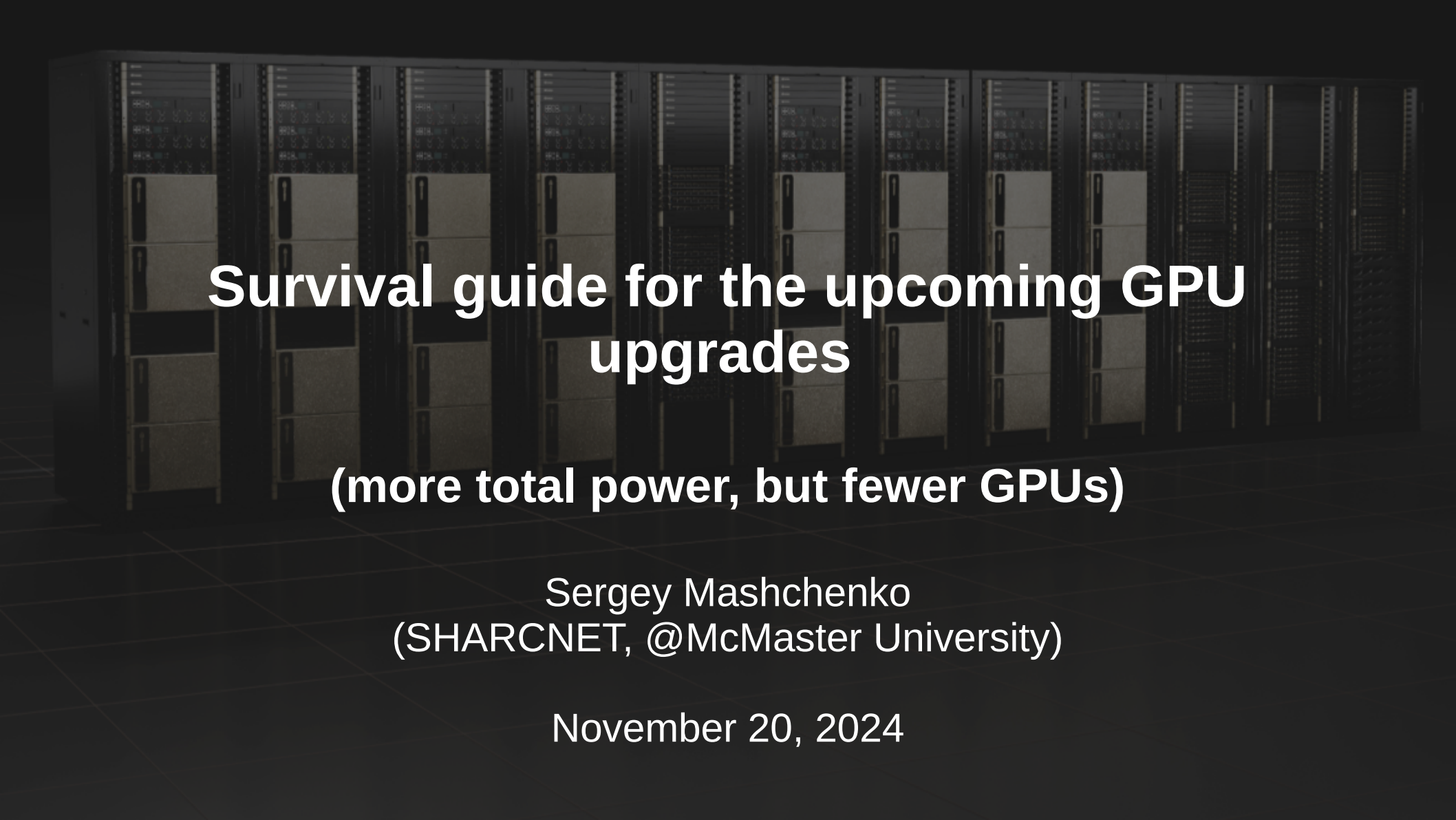


Survival guide for the upcoming GPU upgrades

(more total power, but fewer GPUs)

Sergey Mashchenko
(SHARCNET, @McMaster University)

November 20, 2024

Outline

- What problem are we trying to solve?
- Possible solutions
  - CUDA streams
  - Multi-Instance GPU (MIG)
  - Multi-Process Service (MPS)
- Picking the right solution
- Live demo

What problem are we trying to solve?

