INAF

ISTITUTO NAZIONALE DI ASTROFISICA

GPU Programming

Lecture 3

UniTS Advance HPC Course 2024/2025

Agenda

- OpenMP for GPUs
- Advanced Scheduling
- Data Movement
- Modular programming
- Examples and Exercises

But not everything today….

Multi-level Parallelism

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

Create **`nteams`** “`teams`”.

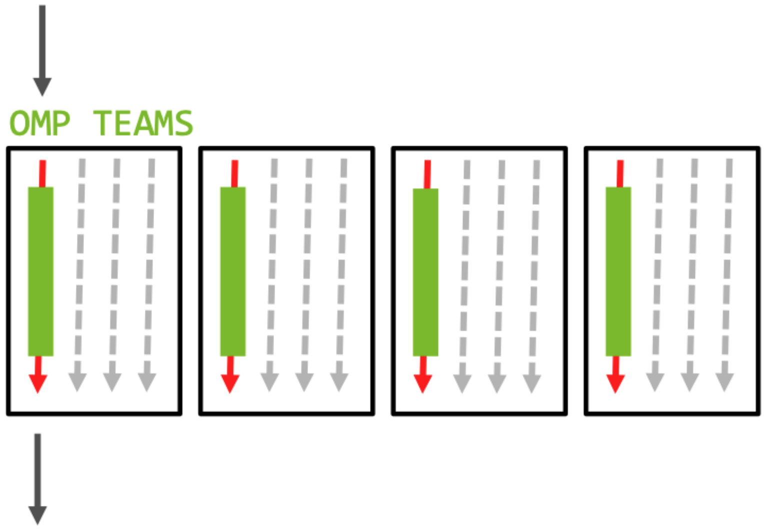
```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(nteams)
    {
        int bs = n / omp_get_num_teams();   // n assumed to be multiple of #teams
        #pragma omp distribute
        for (int i = 0; i < sz; i += bs) {
 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

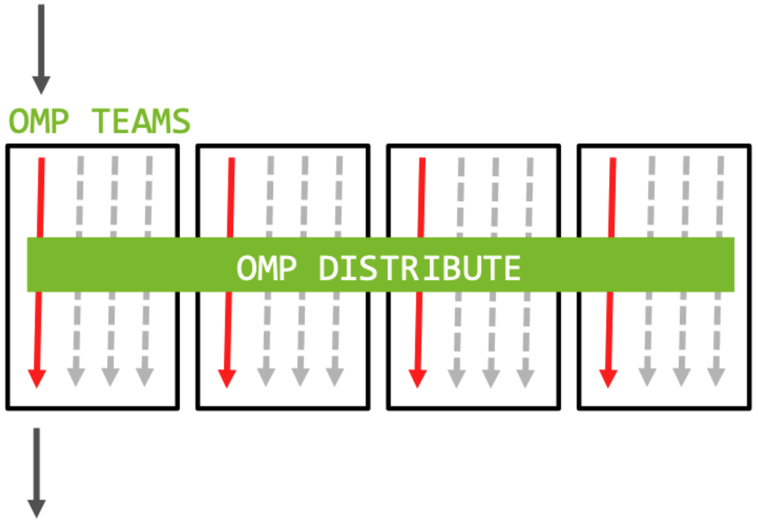
distribute directive

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 **`nteams`** "`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(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 threads

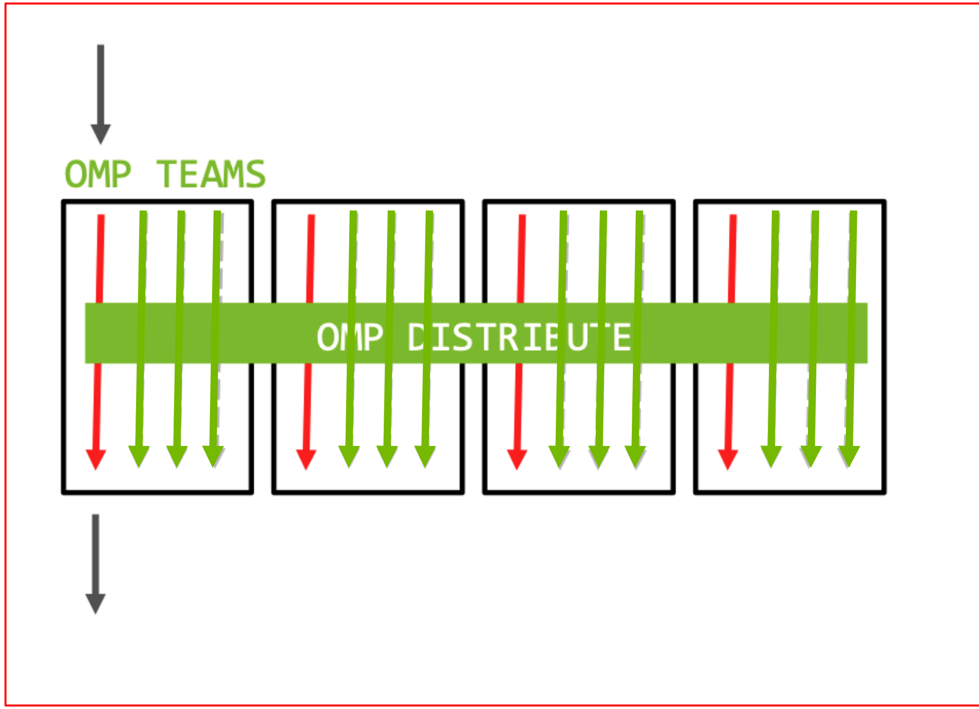
`parallel for` directive

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];
    }
}
```

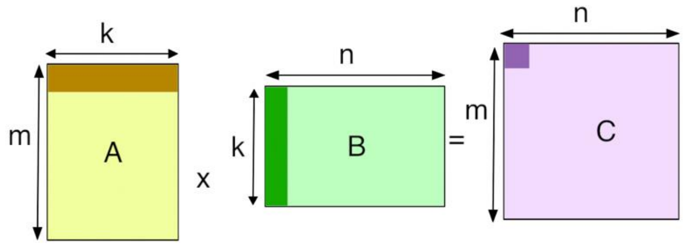
Summing up multi-level Parallelism

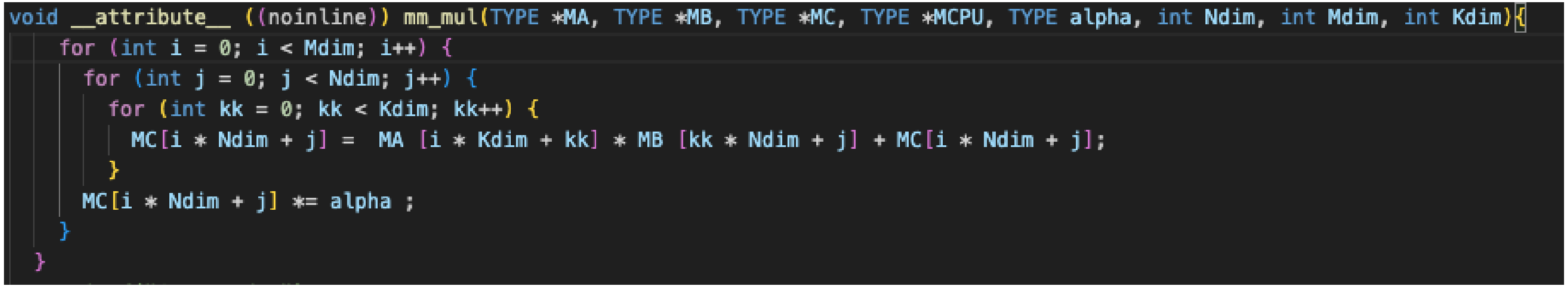
```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams distribute parallel for simd \
            num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

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.

Creating Parallelism on the Target Device

DGEMM $C = \alpha A \times B$





```
void __attribute__ ((noinline)) mm_mul(TYPE *MA, TYPE *MB, TYPE *MC, TYPE *MCPU, TYPE alpha, int Ndim, int Mdim, int Kdim){
    for (int i = 0; i < Mdim; i++) {
      for (int j = 0; j < Ndim; j++) {
        for (int kk = 0; kk < Kdim; kk++) {
          MC[i * Ndim + j] =  MA [i * Kdim + kk] * MB [kk * Ndim + j] + MC[i * Ndim + j];
        }
      MC[i * Ndim + j] *= alpha ;
    }
  }
```

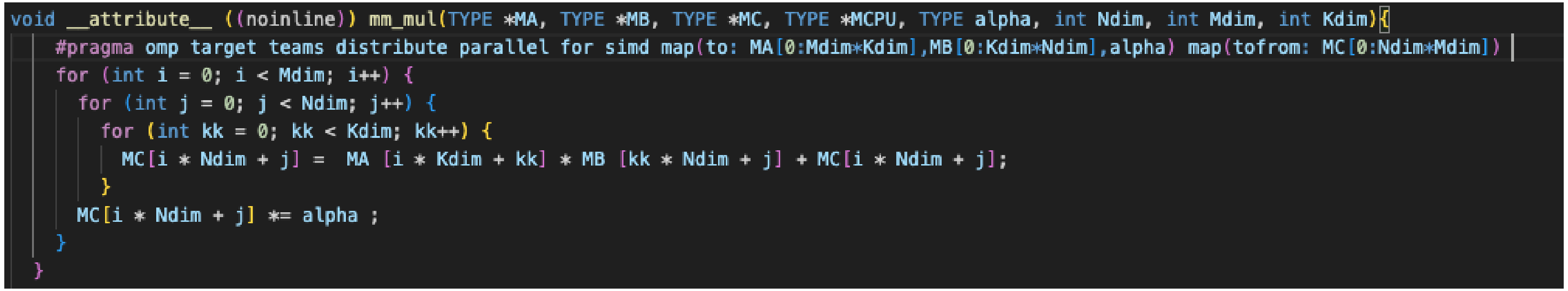
Creating Parallelism on the Target Device

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.



```
void __attribute__ ((noinline)) mm_mul(TYPE *MA, TYPE *MB, TYPE *MC, TYPE *MCPU, TYPE alpha, int Ndim, int Mdim, int Kdim){
  #pragma omp target teams distribute parallel for simd map(to: MA[0:Mdim*Kdim],MB[0:Kdim*Ndim],alpha) map(tofrom: MC[0:Ndim*Mdim])
  for (int i = 0; i < Mdim; i++) {
    for (int j = 0; j < Ndim; j++) {
      for (int kk = 0; kk < Kdim; kk++) {
        MC[i * Ndim + j] =  MA [i * Kdim + kk] * MB [kk * Ndim + j] + MC[i * Ndim + j];
      }
    MC[i * Ndim + j] *= alpha ;
  }
}
```

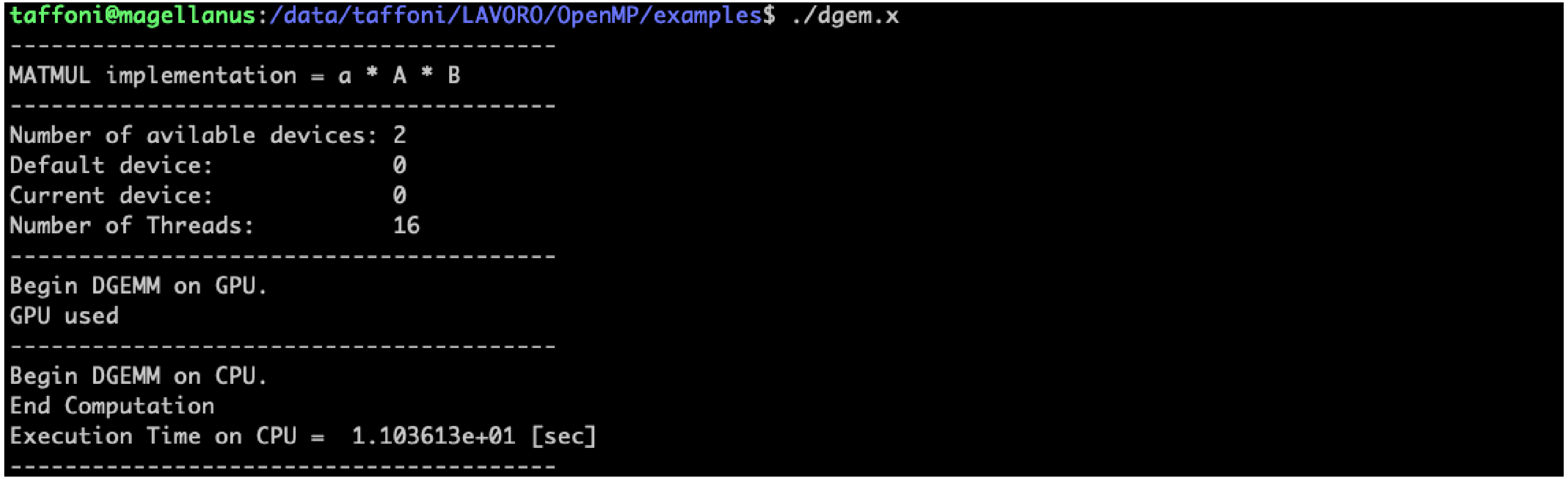
Creating Parallelism on the Target Device

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.



