OpenACC Hands-on

Ruymán Reyes Castro [*], J. Lucas Grillo [†]

March 12, 2013

1 Firsts steps

1.1 Accessing the Cluster

The practical sessions require a system with the appropriate compilers and GPU devices. The High Performance Computing Group from the Universitat Jaume I (UJI) has kindly provided us with access to their `tintorrum` cluster. We are going to use the `cuda` queue, which contains 9 nodes with two Intel Xeon E5520 quad-core processors (total of 8 cores @ 2.27 GHz) and a Tesla C2050 GPU. The nodes are connected via Infiniband QDR (Mellanox MTS3600 switch)

Since the cluster is not directly connected to the internet, we need to use an SSH gateway (Lorca) to access to the internal UJI network. The easiest way to access to `tintorrum` is to create an ssh tunnel from your computer to the target computer (using Lorca).

1. Get the DSA key

```
# Get the DSA key
$ wget -O gpgpu_dsa https://www.dropbox.com/s/7wha3fa5dcd7bjp/gpgpu_dsa
$ chmod 400 gpgpu_dsa
```

2. Launch the following command in your computer (not in Lorca):

```
ssh -i gpgpu_dsa -L 20022:tintorrum.act.uji.es:22 capap@lorca.act.uji.es
```

3. Now you can connect to localhost:20022 as it was **tintorrum** (leave the previous connection open and use another terminal).

```
ssh -p 20022 -XC {username}@localhost
```

[*]EPCC - University of Edinburgh (`rreyesc@epcc.ed.ac.uk`)
[†]GCAP - University of La Laguna

If everything went OK, we should now be logged into `tintorrum` (hostname: **tintorrum.act.uji.es**) To copy files in/out the cluster, we can use the `-P` option of `scp`.

```
# Copy from the cluster
  scp -P 20022 {usuario}@localhost:file  .
# Copy to the cluster
  scp -P 20022  file usuario@localhost:.
```

1.2 Compiling the Hello World

This first exercise only pretends to familiarize the user with the labs environment. All the source codes of the exercises will be written and built in the frontal node of `tintorrum`. However, unless otherwise specified, the codes *must be* run using the queue system.

Several different compilers implement the OpenACC standard. In this hands-on session we will use the following three implementations:

- CAPS OpenACC 3.3.2
- PGI version 13.2

First of all, copy the file `ejercicios.tgz` from the shared directory `/nas/openacc/`into your home folder. This compressed tar file contains all the exercises and their solutions. After uncompressing it on your home folder, you will have a directory named `ejercicios` which contains several directories named `ej01`, `ej02`, ...

First of all, load all the environment variables that you may require by typing `source load.sh`.

In the directory `ej01` you will find the first example. Open the file HELLO.C with your favorite editor. It is a very simple OpenACC example, which assigns a value to each position of an array, and checks that it did the same on both the CPU and the GPU. Notice that we are using the `_OPENACC` macro to differentiate when we are building with OpenACC support and when not.

Now close the source file and open the Makefile. In order to use the CAPS OpenACC compiler to build the source. The CAPS compiler is acts as a wrapper in front of `gcc`. The options in front of `gcc` are CAPS options. Type `capsmc --help` to see available options and some usage examples. The `-f` option we are using rewrites the temporary codelets to avoid having several different versions of the same kernel laying in the directory.

The PGI C compiler is run using the `pgcc` command. In this case, PGI is not a wrapper but a full compiler, and it does not generate itermediate files (unless requested). Type `pgcc -help` to see available options and some usage examples. To enable compilation with OpenACC support, add `-acc` to the list of options. Multiple targets are supported (e.g compiling OpenACC to run on the CPU). To ensure generation of GPU code add `-ta=nvidia` to the command line.

The makefile provided takes care of all this options. Type *make* inside the directory to build both the sequential and the various OpenACC versions.

You should not execute the exercises on the frontend node or you may experience random failures when other users are doing the same as you. Use the queue system to submit jobs.

The file SCRIPT.SH in the directory is an example batch submission script that executes the host and GPU versions of the code and stores the output on two text files. Submit it using `qsub script.sh`. Check the files OUTPUT_PGI, OUTPUT_CAPS and OUTPUT_HOST and verify that everything worked as expected.

