>>( N, u_dev, v_dev, z_dev );  
cudaMemcpy(z, z_dev, N * sizeof(double), cudaMemcpyDeviceToHost);
```

CUDA Synchronization

- **Kernels are asynchronous** with respect to the host → the CPU can execute the following line immediately after having launched the kernel (i.e. it does not wait for the kernel to complete)
- **cudaMemcpy is synchronous** with respect to the host → the CPU waits for data exchange between host and device to be completed before executing the next line
- **cudaMemcpyAsync is asynchronous** with respect to the host
- **cudaDeviceSynchronize** for host-device synchronization

__device__ kernels

Kernel called from within a __global__ kernel cannot be called by the host

```
__device__ float calculate_element(float value) {  
    // Implement the specific mathematical operation here  
    return value * value; // Example: Square the value  
}
```

```
// Device code (typically in a .cu file)
```

```
__global__ void myKernel(float* data, int size) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < size) {  
        float result = calculate_element(data[i]); // Call __device__ func  
        // ... perform further operations using 'result' ...  
    }  
}
```

Atomic operations

atomicAdd, atomicMin, atomicMax (target, value)

- The basics atomic operation **in hardware** is something like a read-modify-write operation performed by a single hardware instruction on a memory location address
- Read the old value, calculate a new value, and write the new value to the location
- The hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current atomic operation is complete
- Any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue

CUDA Streams

A stream is a **sequence of commands** (possibly issued by different host threads) that execute in order

Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently.

Key to getting better performance is using multiple streams to overlap things

```
cudaStream_t streams[8];
float      *data[8];

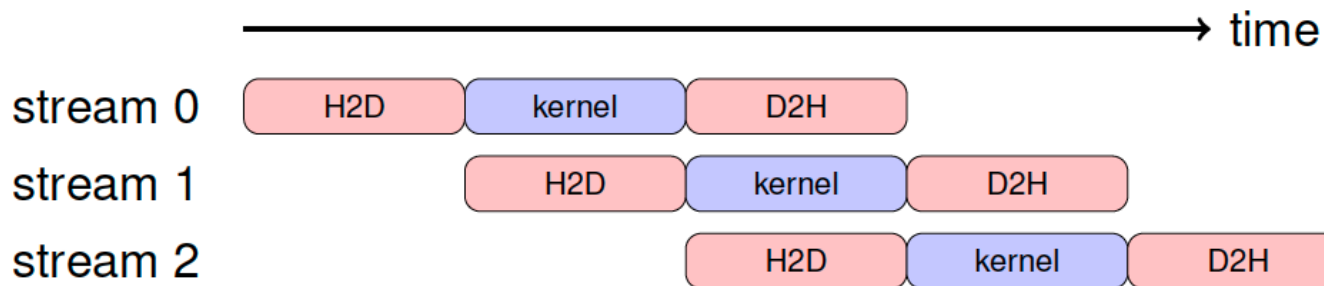
for (int i=0; i<8; i++) {
    cudaStreamCreate(&streams[i]);
    cudaMalloc(&data[i], N * sizeof(float));
}

for (int i=0; i<8; i++) {
    // launch a tiny kernel on default stream
    k<<<1, 1>>>();

    // launch one worker kernel per stream
    kernel<<<1, 64, 0, streams[i]>>>(data[i], N);
}
cudaDeviceSynchronize();
```

CUDA Streams use case...

One important use case for streams is to overlap PCIe transfers with kernel computation for real-time signal processing.

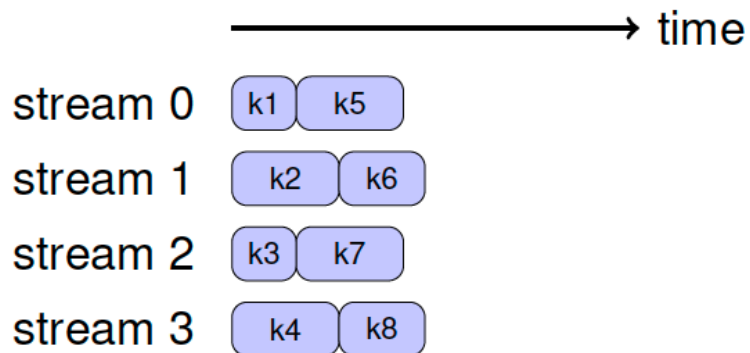


In the best case this gives a factor $3\times$ improvement when the data transfers take as long as the kernel computation

CUDA Streams use case...

A second use case is to overlap the execution of lots of small independent kernels which otherwise would execute sequentially.

Using multiple streams keeps all of the SMs in a big GPU busy.



Some Cuda examples...

deviceQuery