```
taffoni@magellanus:/data/taffoni/LAVORO/OpenMP/examples$ ./dgem.x
----------------------------------------
MATMUL implementation = a * A * B
----------------------------------------
Number of avilable devices: 2
Default device:             0
Current device:             0
Number of Threads:          16
----------------------------------------
Begin DGEMM on GPU.
GPU used
----------------------------------------
Begin DGEMM on CPU.
End Computation
Execution Time on CPU =  1.103613e+01 [sec]
----------------------------------------
```

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.

Step 4: `collapse()` **Merge loops**
Enables to merge all the iterations of several associated loops into a single large iteration loop.

```
#pragma omp target teams dsistribute parallel for simd collapse(3)
for (int i=0;i<nx;i++)
    for (int i=0;i<nx;i++)
        for (int i=0;i<nx;i++)
            ...
```

Creating Parallelism on the Target Device

Step 1: Offload:
`#pragma omp target` sends the loop to the GPU.

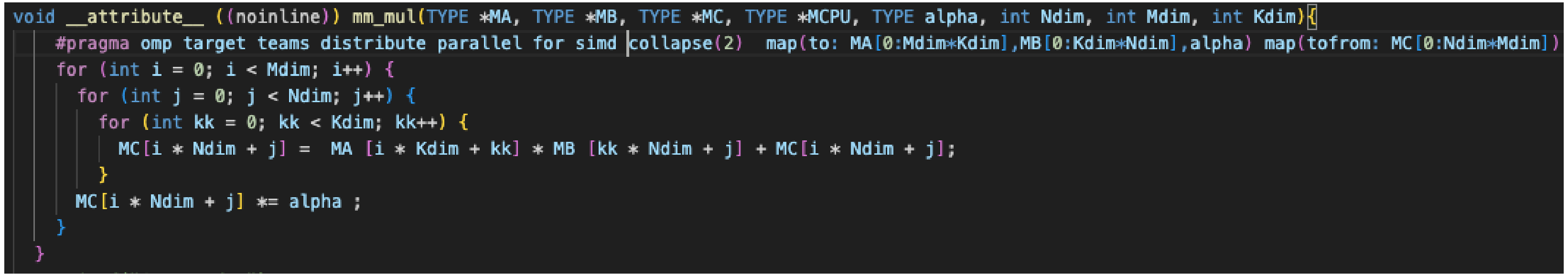
Step 2: Thread-Level Parallelism:
`#pragma omp parallel for` creates threads on the GPU.

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.

Step 4: `collapse()` Merge loops

Enables to merge all the iterations of several associated loops into a single large iteration loop.

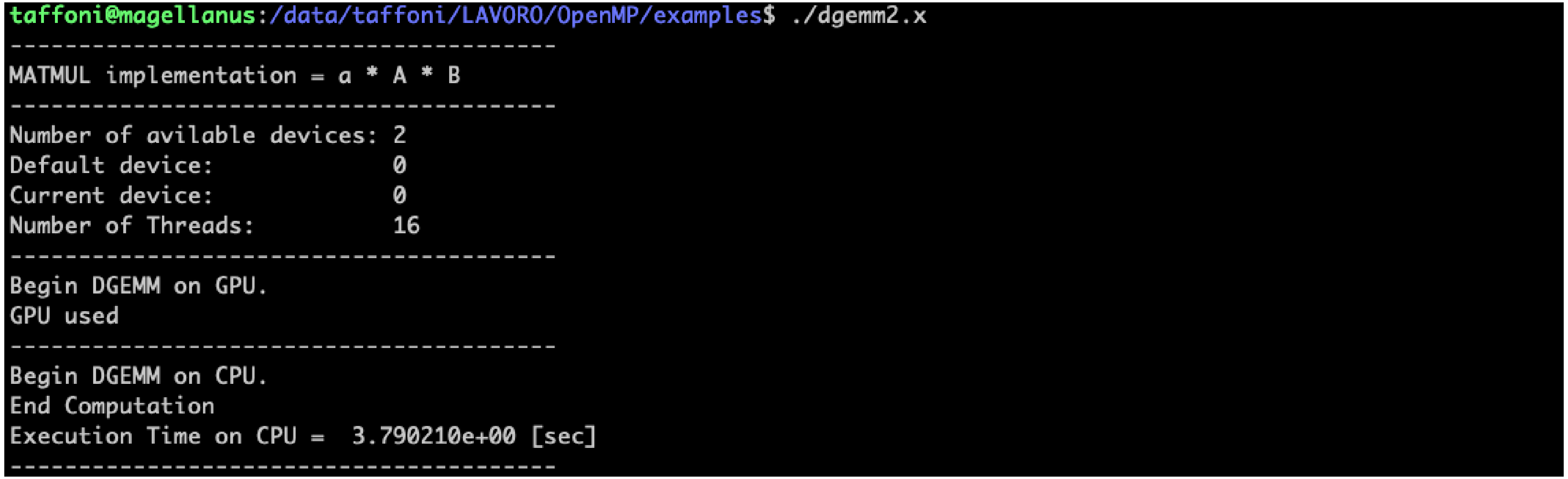


```
void __attribute__ ((noinline)) mm_mul(TYPE *MA, TYPE *MB, TYPE *MC, TYPE *MCPU, TYPE alpha, int Ndim, int Mdim, int Kdim){
    #pragma omp target teams distribute parallel for simd collapse(2)  map(to: MA[0:Mdim*Kdim],MB[0:Kdim*Ndim],alpha) map(tofrom: MC[0:Ndim*Mdim])
    for (int i = 0; i < Mdim; i++) {
      for (int j = 0; j < Ndim; j++) {
        for (int kk = 0; kk < Kdim; kk++) {
          MC[i * Ndim + j] =  MA [i * Kdim + kk] * MB [kk * Ndim + j] + MC[i * Ndim + j];
        }
      MC[i * Ndim + j] *= alpha ;
    }
  }
```

Creating Parallelism on the Target Device

Step 4: `collapse()` **Merge loops**

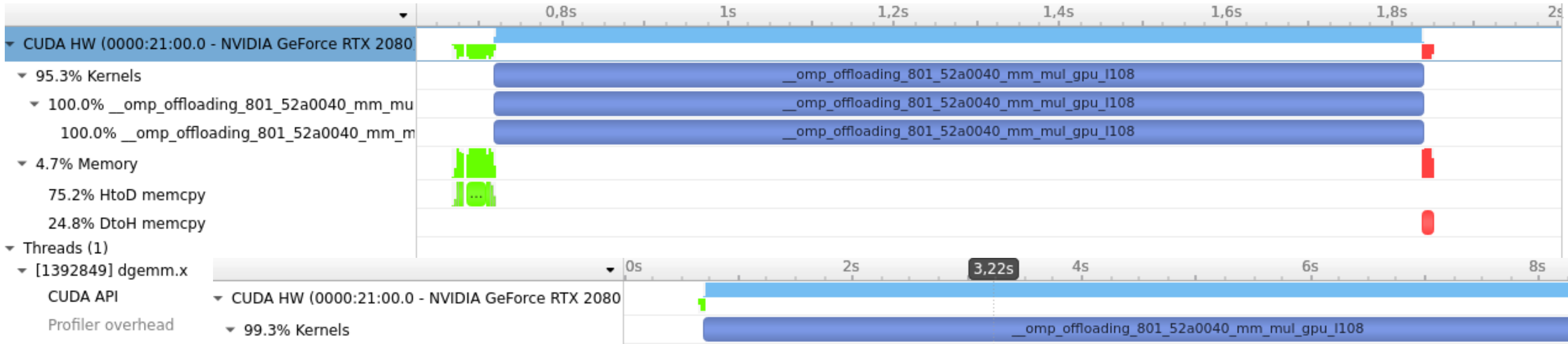
Enables to merge all the iterations of several associated loops into a single large iteration loop.



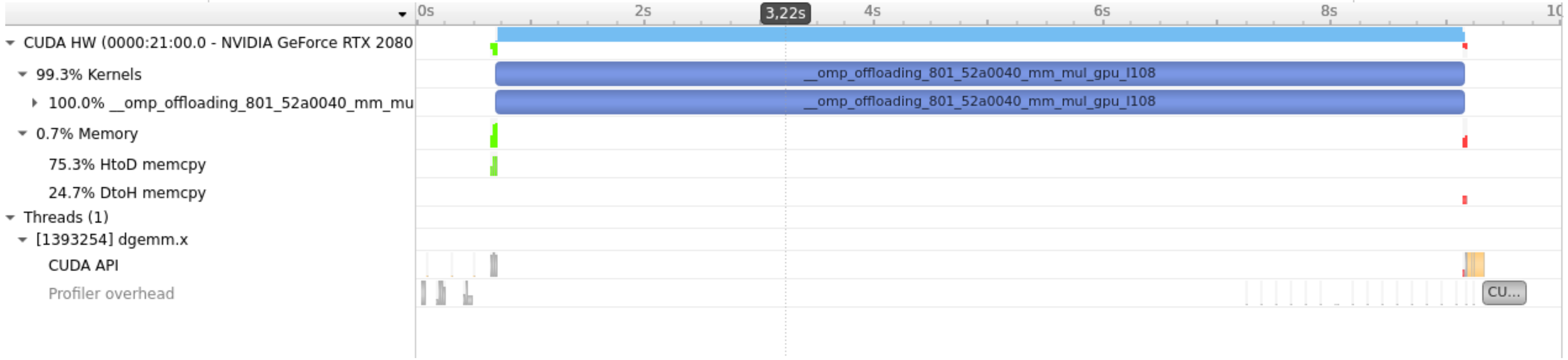
```
taffoni@magellanus:/data/taffoni/LAVORO/OpenMP/examples$ ./dgemm2.x
----------------------------------------
MATMUL implementation = a * A * B
----------------------------------------
Number of avilable devices: 2
Default device:             0
Current device:             0
Number of Threads:          16
----------------------------------------
Begin DGEMM on GPU.
GPU used
----------------------------------------
Begin DGEMM on CPU.
End Computation
Execution Time on CPU =  3.790210e+00 [sec]
----------------------------------------
```

Creating Parallelism on the Target Device

SIMD approach: ~11 sec
SIMD COLLAPSE approach: 3.8 sec



0,8s
1s
1,2s
1,4s
1,6s
1,8s
2s
CUDA HW (0000:21:00.0 - NVIDIA GeForce RTX 2080
95.3% Kernels
100.0% __omp_offloading_801_52a0040_mm_mu
100.0% __omp_offloading_801_52a0040_mm_m
4.7% Memory
75.2% HtoD memcpy
24.8% DtoH memcpy
Threads (1)
[1392849] dgemm.x
CUDA API
Profiler overhead
__omp_offloading_801_52a0040_mm_mul_gpu_l108

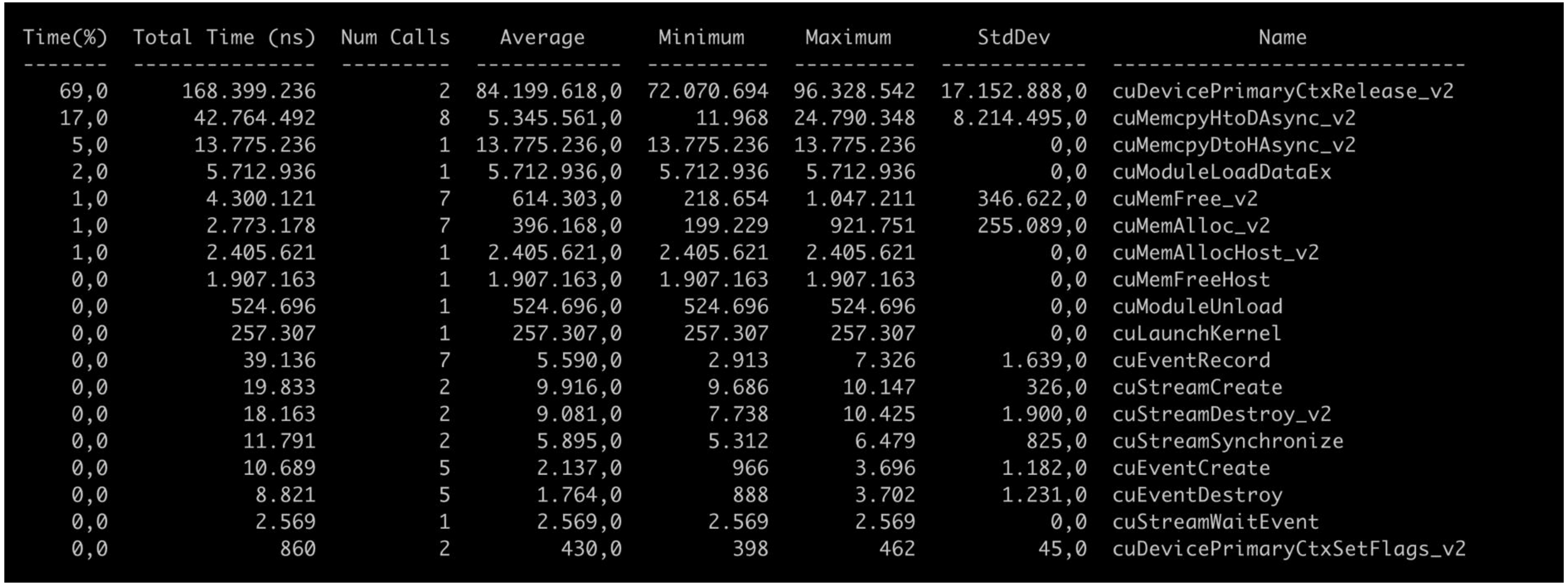


0s
2s
3,22s
4s
6s
8s
CUDA HW (0000:21:00.0 - NVIDIA GeForce RTX 2080
99.3% Kernels
100.0% __omp_offloading_801_52a0040_mm_mu
0.7% Memory
75.3% HtoD memcpy
24.7% DtoH memcpy
Threads (1)
[1393254] dgemm.x
CUDA API
Profiler overhead
__omp_offloading_801_52a0040_mm_mul_gpu_l108
CU...

Profiling Code on GPU

NSYS
NVPROF

