OpenACC Programming Lab

https://doc.itc.rwth-aachen.de/display/VE/PPCES+2019

→ GPGPU Programming with OpenACC → PPCES2019_OpenACC-Lab.tar.gz

Dr. Sandra Wienke, RWTH Aachen University

PPCES, March 2019

Please note that it is not expected that you work on all exercises during this workshop. Rather, they shall give you an option if you have time left or if you are interested in a certain topic.

If you need help or have any question please do not hesitate to ask!

1 RWTH GPU Cluster Environment

1.1 Login & Setup

Use your own laptop or one of the provided laptops for logging in.

1. Login to the frontend node (`login18-1.hpc.itc.rwth-aachen.de`) of the RWTH CLAIX2018 GPU Cluster:

   Use the **`hpclab<XY>` account and the provided password**. Due to special hardware reservations and corresponding restrictions, your own HPC account will *not* work for this lab.

2. Log onto the GPU node whose name was handed out to you by typing:

   ```
   ssh -l hpclab<XY> ncg<number> -Y
   ```

   Use the password of your hpclab account to get access to the node.

For setting up the correct environment, do the following:

1. Since the programs will be run interactively, e. g. with `make run`, and the two GPUs are shared between two people, please set the variable `CUDA_VISIBLE_DEVICES` to the value provided on the hand-out:

   ```
   export CUDA_VISIBLE_DEVICES=<value>
   ```

   This will make sure the execution of your program will not block both GPUs. You can check the value of this environment variable like this:

   ```
   echo $CUDA_VISIBLE_DEVICES
   ```

2. Make PGI's OpenACC compiler available by switching from the default Intel compiler to the PGI compiler:

   ```
   module switch intel pgi/18.10
   ```

3. Load the CUDA toolkit to make its tools available

   ```
   module load cuda/92
   ```

1.2 Getting GPU Information

Before you start programming GPGPUs, check your used GPU hardware by:

```
pgaccelinfo
```

If everything works properly, you will get a list of the most important features of your GPU. Complete Table 1 with the Cluster GPU details.

Table 1: Output of *pgaccelinfo*

Feature	Value
Name of device	
Number of multiprocessors	
CUDA compute capability (cc)[1]	

1.3 Compiling & Executing the Examples

In the `GPU` directory, you can find all sources for the programming lab. The directory structure looks as follows:

- exercises
- solutions
- openmp

In the `exercises` folder, you will find C-skeletons for all tasks that will be covered during this lab. You can compile and run your code with the provided Makefiles:

```
make help
make [jacobi]
       dbg=1
make run
       threads=<numOmpThreads>  |
       time=1                   |
       notify=<verbosityLevel>  |
       rows=<rowsOnCPU>         ]

make gprof
make clean
```

Get information
Compile
  - *with debug information*
Run
  - *with given number of OpenMP threads*
  - *enable timing information*
  - *enable runtime notifications*
  - *modify the no of matrix rows on the CPU (only hetero versions)*
Profile the (CPU) code using gprof
Clean

[1] The compute capability (cc) corresponds to the core architecture of the GPU and describes the features supported by the CUDA-capable GPU. For instance, you need a device of cc 1.3 or higher to enable double precision floating point operations. The PGI compiler calls this `device revision number`.

2 Jacobi Iteration

During the following exercises, you will port a Jacobi solver to OpenACC. This **Jacobi** example solves a finite difference discretization (5-point-stencil) of the Laplace equation (2D):

$$\boldsymbol{\nabla^2 A(x,y) = 0}$$

using the Jacobi iterative method. To this end, the Jacobi method starts with an approximation of the objective function *f(x,y)* and reuses formerly-computed matrix elements to solve the current one (see Figure 2). It iterates only about the inner elements of the 2D-grid (see Figure 1) so that the boundary elements are only used within the stencil. The solving process is aborted if either a certain number of iterations is achieved (see `iter_max`) or the computed approximation is probably close to the solution. In this code, the latter is evaluated by checking whether the biggest change on any matrix element (see array `err` and variable `err`) is smaller than a given tolerance value, in the current iteration.