```
Device 1: "NVIDIA GeForce RTX 2080"
CUDA Driver Version / Runtime Version      11.4 / 11.3
CUDA Capability Major/Minor version number: 7.5
Total amount of global memory:              7982 MBytes (8370061312 bytes)
(046) Multiprocessors, (064) CUDA Cores/MP: 2944 CUDA Cores
GPU Max Clock rate:                        1710 Mhz (1.71 GHz)
Memory Clock rate:                         7000 Mhz
Memory Bus Width:                          256-bit
L2 Cache Size:                             4194304 bytes
Maximum Texture Dimension Size (x,y,z)      1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
Total amount of constant memory:             65536 bytes
Total amount of shared memory per block:     49152 bytes
Total shared memory per multiprocessor:      65536 bytes
Total number of registers available per block: 65536
Warp size:                                  32
Maximum number of threads per multiprocessor: 1024
Maximum number of threads per block:         1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch:                     2147483647 bytes
Texture alignment:                         512 bytes
Concurrent copy and kernel execution:       Yes with 3 copy engine(s)
Run time limit on kernels:                 Yes
Integrated GPU sharing Host Memory:         No
Support host page-locked memory mapping:    Yes
Alignment requirement for Surfaces:         Yes
Device has ECC support:                    Disabled
Device supports Unified Addressing (UVA):    Yes
Device supports Managed Memory:             Yes
Device supports Compute Preemption:         Yes
Supports Cooperative Kernel Launch:         Yes
Supports MultiDevice Co-op Kernel Launch:   Yes
Device PCI Domain ID / Bus ID / location ID: 0 / 45 / 0
Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
> Peer access from NVIDIA GeForce RTX 2080 (GPU0) -> NVIDIA GeForce RTX 2080 (GPU1) : Yes
> Peer access from NVIDIA GeForce RTX 2080 (GPU1) -> NVIDIA GeForce RTX 2080 (GPU0) : Yes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA Runtime Version = 11.3, NumDevs = 2
```

<https://github.com/NVIDIA/cuda-samples>

Programming GPUs with OpenMP

GPU Programming environments

- Nvidia's application programming interface: CUDA
 - Only works with Nvidia GPUs
 - Very well documented, many tutorials, low entry level
- AMD ROCm (HIP): Open-source platform for GPU computing
 - Supports both AMD and Nvidia GPUs
 - New development → still work in progress, not that many examples / tutorials yet
- OpenCL: Framework for heterogeneous platforms
 - CPUs, GPUs, FPGAs, DSPs, etc.
 - Maintained by the Khronos group, based on C99 and C++11
- SYCL: Single source C++ heterogeneous programming platform, built on OpenCL
 - Will be supported by Intel GPUS

GPU Programming environments

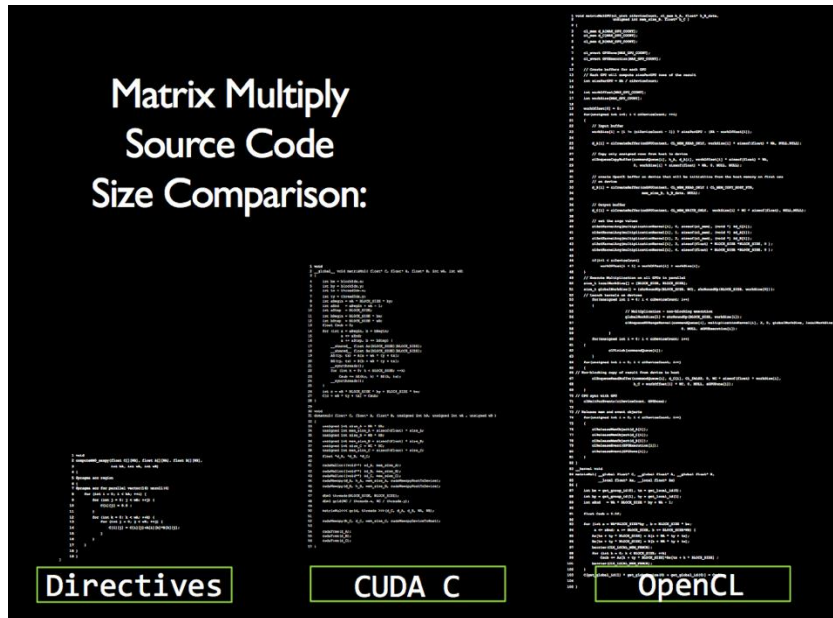
- **NVIDIA OpenACC**
 - Directive-based programming standard
 - Only works with Nvidia GPUs
 - Very well documented, many tutorials, low entry level
- **OpenMP**
 - Directive-based programming standard
 - Supports both AMD and Nvidia GPUs...but also FPGAs etc...
 - Well documented but with less functionalities respect to OpenACC
 - HIGH PORTABILITY!!!

Why a directive-based programming standard

- It is an open standard
- It is a general programming model (not just for GPUs)
- It doesn't require to learn a highly specialized language, supporting Fortran, C and C++
- It doesn't require to develop multiple versions of the code (in principle)
- Simple...almost..

Possible performance sacrifice

10-30% slower than optimized CUDA code



...and of course we need a compiler...

- NVIDIA

```
nvc -mp=gpu -arch=sm_70 -Xcompiler -mno-float128 -std=c++11 -lcudart -lcuda  
-o test test.c
```

- gcc

```
gcc-10 -fopenmp -foffload=-lm -ffast-math -fcf-protection=none  
-fno-stack-protector -foffload=nvptx-none -foffload=misa=sm_35  
-lcudart -lcuda -o test test.c
```

- llvm (including new Intel compiler...)

```
clang++ -fopenmp -fopenmp-targets=nvptx64
```

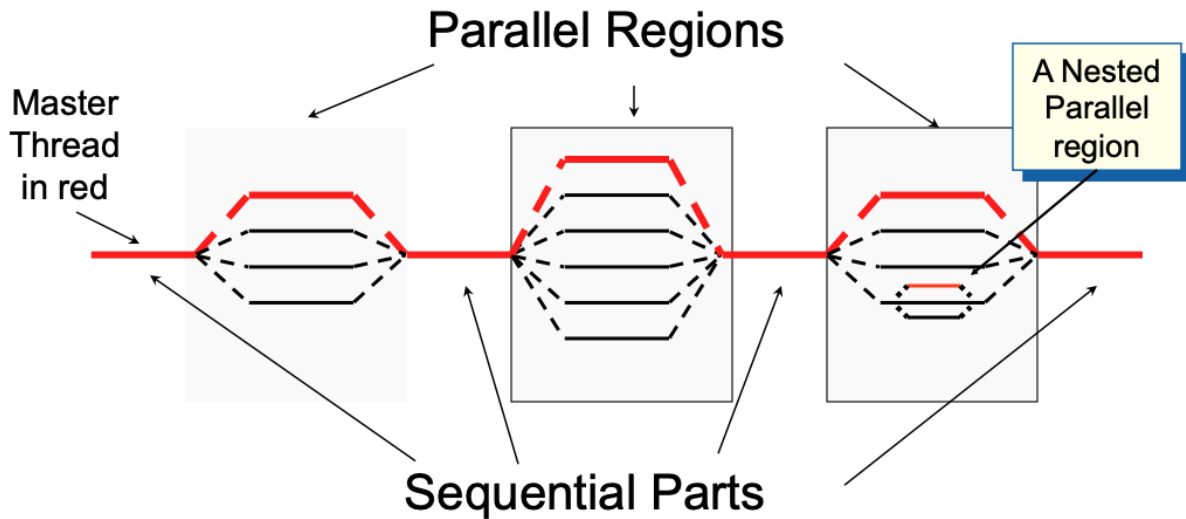
- IBM

