Best Practices for OpenMP

Oct 6, 2023

Chris Daley, Rahul Gayatri, Helen He, Paul Lin, Wael R. Elwasif, Reuben Budiardja, Swaroop Pophale, Suzanne Parete-Koon
GPU Best Practices Overview

1. Use the right combination of OpenMP directives to use *all* GPU parallelism
2. Make sure your loops are large enough to benefit from GPUs
3. Minimize the separation between “teams” and “parallel” directives
4. Use the family of “target data” directives to minimize data movement
5. Don’t map scalar variables (unnecessarily)
6. Avoid Fortran array operations and array sections (:) in target regions
7. Take advantage of compiler diagnostics and runtime tracing
Best Practice #1

Use the right combination of OpenMP directives to use *all* GPU parallelism
GPUs and OpenMP directives

A GPU consists of many SIMD processors (P)

A SIMD processor consists of many physical SIMD lanes (L)

P: Also known as a “Streaming Multiprocessor (SM)” or “Compute Unit (CU)”

L: Also known as a “CUDA core”, “Stream core” or “Shader core”

**OpenMP-5.2**

`#pragma omp teams`
Needed to use more than 1 SIMD processor

`#pragma omp parallel*`
Needed to use more than 1 SIMD lane per SIMD processor

*Cray Fortran compiler sometimes needs “simd” directive. Tip: Always use ”parallel” and “simd”*
Use the right combination of OpenMP directives to use *all* GPU parallelism

<table>
<thead>
<tr>
<th>#pragma omp target parallel for for (int i=0; i&lt;N; ++i)</th>
<th>Missing “teams” directive: will only use 1 SIMD processor (1/108\textsuperscript{th} of an A100 GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp target teams distribute \ parallel for [simd] for (int i=0; i&lt;N; ++i)</td>
<td>“teams” and “parallel” directives needed to use all GPU parallelism</td>
</tr>
<tr>
<td>#pragma omp target teams loop parallel for (int i=0; i&lt;N; ++i)</td>
<td>“parallel” directive not needed “loop” is a worksharing directive that can also generate parallelism</td>
</tr>
</tbody>
</table>
Use the right combination of OpenMP directives to use *all* GPU parallelism

```
#pragma omp target parallel for 
for (int i=0; i<N; ++i)  
```

Missing “teams” directive: will only use 1 SIMD processor (1/108th of an A100 GPU)

OpenMP-4.5 and 5

```
#pragma omp target teams distribute \ 
parallel for [simd] 
for (int i=0; i<N; ++i)  
```

“teams” and “parallel” directives needed to use all GPU parallelism

```
#pragma omp target teams loop 
for (int i=0; i<N; ++i)  
```

“loop” directive has minimal support in the clang compilers

OpenMP-5

```
#pragma omp target teams loop 
for (int i=0; i<N; ++i)  
```

Not needed: parallel directive that can also generate parallelism
Best Practice #2

Make sure your loops are large enough to benefit from GPUs
Reminder: You need a lot of software parallelism to benefit from GPUs

Perlmutter’s NVIDIA A100

108 Streaming Multiprocessors (SM) *
64 warps per SM *
32 work items per warp =
Up to 221,184 active “threads”

Frontier’s AMD MI-250X

110 Compute Units (CU) *
40 wavefronts per CU *
64 work items per wavefront =
Up to 281,600 active “threads”
for each Graphical Compute Die (GCD)
Loops should have at least $O(10^4)$ iterations

The plot shows that 55K GPU threads are needed to get ~90% of NVIDIA A100 memory bandwidth.

