Parallel Programming with OpenACC

For today's session you need to copy the files located at Escritorio/alumnos/OpenACC into your home directory.

1. Checking the system

Firstly, we will check the GPU resources available in the cluster. All the codes have to be compiled with pgcc and run on the nodes equipped with GPUs. Execution has to be done mostly in batch mode, i.e. applications will run in a different node from the one you are logged in and equipped with a modern GPU. The execution will be managed by a batch queue system (SLURM) that grants conflict-free access to available execution nodes.

SLURM requires a script files that both, specifies job requirements and includes the application that has to be executed.

Our basic submission file (*openacc.sub*) is as follows:

```
1. #!/bin/bash
2. #
3. #SBATCH --job-name=gpu_check
4. #SBATCH -N 1 # number of nodes
5. #SBATCH -n 1 # number of cores TOTAL
6. #SBATCH --partition=cuda.q
7. #SBATCH --nodelist=aolin23

8. hostname

9. source /soft/modules-3.2.10/Modules/3.2.10/init/bash

10.  module load pgi/19.4
11.  module add cuda/9.0

12.  pgaccelinfo

13.  exit 0
```

Firstly, the job provides basic definitions and requirements that are managed by SLURM (regard that these lines constitute commands from a bash interpreter point of view): its name is *gpu-check* (line 3), it asks for one node with one core (lines 4 and 5), it is submitted to the queue *cuda.q* (the one that includes all machines with modern GPUs) and, in particular, requests the node with name *aolin23*. Afterwards, the commands executed as part of the job follow: first, it requests the name of the host where the job is running, it sets up some GPU-related environment variables (line9, line 10 and 11) and, finally, runs a command that requests information about GPUs available in the system.

The above script is submitted to execution with command *sbatch*:

```
aa-X-Y@aolin-login:~/home$ sbatch openacc.sub
Submitted batch job 4382
```

You should be able to check its progress using *squeue*.

```
aa-X-Y@@aolin-login:~/home$ squeue
JOBID PARTITION  NAME USER   ST TIME  NODES NODELIST(REASON)
4382 cuda.  gpu-check aa-X-Y  R 0:07   1 aolin23
```

Jobs will run very fast. So, it might be possible that jobs have already finished at the time you run the *squeue* command and you don't see your job listed.

By default, job output is redirected to a file named slurm-<job id>.out

If needed, a job is removed from the queue using the *scancel* command:

```
aa-X-Y@@aolin-login:~/home$ scancel 4382
```

Information about batch queues available at the lab can be queried with the *sinfo* command. *Try it.* As you see, the cuda.q partition contains three nodes (aolin23, aolin24 and aomaster). *Submit the openacc.sub and look at its output.*

The output of the execution should look like:

```
aolin23.uab.es

CUDA Driver Version:           9020
NVRM version:                  NVIDIA UNIX x86_64 Kernel
Module  396.37  Tue Jun 12 13:47:27 PDT 2018

Device Number:                 0
Device Name:                   GeForce GTX 1080
Device Revision Number:        6.1
Global Memory Size:            8513978368
Number of Multiprocessors:     20
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    1733 MHz
Execution Timeout:             No
```

```
Integrated Device:              No
Can Map Host Memory:            Yes
Compute Mode:                   default
Concurrent Kernels:             Yes
ECC Enabled:                    No
Memory Clock Rate:              5005 MHz
Memory Bus Width:               256 bits
L2 Cache Size:                  2097152 bytes
Max Threads Per SMP:            2048
Async Engines:                  2
Unified Addressing:             Yes
Managed Memory:                 Yes
Concurrent Managed Memory:      Yes
Preemption Supported:           Yes
Cooperative Launch:             Yes
  Multi-Device:                 Yes
PGI Default Target:             -ta=tesla:cc60
```

Answer the following questions:

- *How many GPUs are available in this node?*
- *Are they all the same?*
- *How many SMs (multiprocessors) does each GPU have?*

The information about the GPU also includes information for the PGI compiler if we want to generate code for a particular architecture. In particular, the tesla:cc60 specifies the compute capability of the target architecture. For exemple, with PGI 19.4 on a system with CUDA 9.0 driver (our lab configuration), the compiler may target Kepler (cc35), Maxwell (cc50), Pascal (cc60), and Volta (cc70) architectures.

Repeat this exercise for aolin24 and aomaster nodes (modify the submit file accordingly) and answer the previous questions for them as well.