```
xlc -qsmp -qoffload myopenmpprogram.c -o myopenmpprogram
```

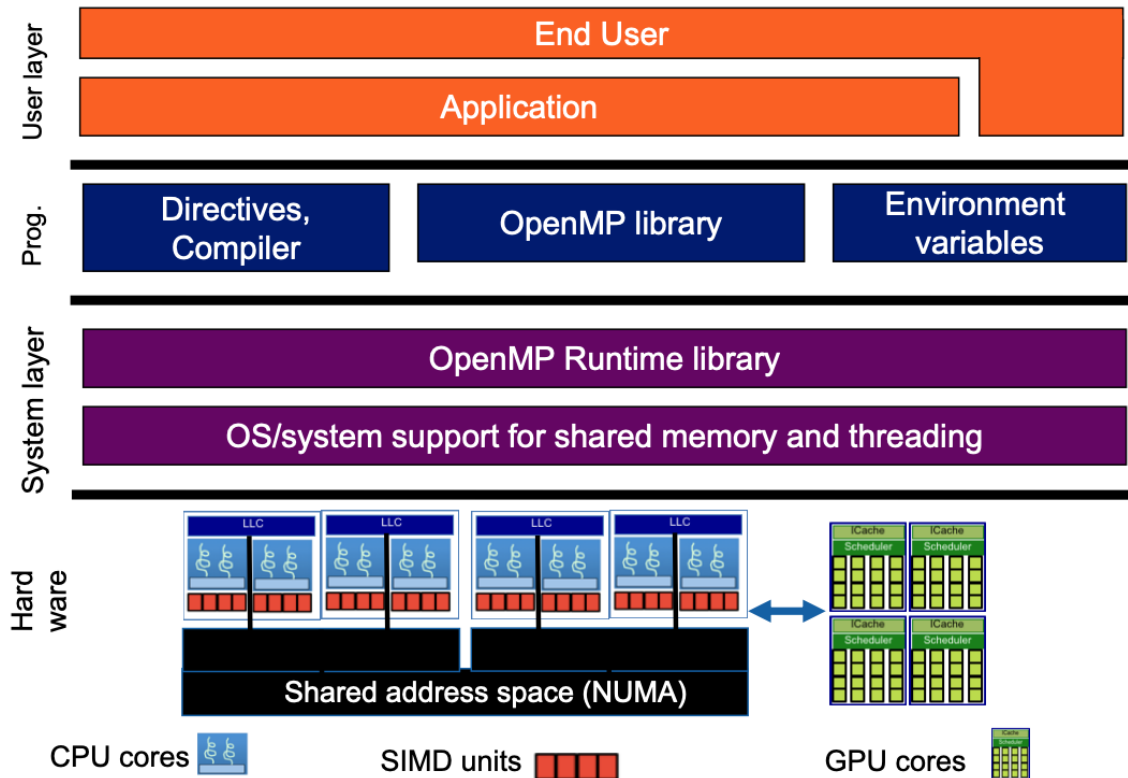
OpenMP programming model recap...

Fork-Join Parallelism:

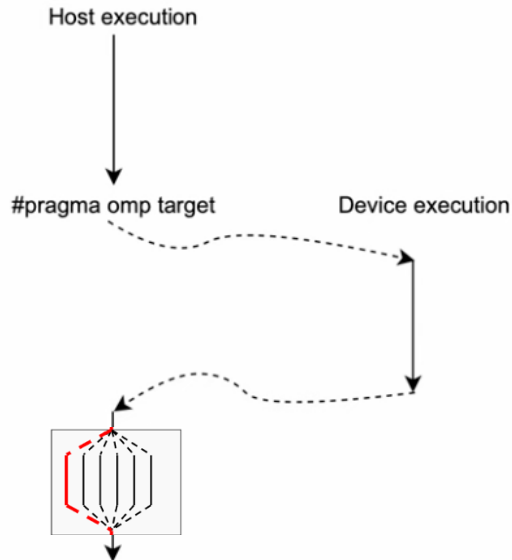
- ◆ Master thread spawns a team of threads as needed.
- ◆ Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



OpenMP programming model recap...



OpenMP programming model with GPUs ...



- Directive has two roles:
 - transfer execution to the device
 - transfer data to/from the device
- Implicit data transfers:
 - firstprivate: scalars
 - copied to and from (map(tofrom)): stack arrays, complete structs
 - zero-length arrays: pointers
 - (pointer itself is firstprivate if in a map clause)

OpenMP in action

```
void saxpy() {  
    float a, x[SZ], y[SZ];  
    // left out initialization  
    double t = 0.0;  
    double tb, te;  
    tb = omp_get_wtime();  
    #pragma omp parallel for firstprivate(a)  
    for (int i = 0; i < SZ; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
    te = omp_get_wtime();  
    t = te - tb;  
    printf("Time of kernel: %lf\n", t);  
}
```

Timing code (not needed, just to have a bit more code to show 😊)

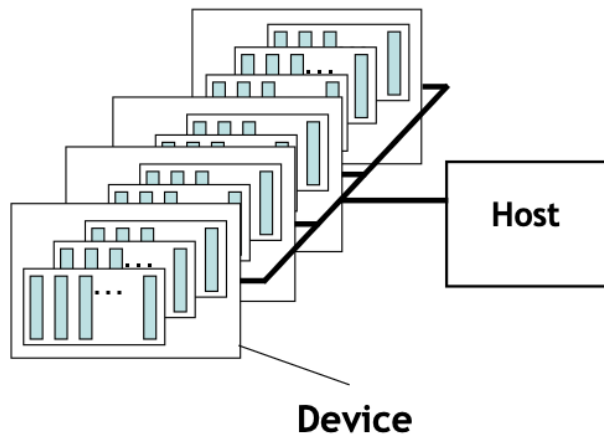
This is the code we want to execute on a target device (i.e., GPU)

Timing code (not needed, just to have a bit more code to show 😊)

Don't do this at home!
Use a BLAS library for this!

OpenMP execution model for devices

- OpenMP uses a host/device model
 - The *host* is where the initial thread of the program begins execution
 - Zero or more *devices* are connected to the host
 - Device-memory address space is distinct from host-memory address space

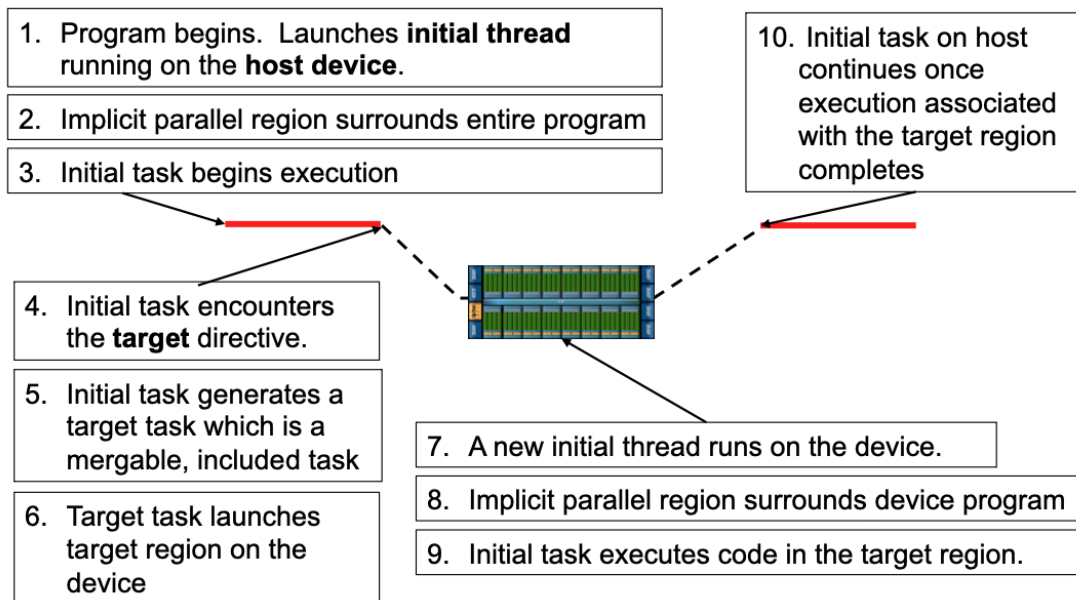


