National Supercomputing Centre

ASPIRE2A+ Advanced Workshop on Profiling and Debugging

Dr Malik M Barakathullah
Fujitsu Managed Services

NSCC.SG

National Supercomputing Centre

Contents

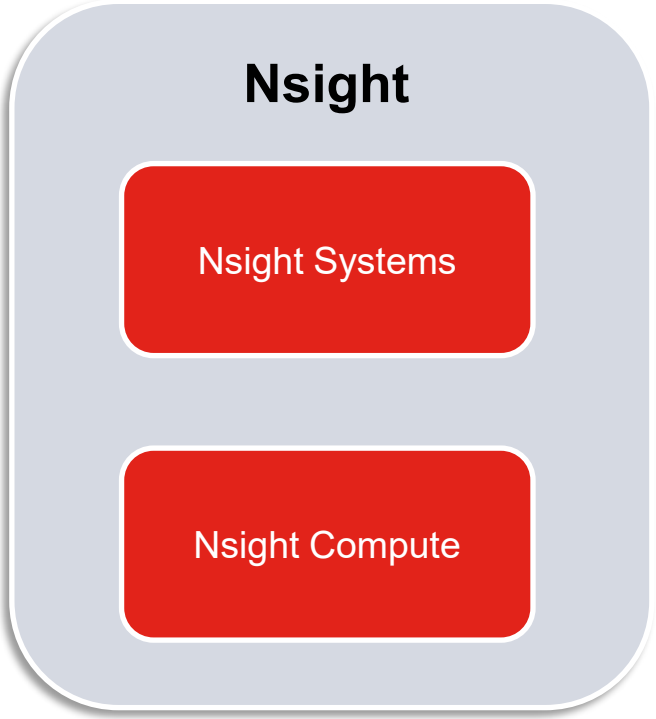
National Supercomputing Centre

1. Profiling Using Nsight Systems

Introduction to Nsight

Nsight comprises:

1. Nsight Systems
2. Nsight Compute
3. Nsight Graphics
   (Not covered here)



- Nsight Systems profiles the overall workload
- Nsight Compute profiles the CUDA kernels in detail

Nsight Systems:

- Helps in identifying the bottlenecks
- Shows activities on a timeline
- Profiles various libraries and APIs.
  Examples:

  CUDA, NVTX ,MPI, OSRT, OpenMP,
  CuBLAS, CuDNN, CuSparse

Nsight Compute:

- Reports on SM/memory utilization and clock cycles
- Correlate to the source code
- Gives advice based on pre-set rules
- Provides launch statistics (influenced by grid and block sizes)
- Provides warp-state statistics (to identify the warp idling)

Introduction to Nsight : Accessing ASPIRE2A+

Nsight is available upon loading a CUDA module

```
malikm@a2ap-login01:~$ qsub -I -l select=1:ngpus=1:mem=200gb -l
walltime=2:00:00 -P 90000002 -q normal
qsub: waiting for job 71854.pbs111 to start
qsub: job 71854.pbs111 ready

malikm@a2ap-dgx038:~$ module load cuda/12.6.2
malikm@a2ap-dgx038:~$ nsys --version
NVIDIA Nsight Systems version 2024.5.1.113-245134619542v0
malikm@a2ap-dgx038:~$
```

The commands:

`nsys`	Nsight Systems
`ncu`	Nsight Compute

Permissions:

Normal users:

- Most of the Nsight Systems' flags (or options) available

Root permission needed for the following:

- All of Nsight Compute
- Nsight Systems' following flags:
  - `--cpuctxsw` CPU context switch
  - `--gpuctxsw` GPU context switch
  - `--backtrace` CPU backtrace
  - `--cudabacktrace` GPU backtrace
  - `--sample -s` Sampling with a frequency
  - `--ftrace` Linux kernel's function trace
  - `--gpu-metrics-devices` Conflicts with DCGM
  - Other flags related to gpu metrics

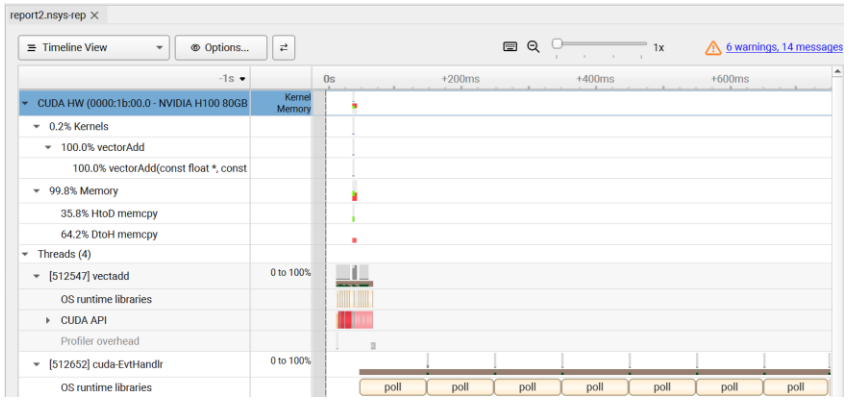
- Nsight Systems generates an output file with extension *.nsys-rep, which will need to be transferred from ASPIRE 2A+ and visualized locally on your laptop. This is due to the fact that x11 forwarding has been disabled on the compute nodes.