Although not shown, there is a similar performance characteristic for AMD MI-250X.
The Collapse clause enables you to create larger parallel loops

- The OpenMP collapse clause specifies the number of loops to collapse
- In the Day 1 exercise, we **collapsed two loops** to enable parallelization of $n_{\text{cells}}^2$ iterations (critical for good utilization of GPUs):

```c
// Tuned Jacobi exercise. First target region: only 4 FLOP per loop iteration
#pragma omp target teams loop collapse(2)
for (unsigned i = 1; i <= n_cells; i++)
    for (unsigned j = 1; j <= n_cells; j++)
        T_new(i, j) = 0.25 * (T(i + 1, j) + T(i - 1, j) + T(i, j + 1) + T(i, j - 1));
```

[https://github.com/olcf/openmp-offload/tree/master/C/7-loop-combined](https://github.com/olcf/openmp-offload/tree/master/C/7-loop-combined)
Note: Kernel launch overhead can be significant when there is minimal work.

Tuned Jacobi. Performance of first target region on NVIDIA A100 GPU using “target teams loop”

The time to execute an OpenMP target region can be dominated by kernel launch time.

Mitigate launch overhead with more loop iterations.

Highly beneficial for our Jacobi kernel which has only 4 FLOP per loop iteration.

Higher is better.

~15x gain

221,184 threads
Best Practice #3

Minimize the separation between “teams” and “parallel” directives
Minimize the separation between “teams” and “parallel” directives

• A single combined directive often gives the best performance, e.g.

```
#pragma omp target teams distribute parallel for [simd]
```
  – All GPU threads are active inside the target region (SPMD execution)

• If you stray from a single combined directive you are more likely to run into compiler correctness or performance issues
  – Sometimes OK: “teams” and “parallel” directives in the same function
  – Problematic: “teams” and “parallel” directives in different functions but in the same compilation unit
  – Very problematic: “teams” and “parallel” directives in different compilation units
Performance may be poor when splitting “teams” and “parallel” directives

The current NVIDIA and Clang compilers don’t always deliver high performance with this code organization:

```c
#pragma omp target teams distribute
for (int i=0; i<N; ++i) {
    #pragma omp parallel for
    for (int j=0; j<N; ++j) {
        x[i][j] += 1.0;
    }
}
```

Tip for NVIDIA compiler: use the “loop” directive to get the highest performance:

```c
#pragma omp target teams loop
for (int i=0; i<N; ++i) {
    #pragma omp loop
    for (int j=0; j<N; ++j) {
        x[i][j] += 1.0;
    }
}
```
Best Practice #4

Use the family of “target data” directives to minimize data movement
Warning: Data movement can dominate runtime

Original Jacobi exercise: T and T_new are copied each time the kernel executes

```
#pragma omp target teams distribute parallel for simd collapse(2) \  
    map(T[:SIZE], T_new[:SIZE])
```

Nsight Systems profile on NVIDIA A100 GPU using n_cells=1000

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Start</th>
<th>Duration</th>
<th>GPU</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Memcpy HtoD</td>
<td>0.728472s</td>
<td>302.558 ms</td>
<td>GPU 0</td>
<td>Stream 13</td>
</tr>
<tr>
<td>2</td>
<td>nvKernel_gpu_teams_parallel_F1L81_2</td>
<td>0.728791s</td>
<td>13.312 ms</td>
<td>GPU 0</td>
<td>Stream 13</td>
</tr>
<tr>
<td>3</td>
<td>MemcpyDtoH</td>
<td>0.728816s</td>
<td>306.270 ms</td>
<td>GPU 0</td>
<td>Stream 13</td>
</tr>
</tbody>
</table>

Kernel time << data movement time to/from GPU !!!
“target data” directives enable us to minimize data movement

Original:

```c
while (residual > MAX_RESIDUAL && iteration <= max_iterations) {
    #pragma omp target teams distribute parallel for simd collapse(2) \
    map(T[:SIZE], T_new[:SIZE])
}
```

Tuned:

```c
#pragma omp target enter data map(to : T[:SIZE]) map(alloc : T_new[:SIZE])
while (residual > MAX_RESIDUAL && iteration <= max_iterations) {
    #pragma omp target teams distribute parallel for simd collapse(2)
```
Use runtime tracing to identify excess data movement (Cray compiler)