2. First steps with OpenACC

Along this set of exercises you are asked to look for information about the OpenACC directives mentioned in this document and generate different versions of the codes by completing them with the appropriate directive(s).

2.1 Understanding the code and compiling

As we'll see, the PGI compiler applies automatically several optimization actions and, therefore before modifying the code, we need to understand what the compiler is actually doing: what optimizations were applied? and what prevented further optimizations, if any?

Because OpenACC provides parallelization clauses that are applied to loops we have to check first the potential parallelization of our codes. In particular, the following questions should be considered:

- *Does one loop iteration affect other loop iterations?*
- *Are there any data dependencies that prevent parallelization of loop iterations?*

We can use the PGI compiler to help in the analysis of our programs. Consider the three loops provided in file loops.c. Are all of them parallelizable?

As you see, each loop has a `#pragma acc kernels` directive. This directive instructs the compiler to automatically parallelize the loop if possible. Let's see the result of the compilation.

Before compiling, the right environment must be set up:

```
module load pgi/19.4
module add cuda/9.0
```

We compile with the command:

```
pgcc –acc –ta=nvidia:cc60 –Minfo=all,intensity,ccff –o loops
loops.c
```

We use the -ta (target accelerator) option in order to enable code generation for a target architecture. Additionally, the PGI compiler offers a -Minfo flag with the following options:

- accel – Print compiler operations related to the accelerator
- all – Print all compiler output
- intensity – Print loop intensity information
- ccff – Add information to the object files for use by external tools

The compilation output is like this:

```
loop_1:
     18, Intensity = 0.0
         Generating implicit allocate(x[:1000])
         Generating implicit copyin(x[1:999])
         Generating implicit copyout(x[:999])
     19, Intensity = 0.50
         Loop carried dependence of x-> prevents parallelization
         Loop carried backward dependence of x-> prevents vectorization
         Accelerator serial kernel generated
         Generating Tesla code
         19, #pragma acc loop seq
loop_2:
```

```
     32, Intensity = 0.0
         Loop not vectorized: data dependency
         Generated vector simd code for the loop
         Loop unrolled 8 times
     36, Intensity = 0.0
         Generating implicit copy(y[:1000])
         Generating implicit copyin(x[:1000])
     37, Intensity = 0.67
         Complex loop carried dependence of x-> prevents parallelization
         Loop carried dependence of y-> prevents parallelization
         Loop carried backward dependence of y-> prevents vectorization
         Accelerator serial kernel generated
         Generating Tesla code
         37, #pragma acc loop seq
loop_3:
     49, Intensity = 0.0
         Generated vector simd code for the loop
     53, Intensity = 0.0
         Generating implicit copy(y[:1000])
         Generating implicit copyin(x[:1000])
     54, Intensity = 0.67
         Loop is parallelizable
         Generating Tesla code
         54, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Among other information about loops, code generation, data dependencies, etc., the compiler informs about loop computational intensity. **Computational Intensity** of a loop is a measure of how much work is being done compared to memory operations.

Computation Intensity = Compute Operations / Memory Operations

Computational Intensity of 1.0 or greater suggests that the loop might run well on a GPU.

According to the information generated by the compiler, which loops were parallelized? What's the reason that prevented parallelization of the other loops?

In the second loop, the problem is pointer aliasing: different pointers are allowed to access the same object. This may induce implicit data dependency in a loop. In this example, it is possible that the pointers x an y access to the same object. Potentially there is data dependency in the loop.

To avoid pointer aliasing, use the keyword *restrict*. *restrict* means: for the lifetime of the pointer ptr, only it or a value directly derived from it (such as ptr + 1) will be used to access the object to which it points. Modify the second loop appropriately, so that the loop is parallelized.

3. Adding OpenACC directives

OpenACC directives are much like OpenMP directives. They take the form of pragmas in C/C++. There are several advantages to using directives. First, since it involves very minor modifications to the code, changes can be done incrementally, one pragma at a time. This is especially useful for debugging purpose, since making a single change at a time allows one to quickly identify which change created a bug. Second, OpenACC support can be disabled at compile time. When OpenACC support is disabled, the pragma are considered comments, and ignored by the compiler. This means that a single source code can be used to compile both an accelerated version and a normal version. Third, since all of the offloading work is done by the compiler, the same code can be compiled for various accelerator types: GPUs, MIC (Xeon Phi) or CPUs. It also means that a new generation of devices only requires one to update the compiler, not to change the code.

3.1. Kernels directive

The *kernels* directive is what we call a descriptive directive. It is used to tell the compiler that the programmer thinks this region can be made parallel. At this point, the compiler is free to do whatever it wants with this information. It can use whichever strategy it thinks is best to run the code, including running it sequentially. Typically, it will

- Analyze the code to try to identify parallelism.
- If found, identify which data must be transferred and when.
- Create a kernel.
- Offload the kernel to the GPU.

One example of this directive is the following code:

```
#pragma acc kernels
{
    for (int i=0; i<N; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

3.2.1. Building, executing and profiling GPU code

In principle, your GPU jobs can be compiled and executed at your login machine. However, most login machines are equipped with old GPUs that do not support modern versions of CUDA. Therefore, all executions shoyld be done in batch mode using machines *aolin23* or *aolin24* as done at section 1. However, the compilation should be

take into account also the CPU available at the target system, otherwise, the application will not run properly.

Compile the *loop_test.c* adding a *-tp* parameter (*-tp=nehalem* or *–tp=sandybridge*, for aolin23 or aolin24, respectively).

Modify now the submision file used in section 1 to run the application either in aolin23 or aolin24, according to the compilation option used previously. The execution section of the submit file should be modified to run the *loop_test* executable. Add also the following changes to get feedback about the execution of the *loop_test*.

```
export PGI_ACC_TIME=1
echo
echo "Running..........."
echo
nvprof ./loop_test
```

With PGI_ACC_TIME=1, we'll get some details about the application execution. The output should be like:

```
Accelerator Kernel Timing data
/home/caos/masenar/OpenACC/AA-2019/loop_test.c
 loop_gpu  NVIDIA  devicenum=0
  time(us): 0
  44: data region reached 2 times
  46: compute region reached 1 time
    47: kernel launched 1 time
      grid: [16]  block: [128]
      elapsed time(us): total=118 max=118 min=118 avg=118
  51: compute region reached 1 time
    52: kernel launched 1 time
      grid: [16]  block: [128]
      elapsed time(us): total=30 max=30 min=30 avg=30
```

GPU profiling is an important activity when it comes time to do your performance optimizations for real. There are several profiling tools provided by different vendors. NVIDIA Visual Profiler and the Portland Group Profiler PGPROF are two of the most commonly used. These tools have a powerful graphic interface that might be very useful when the application is executed interactively. As we are executing in batch mode, our profiling activity will be done with a command line profile (*nvprof*), provided also by NVIDIA. In this example, *nvprof* provides the following profiling information about the application:

```
==26846== Profiling application: ./loop_test
==26846== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   50.69%  9.5050us         1  9.5050us  9.5050us  9.5050us  loop_gpu_47_gpu
                   49.31%  9.2480us         1  9.2480us  9.2480us  9.2480us  loop_gpu_52_gpu
      API calls:   58.03%  208.68ms         1  208.68ms  208.68ms  208.68ms  cuDevicePrimaryCtxRetain
                   41.11%  147.86ms         1  147.86ms  147.86ms  147.86ms  cuDevicePrimaryCtxRelease
```

```
 0.43%  1.5292ms         1  1.5292ms  1.5292ms  1.5292ms  cuMemAllocHost
 0.31%  1.1089ms         3  369.62us  9.9100us  557.38us  cuMemAlloc
 0.07%  244.83us         1  244.83us  244.83us  244.83us  cuModuleLoadData
 0.02%  71.125us         2  35.562us  19.596us  51.529us  cuLaunchKernel
 0.01%  51.319us         1  51.319us  51.319us  51.319us  cuStreamCreate
 0.01%  24.583us         4  6.1450us  4.0870us  10.629us  cuEventRecord
 0.00%  13.148us         3  4.3820us     736ns  9.9880us  cuCtxSetCurrent
 0.00%  10.956us         2  5.4780us  4.7560us  6.2000us  cuEventSynchronize
 0.00%  7.1560us         2  3.5780us  1.9670us  5.1890us  cuPointerGetAttributes
 0.00%  5.8090us         2  2.9040us  2.3880us  3.4210us  cuStreamSynchronize
 0.00%  5.6790us         3  1.8930us     542ns  3.6310us  cuDeviceGetCount
 0.00%  4.4070us         2  2.2030us  1.8840us  2.5230us  cuEventElapsedTime
```

As you see, loops 47 and 52 take a similar time to execute, but the application spends more time in API calls (initialization, completion,…).