Amdahl's Law

- Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

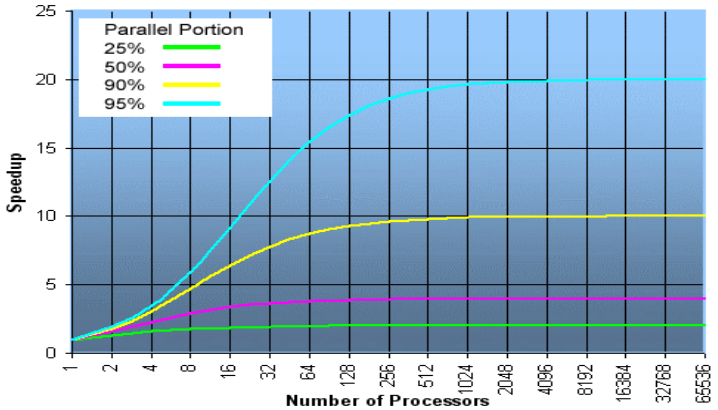
$$\text{speedup} = \frac{1}{\frac{P}{N} + 1-P}$$

where N is the number of processors.

Amdahl's Law visualized

- It soon becomes obvious that there are limits to the scalability of parallelism. For example, at P = .50, .90 and .99 (50%, 90% and 99% of the code is parallelizable):

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02



How to waste GPU cycles?

- From the Amdahl’s law, moving a computational code (solving a *fixed size problem*) from an older smaller GPU to a modern larger GPU might result in wasting a significant fraction of the GPU cycles.
  - This happens **if the problem size is too small** to saturate a modern massively parallel GPU.
  - One possible solution is to increase the size of the problem (use higher resolution in CFD, larger batch size in DL etc.).
    - According to the Gustafson's law, increasing the problem size while increasing the number of computing cores can result in a much better scaling.
  - This is often not the right solution.

More ways to waste GPU cycles

- Another way to waste a significant fraction of a GPU cycles is when your code uses the GPU in short bursts.
  - During such bursts, the GPU may be used quite efficiently (if the problem size is large enough) – but not necessarily!
  - The problem arises from the fact that most of the time the GPU is idle.
  - The interval between bursts can be very short (milliseconds), but still have a dramatic impact on the code efficiency, if the burst time is much shorter than the interval time.

What problem are we trying to solve?

- In the national systems, we have a relatively small number of very powerful (and expensive!) GPUs.
  - As a result, we have a huge demand for GPUs – much more so than for CPUs.
- To add an insult to injury, many (likely most) GPU jobs have low efficiency (effectively just using a fraction of a GPU).
- The solution to this would be some way of sharing a single GPU – either between the user's processes, or between different users.
- Fortunately NVIDIA provides a few ways to achieve this.

Why now?

- The problem of underutilizing modern large GPUs will become much more pronounced when the national systems will undergo upgrades in the next 3-4 months.
- During the upgrades the oldest GPUs (P100, V100) will be removed, and the newer H100 GPUs will be installed instead.
  - Narval cluster will not be upgraded this time, so its 636 A100 GPUs will stay.
- After the upgrades, the combined compute power in GPUs will grow by a factor of 3.5x (from ~6000 RGU to ~21,000 RGU), but **the number of GPUs will actually go down** – from 3200 to 2100.
  - This will exacerbate the problem we already have with inefficient GPU jobs.

