HPC Laboratory Assignment Lab 7

GPUs and OpenACC programming

J.R Herrero and M.A. Senar

Spring 2023-24

OpenACC tutorial

This document has been prepared with the purpose of guiding you through a set of very simple examples that will be helpful to practice the main components of the OpenACC programming model and to execute applications on a computer equipped with a GPU through a batch queue system. As in previous assignments, you will need:

- You account on the UAB cluster to run all the examples. Remember that no X services are available. So you have to use command line and text editors, such as vim and nano.
- Connect to the cluster by using a secure connection
  ssh –p 54022 <your account>@aolin-login.uab.cat

  The first time you stablish the connection, you will get this dialog:

  ```
  The authenticity of host '[aolin-login.uab.cat]:54022 ([158.109.65.225]:54022)' can't be established. ECDSA key fingerprint is
  SHA256:29XgrvpJnHUNxAba86SQ/2bsh/dJWUKp9bG6iJBsWwE.
  Are you sure you want to continue connecting (yes/no)?
  ```

  Answer *yes*. Then the system will ask for your user password:

  ```
  Warning: Permanently added '[aolin-login.uab.cat]:54022,[158.109.65.225]:54022' (ECDSA) to the list of known hosts.
  biohpc-x-x@aolin-login.uab.cat's password:
  ```

  Type your password and, if successfully, you will be logged into the cluster. A credential dialog may be started before you get the system prompt; just hit <Enter> key at each question until you get the system prompt.

- The set of files provided as part of this session are located inside the Escritorio/alumnos/OpenACC directory. Copy them to your home directory or to any other folder created there.
- The set of slides about GPU architecture and OpenACC available through the course Moodle.
- Section 6.2 and 6.3 provides some theoretical background about GPU's architecture, its underlying programming model and OpenACC principles. Section 6.2 might be read at home; section 6.3 can be read as you need to solve the exercises of sections 6.4 and 6.5.

6.1 Deliverable

After the session for this laboratory assignment you will have to deliver a compressed file that includes a **report** in **PDF** format (other formats will not be accepted) containing the answers to the questions and exercises included in this document (questions are in italics and bold). Your compressed file should include also source code files corresponding to the exercises in section 6.4 and 6.5. In your report document provide screenshots or any equivalent method that shows the outputs generated by the programs and the codes created in response to the corresponding questions.

When answering to the questions in this questionnaire, please DO NOT simply answer with yes, no or a number; take a look first at each program and try to figure out its behavior beforehand; afterwards, compile it and run it and check the output to answer the corresponding questions; try to minimally justify all your answers. Answer the questions in this document, not the ones in the source files that may be outdated. Sometimes you may need to execute several times in order to see the effect of data races in the parallel execution or to obtain time measurements free from the effects of other system programs.

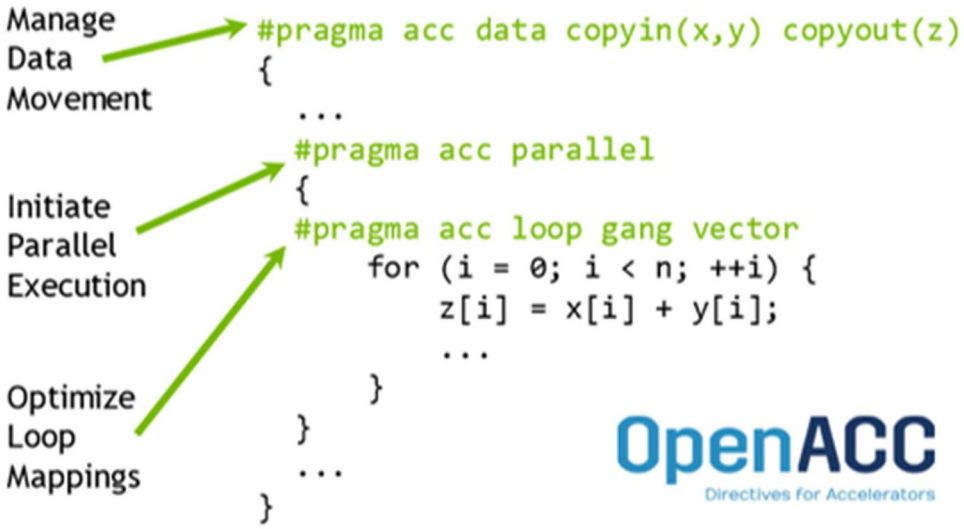
Your professor will open the assignment at the course website and set the appropriate dates for the delivery. Only one file has to be submitted per group through the course website. Late submissions will be accepted but a deduction between 10-30% of the maximum mark available from the actual mark achieved by the student shall be imposed.

