OpenACC using Colab

Sarah

December 5, 2022

1 Part 1: Introduction

This document offers an introduction to porting algorithms on GPU with OpenACC using Colab.

2 Part 2: Running a Fortran code in Colab

2.1 About Colab

Colab is a free cloud service proposed by Google based on the Web Open Source application Jupyter-Notebook.

2.1.1 Environment

Here, we work exclusively with Fortran source code. In Colab, we can see that the GNU Fortran Compiler (gfortran) is already installed and we can check which version is available:

```
[16] !gfortran --version

     GNU Fortran (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
     Copyright (C) 2017 Free Software Foundation, Inc.
     This is free software; see the source for copying conditions.  There is NO
     warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Also, Colab offer a free access to GPUs. We can have a look at the monitoring and management capabilities of the devices by executing the `nvidia-smi` command.

```
Tue Nov 29 15:40:37 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   46C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

2.1.2 Compilation and Execution

Here, we use the tutorial proposed by
Compilation flag To use the OpenACC directives, one should inform the compiler by adding the special flag. For the GNU compiler, the `-fopenacc` flag is required.
Directives Inside the code source, the use of OpenACC directives allow to warn the compute regions that should be performed on the GPU device, and enable the GPU-acceleration.

```
#ifdef _OPENACC
       use openacc
#endif
```

2.1.3 Preview on the CPU code

Before thinking about optimizing an application, one should first make a performance-profiling and find hotspots in the source code. In other words, the idea is to get information about the regions which take a significant percentage of the runtime.
Let's note that in Fortran language, considering a two dimensions array, the reading is made column by line.

2.1.4 OpenACC

First, we could ask why using OpenACC rather than CUDA to port a GPU implementation. The reason is because OpenACC requires less rewriting of code and allow to keep the framework of the source code unchanged and maintain the sequential code.
Also, OpenACC is interoperable with CUDA, which means that later on, one could just add CUDA in some part of the code to adjust regarding some needing performance.
Most of what will be presented here comes from this tutorial.
OpenACC is a directive-bases parallel programming approach used to accelerate in an easy way applications.
One of the limitations using Colab is the timeout of 90 minutes which means that the files uploaded to work with will not be available passing this time.

```
!nvidia-smi
```

Here the outcome of this command:

```
Tue Nov 29 15:40:37 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   46C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

2.1.5 OpenACC Syntax

Only the Fortran syntax will be presented here, but one could easily find the C/C++ Syntax if working with these langages.
The directives are added to a serial code using the syntax:

```
!\$acc \textit{directive clauses}
<code>
```

This syntax is formatted as comments and will be read only if we add the compiler flag `fopenacc` to enable OpenACC. If not enabled the directives will be treated as comment by the compiler.
These directives will allow to inform the compiler how to manage loop parallelization for the computing, but also to manage data transfer between CPU and GPU memory. The latter can be very time consuming as we will see later. And finally, the **Clauses** are here attached to the directives for more specifications.

2.2 Porting to GPU using OpenACC

In this section, we will use an existing program which perform a simple addition between two vectors.

```
program vectorsum
#ifdef _OPENACC
  use openacc
#endif
  implicit none
  integer, parameter :: rk = selected_real_kind(12)
  integer, parameter :: ik = selected_int_kind(9)
  integer, parameter :: nx = 102400

  real(kind=rk), dimension(nx) :: vecA, vecB, vecC
  real(kind=rk) :: sum
  integer(kind=ik) :: i

  ! Initialization of vectors
  do i = 1, nx
     vecA(i) = 1.0_rk/(real(nx - i + 1, kind=rk))
     vecB(i) = vecA(i)**2
  end do

  ! Addition
  do i = 1, nx
     vecC(i) = vecA(i) + vecB(i)
  end do

  ! Compute the check value
  write(*,*) 'Reduction sum: ', sum(vecC)

end program vectorsum
```

2.2.1 Time profiling

In most ressources, some powerful tools are used to perform a an analysis of the time consumption. These tools are part of the PGI compiler. In our case, we are using a GNU Fortran compiler, therefore, we opted for the use of the basic routines `cpu_time` function to highlight the hotspots of our code.
A simple implementation of a routine can give us access to some relevant information ; also some instructions have been added to save them in an output file. The results look like:

```
1  ------------------------------------------------------------------------------
2  |                |   Initialization  |  Vector addition |        Sum         |
3  ------------------------------------------------------------------------------
4   Time consumed   |       2.140       |       0.884      |        0.432       |
5       (ms)
6  ------------------------------------------------------------------------------
7      Percentage   |      61.725       |      25.498      |       12.460       |
8  ------------------------------------------------------------------------------
9
10    Execution Time |       3.467
```

2.2.2 Loops parallelization

2.2.3 Data optimization

2.2.4 Loop optimization

2.3 Index

`gang` : Depending the architecture, `gang` can have different meanings. On a multicore CPU, `gang` means generally a thread. As for a GPU, `gang` means a thread block. The idea behind it is that `gang` is the outer-most level of parallelism for any architecture.