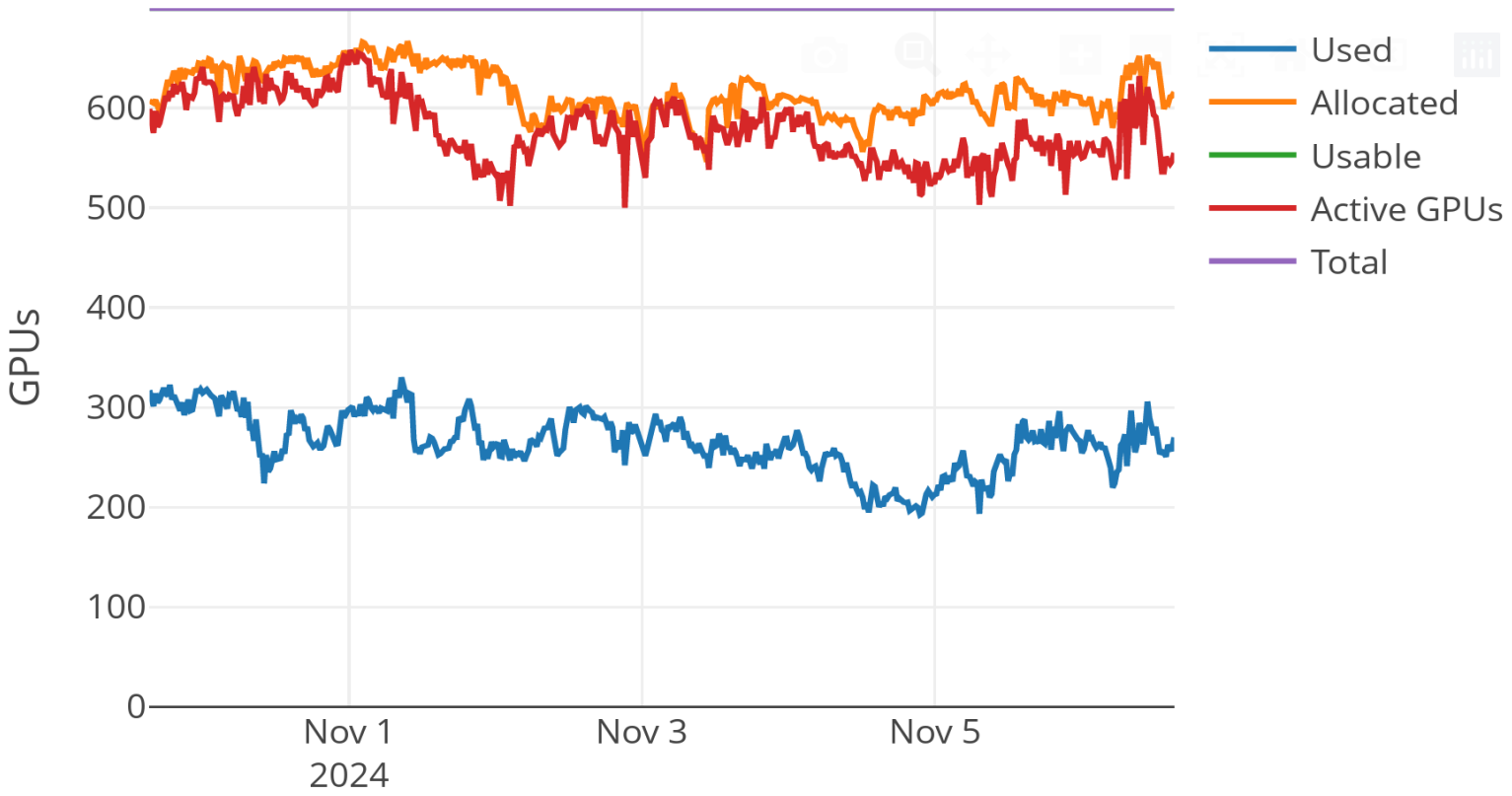
Reference GPU Units (RGU)

	FP32 score	FP16 score	Memory score	Combined score	Available	
Coefficient:	**1.6**	**1.6**	**0.8**	**(RGU)**	**Now**	**2025**
H100-80gb	3.44	3.17	2.0	12.2	No	Yes
A100-80gb	1.00	1.00	2.0	4.8	No	?
A100-40gb	**1.00**	**1.00**	**1.0**	**4.0**	Yes	Yes
V100-32gb	0.81	0.40	0.8	2.6	Yes	?
V100-16gb	0.81	0.40	0.4	2.2	Yes	?
T4-16gb	0.42	0.21	0.4	1.3	Yes	?
P100-16gb	0.48	0.03	0.4	1.1	Yes	No
P100-12gb	0.48	0.03	0.3	1.0	Yes	No

GPUs in upgraded systems

Cluster	Number of GPUs	GPU model	RGU per GPU	Bundle per RGU	Bundle per GPU
Fir (Cedar)	640	H100-80GB	12.2	3.0 cores & 21 GiB	12 cores & 256 GiB
Rorqual (Beluga)	324	H100-80GB	12.2	1.3 cores & 10.3 GiB	16 cores & 124 GiB
Trillium (Niagara)	240	H100-80GB	12.2	2.0 cores & 15 GiB	24 cores & 192 GiB
Graham2	288	H100-80GB	12.2	3.0 cores & 10.3 GiB	12 cores & 124 GiB
Narval	636	A100-40gb	4.0	3.0 cores & 31 GiB	12 cores & 124 GiB

Current GPU jobs efficiency (Narval)