Important:

- Please, follow the same recommendations that we made for the previous deliverable.
- In the front cover of the document, please clearly state the name of all components of the group, the identifier of the group (username biohpc-XX), title of the assignment, date, academic course/semester,... and any other information you consider necessary. As part of the document, you can include any code fragment you need to support your explanations.

6.2 Introduction to GPUs

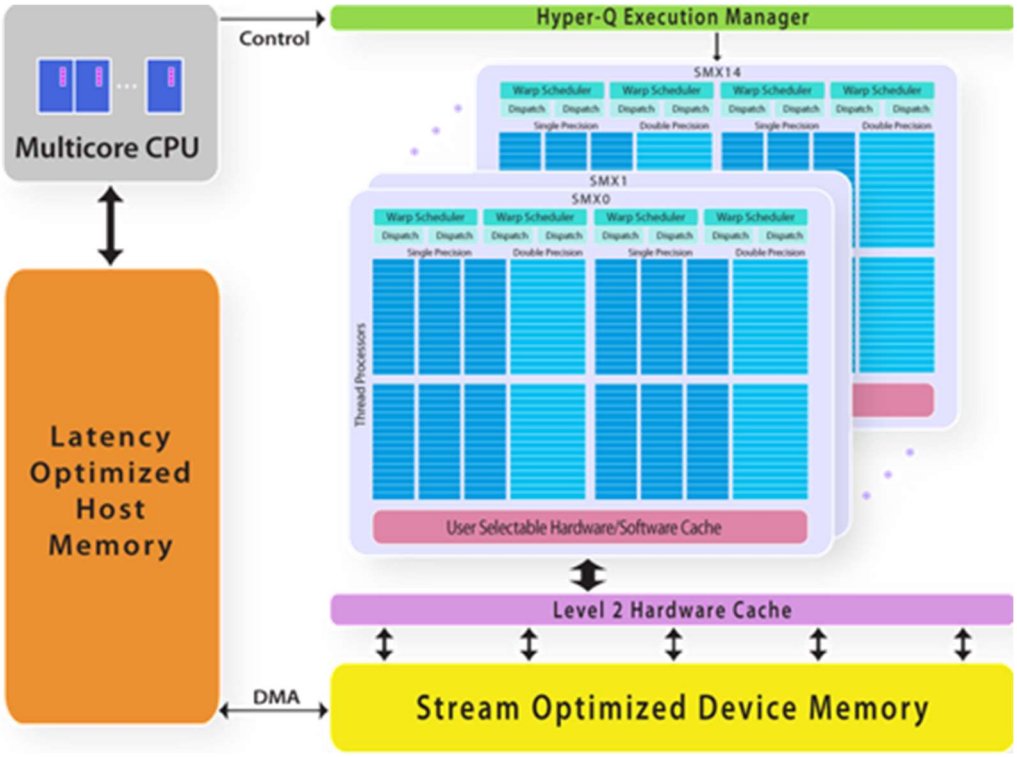
OpenACC is a standard for parallel programming on CPU / GPU heterogeneous systems (Open ACCelerators), developed by Cray, NVIDIA and PGI. Like in the case of OpenMP, the programmer annotates the source code using compiler directives (#pragma) in the C, C++ and Fortran languages in order to identify the areas of code that can be accelerated. The code specified by the programmer can be compiled to be executed in an accelerator device coupled to the CPU (acting as a host), for example, the GPUs of NVIDIA and AMD. This code is downloaded into the device at run time, when the program running in the host invokes the execution.



