Appendix: CUDA Python Performance

In order to achieve optimal performance in CUDA, you must consider several factors:

- Localizing memory access in order to minimize memory latency.
- Maximizing the number of active threads per multiprocessor to ensure high utilization of your hardware.
- Minimization of conditional branching.

In order to overcome the bottleneck between CPU and GPU across the PCIe bus, we want to:

- Minimize the volume of data transferred. Transferring data in large batches can minimize the number of data transfer operations.
- Organize data in a way that complements the hardware architecture.
- Utilize asynchronous transfer features that will allow computation and data transfer to occur simultaneously. Overlapping data transfers with computation can hide latencies caused by data transfers.

Common Pitfalls

The most common mistake is running a CPU-only code on a GPU node. Only codes that have been explicitly written to run on a GPU can take advantage of a GPU. Ensure your codes are using the correct GPU accelerated libraries, drivers, and hardware.

Zero GPU Utilization

- Check to make sure your software is GPU enabled. Only codes that have been explicitly written to use GPUs can take advantage of them.
- Make sure your software environment is properly configured. In some cases certain libraries must be available for your code to run on GPUs. Check your dependencies, version of CUDA Toolkit, and your software environment requirements.

Low GPU Utilization (e.g. less than ~15%)

- Using more GPUs than necessary. You can find the optimal number of GPUs and CPU-cores by performing a scaling analysis.
- Check your process's throughput. If you are writing output to slow memory, making unnecessary copies, or switching between your CPU and GPU, you may see low utilization.

Memory Errors

- Access Violation Errors. Reading or writing to memory locations that are not allowed or permitted can result in unpredictable behavior and system crashes.
- Memory Leaks. When memory is allocated but not correctly deallocated, the application will consume GPU memory resources, but not utilize them. The allocated memory will not be available for further computation.

Debugging and Profiling CUDA Python

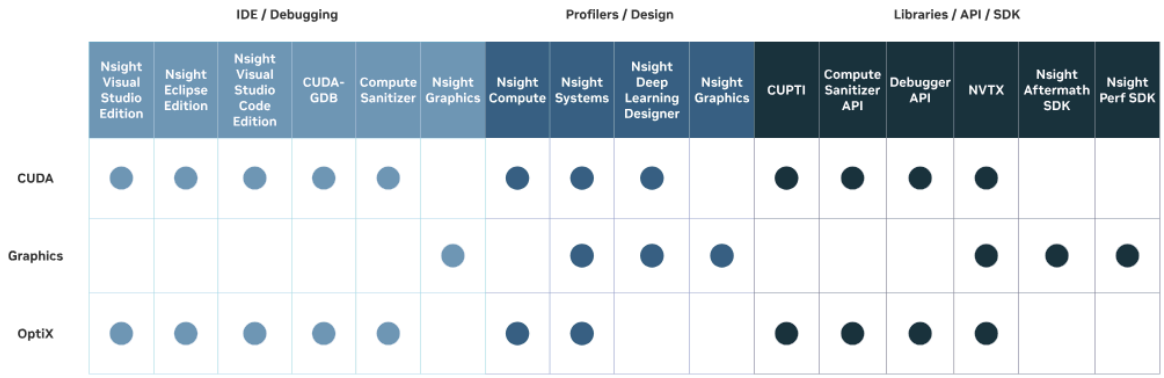
In order to take advantage of the optimizations available through CUDA, debugging and analyzing memory issues is essential to creating accelerated Python applications.

External Tools

Nsight

NVIDIA Nsight™ Systems is a system-wide performance analysis tool designed to visualize an application's algorithms, identify the largest opportunities to optimize, and tune to scale efficiently across any quantity or size of CPUs and GPUs, from large servers to our smallest systems-on-a-chip (SoCs).

This suite of tools offers an array of interactive as well as command-line tools. They can detect and provide insight into kernel execution and memory issues. Ultimately, we need memory usage to align with the GPU hardware preferences. When these two components are out of alignment, applications may use non-performant access or execution patterns.



(Image from https://developer.nvidia.com/tools-overview)

Internal and Integrated Tools

PyNVML

PyNVML provides Python bindings to the NVIDIA Management Library which contains a number of management and monitoring tools for GPU applications.
https://pypi.org/project/pynvml/

NOTE: PyNVML is unmaintained. Use at your own risk.

Cupyx.profiler - the CuPy Profiler

Internal to the CuPy package, there is an interface to an integrated profiler.
https://docs.cupy.dev/en/stable/user_guide/performance.html

NOTE: In CuPy - When you monitor the memory usage (e.g., using nvidia-smi for GPU memory or ps for CPU memory), you may notice that memory not being freed even after the array instance becomes out of scope. This is an expected behavior, as the default memory pool “caches” the allocated memory blocks.