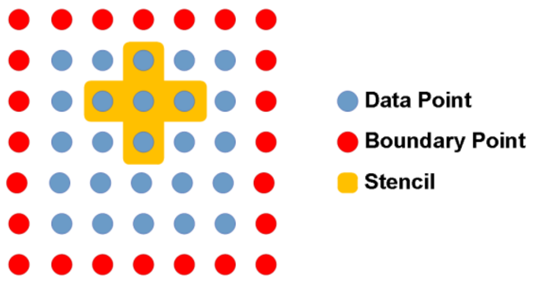
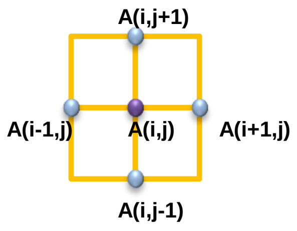


Figure 1: 5-point stencil



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Figure 2: Computation of matrix element A(i,j)

2.1 Reference Version

First, execute the OpenMP reference version:

a) Move to the folder `openmp`.

b) Profile the <u>serial</u> code using gprof by calling `make gprof`. It will take one to two minutes. The console output contains a “Flat profile” that lists the percentages of runtime for compute-intensive code lines. Find out where the three most time-consuming code lines are in the code. They will map to a certain for loops.

c) Now, try executing the OpenMP Jacobi version with 48 threads by running `make run threads=48`

d) Check the output and write down the runtime in Table 2.

Table 2: Runtimes of different Jacobi implementations

Task	Software	Hardware	Runtime [sec]
1	OpenMP	2x Intel Skylake (= 48 cores)	
2	OpenACC-Offload	1x NVIDIA Tesla V100	
4	OpenACC-Data	1x NVIDIA Tesla V100	
5	OpenACC-Collapse	1x NVIDIA Tesla V100	
6	OpenACC-Hetero	1x NVIDIA Tesla V100 + 2x Intel Skylake	
7	OpenACC-MultiGPU	2x NVIDIA Tesla V100 + 2x Intel Skylake	
8	OpenMP-PtrSwap	2x Intel Skylake (= 48 cores)	
8	OpenACC-PtrSwap	1x NVIDIA Tesla V100	

2.2 Offloading Work

Now, you start writing your first OpenACC program. Move to the folder `exercises/task1_basic` and modify the source code file `jacobi.c`. Follow the `TODO`s in the code:

a) Use the `acc parallel` and `loop` directives to parallelize (only) the *one* most compute-intensive loop from section 2.1. Denote all needed clauses.
b) Compile your code using `make` and have a look at the output.
    a) Make sure that, for GPU kernels, the line "`Accelerator kernel generated`" is printed.
    b) Check which data and how many elements are moved forth and back to the GPU.
    c) Execute your program using the command `make run`. It might take a few minutes. How fast does this version execute? Write down the runtime in Table 2.

Optional Task 1 – Pinned Memory

a) Read slide 138 about the usage of pinned memory
b) Take the solution from Task 2.2 as foundation. Open the corresponding `Makefile` and apply the `pinned` flag to the `CFLAGS` and `LDFLAGS` variables.
c) Recompile the code: `make clean & make`
d) Run the code using pinned memory and compare the runtime to the version without pinned memory.

Optional Task 2 – Roofline Model

a) Work on Task 2.9 "Roofline Model"
b) Use the task7_roofline code version as mentioned

2.3 Tools

As you might have recognized, your first OpenACC version is slow. In this task, you will figure out why. To this end, using profiling tools are a good approach. `IMPORTANT`: Please reduce the number of iterations to 5 (variable `iter_max`) when profiling your program to avoid long waiting times.

PGI Timing Environment

The PGI compiler enables a simple way to get some basic timing information of your code. You just have to set the environment variable `PGI_ACC_TIME` to a positive integer. Using the Makefiles provided, you can enable this option by running your code with:

```
make run time=1
```

a) Reduce the number of iteration (variable `iter_max`) in `jacobi.c` to `5`.
b) Compile your code and run it using the timing flag mentioned above. A small runtime overhead might be introduced for collecting corresponding data.
c) Examine the output at the end of the program run. How much time was spent for the kernel execution and how much time was spent for the data transfers?