```
#include <omp.h>
#include <stdio.h>
int main()
{
    printf("There are %d devices\n",
          omp_get_num_devices());
}
```

OpenMP execution model for devices

The target construct offloads execution to a device.

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



OpenMP in action

Example: saxpy

```
void saxpy() {  
    float a, x[SZ], y[SZ];  
    double t = 0.0;  
    double tb, te;  
    tb = omp_get_wtime();  
    #pragma omp target "map(tofrom:y[0:SZ])"  
    for (int i = 0; i < SZ; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
    te = omp_get_wtime();  
    t = te - tb;  
    printf("Time of kernel: %lf\n", t);  
}
```

clang/LLVM: clang -fopenmp -fopenmp-targets=<target triple>
GNU: gcc -fopenmp
AMD ROCm: clang -fopenmp -offload-arch=gfx908
NVIDIA: nvcc -mp=gpu -gpu=cc80

The compiler identifies variables that are used in the target region.

All accessed arrays are copied from host to device and back

a
x[0:SZ]
y[0:SZ]

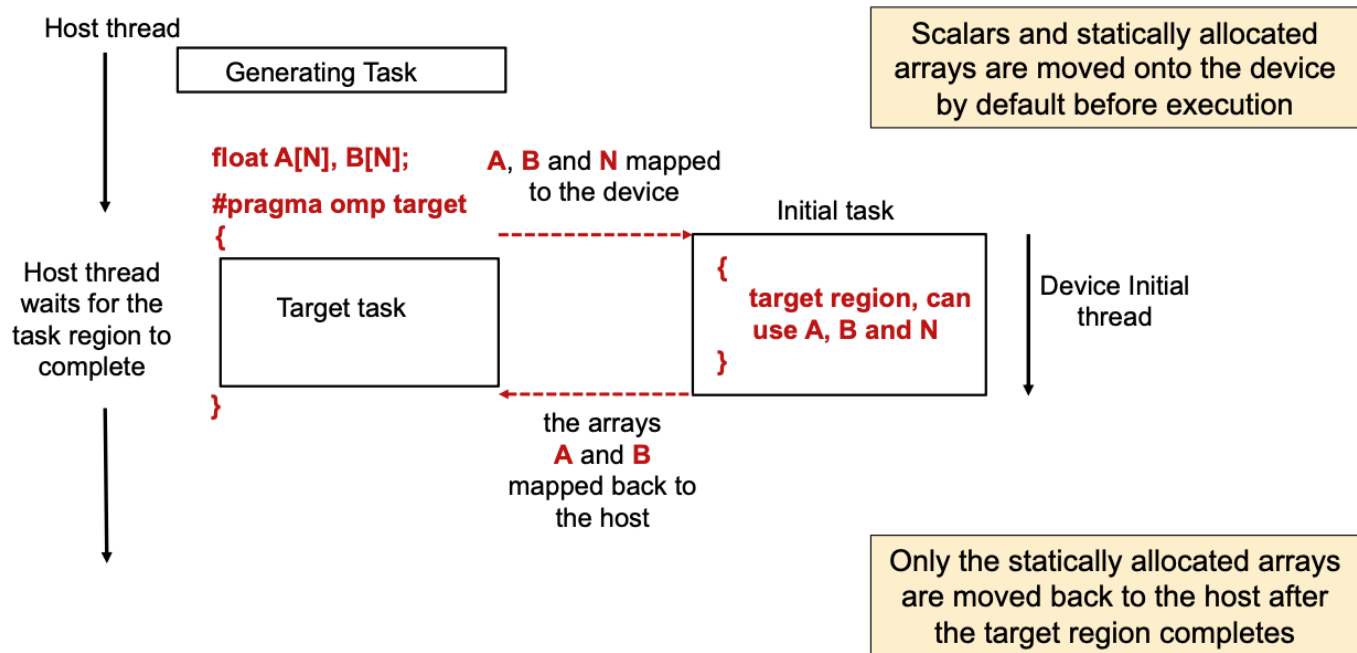
Presence check: only transfer if not yet allocated on the device.

x[0:SZ]
y[0:SZ]

Copying x back is not necessary: it was not changed.

Running code on the GPU:

The target construct and default data movement



The ‘target data’ environment

- **Remember:** there are distinct memory spaces on host and device.
- OpenMP uses a combination of *implicit* and *explicit* data movement.
- Data may move between the host and the device in well defined places:
 - Firstly, at the beginning and end of a **target** region:

```
#pragma omp target
{
    // Data may move from host to device here

    ...