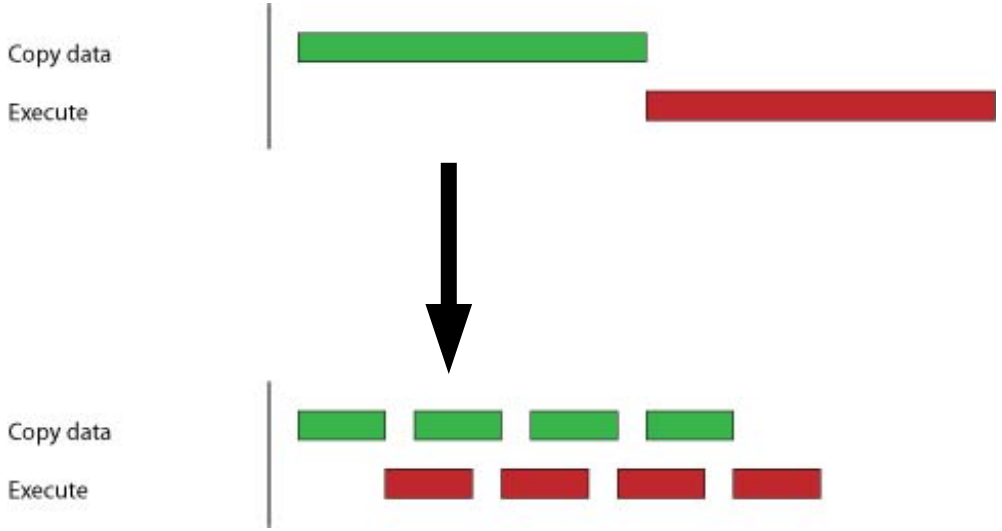
Possible solutions

CUDA streams

- The oldest NVIDIA solution for sharing a GPU between unrelated tasks is CUDA streams.
  - It's been around since the first days of CUDA.
  - A CUDA stream is a queue of GPU facing commands (kernels, memcopy etc.), coming from **a single process**.
  - When more than one stream is defined in the user code, unrelated GPU operations (kernels, memcopy, ...) can run concurrently – if the resources permit.
  - The biggest drawback: this is limited to a single process.
    - If your process simply doesn't have enough of parallelism to saturate a modern GPU, streams are not useful.
  - Another drawback: this requires re-writing the code, which can be very time consuming, or even not possible (if you are using someone else's code).

Stream example

- Streams are often used to hide the high cost of moving the data between the CPU and the GPU.
- This approach is called “staged concurrent copy and execute”.
  - In this approach, when dealing with **data parallel** processing, one single large data copy CPU->GPU followed by a large compute kernel is replaced by concurrent coping and computing in smaller segments, organized as two streams: copy stream, and execute stream.

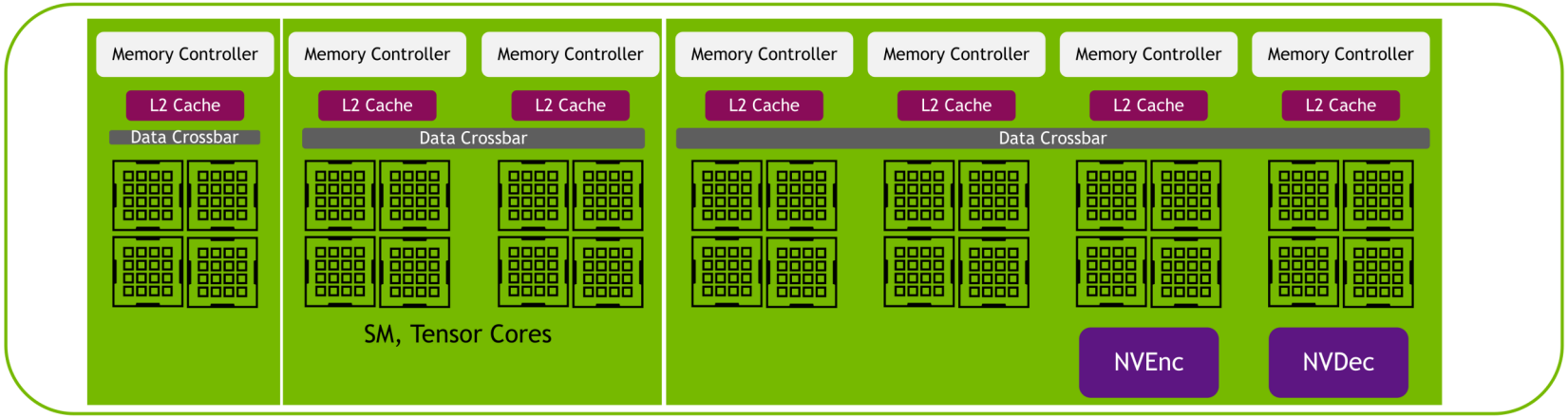


MIG and MPS

- In this presentation, we will focus on two other methods of sharing a GPU which, unlike streams, do not require rewriting the code.
  - In fact, the code doesn't even need to be re-compiled!
- The first method is the **static** GPU fragmentation/virtualization framework called **MIG** (Multi-Instance GPU). Available since Ampere (e.g. A100).
- The second method is the **dynamic** sharing of a GPU by unrelated processes called **MPS** (used to be called Hyper-Q) – Multi-Process Service. Has been around much longer – since Kepler (e.g. K20).
- The two approaches have their Pro's and Con's, so your code might benefit the most from one or another (or both, or neither).

Introduction to MIG

- MIG is a technology that enables the partitioning of a single GPU into multiple, isolated environments, each with its own dedicated memory and resources.
  - This allows multiple applications to run concurrently on a single GPU, increasing overall utilization and efficiency.
  - MIG is particularly useful for applications that require a high degree of isolation and security, such as those in the financial or healthcare industries.