OpenACC directives are designed to reduce the cost of writing simple GPU programs, and to make easier to write large efficient programs. However, writing efficient programs still requires you to understand the target architecture, the CUDA threading model, and how your program is mapped onto the target. This knowledge allows understanding the compiler feedback provided by -Minfo messages and using this feedback to improve the performance of the code.

6.2.1. CUDA threading Model

The next figure shows the model of the CPU-GPU hybrid system considered by OpenACC. A program cannot be executed directly on the GPU (device), which is a coprocessor of a traditional CPU (host). Among others, the device does not incorporate the necessary H/W support to implement OS tasks that allow protection between processes (priorities) or the management of Input / Output devices using interruptions.



Host and device often have separate physical memories (except on low-power embedded computer systems), optimized for different metrics: (1) the host memory is optimized to reduce the access latency and offer the maximum storage capacity; (2) the device memory is optimized to maximize the memory bandwidth for sequential accesses and, for cost reasons, must sacrifice storage capacity and memory latency. As a GPU is designed for stream or throughput computing, it does not rely on a deep cache memory hierarchy for memory performance, and device memory supports very high data bandwidth using a wide data path. The Volta NVIDIA GPU allows fetching 4096 bits in a single cycle from eight 512-bit memory channels; each memory channel can be addressed independently to fetch 16 consecutive 32-bit words. This means there is severe effective bandwidth degradation for strided accesses: a stride-two access, for instance, will fetch those 512 bits, but only use half of them, suffering a 50% bandwidth penalty.

A GPU is connected to a host through a high-speed I/O bus slot, typically PCI-Express, and data is usually transferred between the GPU and host memories using programmed DMA, which operates concurrently with both the host and GPU compute units. These physical memories can be handled as separate memory spaces or can be part of a virtual shared memory known as unified memory.

Another typical feature of many accelerators is that there are small S/W programmable cache memories: the programmer decides which variables are stored in these cache memories during the entire execution of a kernel (GPU program executable). NVIDIA calls this low-latency, high-bandwidth, indexable memory which runs close to register speeds the shared memory. Currently, the use of the S/W cache memories is not possible using OpenACC and can only be controlled using the CUDA programming extensions.

A kernel running on a GPU is made up of thousands, millions, or billions of threads. A thread is a program that specifies the sequential execution of a series of instructions. All threads execute the same code, in what is called an SPMD (Single-Program Multiple-Data) model, but each thread receives a unique identifier that allows it to differentiate the input data it has to use, the results that it must generate, and even the parts of the code that must be executed or not. The threads that make up the kernel share the total work that must be done, and may have to collaborate by sharing data in common variables and synchronizing explicitly.

As shown in the previous figure, the millions of threads that compose the kernel wait in a general queue (Hyper-queue Execution Manager), before being assigned to one of the computation modules of the device (or Stream Multiprocessors or SMs). Each SM has its own queue associated with thousands of threads competing to execute instructions using the SM's compute resources. Once a group (block) of threads is assigned to a SM, the threads can no longer be evicted, and remain in the same SM until their execution is completely finished.

NVIDIA GPUs group threads from 32 to 32 to run in SIMT (Single-Instruction Multiple-Threads) form, that is, synchronized. This scheme represents a different way of programming what we know as SIMD execution and the use of vectorized instructions. The code is actually executed in groups of 32 threads, what NVIDIA calls a warp (like an SIMD instruction with vectors of 32 lanes).

When the 32 threads in a warp group issue a device memory operation, those memory accesses will take a very long time, perhaps hundreds of clock cycles, due to the long memory latency. Mainstream CPU architectures include a two-level or three-level cache memory hierarchy to reduce the average memory latency, and GPUs do include some hardware caches, but mostly GPUs are designed for stream or throughput computing, where cache memories are ineffective. Instead, GPUs tolerate memory latency by using a high degree of multithreading. Volta GPUs support up to 64 active warps on each multiprocessor (up to 2048 concurrent threads per SM). When one warp of threads stalls on a memory operation, another ready warp is selected to execute, so that the cores can be productive as long as there is enough parallelism to keep them busy.

