OpenACC: 2D Laplace Equation

The NVIDIA compiler (and all compilers) automatically applies several **optimization** actions. We need to understand what the compiler is doing, and which code issues are preventing further optimizations.

We will use the NVIDIA compiler and OpenACC to help us analyze the potential parallelism of the loops. In the teaching platform, we provide a code example, ***laplace.c***, with a **#pragma acc kernels** directive in front of some program loops. This directive instructs the compiler to automatically parallelize the loop if possible. Look below for the commands needed to install de NVIDIA compiler and tools, and then compile the code for the local CPU to obtain a text log of the compiler analysis. These results can guide us to find modifications to further improve performance.

```
$ module add nvhpc/21.2                                    # install NVIDIA tools
$ nvc -V                                                 # Look at CPU microarchitecture
$ nvc -fast -acc=host -Minfo=accel laplace.c -o LCPU       # compile for host CPU
```

Q1: (A) Compile the code **laplace.c** and indicate which of the loops (identify them by the line number) the compiler considers to be parallelizable. (B) Run the code on your computer (CPU execution) with the default arguments for the program (4096×4096 mesh and maximum of 100 iterations) and measure the elapsed time, the total machine instructions executed and the IPC. (C) Compile the code, but now with the compilation option –Minfo=all, and check the compiler message carefully: indicate the compiler optimizations performed in the *hotspot* code fragments.

To run a program using a **GPU**, you must use the batch queue system (SLURM), which grants conflict-free access to the available computers that contain a recent **GPU** card. SLURM requires a script file that specifies both job requirements and the statements to be executed. The scripts needed for this assignment are available on your computer and on the student platform. The **squeue** command checks the progress of a SLURM job. The **scancel** command followed by the **job id** (a number) removes a job from the SLURM queue. The **sinfo** command provides information about the batch queues available at the Lab.

The submission file for obtaining information about the **GPU**s in a remote computer is called ***GPUinfo.txt***. You should first analyze the text inside this submission file. Execute the script to obtain information regarding the **GPU** devices attached to the **aolin-gpu-3** remote computer and to the **aolin-gpu-4** remote computer. Notice that each node is attached to a different SLURM queue: **cuda-ext.q** is accessible from everywhere, but **cuda-int.q** is only accessible from the local LAB computers.

```
$ sbatch -p cuda-ext.q -w aolin-gpu-3 --gres=gpu:1 -o GPU3.txt GPUinfo.txt
                                                              # submit job to gpu-3
$ sbatch -p cuda-int.q -w aolin-gpu-4 --gres=gpu:1 -o GPU4.txt GPUinfo.txt
                                                              # submit job to gpu-4
```

Q2: (A) Obtain the following information of the nodes **aolin-gpu-3** and **aolin-gpu-4**: Device (GPU) Name, number of SMs (stream multiprocessors), device clock rate, size of the device **L2** cache, size of the device memory, maximum bandwidth from device to the device memory, and the default CPU target architecture (explained later). (B) Once you know the device name, called <device-name>, search on google the following words "<device-name> specs" (substitute <device-name> by the actual full name) and check your answers.

(Example: https://www.techpowerup.com/gpu-specs/geforce-rtx-3080.c3621)

To run the program on a **GPU** you must compile the code for **GPU** execution. You must replace the –acc=host option by the options –acc and –gpu=cc80 (**cc80** means NVIDIA generation **8.0** = **Ampere** microarchitecture). The option -tp=nehalem or the option -tp=sandybridge tell the compiler to generate **CPU** code for a certain target architecture, like Nehalem or SandyBridge, both older than the local CPU architecture.

```
$ nvc -fast -tp=sandybridge -acc -gpu=cc80 -Minfo=accel laplace.c -o LGPUsandy
```

Once the binary file has been generated, **LGPUsandy**, if you try to execute it on your computer, the execution is run only on **CPU**, since the computer does not have a **GPU** card installed. We provide a script, ***runGPU.txt***, for running and profiling the accelerated binary. Inspect the file to understand the parameters that must be provided to the script.

```
$ sbatch -p cuda-int.q -w aolin-gpu-4 --gres=gpu:1 -o Res.txt runGPU.txt LGPUsandy
```

Q3: (A) Compile the code for **GPU** execution on a **cc80** microarchitecture and a **sandybridge** CPU architecture (see previous command). Explain the messages generated by the compiler. Compile the same code for **GPU** execution on the same **cc80** microarchitecture, but now for a **nehalem** CPU architecture (B) Run the **GPU**-accelerated program on the **GPU** of the **aolin-gpu-3** and **aolin-gpu-4** computers and compare the execution time on both **GPUs** with the execution time in **CPU**. Notice that the CPU architecture of both computers is different.