2 Basic OpenACC

The directory `ej02` contains the files used in this exercise. The file JACOBI.C feature a simplified implementation of a Jacobi iterative solver, which solves the Laplace equation in 2D ($\nabla^2 f(x, y) = 0$) [1]. The main loop is shown below.

```
while ( error > tol && iter < iter_max )
{
    error = 0.0;
    // Loop Nest A
    for( int j = 1; j < n-1; j++)  // A.a
        for( int i = 1; i < m-1; i++ ) // A.b
        {
            Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                + A[j-1][i] + A[j+1][i]);
            error = max( error, fabs(Anew[j][i] - A[j][i]));
        }
    // Loop Nest B
    for( int j = 1; j < n-1; j++)   // B.b
        for( int i = 1; i < m-1; i++ ) // B.b
            A[j][i] = Anew[j][i];
    if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
    iter++;
}
```

The while loop features two loop nests, highlighted as A and B. Loop nest B is a trivial update loop, which copies the values from *Anew* back to *A*, whilst Loop nest A updates the values of each position of the *Anew* matrix in terms of the values of the neighbors in the matrix *A*. It also computes the maximum absolute error between the two matrices. The while loop will stop if the maximum number of iterations is achieved (*iter_max*) or if the error falls below the specified tolerance.

1. Compile the sequential version of the code (`make host`). Execute it on the host and familiarize yourself with the output.

[1] This example is based on the NVIDIA/PGI tutorials. `http://tinyurl.com/d4sxul7`

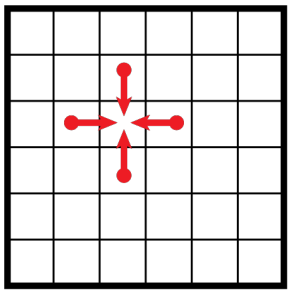

Figure 1: Each cell uses its neighbor values to compute the new value.

2.1 Offloading code to the accelerator

The easiest way to execute a piece of code on the accelerator is to use the `kernels loop` directive. In this example, both A and B loops can be offloaded to the GPU.

1. Apply the `kernels loop` directive to both loops and ensure the result is correct. What happened with the execution time?

2.2 Profiling

Since there is no point in using a programming language for accelerators that runs your code slower, we will use the NVIDIA command line profiler (NVPROF[2]) to analyze the output in an attempt to identify the bottleneck. The code generated by OpenACC compilers for CUDA platforms *is still CUDA code* so traditional tools can be used with it. In this case, for simplicity, we will use only the CAPS OpenACC compiler.

Modify your script file so the execution line looks like the following: `nvprof ./jacobi.acc.cuda.exe &> output_cuda_profiler`

1. Get the profiling information from your code. Identify the major bottleneck of the implementation.

   The output of your profiling output should look similar to the following:

```
======== NVPROF is profiling jacobi.acc.cuda.exe...
======== Command: jacobi.acc.cuda.exe
Jacobi relaxation Calculation: 4096 x 4096 mesh
    0, 0.250000
  100, 0.002363
  200, 0.001204
  300, 0.000804
  400, 0.000603
  500, 0.000483
```

[2] `http://docs.nvidia.com/cuda/profiler-users-guide/index.html`

```
  600, 0.000403
  700, 0.000345
  800, 0.000302
  900, 0.000269
 total: 456.639148 s
======== Profiling result:
 Time(%)      Time   Calls       Avg       Min       Max  Name
   43.66   192.51s    6000   32.08ms    1.86us   57.71ms  [CUDA memcpy DtoH]
   39.13   172.52s    5000   34.50ms    1.44us   53.66ms  [CUDA memcpy HtoD]
   16.00    70.55s    1000   70.55ms   67.56ms   73.07ms  __hmpp_acc_XXXX
    1.21     5.32s    1000    5.32ms    5.28ms    5.37ms  __hmpp_acc_XXXX
```

Notice that more than 70% of the time is invested in memory transfers, and only a 17% in real computation.

2.2.1 Profiling with the PGI compiler

The PGI compiler is able to provide GPU usage statistics by itself. To enable them, add the line `export PGI_ACC_TIME=1` to your batch script, just before executing the GPU version.

1. How much information is being copied to the device?
2. How much information is being copied from the device?

2.3 Creating a data region

Both A and B loops are enclosed within a while loop. Since each `kernels` directive creates its own implicit data region, every time we encounter any kernel directive we transfer all the required variables inside, and then copy everything outside at the end. In order to reduce the amounts of memory transfers, we need to create an explicit data region.

1. Search for the correct place in the code to put the data region and define which variables will be required. Then compile/run again to check for correctness and performance.
2. Profile the application again and compare the timing figures.

2.4 What is going on under the hood?

In some situations it may be of interest to have more details on what the runtime or the compiler is doing.

2.4.1 PGI compiler

It is possible to view the CUDA code generated by the compiler. Change the `PGCC_FLAGS` in the makefile so it looks like the following and recompile.