3.2.2 Loop directive with independent clause

Another way to tell the compiler that loops iterations are independent is to specify it explicitly by using a different directive: *loop*, with the clause *independent*. This is a prescriptive directive. Like any prescriptive directive, this tells the compiler what to do, and overrides any compiler analysis. The initial example above would become:

```
#pragma acc kernels
{
#pragma acc loop independent
for (int i=0; i<N; i++)
  {
    C[i] = A[i] + B[i];
}
```

3.3. Parallel directive

With the *kernels* directive, we let the compiler do all of the analysis. This is the descriptive approach to porting a code. OpenACC supports a prescriptive approach through a different directive, called the *parallel* directive. This can be combined with the loop directive, to form the parallel loop directive. An example would be the following code:

```
#pragma acc parallel loop
for (int i=0; i<N; i++)
{
  C[i] = A[i] + B[i];
}
```

Since parallel loop is a prescriptive directive, it forces the compiler to perform the loop in parallel.

Regard that with the use of this directive we might need to introduce additional clauses to manage the scope of data, such as *private* and *reduction* that control how the data flows through a parallel region.

These additional clauses have the same meaning as in OpenMP. With the private clause, a copy of the variable is made for each loop iteration, making the value of the variable independent from other iterations. With the reduction clause, the values of a variable in each iteration will be reduced to a single value. It supports addition (+), multiplication (*), maximum (max), minimum (min), among other operations. These clauses were not required with the kernels directive, because the kernels directive handles this for you.

3.3.1 Kernels vs Parallel directives

From what we have said so far, the kernels directive is more implicit, gives the compiler more freedom to find and map parallelism and, therefore, the compiler performs parallel analysis and parallelizes what it believes safe.

In contrast, the parallel directive is more explicit, requires analysis by programmer to ensure safe parallelism and constitutes a straightforward path from OpenMP.

#pragma acc kernels
{
for (i=0; i<n; i++)
a[i] = 3.0f*(float)(i+1);
for (i=0; i<n; i++)
b[i] = 2.0f*a[i];
}
	#pragma acc parallel
	{
	#pragma acc loop
	for (i=0; i<n; i++)
	a[i] = 3.0f*(float)(i+1);
	#pragma acc loop
	for (i=0; i<n; i++)
	b[i] = 2.0f*a[i];
	}

In this case, the compiler: • Generates two kernels • There is an implicit barrier between the two loops: the second loop will start after the first loop ends.	In this case, the compiler: • Generate one kernel • There is no barrier between the two loops: the second loop may start before the first loop ends. (This is different from OpenMP)

3.4. Data movement with OpenACC

Sometimes, we see that although we've moved the most compute intensive parts of the application to the accelerator, the process of copying data from the host to the accelerator and back will be more costly than the computation itself. The next step in the acceleration process is to provide the compiler with additional information about data locality to maximize reuse of data on the device and minimize data transfers. It is after this step that most applications will observe the benefit of OpenACC acceleration.

3.4.1. Structured data regions

The *data* directive defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region. Here is an example of defining the structured data region with data directives:

```
#pragma acc data
{
      #pragma acc parallel loop ...
      #pragma acc parallel loop
      ...
}
```

Arrays used within the data region will remain on the GPU until the end of the data region.

3.4.2. Unstructured data regions

We sometimes encounter a situation where scoping does not allow the use of normal data regions. In those cases, we use unstructured data directives:

- enter data: defines the start of an unstructured data lifetime
    - clauses : copyin(list), create(list)
- exit data: defines the end of an unstructured data lifetime
    - clauses: copyout(list), delete(list)

Below is the example of using the unstructured data directives in the code:

```
#pragma acc enter data copyin(a)
...
#pragma acc exit data delete(a)
```

Unstructured data clauses enable OpenACC to be used in C++ classes (using them in object constructors and object destructors). Moreover, unstructured data clauses can be used whenever data is allocated and initialized in a different piece of code than where it is freed.

3.4.3. Data clauses

OpenACC provides several clauses that define how data has to be moved form the host to the device and viceversa. These clauses can be used with data directives or with the parallel directives described before (kernels and parallel).

- copyin(list) - Allocates memory on GPU and copies data from host to GPU when entering region.
- copyout(list) - Allocates memory on GPU and copies data to the host when exiting region.
- copy(list) - Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region. (Structured Only)
- create(list) - Allocates memory on GPU but does not copy.
- delete(list) - Deallocate memory on the GPU without copying. (Unstructured Only)
- present(list) - Data is already present on GPU from another instant in time.