NVIDIA GPUs are programmed as a sequence of kernels: a group of thousands to millions of threads. Typically, each kernel completes execution before the next kernel begins, with an implicit barrier synchronization between kernels. Recent GPUs have support for multiple, independent kernels to execute simultaneously, but many kernels are large enough to fill the entire device capability. On the CUDA data parallel programming model the host program launches a sequence of kernels, and those kernels can spawn sub-kernels. Threads are grouped into blocks (or Cooperative Thread Arrays, CTAs), and blocks are grouped into a grid. Each thread has a unique local index in its block (threadIdx), and each block (or CTA) has a unique index in the grid (blockIdx). Kernels can use these indices to compute array subscripts, for instance.

Threads in a single block are executed on a single multiprocessor (or SM), sharing the software data cache, and can synchronize and share data with threads in the same block. Threads in different blocks may be assigned to run on different multiprocessors (SMs) concurrently, to run on the same multiprocessor concurrently (using multithreading), or

may be assigned to run on the same or different multiprocessors at different times, depending on how the blocks are scheduled dynamically.

There is an upper limit on the size of a thread block (1,024 threads or 32 warps for Volta). Thread blocks are always created in warp-sized units, and a warp will always be a subset of threads from a single block; all thread blocks in the whole grid will have the same size and shape. A multiprocessor can have up to 2,048 threads, or 64 warps, simultaneously active, which can come from 2 thread blocks of 32 warps, or 3 thread blocks of 21 warps, 4 thread blocks of 16 warps, and so on; there is another upper limit of thread blocks simultaneously active on a single multiprocessor.

Performance tuning on NVIDIA GPUs requires: (1) Finding and exposing enough parallelism to populate all the multiprocessors (SMs) and to allow multithreading to keep the cores busy; (2) Optimizing device memory accesses for contiguous data, essentially optimizing for stride-1 memory accesses (these memory accesses are called coalesced); (3) Utilizing the software data cache to store intermediate results or to reorganize data that would otherwise require non-stride-1 device memory accesses.

https://www.openacc.org/resources

https://github.com/OpenACCUserGroup/openacc-users-group/blob/master/

6.3 Introduction to OpenACC

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.

6.3.1. Kernel 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++)
            x[i] = (a+i) * x[i] ;
}
```

6.3.1.1 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++)
  {
    x[i] = (a+i) * x[i] ;
 }
```

6.3.2 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++)
{
      x[i] = (a+i) * x[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.

6.3.3 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.

6.3.3.1 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.

6.3.3.2 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.

6.3.3.3 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.

6.3.3.4 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.

```
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.

6.3.3.5 Array shaping

Sometimes, the compiler cannot determine the size of arrays (for instance, when arrays are allocated dynamically). Therefore, one has to explicitly specify the size with data clauses and array "shape".

The general format is for a 1D array is:

```
array[starting_index:length]
```

Array shaping can be applied to multidimensional arrays and to partial arrays as shown in the following examples:

```
array[0:N][0:M]     (a 2D array)

array[i*N/4:N/4]    (a partial 1D array)
```

Here is an example of array shape in C with *copyin* and *coyout* clauses:

```
#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);
}
```

6.3.3.6 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

  }
```

6.4 Parallel Programming with OpenACC

1. Checking the system

Firstly, we will check the GPU resources available in the cluster. All the codes have to be compiled with nvcc 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 file that both, specifies job requirements and includes the application that has to be executed.

Our basic submission file (*gpu.slurm*) is as follows:

```
1. #!/bin/bash
2. #SBATCH --job-name=gpu-check
3. #SBATCH -N 1 # number of nodes
4. #SBATCH -n 1 # number of tasks
5. #SBATCH --partition=cuda-ext.q
6. #SBATCH --nodelist=aolin-gpu-3

7. hostname
8. module add cuda/11.2
9. module load nvidia-hpc/21.2
10.
11.   echo "****************************************"
12.   pgaccelinfo
13.
```

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 (*line 6). 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 (line 8, line 9) and, finally, runs a command that requests information about GPUs available in the system.