CRAY_ACC_DEBUG=2

ACC: Start transfer 2 items from jacobi.c:84
ACC: allocate, copy to acc 'T_new[:SIZE]' (8032032 bytes)
ACC: allocate, copy to acc 'T[:SIZE]' (8032032 bytes)
ACC: End transfer (to acc 16064064 bytes, to host 0 bytes)
ACC: Execute kernel __omp_offloading_93_282c1f03_kernel_gpu_teams_parallel_l84_cce$noloop$form blocks:7813 threads:128 from jacobi.c:84
ACC: Start transfer 2 items from jacobi.c:84
ACC: copy to host, free 'T[:SIZE]' (8032032 bytes)
ACC: copy to host, free 'T_new[:SIZE]' (8032032 bytes)
ACC: End transfer (to acc 0 bytes, to host 16064064 bytes)
Use runtime tracing to identify excess data movement (NVIDIA compiler)

NVCOMPILER_ACC_NOTIFY=3

upload CUDA data file=jacobi.c function=kernel_gpu_teams_parallel line=81 device=0 threadid=1
variable=T bytes=8032032

upload CUDA data file=jacobi.c function=kernel_gpu_teams_parallel line=81 device=0 threadid=1
variable=T_new bytes=8032032

launch CUDA kernel file=jacobi.c function=kernel_gpu_teams_parallel line=81 device=0 host-threadid=0
num_teams=0 thread_limit=0 kernelname=nvkernel_kernel_gpu_teams_parallel_F1L81_2
grid=<<<7813,1,1>>> block=<<<128,1,1>>> shmem=0b

download CUDA data file=jacobi.c function=kernel_gpu_teams_parallel line=88 device=0 threadid=1
variable=T_new bytes=8032032

download CUDA data file=jacobi.c function=kernel_gpu_teams_parallel line=88 device=0 threadid=1
variable=T bytes=8032032
Best Practice #5

Don’t map scalar variables (unnecessarily)
Specifying the variable as firstprivate is the most optimal

NOTE: Scalar variables are firstprivate by default on a target construct

```c
void scale_array(int scalar, double *x) {
  // Assume "x" already present on the GPU
  // #pragma omp target teams loop map(to:scalar)
  #pragma omp target teams loop firstprivate(scalar)
  for (int i=0; i<N; ++i) x[i] *= scalar;
}
```

The scalar is copied to GPU main memory using e.g. cudaMemcpy (slow 😞)

The scalar is passed as a kernel argument (fast 😊)
Avoid Fortran array operations and array sections (:) in target regions
Issues with Fortran array operations in target regions

```fortran
!$omp target teams distribute parallel do
  do i = 1, N
    x(:) = y(:) ! Error: Redundant execution by each GPU thread
    call mysub(x(i:1)) ! Performance issue: Array descriptor created by each GPU thread
  end do

!$omp target teams
!$omp parallel
!$omp workshare
  x(:) = y(:) ! Error: Workshared over threads only -- not teams
!$omp end workshare
!$omp end parallel
!$omp end target teams
```
Best Practice #7

Take advantage of compiler diagnostics and runtime tracing
All OpenMP compilers provide some compile-time diagnostics and/or tracing

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Diagnostics</th>
<th>Runtime tracing *</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA</td>
<td>-Minfo=mp,accel</td>
<td>NVCOMPILER_ACC_NOTIFY=3</td>
</tr>
<tr>
<td>Cray</td>
<td></td>
<td>CRAY_ACC_DEBUG=3</td>
</tr>
<tr>
<td>Clang</td>
<td>-Rpass=openmp-opt</td>
<td>LIBOMPTARGET_INFO=-1</td>
</tr>
<tr>
<td>GNU</td>
<td></td>
<td>GOMP_DEBUG=1</td>
</tr>
</tbody>
</table>

Tells you how the compiler is parallelizing loops over teams and threads

Shows you a timeline of data movement and kernel execution. This allows you to check that program flow matches your expectations

* Generally multiple trace values – see documentation
Thank You