```
examples$ nsys nvprof -o dgemm_simd ./dgemm.x
```

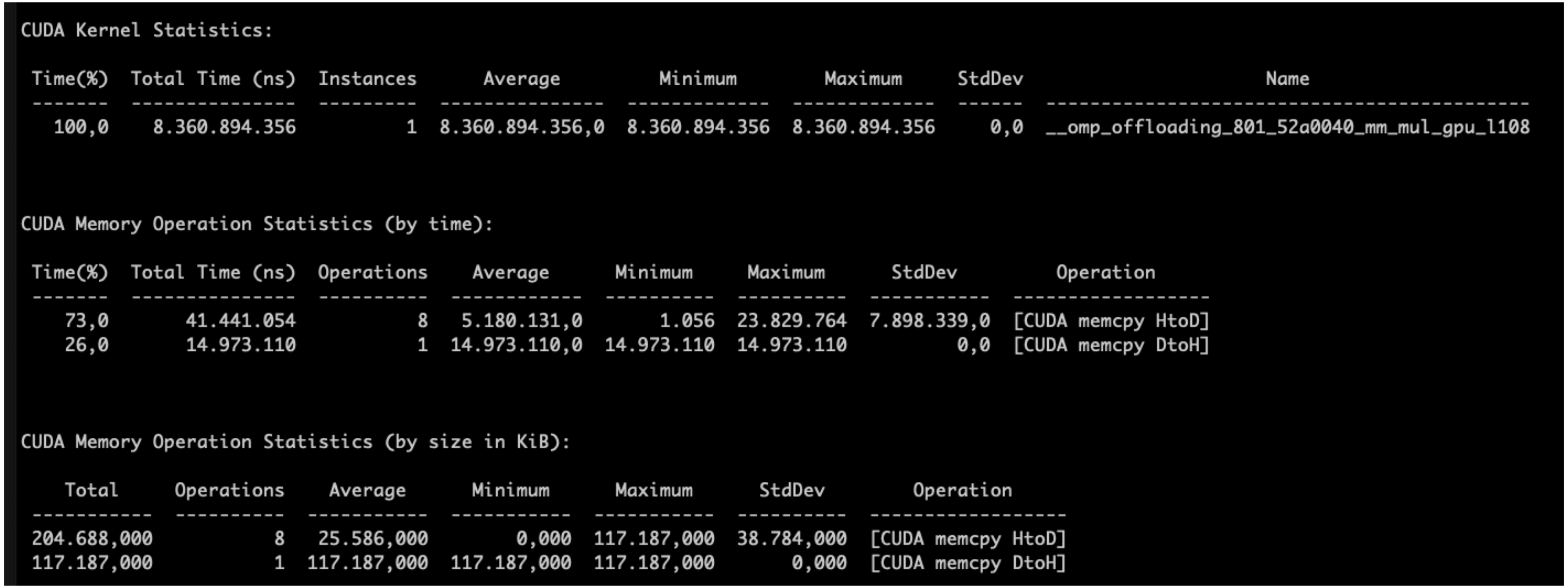


```
Time(%)  Total Time (ns)  Num Calls    Average      Minimum      Maximum      StdDev                Name
-------  ---------------  ---------  ------------  ----------  ----------  ------------  -----------------------------
   69,0      168.399.236          2  84.199.618,0  72.070.694  96.328.542  17.152.888,0  cuDevicePrimaryCtxRelease_v2
   17,0       42.764.492          8   5.345.561,0      11.968  24.790.348   8.214.495,0  cuMemcpyHtoDAsync_v2
    5,0       13.775.236          1  13.775.236,0  13.775.236  13.775.236           0,0  cuMemcpyDtoHAsync_v2
    2,0        5.712.936          1   5.712.936,0   5.712.936   5.712.936           0,0  cuModuleLoadDataEx
    1,0        4.300.121          7     614.303,0     218.654   1.047.211     346.622,0  cuMemFree_v2
    1,0        2.773.178          7     396.168,0     199.229     921.751     255.089,0  cuMemAlloc_v2
    1,0        2.405.621          1   2.405.621,0   2.405.621   2.405.621           0,0  cuMemAllocHost_v2
    0,0        1.907.163          1   1.907.163,0   1.907.163   1.907.163           0,0  cuMemFreeHost
    0,0          524.696          1     524.696,0     524.696     524.696           0,0  cuModuleUnload
    0,0          257.307          1     257.307,0     257.307     257.307           0,0  cuLaunchKernel
    0,0           39.136          7       5.590,0       2.913       7.326       1.639,0  cuEventRecord
    0,0           19.833          2       9.916,0       9.686      10.147         326,0  cuStreamCreate
    0,0           18.163          2       9.081,0       7.738      10.425       1.900,0  cuStreamDestroy_v2
    0,0           11.791          2       5.895,0       5.312       6.479         825,0  cuStreamSynchronize
    0,0           10.689          5       2.137,0         966       3.696       1.182,0  cuEventCreate
    0,0            8.821          5       1.764,0         888       3.702       1.231,0  cuEventDestroy
    0,0            2.569          1       2.569,0       2.569       2.569           0,0  cuStreamWaitEvent
    0,0              860          2         430,0         398         462          45,0  cuDevicePrimaryCtxSetFlags_v2
```

Profiling Code on GPU

NSYS NVPROF

```
examples$ nsys nvprof -o dgemm_simd ./dgemm.x
```

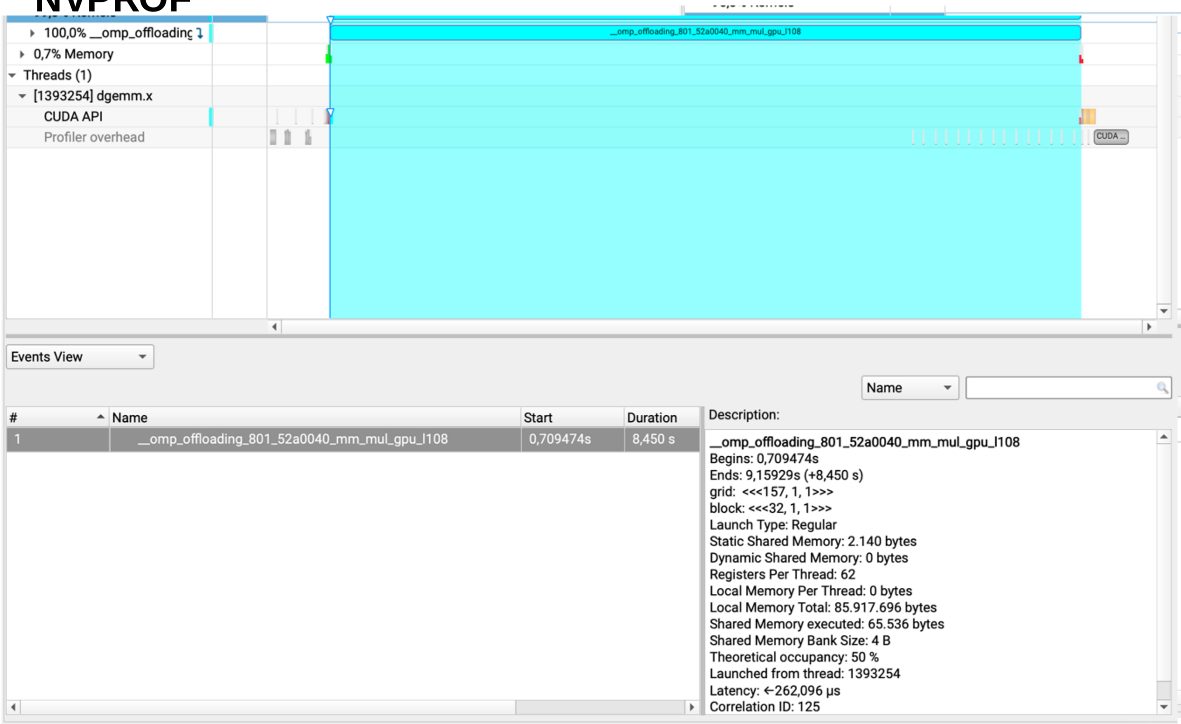


```
CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances      Average          Minimum        Maximum      StdDev                     Name
 -------  ---------------  ---------  ---------------  -------------  -------------  ------  --------------------------------------------
   100,0    8.360.894.356          1  8.360.894.356,0  8.360.894.356  8.360.894.356     0,0  __omp_offloading_801_52a0040_mm_mul_gpu_l108



CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations     Average       Minimum      Maximum      StdDev         Operation
 -------  ---------------  ----------  ------------  ----------  ----------  -----------  ------------------
    73,0       41.441.054           8   5.180.131,0       1.056  23.829.764  7.898.339,0  [CUDA memcpy HtoD]
    26,0       14.973.110           1  14.973.110,0  14.973.110  14.973.110          0,0  [CUDA memcpy DtoH]



CUDA Memory Operation Statistics (by size in KiB):