NVIDIA Visual Profiler

Another way to analyze the performance of your code is NVIDIA's Visual Profiler that ships with the CUDA toolkit. It provides a graphical user interface and more detailed information on kernel executions. If you have any problems with the Java Runtime, set `export JAVA_TOOL_OPTIONS="-Xmx4096m -Dorg.eclipse.swt.internal.gtk.cairoGraphics=false"`.

a) Make sure you have switched the compiler from `intel` to `pgi` and loaded the `cuda` module (see instructions at the beginning of the exercise sheet) before starting the profiler.
b) Then, start the Visual Profiler: `nvvp &`
c) Create a new session.
d) In the `Executable Properties`, choose your executable file.
e) Click `Next`, disable `unified memory profiling` and click `Finish`.
f) In the left pane, click on `MemCpy(HtoD)` and `MemCpy(DtoH)`. Now, you can see the duration of the `Memcpy` command on the right hand side in the tab `Properties`. If you click on the different kernels that are listed under `Compute` (left pane), the `kernel` duration is displayed in the properties tab as the sum of all kernel executions.
g) Now, try the `Guided Analysis`: Activate the `Analysis` tab and `Examine GPU Usage`. Have a look at the first entry. What does it say? If you have lots of time left for the lab session, also have a look at the other entries. Read their explanations. Can we do anything about these issues?
h) Can you see where data is moved between host and device in the timeline? It might be necessary to zoom into the timeline. When do we want to have the data copied between host and device?
i) Close the Visual Profiler's tab as soon as you are done to enable other participants to work with the tool.

If you need help in understanding the plots/tables, ask one of our team members.

PGI OpenACC Debugging

Debugging with PGI's OpenACC is supported by common debuggers such as RogueWave's TotalView or ARM's DDT. However, for the purpose of this task, we rely on PGI's command-line feedback.

a) Some offload information is available during runtime using the environment variable `PGI_ACC_NOTIFY`. Using the provided Makefile, you can enable this by
`make run notify=3`
b) Run your program (`make run notify=3`). Which information do you get?

2.4 Data Transfers

As starting point for the second OpenACC programming task, you can either use your source code that you have just created or you can move to the folder `task2_data` and work on the source files located there (and follow the `TODO`s in the code).

a) Increase the number of iterations back to `100` (variable `iter_max`).
b) Use the `acc data` directive to remove the excess of data transfers. You may offload more loops to the GPU and use the `present` clause for defining the data status.

c) Examine the compiler feedback. Can you see any changes?

d) Use to `make run` to execute your program. How fast is your program now? Write down the runtime in Table 2.

e) Profile your code again using PGI's timing information (`make run time=1`) or the Visual Profiler (`nvvp &`). Can you see any changes? Again, close the profiler as soon as you are done.

Optional Task 1 – Eliminating Data Swapping

a) Work on Task 2.8 "Eliminating Data Swapping"

Optional Task 2 – Roofline Model (if not yet done)

a) Work on Task 2.9 "Roofline Model"
b) Use the task7_roofline code version as mentioned

2.5 Further Tuning

In this task, we want to improve the occupancy of the compute-intensive kernel. You can either use your code from the previous task or use the code located in the folder `task3_tuning`.

a) Reduce the number of iterations to `5` (variable `iter_max`) before starting the performance analysis.

b) Due to a bug in the NVIDIA Visual Profiler, you need to use the command line tool `nvprof`. One advantage of `nvprof` is that you can run it without interaction. Collect all needed metrics for a guided analysis by specifying:
```
nvprof -f -o base.nvprof --analysis-metrics <executable>
```
This may take several minutes.

c) Now, open the NVIDIA Visual Profiler on your GPU cluster node: `nvvp &`

d) Import the file `base.nvprof` that you have created in step b: `File` → `Import` → `Nvprof` → `Single process`. Then select the file as `timeline data file`. Disable `fixed width segments for unified memory timeline`.