Option	Loop A.a	Loop A.b	Time CAPS (s)	Time PGI (s)
1	gang(64) worker(64)			
2	gang(128) worker(128)			
3	gang(256)	worker(256)		
4	gang(128)	worker(128)		
5		gang(256)		
6				

```
PGCC_FLAGS=-acc -ta=nvidia,keepgpu -Minfo
```

If you list the content of the directory after building the sources, you will see the intermediate files It is also possible also to see runtime information regarding whenever the accelerator is used. This is accomplished using the `ACC_NOTIFY` or `PGI_ACC_NOTIFY` variables.

1. Open the intermediate file and try to identify the kernel code.
2. Add the variable to the script file and identify when the kernels are executed.

2.4.2 CAPS OpenACC compiler

The CAPS OpenACC compiler keeps the intermediate files by default. Each generated kernel is stored on a separated file, together with the code required to execute it.

To view detailed execution information (e.g, when a kernel is loaded or memory is being transferred), you can use the `HMPPRT_LOG_LEVEL` environment variable.

```
HMPPRT_LOG_LEVEL=info ./program_name
```

1. Add the variable to the script file and identify when the kernels are executed.
2. Open the intermediate file and try to identify the kernel code.

2.5 Playing around with gangs and workers

Now that the bottleneck is no longer the memory transfers, we can fine tune the performance of the kernels by exploring different values of gangs and worker clauses.

1. Try to find the optimal values of gang/workers for Loop A. The following table includes a set of suggestions, but feel free to try different combinations.
2. Is it the best combination of gang and workers the same for both compilers?

Be careful! Some options might cause the compiler to crash or the final executable may produce incorrect results

3 Slightly less basic OpenACC

```
1  int main(...) {
2    ...
3    // Initial energy calculation
4    compute(position, velocity, mass, force, &pe, &ke);
5    ...
6    // (S) Simulation
7    for (i = 0; i < NSTEPS; i++) {
8      compute(position, velocity, mass, force, &pe, &ke);
9      // Compute error
10     ....
11     printf(..., pe, ke);
12     update (position, velocity, mass, force, &pe, &ke);
13   }
14   ...
15 }
16 void compute(...) {
17     // (C) Compute forces
18     for (...) {
19       .... compute pe, ke
20     }
21 }
22 void update(...) {
23     // (U) Update velocity/position
24     for (...)
25         ... update pos, vel, mass
26 }
```

Listing 1: Sketch of MD simulation

Given positions, masses and velocities of `np` particles, the pseudo code shown in Listing 1 computes the energy of the system and the forces on each particle.

The code found in `ej03` is a C implementation of a simple Molecular Dynamics (MD) simulation. It employs an iterative numerical procedure to obtain an approximate solution whose accuracy is determined by the time step of the simulation. Particles are represented by three three-dimensional double precision matrices: Position, Velocity and Force (parameters). Rows of each matrix represent a particle, whereas columns represent a dimension. For example, the coordinate $\{3, 1\}$ contains the parameter value for the particle number three in dimension one.

After an initial forces computation, on each simulation step, the algorithm performs two basic operations: *compute* (C) and *update* (U). C operation consists of several nested loops computing the forces for each position. An external loop iterates over all particles computing its forces in the current simulation step. This requires computing the distance among all other particles, hence accessing the `position` matrix, and computes the total potential and kinetic energy of the system, which requires access to the `velocity` matrix. In terms of the data access pattern, the code is highly un-coalesced, requiring several non-contiguous loads to compute each particle. In addition, it features several costly double precision operations (*sqrt*, *sin* and *cos*) which traditionally perform badly on GPU devices. The U operation

is simply a for loop that runs over the particles, updating their positions, velocities and accelerations. C is more compute–intensive than U.

3.1 Multiple offload regions

Both C and U routines are suitable to be executed on the GPU. Since the C routine is more compute intensive, it should be our first optimization target.

Sometimes, when porting applications to OpenACC, the compiler needs some help refactoring the code before being able to generate the CUDA kernel. In this case, the main loop of the C routine contains a call to a function. This calls are typically inlined by the compiler. However, in this case the parameters to this functions are pointers. Pointer addressing is not supported on the majority of implementations. Although this may be solved in the future, or for some simple cases, it is a general good idea to avoid pointer addressing inside offloaded code. This simplifies the work of the compiler, generating a better dependency analysis. It is important to take into account that, although most GPU accelerators nowadays support the majority of the IEEE floating-point standard, they do not support all the rounding modes. This implies that some operations (particularly square root) may not deliver full precision results.

For this first set of questions, we will use 8192 particles and 10 simulation steps.

Fill the table below with the time of each of the following combinations (and any other you come up). Notice that there is a column for the Host as well. Some code modifications may affect (or benefit) serial performance!