    Total     Operations    Average      Minimum      Maximum      StdDev        Operation
 -----------  ----------  -----------  -----------  -----------  ----------  ------------------
 204.688,000           8   25.586,000        0,000  117.187,000  38.784,000  [CUDA memcpy HtoD]
 117.187,000           1  117.187,000  117.187,000  117.187,000       0,000  [CUDA memcpy DtoH]
```

Profiling Code on GPU

NSYS NVPROF

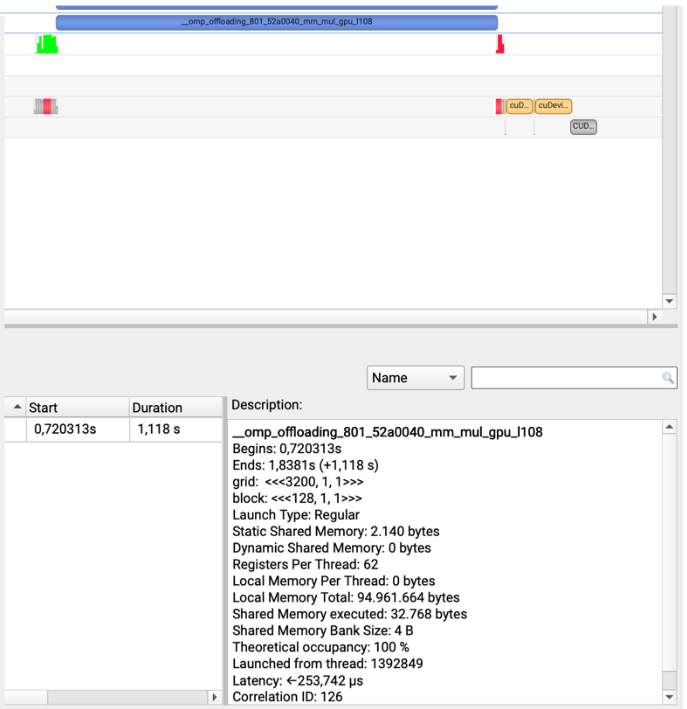
```
examples$ nsys nvprof -o dgemm_simd ./dgemm.x
```



#	Name	Start	Duration
1	__omp_offloading_801_52a0040_mm_mul_gpu_l108	0,709474s	8,450 s

Description:

__omp_offloading_801_52a0040_mm_mul_gpu_l108
Begins: 0,709474s
Ends: 9,15929s (+8,450 s)
grid: <<<157, 1, 1>>>
block: <<<32, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 2.140 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 62
Local Memory Per Thread: 0 bytes
Local Memory Total: 85.917.696 bytes
Shared Memory executed: 65.536 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 50 %
Launched from thread: 1393254
Latency: ←262,096 µs
Correlation ID: 125

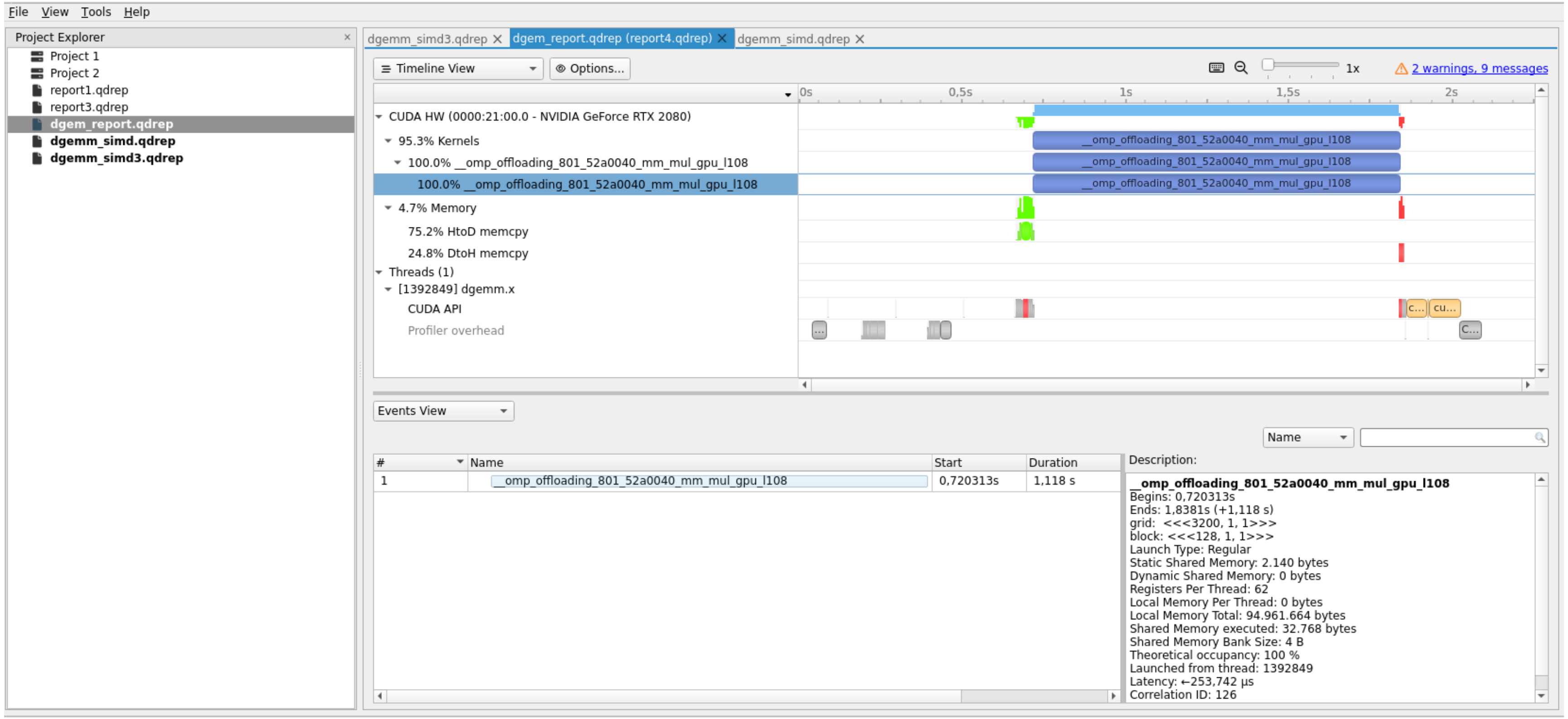


Start	Duration
0,720313s	1,118 s

Description:

__omp_offloading_801_52a0040_mm_mul_gpu_l108
Begins: 0,720313s
Ends: 1,8381s (+1,118 s)
grid: <<<3200, 1, 1>>>
block: <<<128, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 2.140 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 62
Local Memory Per Thread: 0 bytes
Local Memory Total: 94.961.664 bytes
Shared Memory executed: 32.768 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 100 %
Launched from thread: 1392849
Latency: ←253,742 µs
Correlation ID: 126

Profiling Code on GPU



CUDA HW (0000:21:00.0 - NVIDIA GeForce RTX 2080)
95.3% Kernels
100.0% __omp_offloading_801_52a0040_mm_mul_gpu_l108
100.0% __omp_offloading_801_52a0040_mm_mul_gpu_l108
4.7% Memory
75.2% HtoD memcpy
24.8% DtoH memcpy
Threads (1)
[1392849] dgemm.x
CUDA API
Profiler overhead

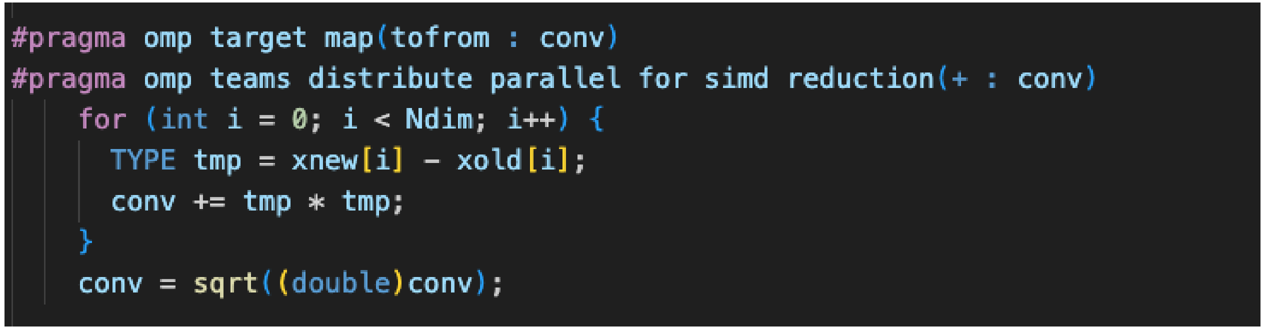
#	Name	Start	Duration
1	__omp_offloading_801_52a0040_mm_mul_gpu_l108	0,720313s	1,118 s

Description:

__omp_offloading_801_52a0040_mm_mul_gpu_l108
Begins: 0,720313s
Ends: 1,8381s (+1,118 s)
grid: <<<3200, 1, 1>>>
block: <<<128, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 2.140 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 62
Local Memory Per Thread: 0 bytes
Local Memory Total: 94.961.664 bytes
Shared Memory executed: 32.768 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 100 %
Launched from thread: 1392849
Latency: ←253,742 µs
Correlation ID: 126

OpenMP Reduction

*“The reduction clauses are **data-sharing attribute** clauses that can be used to perform some forms of recurrence calculations in parallel.”*



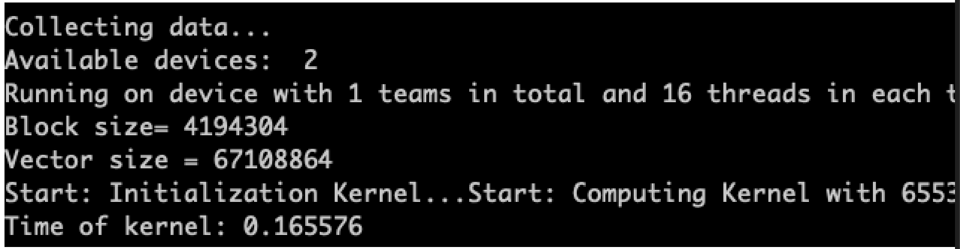
```
#pragma omp target map(tofrom : conv)
#pragma omp teams distribute parallel for simd reduction(+ : conv)
    for (int i = 0; i < Ndim; i++) {
      TYPE tmp = xnew[i] - xold[i];
      conv += tmp * tmp;
    }
    conv = sqrt((double)conv);
```

```
#pragma omp target parallel for reduction(operation:variable_list)
```

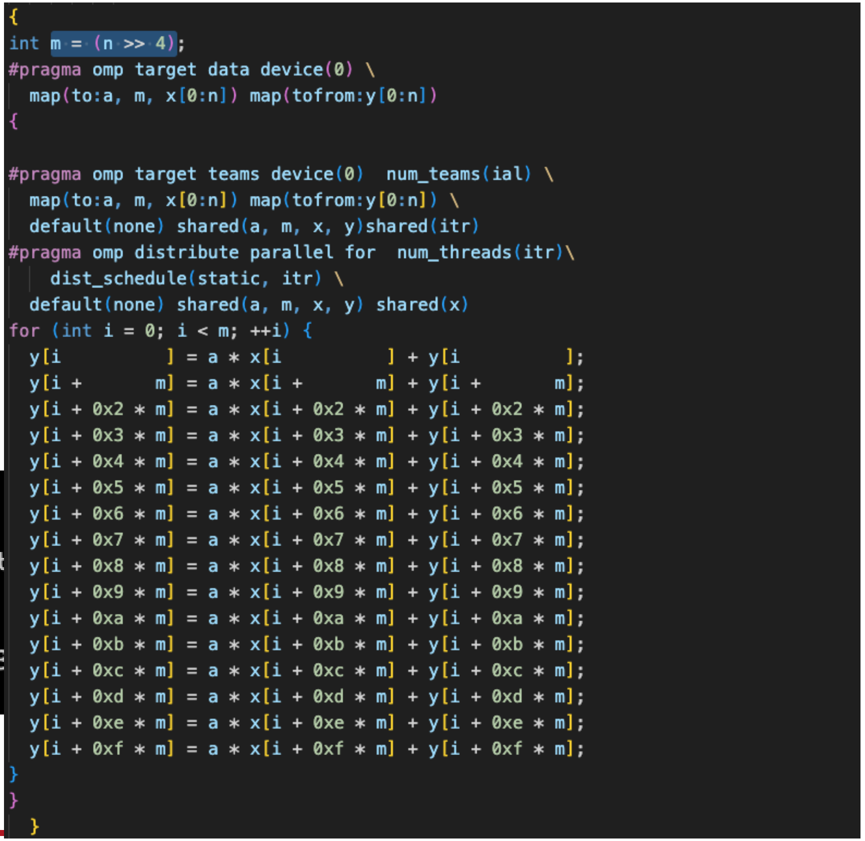
Multi-level Parallelism

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

Block size = 16



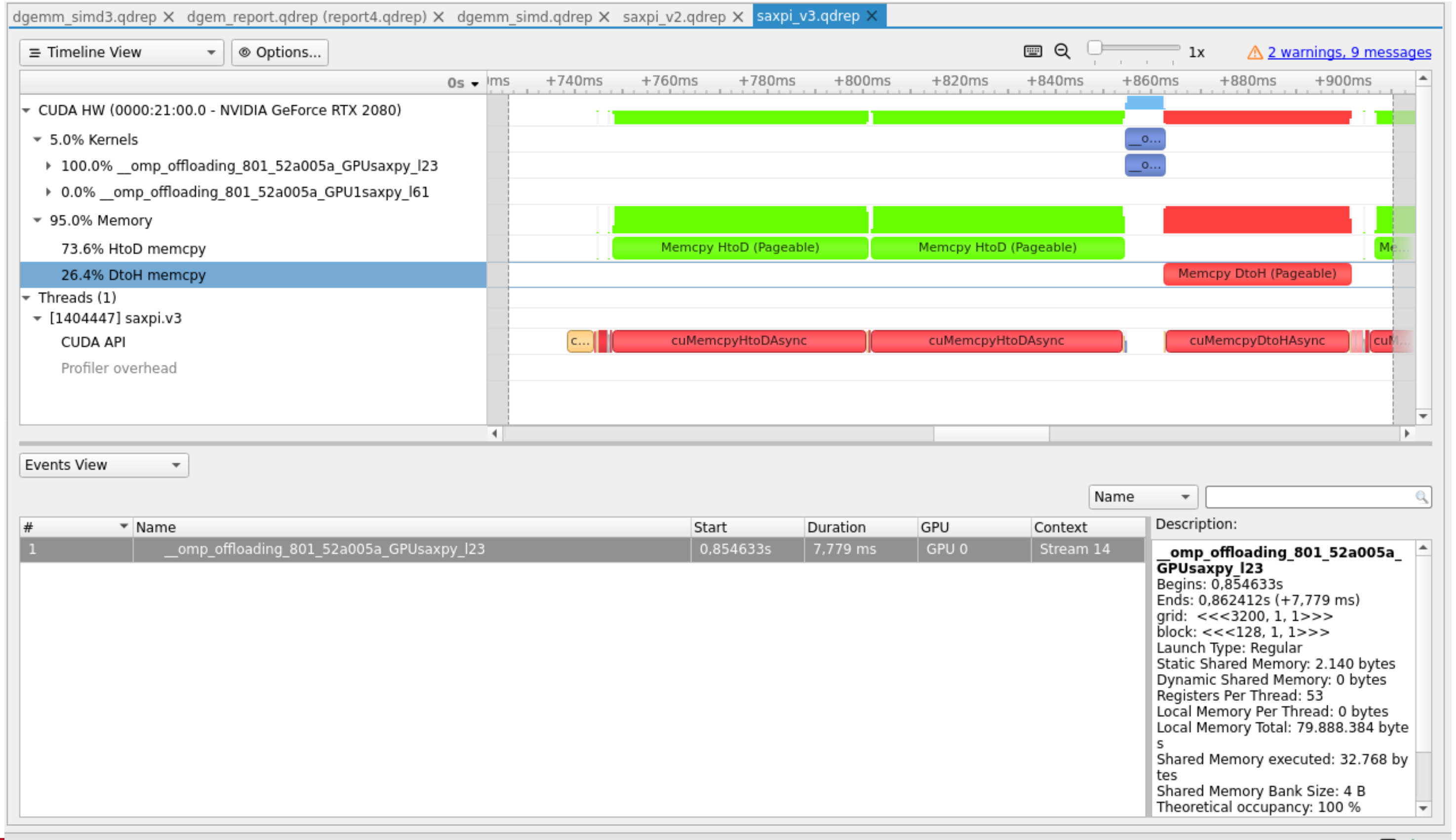
```
Collecting data...
Available devices:  2
Running on device with 1 teams in total and 16 threads in each t
Block size= 4194304
Vector size = 67108864
Start: Initialization Kernel...Start: Computing Kernel with 6553
Time of kernel: 0.165576
```