}
// and from device to host here
```

Default Data Mapping: implicit movement with a target region

- Scalar variables:
 - Examples:
 - `int N; double x;`
 - OpenMP implicitly maps scalar variables as **firstprivate**
 - A new value per work-item is initialized with the original value (in OpenCL nomenclature, the firstprivate goes in private memory).
 - The variable is not copied back to the host at the end of the target region.
 - In CUDA/OpenCL parlance, a firstprivate scalar can be launched as a parameter to a kernel function without the overhead of setting up a variable in device memory.

Default Data Mapping: implicit movement with a target region

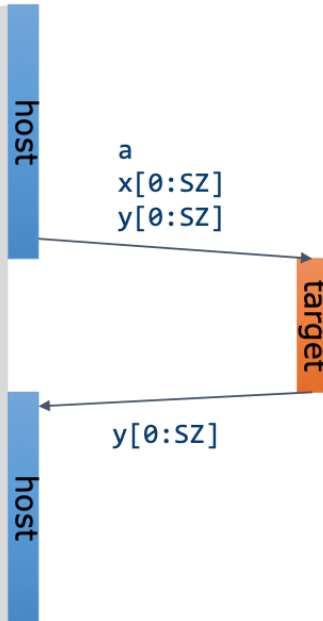
- Non-scalar variables:
 - Must have a ***complete type***.
 - Example: fixed sized (stack) array:
 - `double A[1000];`
 - Copied to the device at the start of the **target** region, *and* copied back at the end. In OpenCL nomenclature, these are placed in device global memory.
 - A new memory object is created in the target region and initialized with the original data, but it is shared between threads on the device. Data is copied back to the host at the end of the target region.
 - OpenMP calls this mapping **tofrom**

Default Data Mapping: implicit movement with a target region

- Pointers are implicitly copied, but **not** the data they point to:
 - *Example: arrays allocated on the heap*
 - `double *A = malloc(sizeof(double)*1000);`
 - The pointer **value** will be mapped (i.e. the address stored in A).
 - But the data it points to ***will not*** be mapped by default.
 - We'll show you how to map a pointer's data shortly.

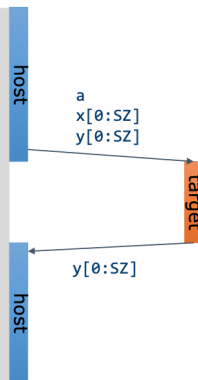
Default Data Mapping: implicit movement with a target region

```
void saxpy() {
    double a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target map(to:x[0:SZ]) \
                      map(tofrom:y[0:SZ])
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```



...what happens in the GPU?

```
void saxpy() {  
    double a, x[SZ], y[SZ];  
    double t = 0.0;  
    double tb, te;  
    tb = omp_get_wtime();  
    #pragma omp target map(to:x[0:SZ]) \  
        map(tofrom:y[0:SZ])  
    for (int i = 0; i < SZ; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
    te = omp_get_wtime();  
    t = te - tb;  
    printf("Time of kernel: %lf\n", t);  
}
```



With just `#pragma omp target`, the default behavior is to use one **team**

All threads in the team will redundantly execute the entire loop (i.e., each thread will process all iterations from 0 to SZ).

There is no parallelization of the loop across threads.

Default Data Mapping: explicit movement with a target region

Example: saxpy

```
void saxpy(float a, float* x, float* y,
          int sz) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
    #pragma omp target map(to:x[0:sz]) \
                      map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

The compiler cannot determine the size of memory behind the pointer.

Observation when running this: the loop is a sequential loop, and the capabilities of the GPU are not really used! ☹️

target

y[0:sz]

host

Programmers have to help the compiler with the size of the data transfer needed.

Default Data Mapping: explicit movement with a target region

- For mapping data arrays/pointers you must use array section notation:
 - In C, notation is **pointer[lower-bound : length]**
 - **map(to: a[0:N])**
 - Starting from the element at a[0], copy N elements to the target data region
 - **Be careful!**
 - Common to misremember this as begin : end, but it is **length**
 - Without the map, OpenMP defines that the pointer itself (**a**) is mapped as a zero-length array section.
 - Zero length arrays: A[:0]

Default Data Mapping: explicit movement with a target region