We provide a script, ***profGPU.txt***, for profiling the execution of the accelerated binary in a **GPU**. Inspect the file to understand the parameters that must be provided to the script. Identify

```
$ sbatch -p cuda-ext.q -w aolin-gpu-3 --gres=gpu:1 -o Prof3.txt profGPU.txt LGPUnehal
```

Q4: Run the profiling task and analyze the output text. Ignore the section "CUDA API Statistics". (A) Identify the different ***kernel*** functions that are executed on the GPU: the name of each function identifies the line number of the loop that generates the function (see section "CUDA Kernel Statistics"); compute the total time devoted by the kernel functions. (B) Identify the **memory-copy** operations performed during the execution (sections "CUDA Memory Operation Statistics"): (1) copies of data from Host to Device (**memcpy HtoD**); (2) copies of data from Device to Host (**memcpy DtoH**); and copies of a constant value into a region of memory (**memset**); how much data is moved between **CPU** and **GPU** and how much time takes the movement? (C) Find out the main performance bottleneck: where is the **GPU** consuming most of the time? (D) Repeat the previous steps for the execution on the **aolin-gpu-4** computer.

One way to improve execution time in this example is to use the **data** directive to indicate the compiler where and how the program's data must be transferred between host memory and device memory. Once the code region is identified, the appropriate clause must be used to indicate which data have to be moved in and/or out of the device memory: **copyin**(list), **copyout**(list), **copy**(list), **create**(list), **present**(list). In some cases, the sizes of the arrays must be declared explicitly.

```
$ diff laplace.c laplace2.c
```

Q5: (A) Find out the differences between both source programs (**diff** command). (B) Compile the new program, **laplace2.c**, and identify the differences in the *log* message provided by the compiler. (C) Run and profile the program; calculate the speedup versus the single-thread **CPU** execution and versus the first **GPU**-accelerated version; why is this speedup achieved? (D) Compare the amount of data **copy** operations and the number of Bytes moved in the new **GPU** version and the previous **GPU** version.

Q6: (A) Execute the previous program but increasing the number of iterations of the convergence loop from 100 to 10,000. You must provide the parameter in the command line, and do not need to modify the source code and re-compile it. (B) Analyze the performance results and find out the main performance bottleneck: where is the **GPU** consuming most of the time?

Optimizing the lap2.c code

One problem with the previous version of the code (**laplace.c**) is that the size of the mesh is fixed at compile time. The code **lap.c** provides an implementation that allocates dynamically the matrices required for the computation, and allows defining the matrix size at runtime, instead of at compile time. We provide an improved version of the code, **lap2.c**, which runs considerably faster on a CPU. Remember to install the NVIDIA compiler and tools and compile the **lap2.c** code for the local CPU to obtain a text log of the compiler analysis.

Q7: Compile the code **lap2.c** and describe the transformations done by the compiler on the **laplace_error** function, which is the function responsible for almost of all the execution time. Run the executable code (**lap2CPU**) on your computer (CPU execution) with the default arguments for the program (4096×4096 mesh and maximum of 100 iterations) and measure the elapsed time, the total machine instructions executed and the IPC.

The problem of *pointer aliasing* means that different pointers may access the same data object, and then **implicit dependences** may happen in a loop (for example, pointers **x** and **y** may be used to access the same object). The keyword **restrict**, applied as in the next example, indicates the compiler that for the lifetime of a pointer **ptr**, only the pointer or a value directly derived from it (such as **ptr**+1) will be used to access the object to which it points.

```
float * restrict ptr;
```

Q8: Modify the declaration of the arguments to the function **laplace_error** using the ***restrict*** word, to help the compiler understand that the inner loop can be vectorized. Compile and run the code as in the previous question. Explain why the performance results have improved. From now on, this modified version of **lap2.c** will always be used.

Q9: Generate a GPU-accelerated version of the **lap2.c** code. You do not need to make any change to the C program, except for adding **#pragma acc** directives. You must add pragmas in two places: one before the **while-loop** in line 62 and one in line 15, inside the **laplace_error**() function. Once your program provides correct results and runs very fast using the GPU, you can increase the number of iterations of the convergence loop from 100 to 1,000 or 10,000 and measure the effect on performance. Notice that this version of the program is designed to get three parameters from the command line: <*n*> <*m*> <*max_iterations*>

Q10: Compute the **effective main memory bandwidth** consumed by the two versions of the program (single-thread CPU and GPU-accelerated), for execution with a 4096 x 4096 mesh, and for **Iter**=1000 and 100.000. You must estimate the total number of Bytes moved by the program between CPU/GPU and the Host/Device main memory, and then divide this value by the total execution time. **Hint**: You must be aware that not all data requested by the program (by LOAD Instructions) must be read from main memory but are read from a cache memory.
Hint: the *write-miss-allocate* policy means that blocks of the output matrix are first read from memory into the cache memory, then are modified into the cache memory, and finally are written to main memory, when replaced by other cache blocks.