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

```
sbatch gpu.slurm
```

```
Submitted batch job 4382
```

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

```
biohpc-XX@@aolin-login:~/home$ squeue
```

```
JOBID PARTITION  NAME USER    ST TIME  NODES NODELIST(REASON)
```

```
4382 cuda-ext.q  gpu-check biohpc-XX  R 0:07   1 aolin23
```

This job will run very fast. So, it might be possible that has 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

Information about batch queues available at the lab can be queried with the *sinfo* command. As you may see, the cuda.q partition contains four nodes (aolin23, aolin24, aoclsd and aomaster). ***Submit the gpu.slurm and look at its output***.

The output of the execution should look like this:

```
aolin-gpu-3.uab.cat
****************************************

CUDA Driver Version:           12020
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module
535.54.03  Tue Jun  6 22:20:39 UTC 2023

Device Number:                 0
Device Name:                   NVIDIA GeForce RTX 3080
Device Revision Number:        8.6
Global Memory Size:            10495524864
Number of Multiprocessors:     68
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:                    1740 MHz
Execution Timeout:             Yes
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   No
Memory Clock Rate:             9501 MHz
```

```
Memory Bus Width:              320 bits
L2 Cache Size:                 5242880 bytes
Max Threads Per SMP:           1536
Async Engines:                 2
Unified Addressing:            Yes
Managed Memory:                Yes
Concurrent Managed Memory:     Yes
Preemption Supported:          Yes
Cooperative Launch:            Yes
  Multi-Device:                Yes
Default Target:                cc80
```

Q1: Answer the following questions:

- ***How many GPUs are available in this node?***
- ***How many SMs has this GPU?***
- ***What's the size of GPU's main memory?***

2. Running on a specific GPU

We can get the information of GPUs available in our cluster nodes using the command *scontrol*. For instance,

scontrol show nodes aolin-gpu-1,aolin-gpu-2,aolin-gpu-3,aolin-gpu-4

would show you a detailed information of the corresponding nodes.

Verify that the information that appears about the GPU of this node is the same as that obtained in the execution of the slurm job (look at the gres field).

Q2: Modify the gpu.slurm file so that the job runs only on a node with a GeForceGTX1080Ti GPU. You have to add a #SBATCH --gres option with the appropriate information. Verify that your solution works correctly.

3. First steps with OpenACC

Along with 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).

As we'll see, the NVIDIA compiler (nvc) applies automatically several optimization actions and, therefore, we need to understand what the compiler is actually doing before any modification to the code is applied: 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 nvc 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 by executing the following commands:

```
module load nvhpc/21.2
```

The program will be compiled with version 21.2 of the nvc compiler and a *Makefile* is provided to simplify the compilation execution. This *Makefile* can be used in next sections to compile other examples; just pass the name of the target executable as argument to the *make* command (i.e., *make loops* will compile program *loops.c* and generate executable *loops*). The contents of the *Makefile* are as follows:

```
CXX=nvc
CXXFLAGS= -acc=gpu -Minfo=all,intensity,ccff
LDFLAGS=${CXXFLAGS}

%: %.c
	$(CXX) ${LDFLAGS} -o $@ $<

.SUFFIXES: .o .c .cpp .h
.PHONY: clean
clean:
	rm *.qdrep *.sqlite *.o core
```

```
CXX=nvc
CXXFLAGS= -acc=gpu -Minfo=all,intensity,ccff
LDFLAGS=${CXXFLAGS}

% : %.c
	$(CXX) ${LDFLAGS} -o $@ $<

.SUFFIXES: .o .c .cpp .h
.PHONY: clean
clean:
	-rm *.sqlite *qdrep  *.o core
	-rm `find . -maxdepth 1 -perm /111 -type f`
```

Application is compiled with the nvc compiler. We use the –acc=gpu option to enable offloading to accelerators. On the other hand, the PGI compiler offers a -Minfo flag with several options. In our case, we are using the following:

- all – Print all compiler output

- intensity – Print loop intensity information
- ccff – Add information to the object files for use by external tools