e) Go to the `Analysis` tab and make sure that the `Guided analysis` button is selected. Click on `Examine Individual Kernels`.

f) Choose the top kernel (most compute-intensive one) in the right pane. Then, `Perform Kernel Analysis`. What does the tool suggest to be the main performance limiter of that kernel?

g) Continue to investigate this performance limiter. Click on `Perform Memory Bandwidth Analysis` and investigate `GPU Utilization`.

h) The NVIDIA Profiler suggests to work on `Shared Memory`. While this is a good idea for stencil codes, the given hints will not help a lot. Instead, you should try to increase the amount of shared memory used. This can be done by the `cache` clause or implicitly by restructuring the data access so that the compiler can optimize on shared memory itself. Follow the latter approach by using the `tile` clause. Apply the `tile` clause to the most-compute intensive loop and also the second GPU loop. Play around with different `tile` sizes.

i) Collect `nvprof` data from this tuned program version by running:
```
nvprof -f -o tuned.nvprof --analysis-metrics <executable>
```

j) How much speedup did you get? Write down the runtime in Table 2.
k) Now, redo the performance analysis by importing `tuned.nvprof`. Directly compare the performance limiters and the shared memory bandwidth.

Optional Task 1 – Loop Schedules

a) Read slides 110-122 on launch configurations and loop schedules
b) Check out the loop schedule that is automatically applied to the Jacobi code (see compiler feedback)
c) Try out different launch configurations and loop schedules and investigate the performance difference.
d) Especially, try `gang vector` on the outer loop and `seq` on the inner loop of the loop nests. Investigate the performance. Why is the runtime so much slower? You can use the Visual Profiler to get an idea.

Optional Task 2 – Eliminating Data Swapping (if not yet done)

a) Work on Task 2.8 “Eliminating Data Swapping”

Optional Task 3 – Roofline Model (if not yet done)

a) Work on Task 2.9 “Roofline Model”
b) Use the task7_roofline code version as mentioned

2.6 Heterogeneous Computing

So far, the compute-intensive code ran only on the GPU. However, it is often a good idea to utilize all available compute resources. Therefore, you should enable heterogeneous computing in this task by letting the CPU compute simultaneously to the GPU.
In this case, you should give the GPU more work to do than the CPU by distributing matrix rows to the GPU (see

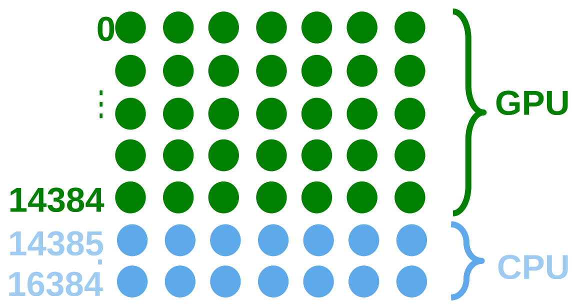


Figure 3). Be aware that you have to exchange some halo data between host and device (Figure 4) in each iteration.

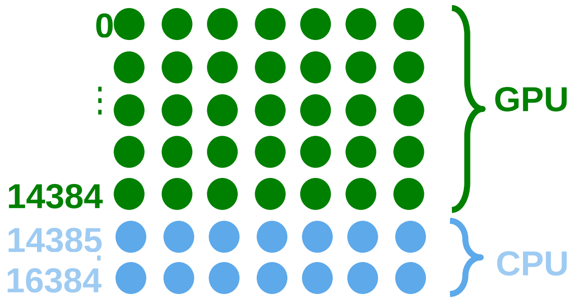


Figure 3: Work distribution between CPU and GPU

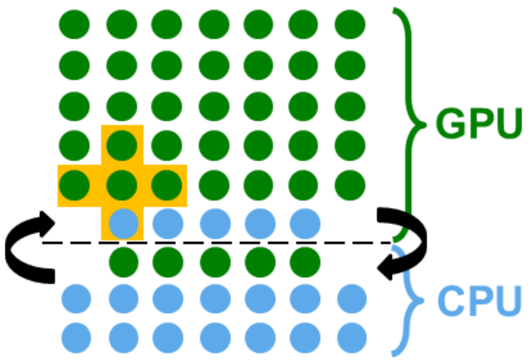


Figure 4: Halo exchange between CPU and GPU

Please use the skeleton in folder `task4_hetero` and follow these instructions and the `TODO`s in the code (recommended because of limited lab time) [or see "or" below].