```
{
int m = (n >> 4);
#pragma omp target data device(0) \
  map(to:a, m, x[0:n]) map(tofrom:y[0:n])
{

#pragma omp target teams device(0)  num_teams(ial) \
  map(to:a, m, x[0:n]) map(tofrom:y[0:n]) \
  default(none) shared(a, m, x, y)shared(itr)
#pragma omp distribute parallel for  num_threads(itr)\
    dist_schedule(static, itr) \
  default(none) shared(a, m, x, y) shared(x)
for (int i = 0; i < m; ++i) {
  y[i          ] = a * x[i          ] + y[i          ];
  y[i +       m] = a * x[i +       m] + y[i +       m];
  y[i + 0x2 * m] = a * x[i + 0x2 * m] + y[i + 0x2 * m];
  y[i + 0x3 * m] = a * x[i + 0x3 * m] + y[i + 0x3 * m];
  y[i + 0x4 * m] = a * x[i + 0x4 * m] + y[i + 0x4 * m];
  y[i + 0x5 * m] = a * x[i + 0x5 * m] + y[i + 0x5 * m];
  y[i + 0x6 * m] = a * x[i + 0x6 * m] + y[i + 0x6 * m];
  y[i + 0x7 * m] = a * x[i + 0x7 * m] + y[i + 0x7 * m];
  y[i + 0x8 * m] = a * x[i + 0x8 * m] + y[i + 0x8 * m];
  y[i + 0x9 * m] = a * x[i + 0x9 * m] + y[i + 0x9 * m];
  y[i + 0xa * m] = a * x[i + 0xa * m] + y[i + 0xa * m];
  y[i + 0xb * m] = a * x[i + 0xb * m] + y[i + 0xb * m];
  y[i + 0xc * m] = a * x[i + 0xc * m] + y[i + 0xc * m];
  y[i + 0xd * m] = a * x[i + 0xd * m] + y[i + 0xd * m];
  y[i + 0xe * m] = a * x[i + 0xe * m] + y[i + 0xe * m];
  y[i + 0xf * m] = a * x[i + 0xf * m] + y[i + 0xf * m];
}
}
  }
```

Profiling Code on GPU

NSYS
NVPROF



#	Name	Start	Duration	GPU	Context
1	__omp_offloading_801_52a005a_GPUsaxpy_l23	0,854633s	7,779 ms	GPU 0	Stream 14

Description:
__omp_offloading_801_52a005a_GPUsaxpy_l23
Begins: 0,854633s
Ends: 0,862412s (+7,779 ms)
grid: <<<3200, 1, 1>>>
block: <<<128, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 2.140 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 53
Local Memory Per Thread: 0 bytes
Local Memory Total: 79.888.384 bytes
Shared Memory executed: 32.768 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 100 %

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/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:

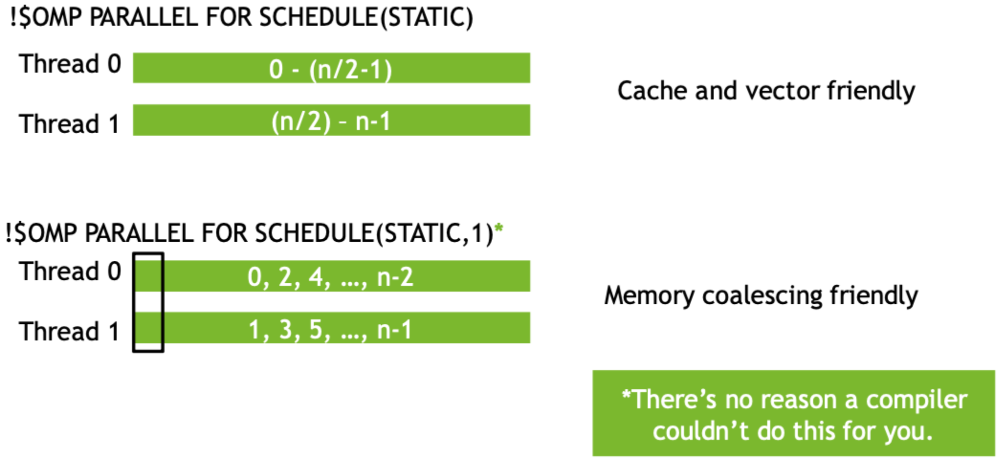
- Dynamically assigns chunks to teams and threads.
- Useful for unbalanced workloads.

Improving Parallelism: scheduling

Most OpenMP compilers will apply a **static** schedule to workshared loops, assigning iterations in (`N/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)
```

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

*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	The iterations of the loop are divided into chunks of fixed size (as specified in the chunk_size argument) and assigned to threads in a round-robin fashion. If no chunk_size is specified, iterations are divided into chunks of approximately equal size.
dynamic	Iterations are divided into chunks of chunk_size, 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.
guided	Iterations are divided into chunks, but the size of each chunk decreases exponentially as the computation progresses. •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.

How OpenMP Leverages GPU Hardware

SMs (Streaming Multiprocessors):

- Each **team** is assigned to one or more SMs.
- Multiple teams can share an SM depending on the workload and hardware capabilities.

Warp-Level Execution:

- Threads within a team are grouped into **warps** and execute SIMD-style instructions.

Resource Sharing:

- Teams and threads within an SM share the same hardware resources, such as L1 cache, registers, and shared memory.

Key OpenMP Directives

- **Offloading:** `#pragma omp target`
Specifies that the computation should run on the GPU.

- **Teams Creation**: `#pragma omp teams`
Creates a grid of teams, each consisting of multiple threads.

- **• Parallel Threads Within a Team:** `#pragma omp parallel`
Specifies parallel execution within a team.

- **Combined Directive:** `#pragma omp target teams distribute parallel for`
Combines offloading, team creation, and thread-level parallelization.

Key Mapping

Teams are mapped to SMs.

Threads within a team are mapped to CUDA cores and grouped into warps.

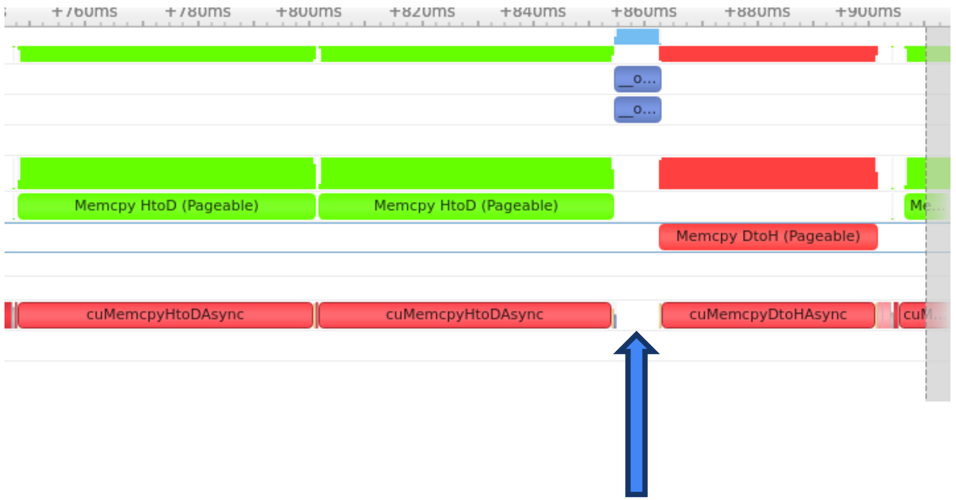
OpenMP vs CUDA: Key Differences

Aspect	OpenMP	CUDA
Grid/Block Management	Automatically handled by OpenMP runtime	Explicitly defined by the programmer
Thread Control	Threads are abstracted; programmer controls teams and loops	Fully controlled by the programmer
Ease of Use	High, suitable for quick GPU programming	Requires more detailed understanding of GPU hardware
Performance Tuning	May require effort to optimize (less low-level control)	Allows fine-tuning of hardware resources
Memory Management	Managed implicitly via OpenMP clauses	Explicitly managed by the programmer

OpenMP: modular programming

Some considerations:

- The OMP implementation of memCopy is synchronous (and implies CPU operations)
- The OMP implementation of Kernel execution is synchronous (NO CPU operations)
- Copy data is expensive



CPU can process something here!!!!

OpenMP: optimizing data transfer

If you want to allocate the memory of some variables on the device at a given point of your program but it is not possible to free the memory within the same scope of the program, you can then use the enter data and exit data constructs.