When compiling our example (*make loops*) you get an output like:

```
loop_1:
     16, Intensity = 0.0
         Generating implicit allocate(x[:1000]) [if not already present]
         Generating implicit copyin(x[1:999]) [if not already present]
         Generating implicit copyout(x[:999]) [if not already present]
     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
     35, Intensity = 0.0
         Generating implicit copyin(x[:1000]) [if not already present]
         Generating implicit copy(y[:1000]) [if not already present]
     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
     52, Intensity = 0.0
         Generating implicit copyin(x[:1000]) [if not already present]
         Generating implicit copy(y[:1000]) [if not already present]
     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.

Once compiled, the executable is located at your directory. This is a sequential version that can be executed on a standard CPU. *Run it and see the execution results.*

Q3: According to the information generated by the compiler,

- ***Which loops were parallelized? How do you know it?***
- ***What are the reasons that prevented parallelization of the other loops?***

In the second loop, the problem is pointer aliasing: different pointers may 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, the keyword *restrict* can be used. Adding this keyword 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 (addind the restrict keyword at the definition of both arrays), so that the loop is parallelized. Before asking your lecturer, try to find the right syntax for using *restrict* by yourself.

4. Building, executing and profiling GPU code

We are ready to run our *loops* program in a node with a GPU.

Make sure you have this line in *gpu.slurm*

```
#SBATCH --gres=gpu:GeForceGTX1080Ti:1
```

Uncomment the following three lines of *gpu.slurm*. The first two lines will force a recompilation of the program in the machine where it will run. This recompilation guarantees that the code generated by the compiler will be compatible with the CPU where the application will be executed (which might be different to the CPU of the submission machine). In any case, it is good practice to compile the program first on the submit node and make sure it is bug-free before submitting it to SLURM.

```
touch loops.c

make loops

nsys nvprof --print-gpu-trace loops
```

With this submit file loops application will be executed and NVIDIA's *nsys* profiling tool will collect execution statistics. 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 has 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*) and we will collect only some summary data about the execution of the application.

Submit your program and check the information provided by nsys at the output file generated by SLURM.

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 has 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*) and we will collect only some summary data about the execution of the application.

Q4: Look for the sections related to CUDA kernel statistics and CUDA Memory Operation Statistics.

- ***What's the total time spent by each loop executed at the GPU?***

- ***What's the total time and data size moved from the CPU (host) to the GPU (device) and viceversa? Can you explain these values according to the operations required by each loop?***

Q6: Modify the loops.c application to use now the parallel loop directive. Execute the new version and check if there are any differences in performance compared to the kernels version. Is this execution correct? Why? Modify the program to get a proof that supports your answers.

Q7: Implicit data movement actions are taken by the compiler if no explicit OpenACC data clauses are used. Take a look again at the compilation output of the loops.c program (pg. 10 an 11) and identify which data movement operations were implicitly generated by the compiler. Do these data movements correlate with the memory operation statistics collected at the execution of the program (Question 5)?

6.5 Parallelization exercises

1. Vector addition

Calculate the sum of two vectors (C = A + B) in parallel using OpenACC. A skeleton code is provided in *vector-sum.c* (a TODO comment shows the place where your code should go).

The main computation loop should be parallelised using OpenACC. Try both acc parallel and acc kernels and check the compiler diagnostics output. Run the programs and compare the results.

2. Stencil

Parallelise a simple stencil update kernel with OpenACC parallel or kernels pragmas.

The file stencil-loop.c implements a simple stencil update kernel. Search for a TODO tag and try to parallelize the given loop nest with OpenACC parallel or kernels pragmas. Regard that the outer most loop (*iter*) simulates different time iterations that should not be parallelized. Your ACC parallelization clauses (*parallel, kernels, loop*),.. should affect only the i and j loops. Data movement clauses should be placed in the right place to avoid excessive data movements. Pay attention to the compiler diagnostics output. Initially, you can set the number of iterations (*niter* variable) to 4, so that you can quickly check that your parallel code is working properly.

Write a first version using only parallel or kernel directives. Write a second version adding explicit data movement clauses. Execute and profile both versions and compare the differences in data movement and execution times.

Compile a reference version without OpenACC setting number of iterations (*niter* variable) to 2000 and compare the results with your best parallel version. What speedup are you getting with your gpu versions of the program?