```
int i, a[N], b[N], c[N];  
#pragma omp target map(to:a,b) map(tofrom:c)
```

Data movement
defined from the
host perspective.

- The various forms of the map clause
 - **map(to:list)**: On entering the region, variables in the list are initialized on the device using the original values from the host (host to device copy).
 - **map(from:list)**: At the end of the target region, the values from variables in the list are copied into the original variables on the host (device to host copy). On entering the region, the initial value of the variables on the device is not initialized.
 - **map(tofrom:list)**: the effect of both a map-to and a map-from (host to device copy at start of region, device to host copy at end).
 - **map(alloc:list)**: On entering the region, data is allocated and uninitialized on the device.
 - **map(list)**: equivalent to **map(tofrom:list)**.

Creating Parallelism on the Target Device

- The **target construct transfers** the control flow to the target device
 - Transfer of control is sequential and synchronous
 - This is intentional!
- OpenMP separates offload and parallelism
 - Programmers need to explicitly create parallel regions on the target device
 - In theory, this can be combined with any OpenMP construct
 - In practice, there is only a useful subset of OpenMP features for a target device such as a GPU, e.g., no I/O, limited use of base language features.

Creating Parallelism on the Target Device

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

host

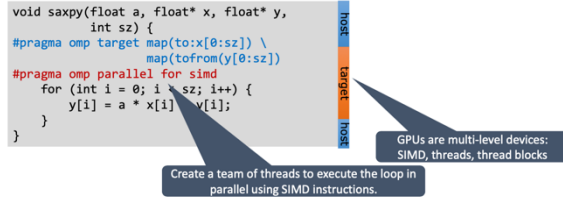
target

host

Create a team of threads to execute the loop in parallel using SIMD instructions.

GPUs are multi-level devices:
SIMD, threads, thread blocks

Creating Parallelism on the Target Device



Step 1: Offload:

`#pragma omp target` sends the loop to the GPU.

Step 2: Thread-Level Parallelism:

`#pragma omp parallel for` creates threads on the GPU.

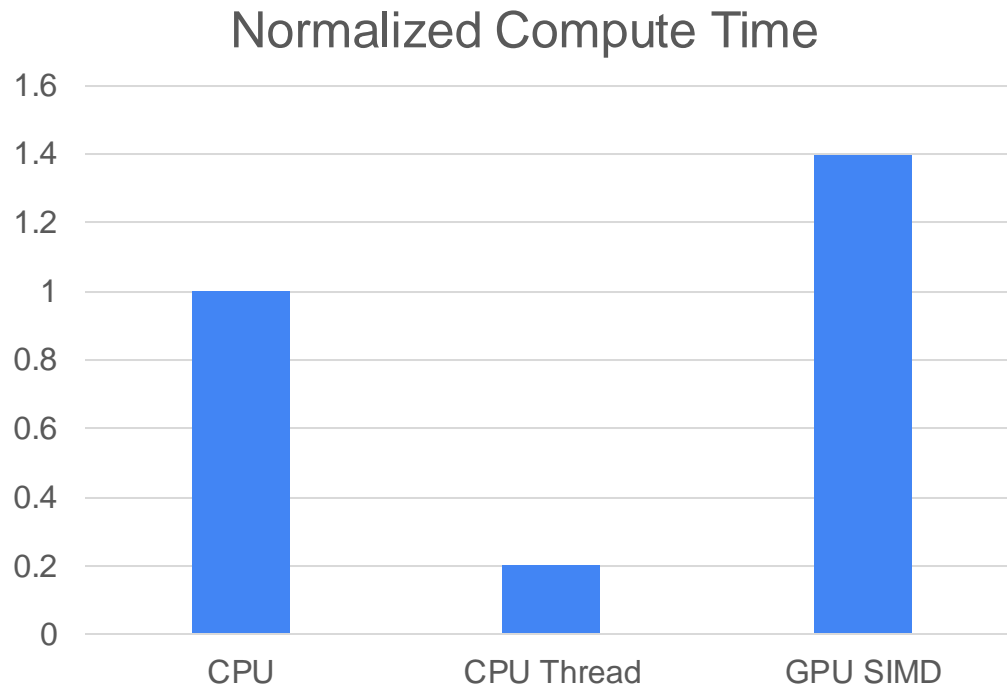
If the runtime decides to use 128 threads, for example, the loop's 1,000 iterations are split across these threads.

Step 3: `simd` Vectorization

Each thread processes multiple iterations at once using SIMD vectorization.

~~For instance, if each SIMD unit handles 4 iterations simultaneously, a thread might execute iterations [0, 1, 2, 3], [4, 5, 6, 7], and so on.~~

Execution time...too bad



Multi-level Parallelism

Tile the loop into an outer loop and an inner loop.

Create `nteams` “teams”.

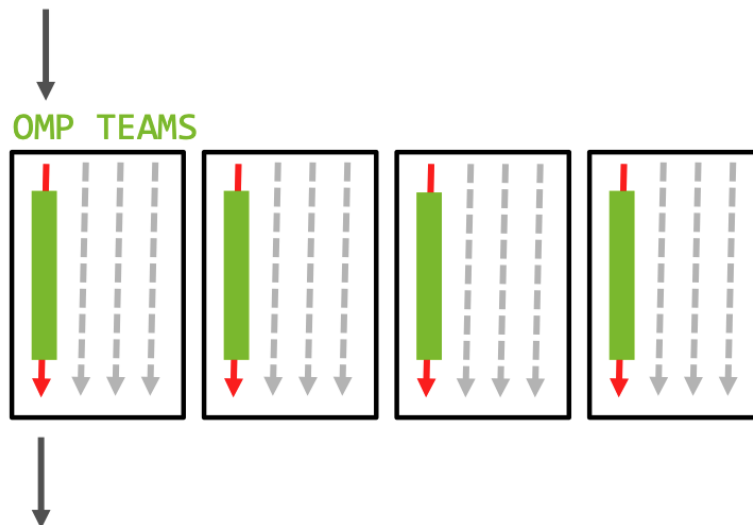
```
void saxpy(float a, float* x, float* y, int sz) {  
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(nteams)  
    {  
        int bs = n / omp_get_num_teams(); // n assumed to be multiple of #teams  
        #pragma omp distribute  
        for (int i = 0; i < sz; i += bs) {  
            #pragma omp parallel for simd firstprivate(i,bs)  
            for (int ii = i; ii < i + bs; ii++) {  
                y[ii] = a * x[ii] + y[ii];  
            }  
        }  
    }  
}