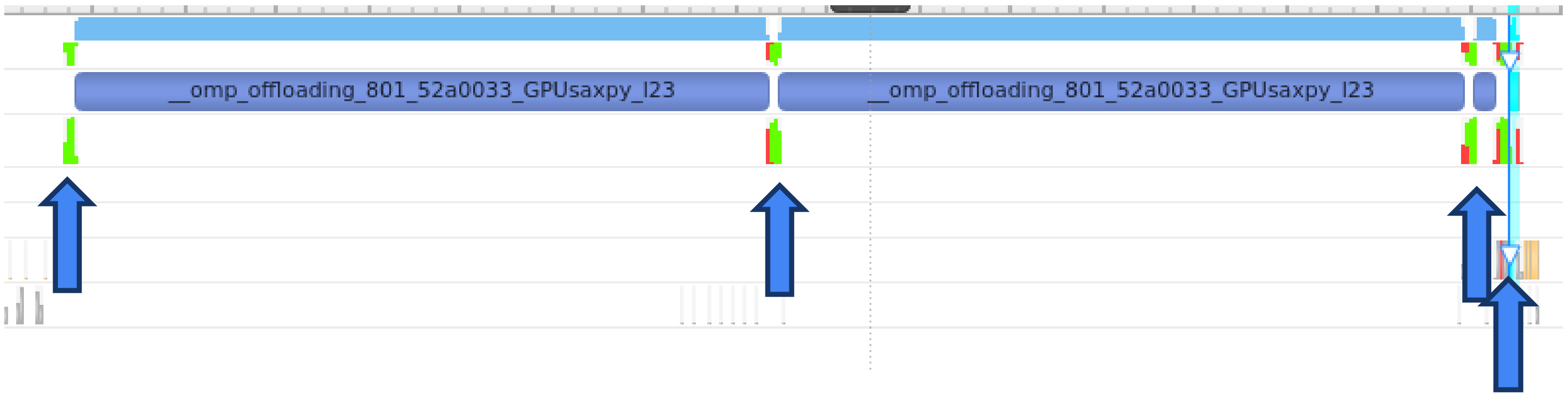
```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(nteams)
    {
        int bs = n / omp_get_num_teams();   // n assumed to be multiple of #teams
        #pragma omp distribute
        for (int i = 0; i < sz; i += bs) {
 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

Memory is deallocated

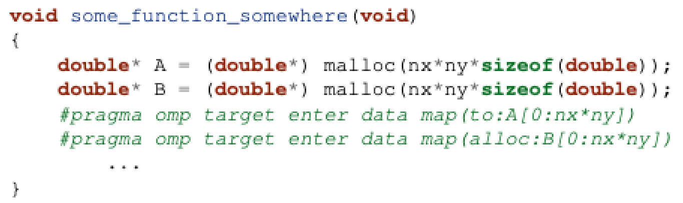
OpenMP: optimizing data transfer

If you want to allocate the memory of some variables on the device at a given point of your program but it is not possible to free the memory within the same scope of the program, you can then use the enter data and exit data constructs.

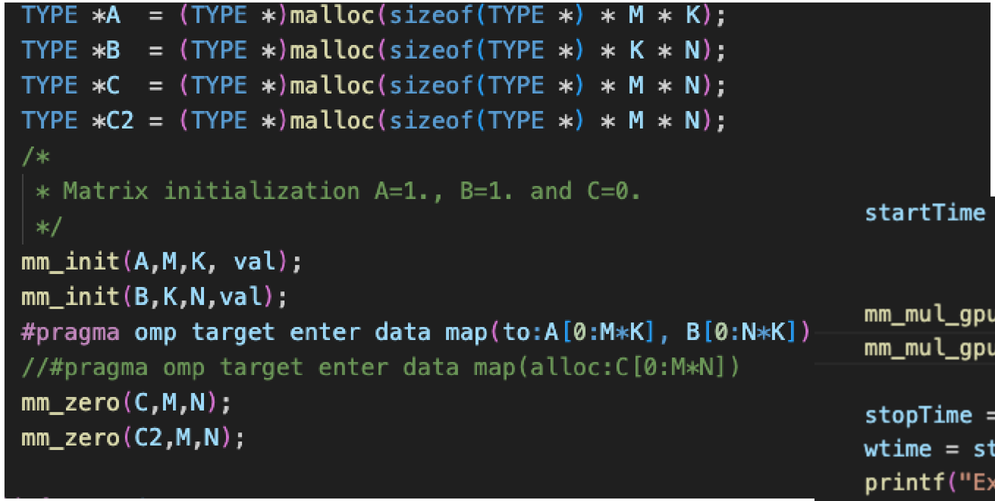
SAXPI.V2


__omp_offloading_801_52a0033_GPUsaxpy_l23
__omp_offloading_801_52a0033_GPUsaxpy_l23

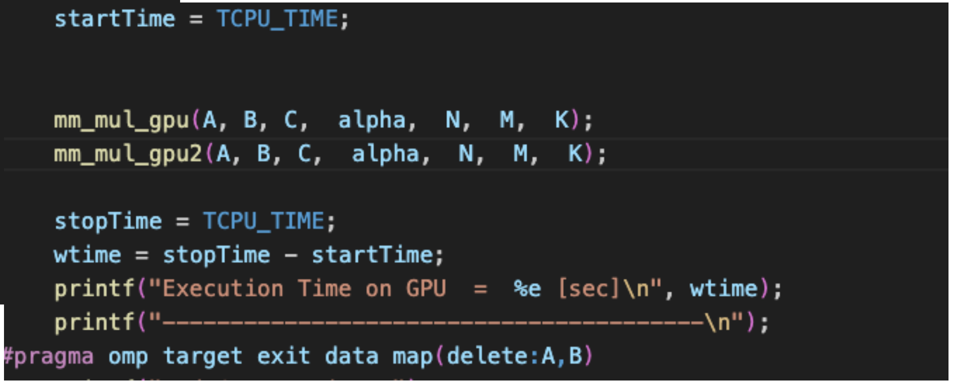
OpenMP: Persistent data



```
void some_function_somewhere(void)
{
    double* A = (double*) malloc(nx*ny*sizeof(double));
    double* B = (double*) malloc(nx*ny*sizeof(double));
    #pragma omp target enter data map(to:A[0:nx*ny])
    #pragma omp target enter data map(alloc:B[0:nx*ny])
        ...
}
```

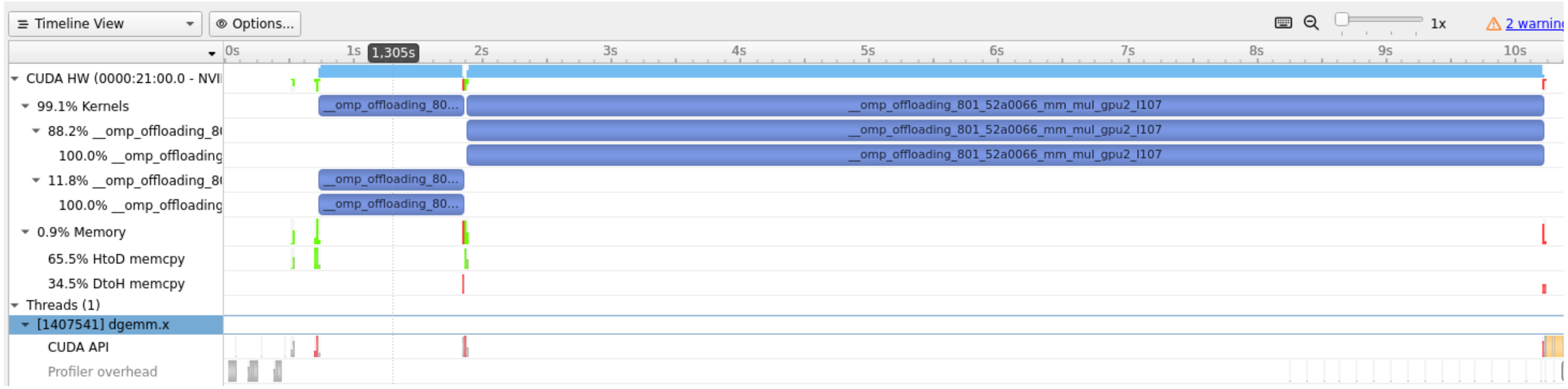


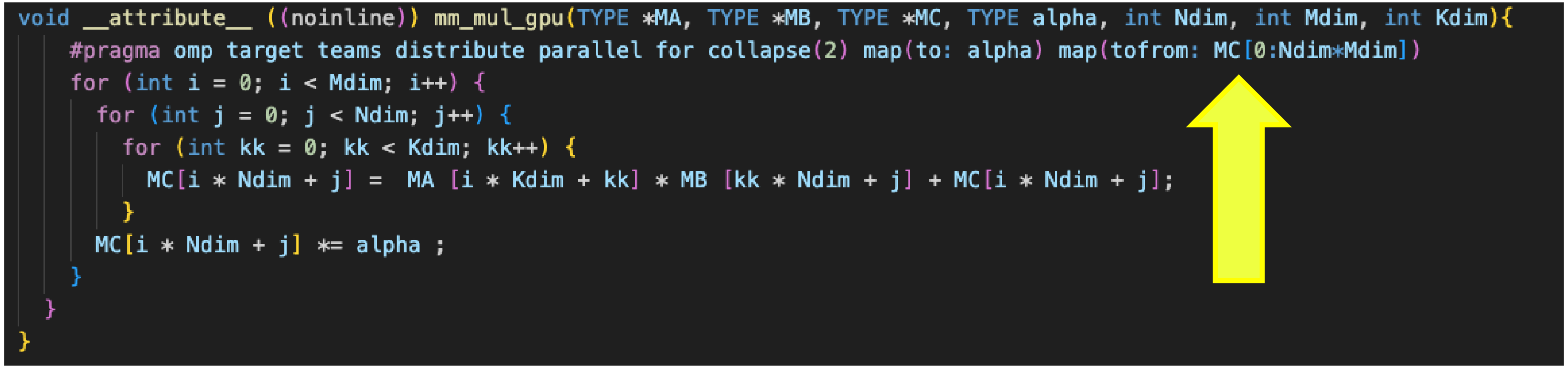
```
TYPE *A  = (TYPE *)malloc(sizeof(TYPE *) * M * K);
TYPE *B  = (TYPE *)malloc(sizeof(TYPE *) * K * N);
TYPE *C  = (TYPE *)malloc(sizeof(TYPE *) * M * N);
TYPE *C2 = (TYPE *)malloc(sizeof(TYPE *) * M * N);
/*
 * Matrix initialization A=1., B=1. and C=0.
 */
mm_init(A,M,K, val);
mm_init(B,K,N,val);
#pragma omp target enter data map(to:A[0:M*K], B[0:N*K])
//#pragma omp target enter data map(alloc:C[0:M*N])
mm_zero(C,M,N);
mm_zero(C2,M,N);
```



```
  startTime = TCPU_TIME;


  mm_mul_gpu(A, B, C,  alpha,  N,  M,  K);
  mm_mul_gpu2(A, B, C,  alpha,  N,  M,  K);

  stopTime = TCPU_TIME;
  wtime = stopTime - startTime;
  printf("Execution Time on GPU  =  %e [sec]\n", wtime);
  printf("-----------------------------------------\n");
#pragma omp target exit data map(delete:A,B)
```

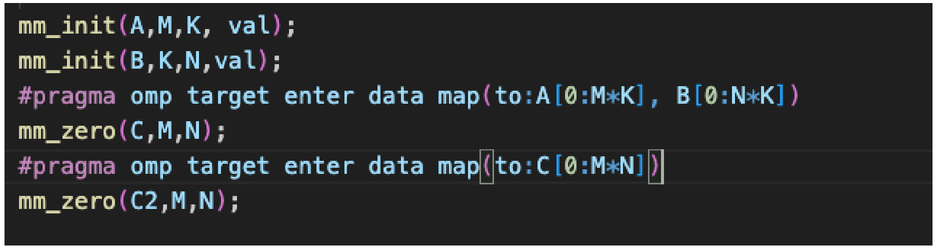
OpenMP: Persistent data



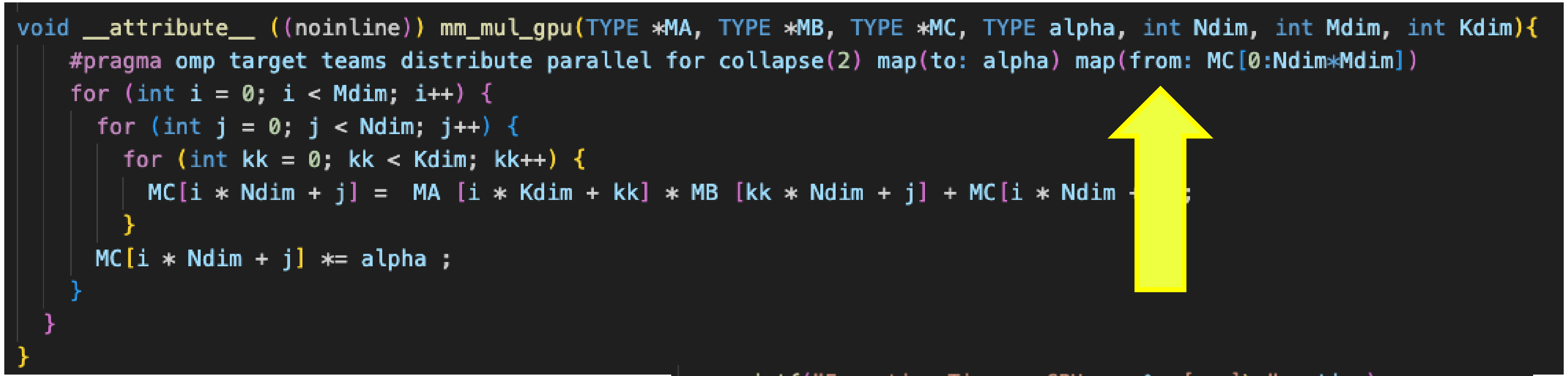