a) Decompose the matrix rows to GPU and CPU. See the variable `n_cpu` that denotes the number of rows that shall be computed on the CPU. Use it and the corresponding index `j_cpu_start` to distribute the loops. Later you will have to adapt this value.
b) Use the `async` clause to enable overlap of CPU and GPU computational work.
c) Use OpenMP to fully utilize the CPU. You need `#pragma omp parallel for` on the for loops of the iteration process. However, you should also specify OpenMP's data and `reduction` clauses if appropriate. Some loops were already parallelized with OpenMP for you to ensure good data locality. If you run the application at the end, don't forget to increase the number of threads by `make run threads=<noThreads>`.
d) Use the `update` directive to manage the halo data exchange. While it is possible to also overlap data transfers with computations using the `async` clause on `update`s directives, you should *not* implement it for this task.
e) The OpenACC specification says that reduction variables are directly copied back to the host after the loop. However, this would prevent simultaneous execution of the first loop on CPU and GPU. Therefore, you have to decouple the reduction copy of `acc_err` from the computation and update its value manually: Put `acc_err` into the data region and `update` it before and (when needed) after the computation loop.
f) Think about synchronizing data again. Do you have to insert a `wait` directive to avoid inconsistent data?
g) Fix the calculation of `err`.
h) What is the runtime of this heterogeneous version? Play around with the decomposition size of the matrix. Use `make run rows=<rowsOnCPU> threads=<noThreads>` to do so. Write down the shortest runtime in Table 2.

or start from your source code that you have already implemented and follow these steps

a) Define a variable that denotes the matrix split between host and device. For this domain decomposition, which data is needed on which architecture? Don't forget that we need a stencil for updating one matrix element.
b) Split all compute-intensive loops along this variable (matrix element computation and swap loop). Make sure to reduce the error variable of both loop parts after splitting the reduction loop.

c) Afterwards, execute one part of the loop on the GPU and simultaneously the other part of the loop on the CPU. You might need the `async` clause.

d) Before the swap loop, the halo data must be exchanged (see Figure 3) which can be done using the `update` directive. While it is possible to also overlap data transfers with computations using the `async` clause on `update`s directives, you should *not* implement it for this task.

e) The OpenACC specification says that reduction variables are directly copied back to the host after the loop. However, this would prevent simultaneous execution of the first loop on CPU and GPU. Therefore, you have to decouple the reduction copy of `acc_err` from the computation and update its value manually: Put `acc_err` into the data region and `update` it before and (when needed) after the computation loop.

f) Think about synchronizing data again. Where must a `wait` directive be used to avoid inconsistent data?

g) Parallelize the CPU code using OpenMP. You will usually only need `#pragma omp parallel for`. You should also use OpenMP's `reduction` clause where appropriate. If you run the application at the end, don't forget to increase the number of threads by `make run threads=<noThreads>`.

h) What is the runtime of this heterogeneous version? Play around with the decomposition size of the matrix. Write down the shortest runtime in Table 2.

2.7 Multiple GPUs

If you have a cluster of GPU nodes, you can utilize their compute power by having an MPI program that runs on different nodes with GPUs. If you have several GPUs within one node, you can use both accelerators simultaneously even easier by specifying in your program which one to use by OpenACC API calls. Here, you will do the latter.
To distribute the work, we follow the strategy described in section 2.6 *Heterogeneous Computing*. The only difference is that we now have three partitions: one on GPU 0, one on GPU 1 and one on the CPU (as shown in Figure 5).