MIG: some details

- We can segment the GPU into up to 7 physically discrete instances (for both A100 and H100).
  - Memory is split into 8 equal size segments.
  - Compute (SMs) is also split into 8 segments, but only 7 segments are available for MIGs.
  - This implies the MIG overhead in terms of computing power of ~10%.
- Each instance has dedicated memory and processing.
- This technology allows for an easy and safe sharing of a GPU between different jobs (and users).
- Only a limited set of compute+memory configurations (“MIG profiles”) is available.

Partitioning: A100

- On narval, currently only two profiles are available: **3g.20gb** and **4g.20gb**.
- To request one of these profiles, or a full sized A100, use one of the following sbatch arguments:

```
--gres=gpu:a100_3g.20gb:1
--gres=gpu:a100_4g.20gb:1
--gres=gpu:a100:1
```

Profile Name	Fraction of Memory	Fraction of SMs
MIG 1g.5gb	1/8	1/7
MIG 1g.5gb+me	1/8	1/7
MIG 1g.10gb	1/8	1/7
MIG 2g.10gb	2/8	2/7
MIG 3g.20gb	4/8	3/7
MIG 4g.20gb	4/8	4/7

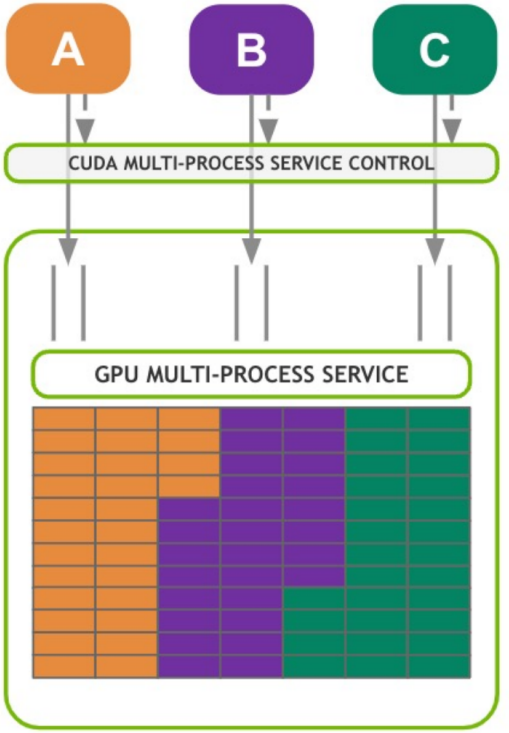
Partitioning: H100

- We have not decided yet which profiles to make available on upgraded systems.
- It is likely that all the flavours will be provided: from 1g.10gb to 4g.40gb.

Profile Name	Fraction of Memory	Fraction of SMs
MIG 1g.10gb	1/8	1/7
MIG 1g.10gb+me	1/8	1/7
MIG 1g.20gb	1/4	1/7
MIG 2g.20gb	2/8	2/7
MIG 3g.40gb	4/8	3/7
MIG 4g.40gb	4/8	4/7

Introduction to MPS

- MPS is a technology that allows multiple processes to share a single GPU, but with a focus on maximizing performance and minimizing overhead.
  - Unlike MIG, MPS does not provide isolation between processes, but instead, it optimizes the allocation of GPU resources to achieve the best possible performance.
  - MPS is ideal for applications that require high-performance computing and can tolerate some level of resource sharing.
- In MPS, the GPU is **dynamically** shared between multiple (could be unrelated) processes. Examples:
  - A group of MPI ranks sharing a single GPU.
  - GPU farming (sharing a GPU between multiple instances of a code).
    - Used for Monte Carlo simulations, parameter sweeps etc.



MPS: some details

- To make use of the MPS feature, include the following lines in your job script:

```
echo quit | nvidia-cuda-mps-control
export CUDA_MPS_LOG_DIRECTORY=/tmp/nvidia-log
nvidia-cuda-mps-control -d
```

Kills the MPS daemon

Launches the MPS daemon

- With MPS, the GPU can be shared between up to 48 processes.
- There is a memory overhead; for A100 it is 432MB + the execution code copy, for each process.
- **MPS can be used with MIGs.**

MPS for GPU farming

- To share a GPU between unrelated processes using MPS, follow these steps:
  - In your job script, request one GPU and multiple CPU cores (one or more core for each process), e.g.
    ```
    $ salloc --time=0-03:00 -c 16 -gres=gpu:a100:1 -A def-myaccount
    --mem=64G
    ```
  - Launch the MPS daemon inside the job script (previous slide)
  - You can launch multiple instances of your code (sharing a single GPU) inside the job script using the **for** loop, e.g.

```
for ((i=0; i<$N; i++))
  do
  ./my_GPU_code  &>$i.out  &
  done
wait
```