3.4.4. Array shaping

Sometimes, the compiler cannot determine the size of arrays (for insance, when arrays are allocated dynamically). Therefore, one have to explicitly specify the size with data clauses and array "shape". Here is an example of array shape in C:

```
#pragma acc data copyin(a[0:nelem]) copyout(b[s/4:3*s/4])
```

Moreover, OpenACC is not automatically able to copy dynamic pointers to the device. You must first copy the struct into device memory. Then you must allocate/copy the dynamic members into device memory. To deallocate, you must first deallocate the dynamic members and then deallocate the struct.

For example, copying and deleting a struct that contains a dynamically allocated array:

```
typedef struct {
float *arr;
int n;
} vector;

int main(int argc, char* argv[]){
      vector v;
      v.n = 10;
      v.arr = (float*) malloc(v.n*sizeof(float));

      #pragma acc enter data copyin(v)
      #pragma acc enter data create(v.arr[0:v.n])
      ...

      #pragma acc exit data delete(v.arr)
      #pragma acc exit data delete(v)
      free(v.arr);
}
```

3.4.5. The present clause

When managing the memory at a higher level, it's necessary to inform the compiler that data is already present on the device to save time in execution. Local variables should still be declared in the function where they're used. For example:

```
function main(int argc, char **argv) {
#pragma acc data copy(A) {
    laplace2D(A,n,m);
    }
}
...
function laplace2D(double[N][M] A,n,m){
   #pragma acc data present(A[n][m]) create(Anew)
   while ( err > tol && iter < iter_max ) {
      err=0.0;
      ...
   }
}
```

High-level data management and the *present* clause are often critical to good performance. Therefore use it when possible.

3.4.6. Update Directive

In OpenACC it is possible to specify an array (or part of an array) that should be refreshed within the data region. In order to do so we use the update directive:

```
do_something_on_device()
#pragma acc update self(a)  //  Copy "a" from GPU to CPU
do_something_on_host()
#pragma acc update device(a) // Copy "a" from CPU to GPU
```

The following example demonstrates the usage of the update directive. First, we modify a vector on the CPU (host), then copy it to the GPU (device). Vector *v* is a struct of arrays, in this case, and *v.n* contains the size of the array *v.coefs[ ]*.

```
void initialize_vector(vector &v,double val) {
   for(int i=0;i<v.n;i++)
      v.coefs[i]=val;          // Updating the vector on the CPU
   #pragma acc update device(v.coefs[:v.n])  // Updating the
                                            // vector on the GPU
  }
```

Take a look at the *data_test.c*, compile it and execute it (modify the submition file as needed). Analyse the data movements carried out by each loop (check the compiler output as well as the PGI and the nvprof profile data). Look at the GPU activities: what's the impact of data movement in the overall execution time?

4. **Putting it all together**

In this section, you are asked to parallelize a simple application by using basic OpenACC directives described in previous sections.

Our basic application (*addition.c)* is a C program that adds two vectors and a constant. The operation is carried out for vectors of different sizes and, for each size, it is repeated several times in order to have significant execution times. The addition operation is repeated twice (first in the CPU and then in the GPU) and the time consumed in each case is measured in order to compare the corresponding performance.

Modify the code of *addition.c* and add additional clauses to parallelize the main loops and use eventually explicit data directives. You may use either kernels or parallel loop directives (or both, if you have time).
A *makefile* is provided to simplify the compilation execution. See the contents of the *makefile* below. The makefile can generate executables for different architectures:

make ARCH=nehalem (for aolin23)

make ARCH=sandybridge (for aolin24)

```
# Makefile for OpenACC Examples

CC = pgcc

ifeq ($(ARCH),nehalem)
CFLAGS = -tp=nehalem -fast -acc -ta=nvidia:cc60 -Minfo=all,intensity,ccff
LFLAGS =
else ifeq ($(ARCH),sandybridge)
CFLAGS = -tp=sandybridge -fast -acc -ta=nvidia:cc60 -Minfo=all,intensity,ccff
LFLAGS =
else ifeq ($(ARCH),aomaster)
CFLAGS = -tp=nehalem -fast -acc -ta=nvidia:cc50 -Minfo=all,intensity,ccff
LFLAGS =
endif

objects = addition.o

default: addition

%.o : %.c
	$(CC) $(CFLAGS) -c $<

addition : $(objects)
	$(CC) $(CFLAGS) -o $@ $^ $(LFLAGS)

.PHONY: clean

clean:
	rm -f *.o
```