基于 DCU 的 OpenMP Offload 多卡编程实践

张文帅

中国科学技术大学 超级计算中心

随着 OpenMP Offload 技术的发展，特别是相关的 OpenACC 技术证明了该方向的显著编程优势以及性能方面劣势的补足，该技术方向已经逐渐脱离了简单的导语编程模式，逐渐具备了跟原生编程技术同样的编程控制力与性能，本文简单的介绍下基于 DCU 的 OpenMP Offload 编程经验，特别是在针对多卡计算等场景，给出一个编程示范，并讨论两个技术建议。

(一) 一个最简单的 OpenMP Offload 程序示例

本示例中，简单的计算一个数组 x 乘以一个标量 a 的结果，并记为 y 。具体的代码见本文最后面的附件中。

核心的代码如下所示：

```
#pragma omp target device(0) map(tofrom: x[0:n],y[0:n])
{
#pragma omp parallel for
for (int i=0;i<n;i++){
    y[i] = a * x[i];
}
```

其中，pragma omp target device(0) map(tofrom: x[0:n],y[0:n],a) 依次的表示，启动一个 OpenMP Offload target 任务，使用 0 号 device，将主机内存中的 x、y 变量在计算前复制到设备中并在计算完成后立刻复制回主机内存。其中，变量 a 没有被指定，但会被自动识别进行双向数据传输，其实 x、y 也是不必被指定的，这体现了 OpenMP 语法确实可以简化与隐藏很多的细节操作。

第二行的 pragma omp parallel for 表示对以下 for 循环在设备执行并行的计算。

这是一个最简单的示例，代码也很少，因为 OpenMP 技术隐藏了背后的很多操作细节，提供了简洁的编程，但同时也限制了功能，特别是限制了优化性能所必须的细节控制能力，最显著的是，我们可能需要一些数据按需的复制到内存，同时再多次 Offload 后，再回传到主机内存，因此我们需要自主的控制数据的传输。

（二）改进：通过 omp_target_memcpy 自主的控制数据传输

控制数据传输并不总是需要 omp_target_memcpy 函数，实际上，可以通过如下 target enter date map 语句来在设备上建立同样变量名的数据，并通过 target update 来随时的

双向传输数据：

```
#pragma omp target enter data map(alloc: x[10*idev:10*idev+10], y[10*idev:10*idev+10], a) device(idev)
#pragma omp target update from(y[c:c+CHUNKSZ])  device(idev)
```

该方法应该是一个好的技术趋势，但是其隐藏了设备变量，不利于清晰的完成多设备关联的数据控制，因此本文使用更加底层的函数 omp_target_memcpy 来示例。

为了做主机与设备间的数据传输，首先需要建立设备上的变量，并分配空间：

```
double *y_d0;
double *x_d0;
double *a_d0;
x_d0 = (double *) omp_target_alloc( sizeof(double) * n/ndev, 0);
y_d0 = (double *) omp_target_alloc( sizeof(double) * n/ndev, 0);
a_d0 = (double *) omp_target_alloc( sizeof(double) , 0);
double *y_d1;
double *x_d1;
double *a_d1;
x_d1 = (double *) omp_target_alloc( sizeof(double) * n/ndev, 1);
y_d1 = (double *) omp_target_alloc( sizeof(double) * n/ndev, 1);
a_d1 = (double *) omp_target_alloc( sizeof(double) , 1);
```

如上所示，omp_target_alloc 可以方便的帮助分配数据空间，之后可以使用 omp_target_memcpy 来显式的在任意需要时拷贝数据。

```
omp_target_memcpy( x_d0, &x[ipart*CHUNKSZ], sizeof(double)*CHUNKSZ, sizeof(double)*deviC*CHUNKSZ, 0, idev, hdev);
omp_target_memcpy( &x[ipart*CHUNKSZ], x_d0, sizeof(double)*CHUNKSZ, 0, sizeof(double)*deviC*CHUNKSZ, hdev, idev);
omp_target_memcpy( a_d0, &a, sizeof(double), 0, 0, idev, hdev);
```

以上分别是，将 x 从主机内存 hdev 拷贝到设备 idev，将 x 从设备 idev 拷贝到主机内存 hdev，将 a 变量从主机内存 hdev 拷贝到设备 idev，其中 hdev 是通过 int hdev = omp_get_initial_device() 函数来获取，idev 分别是从 0 开始的 GPU 卡编号。

此外，程序在标记 OpenMP 计算时，还需要通过 is_device_ptr 指定 x_d0, y_d0, a_d0 是设备指针，完整的代码段为：

```
>     #pragma omp target device(idev) is_device_ptr( x_d0, y_d0, a_d0 )
>     {
>     #pragma omp teams num_teams(2) thread_limit(5)
>     {
>                 #pragma omp distribute parallel for dist_schedule(static,10)
schedule(static,1)
>       for (i = deviC*ChunkSize; i < deviC*ChunkSize+ChunkSize; i++){
>         y_d0[i] = a_d0[0] * x_d0[i];
>       }
>     }
>     }
```