```

OpenMP Teams

teams directive

To better utilize the GPU resources, use many thread teams via the TEAMS directive.

- Spawns 1 or more thread teams with the same number of threads
- Execution continues on the master threads of each team (redundantly)
- No synchronization between teams



Multi-level Parallelism

Tile the loop into an outer loop and an inner loop.

Create `ntteams` “`teams`”.

Assign the outer loop to “`teams`”.

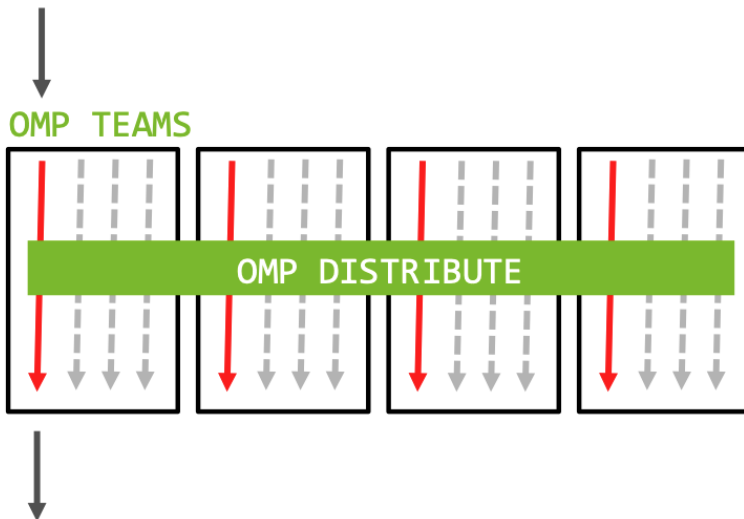
```
void saxpy(float a, float* x, float* y, int sz) {  
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(ntteams)  
    {  
        int bs = n / omp_get_num_teams(); // n assumed to be multiple of #teams  
        #pragma omp distribute  
        for (int i = 0; i < sz; i += bs) {  
            #pragma omp parallel for simd firstprivate(i,bs)  
            for (int ii = i; ii < i + bs; ii++) {  
                y[ii] = a * x[ii] + y[ii];  
            }  
        }  
    }  
}
```

OpenMP Teams

distribute directive

The distribute clause splits the loop iterations into chunks and assigns each chunk to a team.

- A team corresponds to a group of threads that work together (similar to a **CUDA block**).
- Each team works independently on its assigned chunk of the loop.
- Enables **team-level parallelism**, distributing work across multiple Streaming Multiprocessors (SMs).



Multi-level Parallelism

Tile the loop into an outer loop and an inner loop.

Create `ntteams` “`teams`”.

Assign the outer loop to “`teams`”.

Assign the inner loop to the “`threads`”.

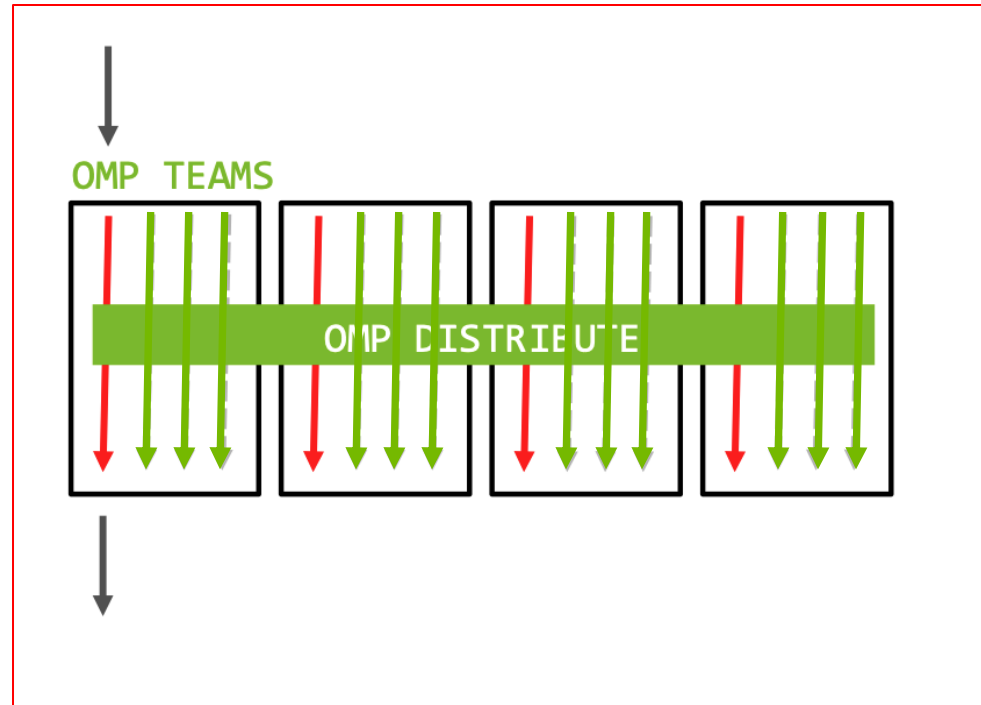
```
void saxpy(float a, float* x, float* y, int sz) {  
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(ntteams)  
    {  
        int bs = n / omp_get_num_teams(); // n assumed to be multiple of #teams  
        #pragma omp distribute  
        for (int i = 0; i < sz; i += bs) {  
            #pragma omp parallel for simd firstprivate(i,bs)  
            for (int ii = i; ii < i + bs; ii++) {  
                y[ii] = a * x[ii] + y[ii];  
            }  
        }  
    }  
}
```

OpenMP threads

`parallel for` directive

The `parallel for` clause takes the chunk of iterations assigned to each team (by `distribute`) and further parallelizes it across the threads within that team.

- Threads in the team work on individual loop iterations, dividing the chunk of work among themselves.
- Threads within a team map to CUDA threads (within a CUDA block).
- Enables thread-level parallelism, distributing the workload among the threads of a team.



Multi-level Parallelism

For convenience, OpenMP defines composite constructs to implement the required code transformations

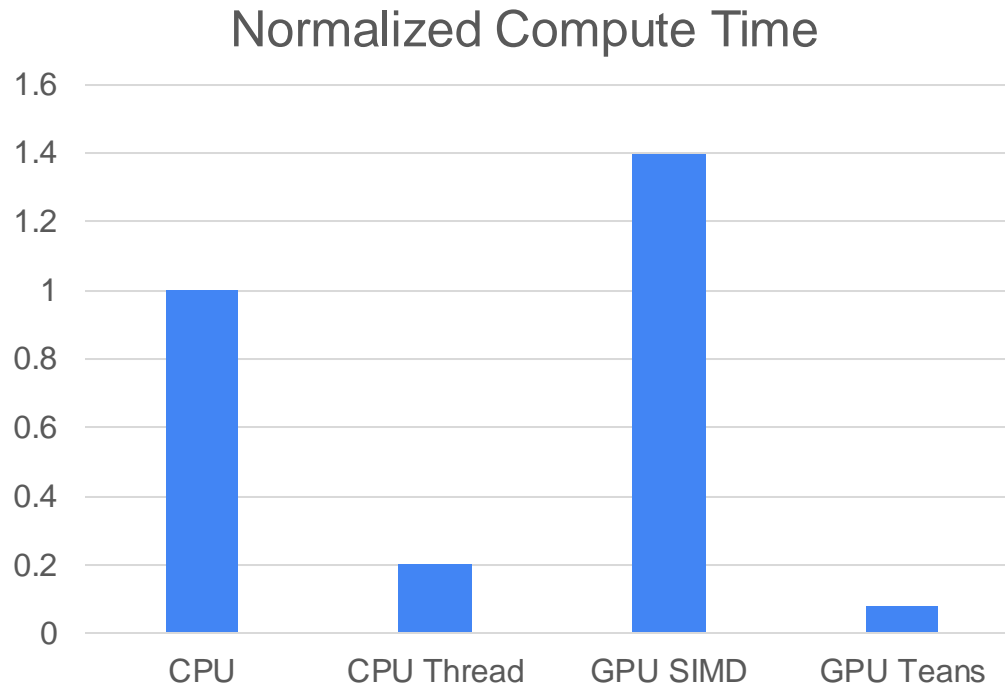
```
void saxpy(float a, float* x, float* y, int sz) {  
    #pragma omp target teams distribute parallel for simd \  
        num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])  
    for (int i = 0; i < sz; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