MPS: limiting compute resources per process

```
$ setenv CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=percentage
```

- Environment variable: configures maximum fraction of a GPU available to an MPS-attached process
- Guarantees a process will use at most percentage execution resources (SMs)
- Over-provisioning is permitted: sum across all MPS processes may exceed 100%
- Provisions only execution resources (SMs) – does not provision memory bandwidth or capacity

MPS: limiting memory per process

- What if some MPS clients try to monopolize all the available GPU memory?
- One can prevent that via a global or per-client memory limits.
- Default Global Limit

```
$ echo set_default_device_pinned_mem_limit 0 2G | nvidia-cuda-mps-control
```

- Per-Client Limit

```
$ export CUDA_MPS_PINNED_DEVICE_MEM_LIMIT="0=1G,1=2G”
```

Side by side comparison

	CUDA Streams	MPS	MIG
Partition Type	Single-Process	Logical	Physical
Max Partitions	Unlimited	48	7
SM Performance Isolation	No	By Percentage	Yes
Memory Protection	No	Yes	Yes
Memory Bandwidth QoS	No	No	Yes
Error Isolation	No	No	Yes
Cross-Partition Interop	Yes	IPC	Limited IPC
Reconfiguration	Dynamic	At Process Launch	When Idle

Picking the right solution

Does your code need this?

- Most likely **YES**.
  - Majority of GPU jobs running on our clusters cannot properly utilize whole GPUs, and should be subjected to this analysis.
- GPU utilization depends not only on the code, but also very strongly on the **problem size** (Amdahl's law!).
  - So even if you had good GPU utilization with your code before, when you change your problem, you need to re-evaluate the code performance.
- One exception: you cannot improve the GPU utilization using MPS and/or MIG technologies when your code needs all (or almost all) of the GPU memory
  - 40GB for A100, 80GB for H100.

Step 1: measure the GPU utilization

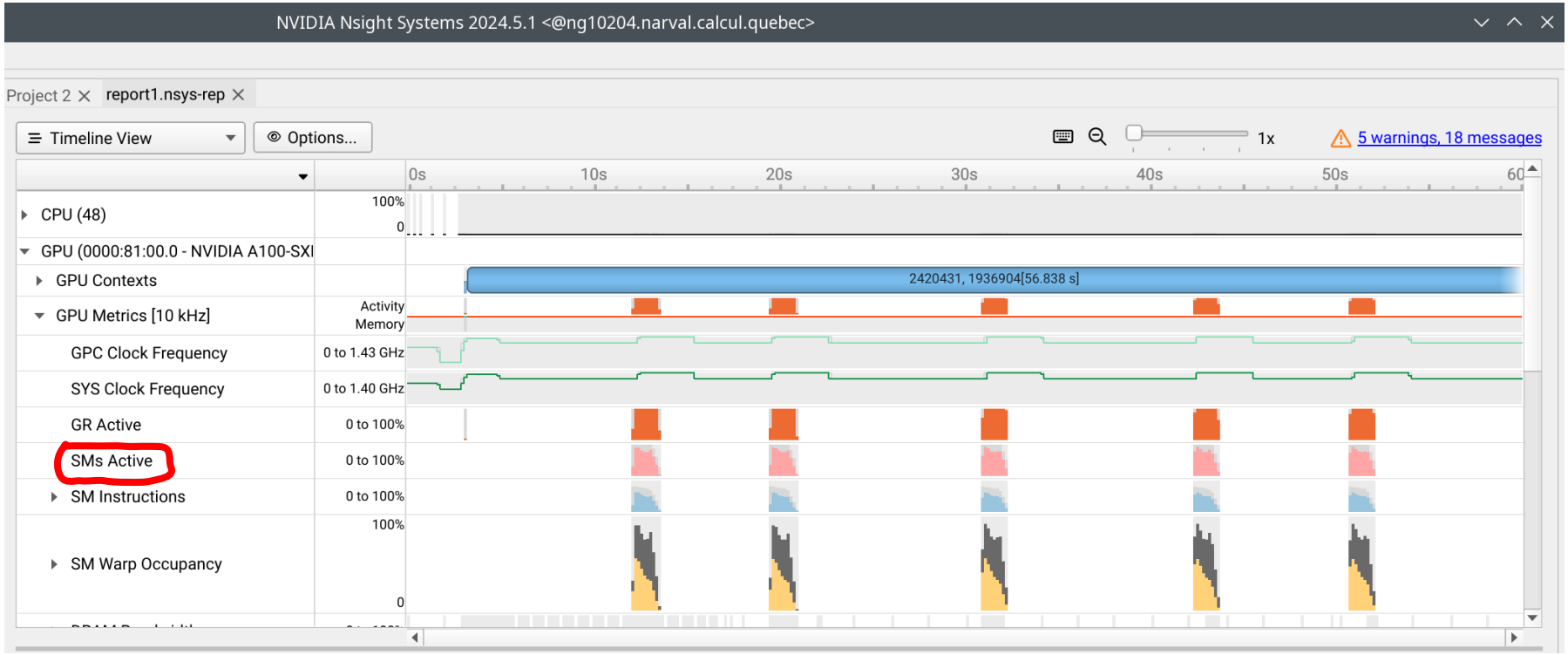
- You should start by running some test jobs, then analyzing the overall GPU utilization.
- A convenient way to do it is by accessing the “Jobs stats” tab of a cluster portal.
  - Only narval for now, but other clusters will be added as well, after the upgrades.
  - Search for the cluster page on our documentation site (e.g. https://docs.alliancecan.ca/wiki/Narval), then click on the Portal link at the top.
- If your GPU job
  - requires <50% of the GPU memory (<20GB on A100, <40GB on H100), and
  - has a GPU utilization <75%,