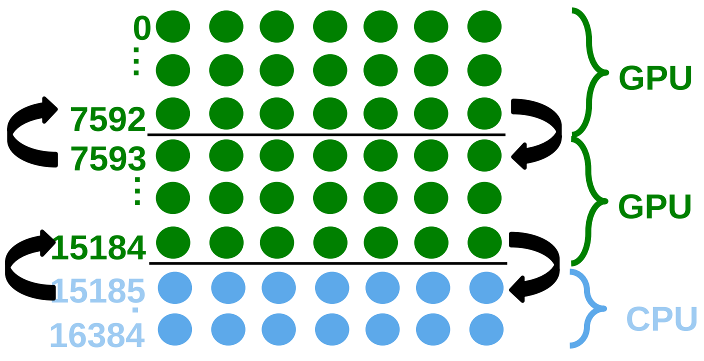


Figure 5: Work distribution between two GPUs and a 2-socket CPUs

You can either start from your source code that you have already implemented or use the skeleton in the folder `task5_multigpu` (and follow the `TODO`s in the code).

a) On the CLAIX-GPU nodes, you have 2 GPUs available. Make sure to temporarily (!) set `export CUDA_VISIBLE_DEVICES=0,1` to gain access to both.

b) Leave the CPU matrix size as it is and divide evenly the number of rows, which were located on one GPU so far, to both GPUs.
c) For specifying that the following OpenACC pragmas shall be executed on a certain GPU, the OpenACC API call `acc_set_device_num(<id>,acc_device_nvidia)` is used. You have to include `openacc.h` for using API routines.
d) Start by moving only the necessary data to GPU 0 and GPU 1. Therefore, exchange the structured `data` region to unstructured `enter data` directives. Don't forget to `create` the reduction variables for both GPUs explicitly on the devices. After the equation system was solved, delete temporary data and copy back the result matrix. Use `exit data` for that.
e) Note that you have to manually `update` the reduction variables on both GPUs now.
f) Before the swap loop, the halo data must be exchanged (see Figure 5) which can be done using the `update` directive. The first GPU needs to update its last matrix row, the second GPU needs to update its first and last row and the CPU needs to update its first row. Make sure that the data is synchronized.
g) Combine the reduction variables of both GPUs and the CPU.
h) Determine a good value for the number of rows on the CPU by playing around with `make run rows=<rowsOnCPU> threads=<noThreads>`. Use the results from the previous task to obtain reasonable starting values.
i) What is the runtime of this heterogeneous multi-device version (`make run rows=<rowsOnCPU> threads=<noThreads>`)? Write down the shortest runtime in Table 2.

2.8 Eliminating Data Swapping

When (or even before) diving into GPU and kernel tuning, you should usually also consider algorithmic optimizations. In this task, you will eliminate the second computational loop and thereby decrease runtime.

As starting point for this task, you can either use your source code that you created in Task 2.5 (!) or you can move to the folder `task6_ptrswap` and work on the source files located there (and follow the `TODO`s in the code).

a) Make sure to `copyin` matrix `A` and `create` matrix `Anew` on the device. To avoid accessing uninitialized elements, the boundaries of matrix `Anew` need to be initialized. Copy the required values from `A` into `Anew` using an additional accelerator kernel.
b) Eliminate the second loop which just copies the matrix `Anew` into the original matrix `A` by using *host pointer swapping*.
c) How fast is your program now (`make run`)? Write down the runtime in Table 2.
d) To obtain a comparable runtime for the reference OpenMP version, go to the `openmp` directory and run it with pointer-swapping enabled by using
`make run ptr_swap=1 threads=24`
Write down the runtime in Table 2.

2.9 Roofline Model

Finally, we might want to ask how close our current version is compared to sustainable peak performance on that particular GPU. For that, the roofline model[2] is a good approach. The

[2] Williams, S., Waterman, A., Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM, 65–76, 2009.

roofline model defines the peak performance of an architecture by looking at the memory bandwidth (for memory-bound kernels) and on the theoretical peak GFlop/s (for compute-bound kernels). The operational intensity [Flop/Byte] is given by the algorithm and thereby defines the performance limit.

In Figure 6, you can find the roofline for an NVIDIA Volta V100[3]. The kernel's operational intensity can be determined either by manual counting Flops and Bytes in the source code, or by approximating the corresponding values by measurement. Here, you will look at the measured values by using the NVIDIA Profiler.