Summing up multi-level Parallelism

```
void saxpy(float a, float* x, float* y, int sz) {  
    #pragma omp target teams distribute parallel for simd \  
        num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])  
    for (int i = 0; i < sz; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

1. **Teams**: create multiple independent teams of threads
2. **Distribute**: Splits the loop into chunks, assigning chunks to teams. Each team works **independently** on a specific subset of the loop iterations.
3. **parallel for**: Splits the team's assigned chunk of iterations among threads within the team.
4. Hierarchy:
 - **distribute** controls **team-level parallelism**.
 - **parallel for** controls **thread-level parallelism** within each team.

Execution time...not too bad



Improving Parallelism with distribute parallel for

Create enough teams to **fully utilize all Streaming Multiprocessors** on the GPU.

Use the `num_teams` clause to explicitly control the number of teams.

For example, on an NVIDIA A100 with 108 SMs:

- Set `num_teams` to at least 108 to assign one team per SM.
- You can go higher (e.g., 512 teams) to **saturate** the GPU, as multiple teams can execute on the same SM.

```
#pragma omp target teams distribute parallel for num_teams(128)
for (int i = 0; i < N; i++) {
    // Your compute-intensive kernel here
}
```


Improving Parallelism with distribute parallel for

Maximize the number of threads within each team to fully utilize the GPU cores.

Use the `thread_limit` clause to control the number of threads per team.

For example, on an NVIDIA A100:

- Use a multiple of 32 (warp size) for `thread_limit` to ensure warp efficiency.
- Common values are 256 or 512 threads per team.
- Avoid exceeding **1,024 threads per team**, as this is the hardware limit.

```
#pragma omp target teams distribute parallel for num_teams(128) thread_limit(256)
for (int i = 0; i < N; i++) {
    // Your compute-intensive kernel here
}
```

Improving Parallelism with distribute parallel for

Achieve a good balance between the work assigned to teams (**distribute**) and the work assigned to threads (**parallel for**).

Ensure the **chunk size** assigned to each team is neither too large nor too small:

- If chunks are **too large**, some teams may idle while others are still working.
- If chunks are **too small**, the overhead of team creation may reduce performance.

```
#pragma omp target teams distribute parallel for schedule(static, chunk_size)
for (int i = 0; i < N; i++) {
    // Your compute-intensive kernel here
}
```

Improving Parallelism: scheduling

Most OpenMP compilers will apply a **static** schedule to workshared loops, assigning iterations in $(N/\text{num_threads})$ chunks.

- Each thread will execute contiguous loop iterations, which is very cache & SIMD friendly
- This is great on CPUs, but bad on GPUs

The `SCHEDULE()` clause can be used to adjust how loop iterations are scheduled.

Static Scheduling:

- Assigns fixed-size chunks to teams and threads.
- Good for balanced workloads.

Dynamic Scheduling:

- Dynamically assigns chunks to teams and threads.
- Useful for unbalanced workloads.

Improving Parallelism: scheduling

Most OpenMP compilers will apply a **static** schedule to workshared loops, assigning iterations in $(N/\text{num_threads})$ chunks.

- Each thread will execute contiguous loop iterations, which is very cache & SIMD friendly
- This is great on CPUs, but bad on GPUs

The `SCHEDULE()` clause can be used to adjust how loop iterations are scheduled.

`!$OMP PARALLEL FOR SCHEDULE(STATIC)`

Thread 0  0 - $(n/2-1)$

Thread 1  $(n/2) - n-1$

Cache and vector friendly

`!$OMP PARALLEL FOR SCHEDULE(STATIC,1)*`

Thread 0  0, 2, 4, ..., $n-2$

Thread 1  1, 3, 5, ..., $n-1$

Memory coalescing friendly

*There's no reason a compiler couldn't do this for you.

Improving Parallelism with distribute parallel for

Achieve a good balance between the work assigned to teams (**distribute**) and the work assigned to threads (**parallel for**).

Ensure the **chunk size** assigned to each team is neither too large nor too small:

- If chunks are **too large**, some teams may idle while others are still working.
- If chunks are **too small**, the overhead of team creation may reduce performance.

```
#pragma omp target teams distribute parallel for num_teams(128) thread_limit(256) schedule(static, 16)
for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
```

- num_teams(128): Launches 128 teams, targeting SMs on the GPU.
- thread_limit(256): Each team uses 256 threads (8 warps).
- schedule(static, 16): **Distributes chunks of 16 iterations to threads in each team, ensuring load balance.**



Why?

Improving Parallelism: scheduling

| Schedule Type | Behavior | Use Case | Overhead |
|---------------|--|--|----------|
| static | Fixed-size chunks, assigned in advance. | Balanced workloads. | Low |
| dynamic | Chunks assigned dynamically as threads finish. | Unbalanced workloads. | Medium |
| guided | Exponentially decreasing chunk sizes. | Unbalanced workloads with decreasing cost. | Medium |
| auto | Determined by runtime. | When unsure of the best strategy. | Varies |
| runtime | Controlled by OMP_SCHEDULE. | Experimenting with schedules. | Varies |

Improving Parallelism: scheduling

| Schedule | Description |
|----------|---|
| static | <p>The iterations of the loop are divided into chunks of fixed size (as specified in the <code>chunk_size</code> argument) and assigned to threads in a round-robin fashion.</p> <p>If no <code>chunk_size</code> is specified, iterations are divided into chunks of approximately equal size.</p> |
| dynamic | <p>Iterations are divided into chunks of <code>chunk_size</code>, and chunks are assigned to threads dynamically as threads finish their previous chunks. Threads request new chunks from a queue, which introduces some scheduling overhead.</p> |
| guided | <p>Iterations are divided into chunks, but the size of each chunk decreases exponentially as the computation progresses.</p> <ul style="list-style-type: none">• Chunks start large and gradually become smaller, aiming to reduce overhead while still balancing the workload. |

Multi-level Parallelism final considerations...

```
void saxpy(float a, float* x, float* y, int sz) {  
    #pragma omp target teams distribute parallel for simd \  
    num_teams(128) thread_limit(256) schedule(static, 16) \  
    map(to:x[0:sz]) map(tofrom:y[0:sz])  
    for (int i = 0; i < sz; i++) {  
        y[i] = a * x[i] + y[i];  
    }  
}
```

Minimize Overhead

1. Avoid Overloading Teams: Don't assign too many iterations to each team. This can create bottlenecks if threads within a team cannot efficiently distribute the work.
2. Coalesce Memory Accesses: Ensure that threads within a team access contiguous memory locations to maximize memory bandwidth. For example, structure data so that threads process adjacent elements of an array.
3. Avoid Warp Divergence: Minimize branching (e.g., **if statements**) within the loop to ensure all threads in a warp execute the same instructions.

THANK YOU.

Now, it's your time
QUESTIONS?

GPU programming with OpenMP