```
void __attribute__ ((noinline)) mm_mul_gpu(TYPE *MA, TYPE *MB, TYPE *MC, TYPE alpha, int Ndim, int Mdim, int Kdim){
    #pragma omp target teams distribute parallel for collapse(2) map(to: alpha) map(tofrom: MC[0:Ndim*Mdim])
    for (int i = 0; i < Mdim; i++) {
      for (int j = 0; j < Ndim; j++) {
        for (int kk = 0; kk < Kdim; kk++) {
          MC[i * Ndim + j] =  MA [i * Kdim + kk] * MB [kk * Ndim + j] + MC[i * Ndim + j];
        }
        MC[i * Ndim + j] *= alpha ;
    }
  }
}
```

OpenMP: Persistent data

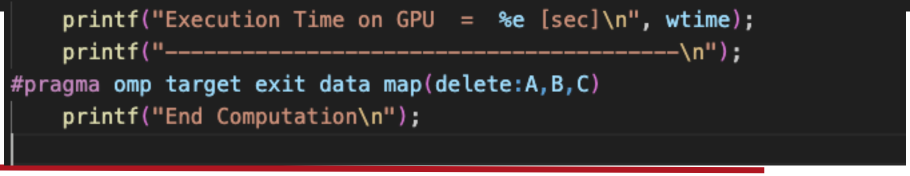


```
mm_init(A,M,K, val);
mm_init(B,K,N,val);
#pragma omp target enter data map(to:A[0:M*K], B[0:N*K])
mm_zero(C,M,N);
#pragma omp target enter data map(to:C[0:M*N])
mm_zero(C2,M,N);
```



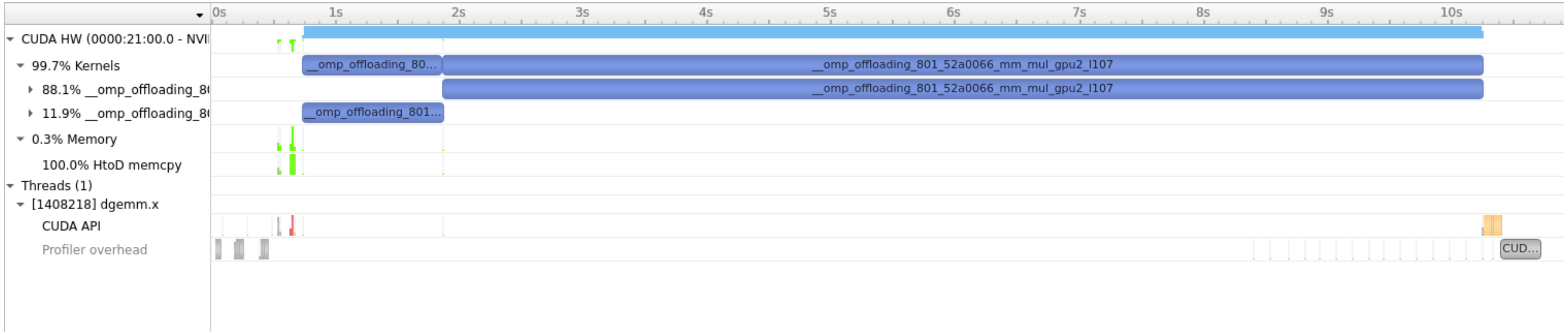
```
void __attribute__ ((noinline)) mm_mul_gpu(TYPE *MA, TYPE *MB, TYPE *MC, TYPE alpha, int Ndim, int Mdim, int Kdim){
    #pragma omp target teams distribute parallel for collapse(2) map(to: alpha) map(from: MC[0:Ndim*Mdim])
    for (int i = 0; i < Mdim; i++) {
      for (int j = 0; j < Ndim; j++) {
        for (int kk = 0; kk < Kdim; kk++) {
          MC[i * Ndim + j] =  MA [i * Kdim + kk] * MB [kk * Ndim + j] + MC[i * Ndim -      ;
        }
      MC[i * Ndim + j] *= alpha ;
    }
  }
}
```

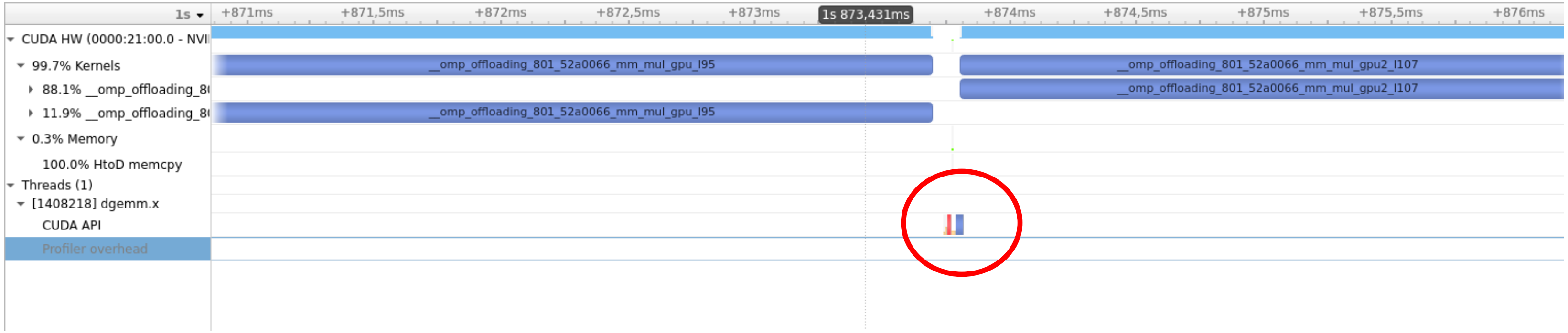
Initialize one (copy one) use two times!!!



```
    printf("Execution Time on GPU  =  %e [sec]\n", wtime);
    printf("----------------------------------------\n");
#pragma omp target exit data map(delete:A,B,C)
    printf("End Computation\n");
```

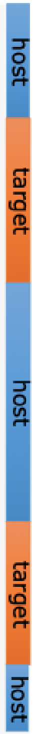
OpenMP: Persistent data



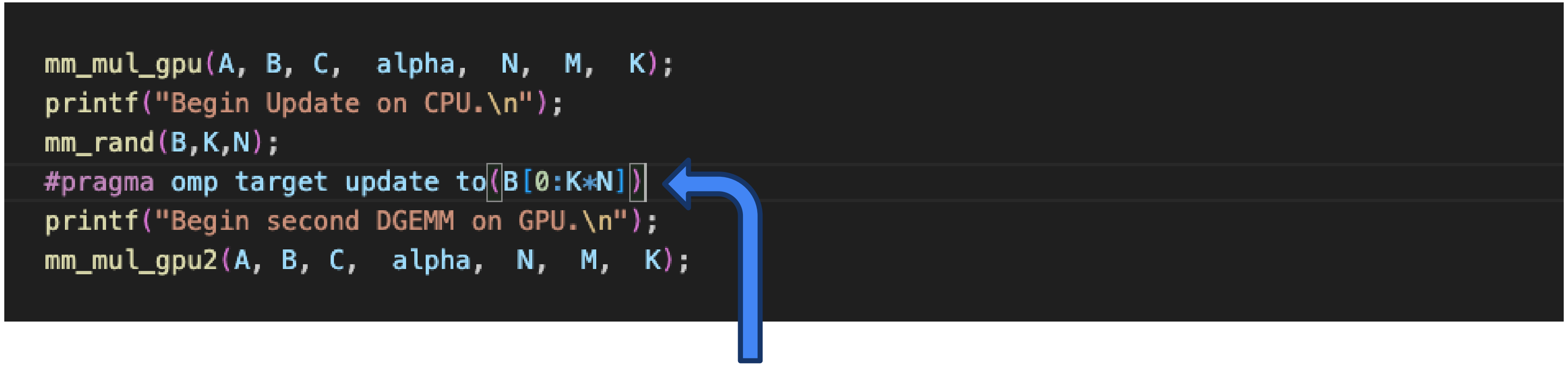


OpenMP optimizing data transfer

```
#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)
  }
```



OpenMP: Persistent data

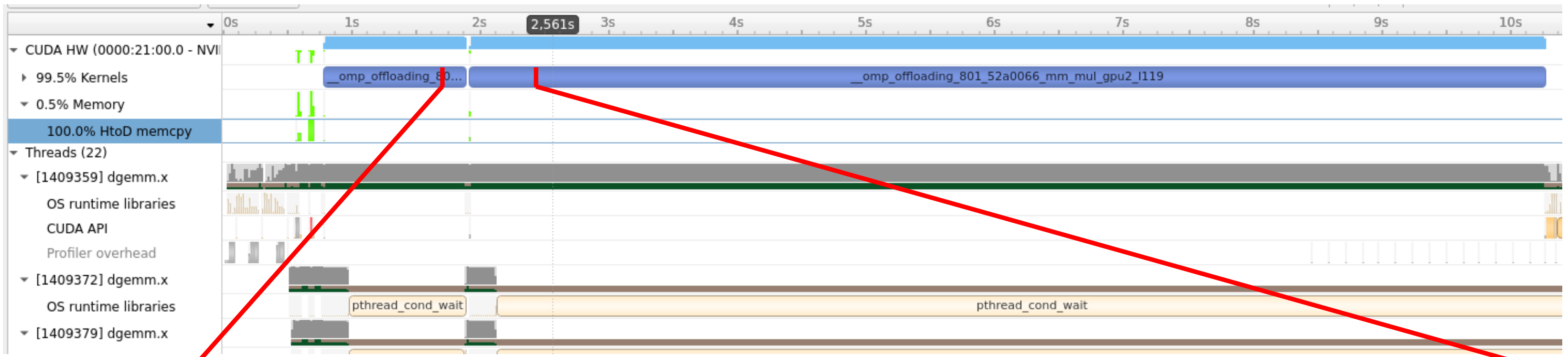


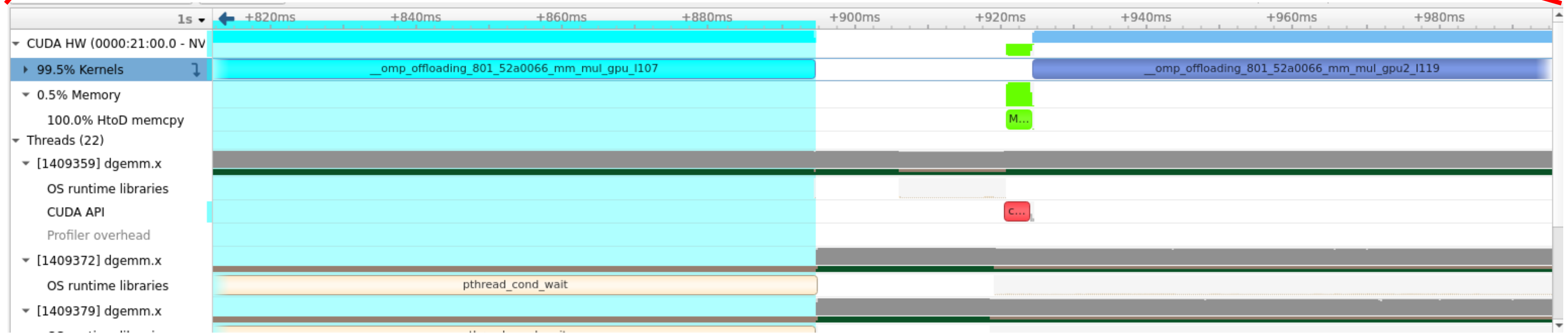
```
mm_mul_gpu(A, B, C,  alpha,  N,  M,  K);
printf("Begin Update on CPU.\n");
mm_rand(B,K,N);
#pragma omp target update to(B[0:K*N])
printf("Begin second DGEMM on GPU.\n");
mm_mul_gpu2(A, B, C,  alpha,  N,  M,  K);
```

When you want to update the values of a given variable, or a set of variables, either on the GPU or on the CPU, you can use the target update construct to **avoid doing it by closing a data structure.**

Warning: you can update the whole array or a part of it.

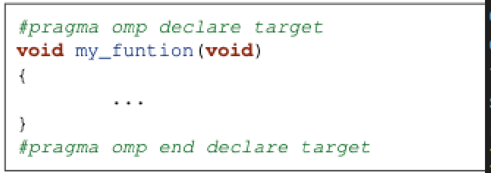
OpenMP: Persistent data



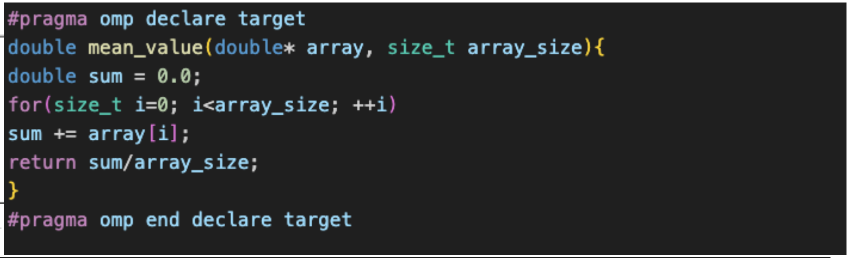


OpenMP: Modular Programming

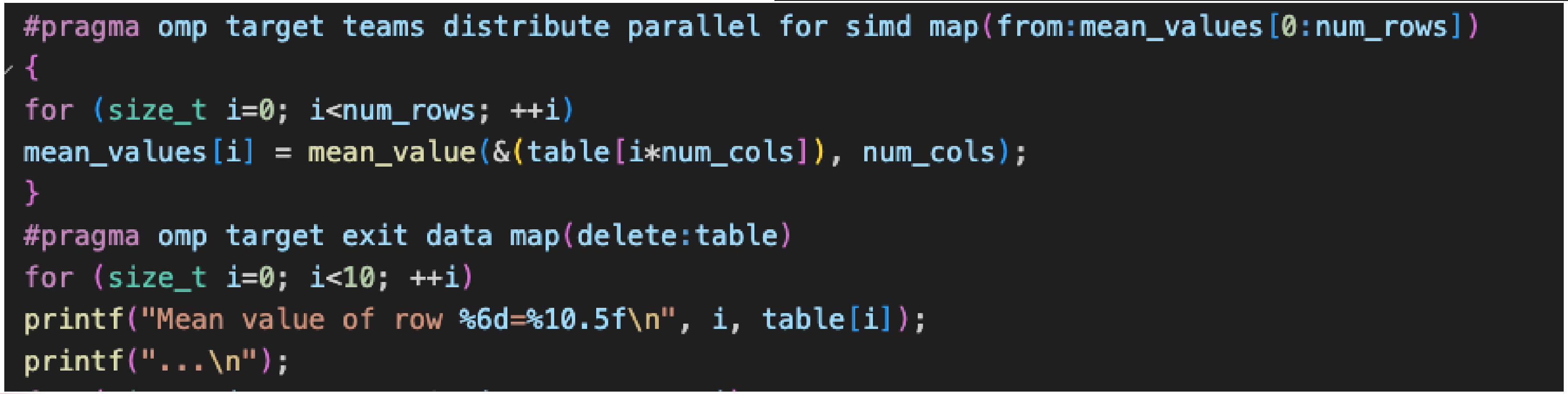
Functions that are call inside a kernel should be executed on the accelerator.
You should use the `declare` construct to inform the compiler that it should produce such an executable.