We only consider the remaining compute-intensive kernel (without the reduction). You can either use your code from the previous exercises (but no heterogeneous/multi GPU computations) or the code from the directory `task7_roofline`.

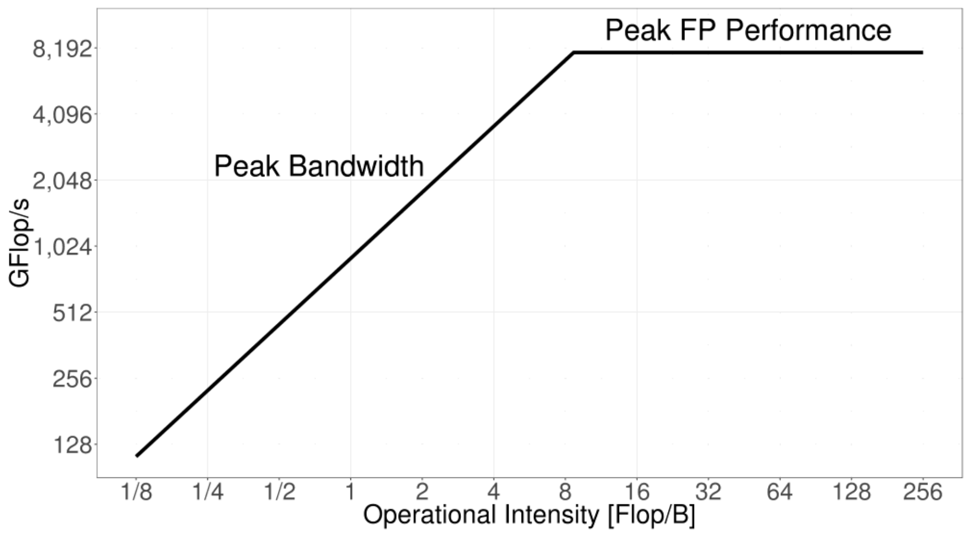


Figure 6: Performance roofline for a NVIDIA Volta V100 GPU. Peak double precision floating point performance are 7.8 TFlop/s. The peak sustainable memory bandwidth (to device memory) is 900 GB/s.

a) We use performance counters on floating point operations and memory instructions to compute the operational intensity. For this task, you will use the command line tool of the NVIDIA Profiler (named `nvprof`) - although all measurements are also configurable in the GUI. `IMPORTANT`: Please reduce the number of iterations to 5 (variable `iter_max`) when profiling your program to avoid long waiting times.
```
nvprof --metrics flop_count_dp,dram_read_transactions,dram_write_transactions ./jacobi
```
Also collect the kernel runtimes:
```
nvprof --print-gpu-trace ./jacobi
```
b) Collect the average values for the kernel, and compute the operational intensity for double precision by:

$$\frac{\text{FLOP}}{\text{Byte}} = \frac{\text{flop\_count\_dp}}{(\text{dram\_read\_transactions} + \text{dram\_write\_transactions}) \cdot 32}$$

Note that you have to multiply the number of transaction to/from the device memory by 32 since each transactions takes place in 32 Byte chunks.

[3] NVIDIA: Specifications: http://www.nvidia.com/object/product-quadro-6000-us.html

c) Read the maximum theoretical performance from Figure 6 or compute it:

$$\min(\text{operational intensity [Flop/B]} \cdot 900\ \text{[GB/s]},\ 7.8\ \text{[TFlop/s]})$$

d) To evaluate which portion of the theoretical peak performance is reached, you have to read the duration (in seconds) of the kernel from the GPU trace:

$$\text{flop\_count\_dp [Flop]} \ /\ \text{duration [s]}$$

e) How close does the kernel get to the theoretical peak performance?

Note that this approach (that evaluates the operational intensity by measuring) is not the most accurate one, but a good first approach. Also, be aware that the measured values must not be the optimal ones. For example, if you do additional floating point operations in the kernel that do not contribute to the actual result, the measured operational intensity might get mistakenly increased.