  it has to be used with either MPS, or MIG, or both.

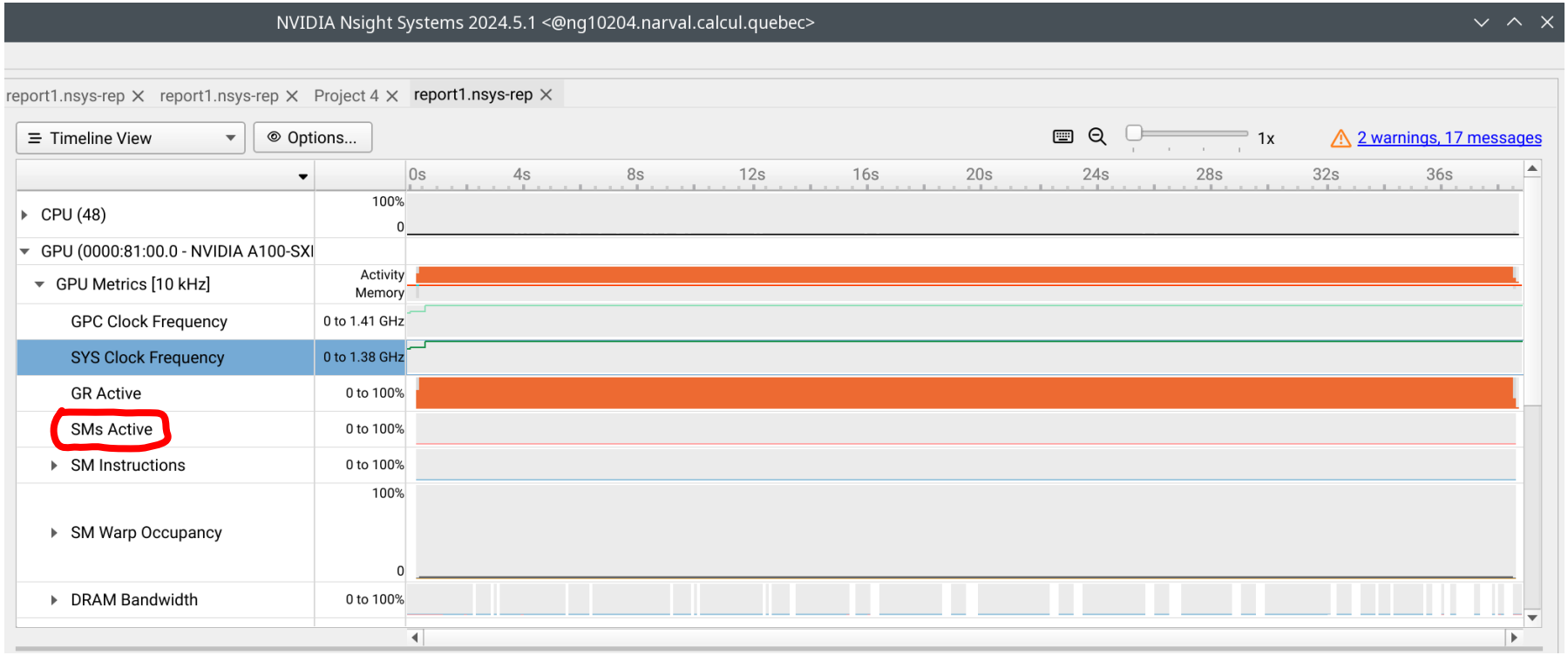
Optional: code profiling

- You may want to know where are the inefficiencies inside the code.
  - E.g., you want to know whether the inefficiency is due to the code being bursty, due to a small problem size, or perhaps both.
- The best profiler for NVIDIA GPUs is Nsight.
  - It is already installed on our systems (part of the “cuda” module).
  - It is a GUI application, so you need to use VNC or Jupyterhub to run it on the cluster.
    - Check the Jupyterhub page for details: https://docs.alliancecan.ca/wiki/JupyterHub
  - It comes in two packages:
    - Nsight Systems (nsys-ui): high level code profiling
    - Nsight Compute (ncu-ui): low level, line-by-line code profiling (individual kernels)
  - Check out my webinar “Profiling GPU codes with Nsight” on SHARCNET youtube channel: http://youtube.sharcnet.ca/