```
#pragma omp declare target
void my_funtion(void)
{
        ...
}
#pragma omp end declare target
```



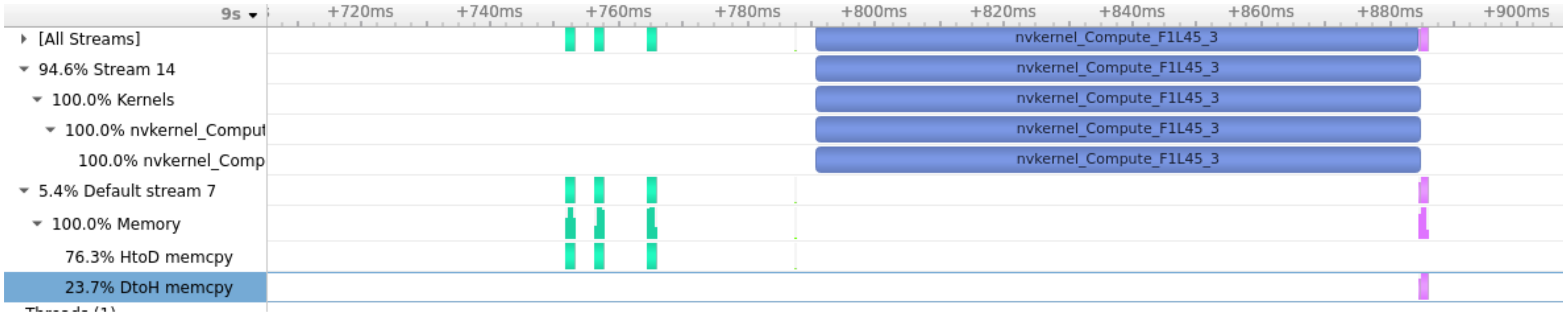
```
#pragma omp declare target
double mean_value(double* array, size_t array_size){
double sum = 0.0;
for(size_t i=0; i<array_size; ++i)
sum += array[i];
return sum/array_size;
}
#pragma omp end declare target
```



```
#pragma omp target teams distribute parallel for simd map(from:mean_values[0:num_rows])
{
for (size_t i=0; i<num_rows; ++i)
mean_values[i] = mean_value(&(table[i*num_cols]), num_cols);
}
#pragma omp target exit data map(delete:table)
for (size_t i=0; i<10; ++i)
printf("Mean value of row %6d=%10.5f\n", i, table[i]);
printf("...\n");
```

OpenMP: Modular Programming

Functions that are call inside a kernel should be executed on the accelerator.
You should use the `declare` construct to inform the compiler that it should produce such an executable.



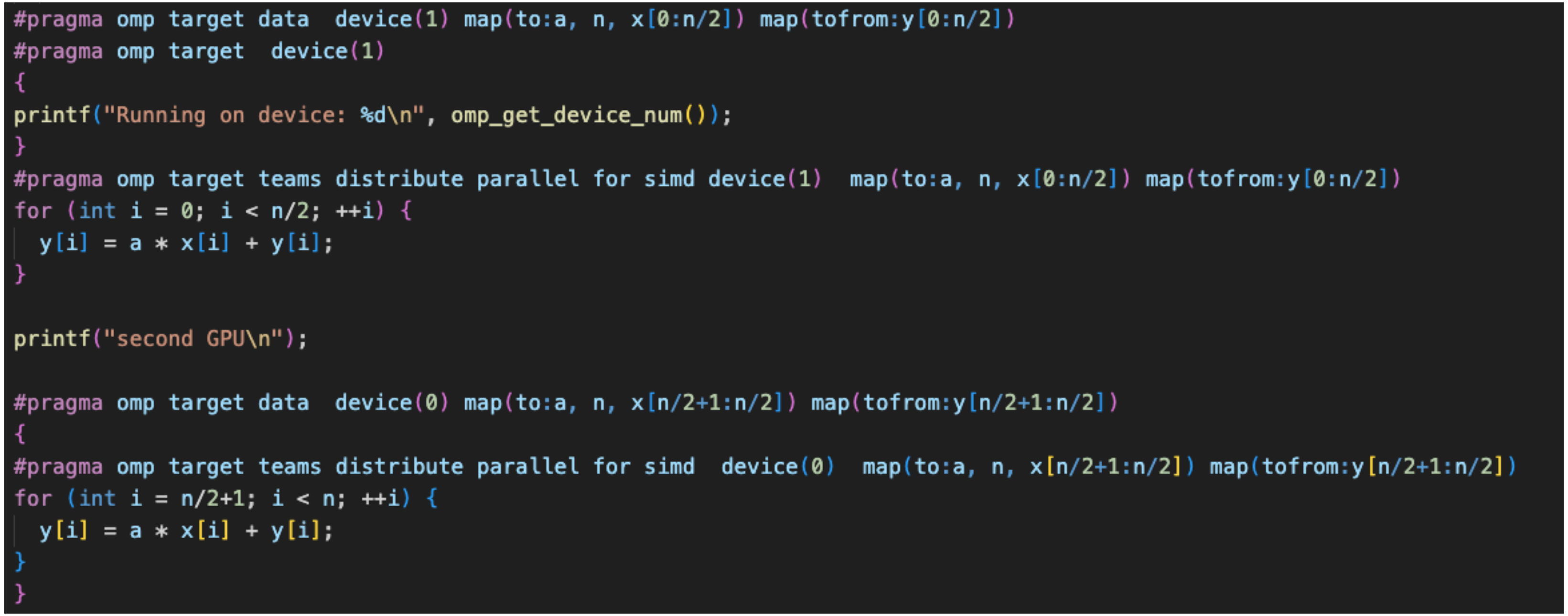
OpenMP: Modular Programming

If you have multiple accelerators available, you can select the one on which you run the kernels with the `device` clause of the `target` construct. It includes both target data constructs and `target teams/parallel` constructs.

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int num_gpus = omp_get_num_devices();
int my_gpu = my_rank%num_gpus
#pragma omp target data map(...) device(my_gpu)
{
    ...
}
```

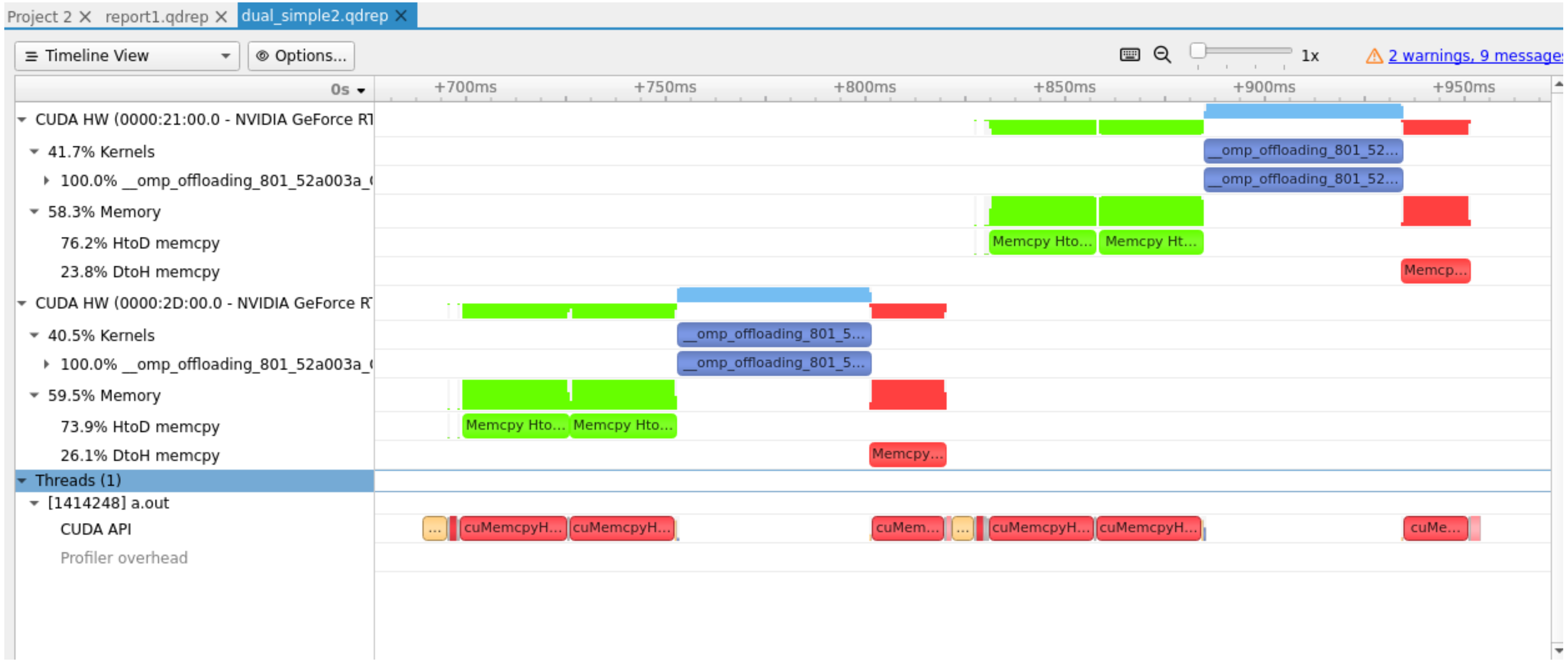
`device` number is 0, 1, 2, …
On `target data` construct you can partition data between GPUs

OpenMP: Modular Programming



```
#pragma omp target data  device(1) map(to:a, n, x[0:n/2]) map(tofrom:y[0:n/2])
#pragma omp target  device(1)
{
printf("Running on device: %d\n", omp_get_device_num());
}
#pragma omp target teams distribute parallel for simd device(1)  map(to:a, n, x[0:n/2]) map(tofrom:y[0:n/2])
for (int i = 0; i < n/2; ++i) {
  y[i] = a * x[i] + y[i];
}

printf("second GPU\n");

#pragma omp target data  device(0) map(to:a, n, x[n/2+1:n/2]) map(tofrom:y[n/2+1:n/2])
{
#pragma omp target teams distribute parallel for simd  device(0)  map(to:a, n, x[n/2+1:n/2]) map(tofrom:y[n/2+1:n/2])
for (int i = n/2+1; i < n; ++i) {
  y[i] = a * x[i] + y[i];
}
}
```

OpenMP: Modular Programming



INAF
ISTITUTO NAZIONALE
DI ASTROFISICA

THANK YOU.

Now, it’s your time
QUESTIONS?

GPU programming with OpenMP