1. Try to offload the main computational part of the C routine. A *kernels loop* directive on the outermost loop could be a good idea. Check the compiler output for hints on potential problems.

2. Manually inline the `dist` routine into the main computational loop of the C routine. Verify the results.

3. What happen to the magnitude of the error (third column) when running the program?

4. How many different kernels were generated by the PGI compiler? And by the CAPS OpenACC?

5. Try to manually unroll the loops within the C loop. **Suggestion**: Decompose `rij` into three variables (`rij_x`,`rij_y`,`rij_z`). Check correctness and measure performance.

6. If you look closely, within the `C` loop there are three different nested loops. Try splitting those loops into different kernel regions. This may allow you to use nested loop clauses and generate finer-grainer kernels. As usual, ensure correctness and see the performance. **Suggestion**: Sometimes compilers may have problems with reductions, particularly if the same reduction appears on different branches of a conditional sentence. Try to put reduction operations on a statement outside the branch.

7. Explore various gang/workers/vector combinations. Check the results - some combinations may produce incorrect results on some compilers.

Option	Time Host (s)	Time CAPS (s)	Time PGI (s)
1			
2			
3			
4			
5			
6			

3.2 Orphaned directives

Although the majority of the time in this example is spent on the computation part, we are going to explore other features of the OpenACC language that are useful for larger source codes. If you take a look to the general structure of the source, the compute and update parts are in different routines. However, both routines use the same data. If we only offload the computation of the `C` routine, we need to copy in and out the data from the CPU to the GPU and back in each simulation step. This is not desirable, particularly if we have to transfer large matrices to the device. OpenACC directives may be orphaned, increasing the expressiveness of the language.

1. Create a data region that includes the main simulation loop
2. Replace the *copy* clauses with *pcopy* clauses
3. Ensure correctness and measure performance
4. Use the profiler to check the amount of memory transfers have been reduced. **Note:** Depending on the compiler used and its version, the effect could be different. As always, check results.
5. Modify the source code so the program writes to the screen the force vector in each simulation step. Do it by adding a print statement on the main simulation loop. **Suggestion:** An update clause may be needed. Remember that you can use the indexing features to specify a contiguous section of an array.

4 Interacting with native CUDA applications

One of the most interesting features of OpenACC is its ability to interact with existing accelerator code, such as libraries or handwritten codes. The files in the `ej04` directory represent easy examples on this usage. The file `mxm_deviceptr` uses `acc_malloc` to allocate memory on the accelerator, and then uses it on the accelerator regions across the code. Note that `acc_malloc` is completely transparent to the platform we are working on, so this code will work on any kind of accelerator.

The file `cublas` is a modified example of the `Sgemm` routine from the `CUBLAS` library.

1. This exercise would be too simple if nothing else would be required. A ghost in the shell deleted some clauses from the above mentioned files, so they do not compile now. Fix them to ensure they build and the output is correct.

5 Various advanced topics

In the directory `ej05` there are two files: GEMM.C - a very simple implementation of the general (square) matrix multiplication, and DRIVER.C - a program that performs three matrix multiplications and check the results. You can specify the problem size as a parameter when running the program.

The driver program also contains a `fill` routine that puts data into the matrix, and a `identity` routine that creates identity matrices.

The program checks for correctness at the end, and returns Ok with the time required to compute the main part if is successful.

1. Add the required directives so that `fill`, `identity` and `gemm` routines are offloaded to the accelerator. Check that everything works OK.

2. Create a data directive in the main program enclosing the calls to the above mentioned routines. **Note:** Check that the copy clauses of the orphaned directives are `present_or_???` to avoid producing incorrect results.

In this exercise we are going to use the OpenACC **if** clause to illustrate how easily we can overlap computation between the host and the accelerator. The computation of the two first matrix multiplication operations is independent, and it can be done in parallel.

1. Using the `if` clause on the gemm routine, modify the source so that the first multiplication runs on the CPU and the second one on the GPU. **Note:** You may need to add `update` clauses to keep the memory consistency.

2. Use the `async` and `wait` clauses to asynchronously offload the computation of the second multiplication to the accelerator. You will need to ensure that all the data for this operation is on the accelerator, and that the results of both multiplications are ready for the third operation to be executed.

If combining the Host CPU with the GPU is still slower than using only the GPU, do not loose hope. It is possible to combine OpenMP with OpenACC in the same source file, although the behaviour of putting together OpenMP and OpenACC directives is compiler dependant. For example: PGI uses OpenMP with OpenACC to implement multiple device support (one device per thread).

1. Create an OpenMP parallel region within the gemm. Use the `if` clause of OpenMP to enable it when the OpenACC region is disabled. **Note: Do not panic if it does not compile or hangs when running. This feature is not fully supported**.