Bursty code example



Small problem size example



Step 2: first, consider MPS

- MPS is using dynamic work load balancing (vs. static fragmentation under MIG), so can result in better GPU utilization – **if the following condition is met**:
  - you either need to run multiple instances of your code (job farming), or
  - your code is an MPI code with GPU acceleration.
    - In other words, if you just need to run one GPU job which doesn't use MPI, MPS is the wrong tool.
    - This limitation is due to the fact that MPS cannot be used to (safely) share a GPU between different users.
- Double check – your job may be already using MPS.
  - pytorch-gpu-mps.sh is currently the only example on our documentation site (on PyTorch page: https://docs.alliancecan.ca/wiki/PyTorch).

MPS: how many clients per GPU?

- If your job(s) qualify for MPS, run a few more test jobs, using a different number of processes (or MPI ranks) per GPU under MPS control.
  - You can use any number **between 2 and 48**.
  - The upper limit will most likely be set by the memory available on the GPU (or CPU).
    - Once the number of clients is too high, your code will crash as it will run out of memory.
    - Memory crash on a GPU will not result in a classical SEGFAULT error.
    - Instead, test for the correctness of results. If the code has a GPU debugging option, you may want to turn it on, for better error catching.
  - Another factor to consider is the CPU-core to GPU ratio, which ranges from 12 (Narval, Graham2, Fir) to 16 (Rorqual) to 24 (Trillium).
  - Pick the smallest number of clients which will make the **GPU utilization 75% or better**.

Step 3: next, consider MIG

- If MPS doesn't work for you, test your job performance using different MIG profiles.
  - Use only the profiles which have enough of memory to run your job.
    - If not sure, try profiles with different memory sizes, and pick the profile with the smallest memory size which can still run your job without crashing.
  - On narval (A100 GPUs) we currently only have two profiles: 3g.20gb and 4g.20gb .
  - On upgraded clusters (H100 GPUs) we will most likely have all possible sizes: 1g, 2g, 3g, 4g.
  - When testing an MPI+GPU code, use equal size MIGs for your job.
- Pick the profile which gives you **>75% GPU utilization**.

What about MIG + MPS?

- It is perfectly fine to use MPS on a MIG.
  - The maximum number of MPS clients is reduced accordingly.
    - So instead of 48, you will get 24 for 4g, 18 for 3g, 12 for 2g, and 6 for 1g.
- Use MPS on a MIG under the following circumstances:
  - Your code gets the best GPU utilization (>75%) with MPS (and does worse with pure MIG), but
    - either you do not have enough of processes (MPI ranks) per whole GPU to get to the efficient regime,
    - or MIGs are much more available (resulting in significantly shorter queue wait time) than whole GPUs.

Step 4: switch to CPUs

- If everything else fails, you should test the CPU-only version of your code.
  - Large memory, low GPU utilization codes are the primary candidates.
- Chances are, you will get comparable runtime, and shorter queue wait time, if you do the transition.

Live Demo

Main takeaways

- There are lots of inefficient GPU jobs on our clusters. (Your job is likely one of them!)
  - This results in long queue wait times.
- The situation will get much worse after the upcoming cluster upgrades. (Despite the increased combined GPU compute power.)
- The two Nvidia technologies to share a GPU – MPS and MIG – can rectify the situation, and are very easy to use.

References

- **Alliance Infrastructure Renewal page:** https://docs.alliancecan.ca/wiki/Infrastructure_renewal
- **MIG User Guide (Nvidia):** https://docs.nvidia.com/datacenter/tesla/mig-user-guide/
- **Multi-Process Service (Nvidia):** https://docs.nvidia.com/deploy/mps/index.html
- **Optimizing GPU Utilization: Understanding MIG and MPS*:** https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41793/
- **Multi-Instance GPU (Alliance page):** https://docs.alliancecan.ca/wiki/Multi-Instance_GPU
- **Hyper-Q / MPS (Alliance page):** https://docs.alliancecan.ca/wiki/Hyper-Q_/_MPS

Questions?

You can contact me directly
(syam@sharcnet.ca)

or send an email to
help@sharcnet.ca