其中，变量后缀 d0 表达了该变量为设备 0 中的指针，最后一行将于设备中新计算得到的 y_d0 数据传输回主机内存。

（三）优化 Offload 计算中线程分组以及与 loop 计算任务的对应关系

线程会被分组，单组的线程数量由 thread_limit 限制，组数量由 num_teams 限制，更具体的规则不被 OpenMP 规则所明确，由各个实现决定，一般情况下此两个数据指定了程序使用的组数量与线程数量。每个组具有一个主线程，可以独立执行任务，例如使用 #pragma omp master 可以开启一段计算，team 内的除主线程之外的其他线程均不会参与计算。

如下是一个较完整的示例：

```
>   #pragma omp target device(idev) is_device_ptr( x_d0, y_d0, a_d0 )
>   {
>   #pragma omp teams num_teams(2) thread_limit(5)
>     {
>     #pragma omp distribute parallel for dist_schedule(static,10) schedule(static,1)
>     for (i = deviC*ChunkSize; i < deviC*ChunkSize+ChunkSize; i++){
>         y_d0[i] = a_d0[0] * x_d0[i];
>     }
>   }
```

其中，distribute 指示将 loop 任务分配给各个组，dist_schedule 指示了 loop 任务分配给各个组的分配策略，schedule 指示了将 team 组中的 loop 任务分配给各个组内 threads 的分配方式。

这些配置参数显著的提升了开发者对计算的控制力，特别是调整这部分划分与分配规则，

可以提升对数据的访问效率，对计算速度会起到很大的影响，应该根据具体问题进行具体的配置。

（四）编译与执行示例

下载附件后，首先在昆山中心申请一个交互式节点，载入 DTK 环境 compiler/rocm/dtk-22.10.1，然后采用 hipcc 来编译，编译命令为 hipcc -target x86_64-pc-linux-gnu -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=gfx906 main.c -o multi-dev.x，其已经被封装到 Makefile 中，然后./multi-dev.x 来执行。
简单的操作如下：

```
$ salloc -p kshdtest -J test --time=0-0:30:00 -N 1 -n 1 --cpus-per-task=2 --gres=dcu:2 --qos=partition_hgdcutest srun --pty bash
$ module purge
$ module load compiler/rocm/dtk-22.10.1 2>&1
$ make clean
$ make
$ export OMP_TARGET_OFFLOAD=MANDATORY
$ export LIBOMPTARGET_KERNEL_TRACE=1
$ make test
```

其中，在 make test 执行测试之前，通过配置 OMP_TARGET_OFFLOAD 环境变量可以避免代码在 CPU 中执行，通过设置 LIBOMPTARGET_KERNEL_TRACE 环境变量，可以使输出更详细的 Kernel 执行信息，包括启用的 team 数，以及每个 team 中的 thread 数。

（五）讨论

本文演示了一个简单的 OpenMP Offload 程序，不过其中采用了较底层的函数 API，可以为开发者提供接近原生编程语言的控制力。

在扩展到多卡的过程中，需要对不同卡上的数据分配独立的变量，以后缀“_d0”、“_d1”来区分，以便用 is_device_ptr 来标记设备指针，这导致如下的多卡编程方式：

```
>   if (idev==0){
>     #pragma omp target device(idev) is_device_ptr( x_d0, y_d0, a_d0 )
>     ...
>   }
>   else if (idev==1) {
>     #pragma omp target device(idev) is_device_ptr( x_d1, y_d1, a_d1 )
>     ...
```

```
>   }
```

在成熟的开发中，我们希望对不同卡进行统一的编程，使用一个指针数组变量 x_d[idev] 来统一描述多个设备上的数据，例如：

```
>   for (int idev=0;idev<ndev;idev++) {
>           x_d[idev] = (double *) omp_target_alloc( sizeof(double) * n/ndev, idev);
>           y_d[idev] = (double *) omp_target_alloc( sizeof(double) * n/ndev, idev);
>           a_d[idev] = (double *) omp_target_alloc( sizeof(double) , idev);
>   ...
>               #pragma omp target device(idev) is_device_ptr( x_d[idev], y_d[idev], a_d[idev] )
>   ...
>   }
```

如上模式需要底层工具的进步支持。

此外，当前的 DCU 开发支持的 OpenMP4.5 规范中，没有异步的内存拷贝 API，导致代码编写过程中需要将更多的数据存储在 GPU 中，这对一部分逻辑处理代码的编写不太友好，也是主流的 OpenMP 与 OpenACC 编译器实现之间的一个关键区别，特别是导致一些隐式数据传输的 offload 代码之间具有较大的速度差异。OpenMP5.1 中规范了新的 API：omp_target_memcpy_async 来提供异步拷贝的支持，弥补了 OpenMP 之于 OpenACC 的关键短板，也同样会大大缩小 OpenMP 跟原生的 GPU 开发得到的代码之间的速度差异。

未来，在改进如上两个技术支持后，OpenMP Offload 技术应该可以扩展其应用范围，获得更多用户与开发者的支持。因为当前最先进的 GPU 单节点已经具备非常强大的性能，足够多数用户的正常计算需求，因此高效的开发方法可以使得更多的计算过程被快速的 GPU 化，使得采用相同的人力时间与工作量可以获得更好的程序性能，这在一部分成熟的应用（如 VASP）中已有所体现。

考虑到 CPU 算例相比 GPU 算例的巨大差距，节点内的 CPU 算力可以被开发者忽略，OpenMP 也将可以基于 GPU-GPU 直接传输的通信库做到多节点的高效互联计算。