- Install the client for Nsight Systems on your laptop by using https://developer.nvidia.com/nsight-systems/get-started

Introduction to Nsight Systems

- Nsight Systems is the profiling tool from NVIDIA used in ASPIRE 2A+
- **System-wide profiling** (all components including computations, OS runtime functions, code annotations by the user, memory access and communications)
- **Visualization:** Reports can be visualized to spot the locations of the code that requires optimization

Visualization Window in local laptop:



Important switch options:

- **Profile:** Launching the application, profiling, and generating the report in one go.
- **Launch, Start and Stop:** Used in an interactive mode for toggling the start and end of the profiling period while the application is running.
- **Analyze:** Get a condensed table report on the screen to quickly view the time spent on CUDA and memory movements between host and device
- **Stats:** Used for generating a detailed table report. A text equivalent of the timeline view on the GUI. Useful for parsing and pipelining as input for further processing on command-line

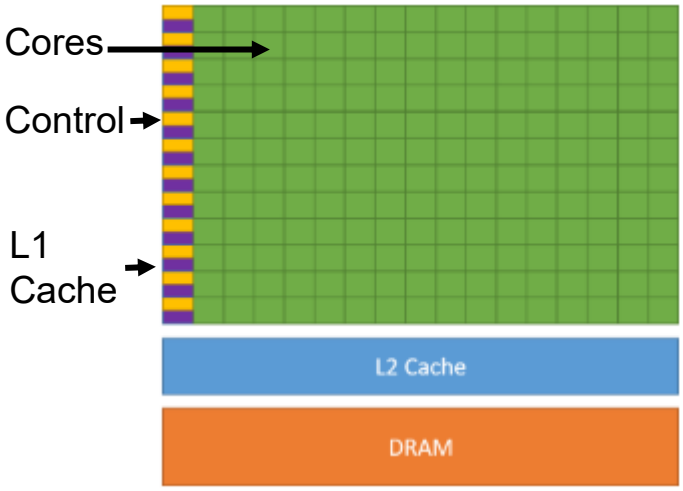
Example below uses the switch, "`analyze`" on a report generated earlier by profile:

```
malikm@a2ap-dgx034:~/cpp$ nsys analyze report1.nsys-rep
```

National Supercomputing Centre

2. Profiling CUDA Activities Using Nsight Systems

Basics: GPU Memory Hierarchy and Thread Blocks



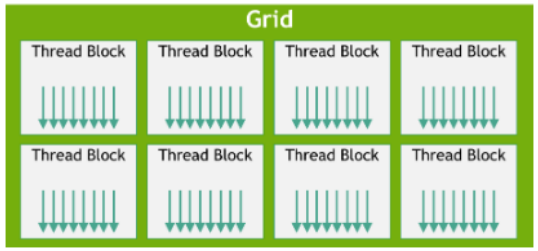
GPU Memory consists of:

- DRAM (80 GB in H100)
- L2 Cache (50 MB in H100)
- L1 Cache (256 KB in H100):

Global memory

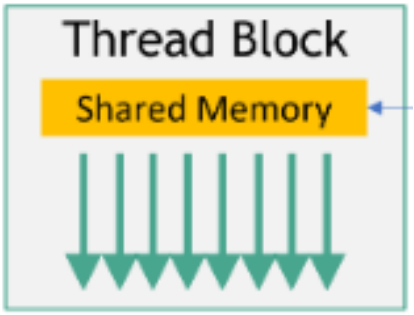
  - ➢ Located within each SM
  - ➢ A portion of this can be used as **Shared Memory**

Each H100 has 132 SM



Thread blocks:

- A thread block can contain up to 2048 threads in H100.
- Each block is executed in each SM in round-robin fashion



Threads in a block can share the “shared memory” address space, if used with __shared__ specifier

Warp:

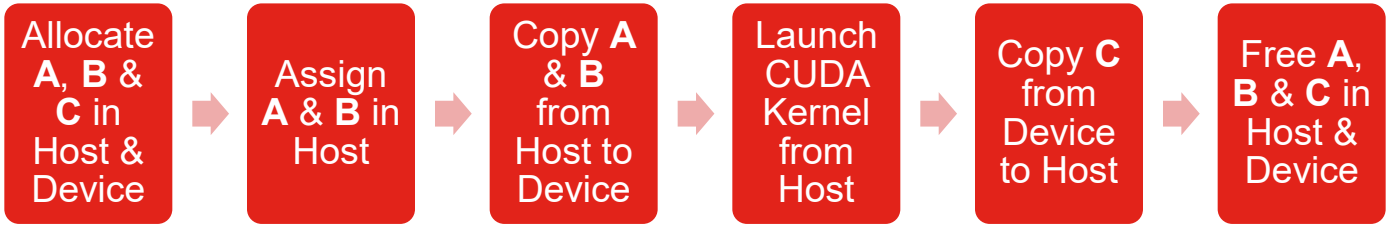
- A group 32 threads executed in locked step using SIMT parallelism.
- H100 Allows 64 warps per SM

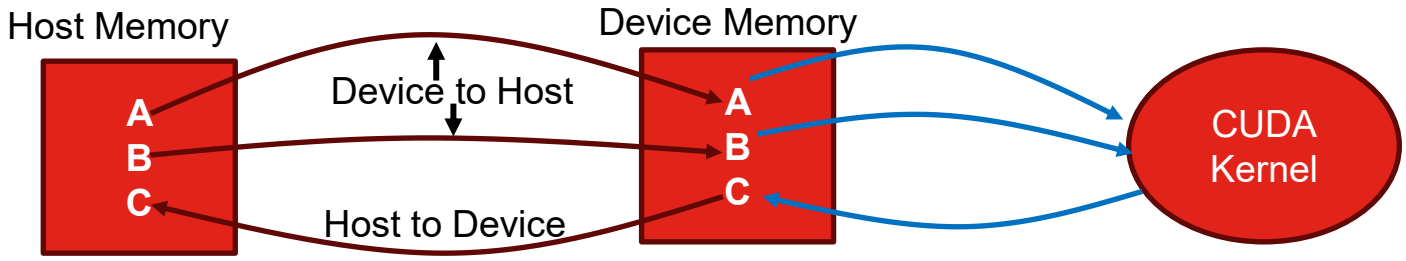
nsys profile: Example with Vector Addition

Let us consider a simple CUDA application:

Vector Addition **C = A + B**

The workflow can be described by the following diagram:





nsys profile: Example with Vector Addition

Let us consider the following source code:

```
malikm@a2ap-dgx036:~/cpp$ cat vectoradd.cu
#include <iostream>
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

__global__ void vectorAdd(const float *a, const float *b, float *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // Compute thread index
    if (i < n) {
        c[i] = a[i] + b[i]; // Perform addition
    }
}

int main(int argc, char* argv[]) {
    int n = std::stoi(argv[1]); size_t size = n * sizeof(float);
    float *h_a, *h_b, *h_c; h_a = new float[n]; h_b = new float[n]; h_c = new float[n];

    for (int i = 0; i < n; i++) {
        h_a[i] = i; h_b[i] = i * 2;
    }

    float *d_a, *d_b, *d_c;
    cudaProfilerStart();
    cudaMalloc((void**)&d_a, size); cudaMalloc((void**)&d_b, size); cudaMalloc((void**)&d_c, size);
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    int blockSize = 512; int gridSize = (n + blockSize - 1) / blockSize;
    vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    for (int i = 0; i < 10; i++) {
        std::cout << h_a[i] << " + " << h_b[i] << " = " << h_c[i] << std::endl;
    }

    delete[] h_a; delete[] h_b; delete[] h_c; cudaFree(d_a); cudaFree(d_b);  cudaFree(d_c);
    cudaProfilerStop();
}
malikm@a2ap-dgx036:~/cpp$
```

This code uses a CUDA kernel (GPU Compute function), “vectoradd()”. It also uses “CUDA profiler API”.

This code has the following important operations:

1. Memory allocation in the host and device
2. Copy two buffers from host to device
3. Launching kernel
4. Copy a buffer from the device to host.
5. Freeing the memory in the host and the device.
6. The code takes a parameter from the command line: the length of the vectors.

nsys profile

Compiling the code:

```
malikm@a2ap-dgx036:~/cpp$ nvcc -gencode=code=sm_90,arch=compute_90 -o vectadd vectoradd.cu
```

Profiling the code:

```
malikm@a2ap-dgx036:~/cpp$ nsys profile --trace=cuda,osrt --capture-range=cudaProfilerApi ./vectadd 10000000
WARNING: CPU IP/backtrace sampling not supported, disabling.
Try the 'nsys status --environment' command to learn more.

WARNING: CPU context switch tracing not supported, disabling.
Try the 'nsys status --environment' command to learn more.

Capture range started in the application.
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
3 + 6 = 9
4 + 8 = 12
5 + 10 = 15
6 + 12 = 18
7 + 14 = 21
8 + 16 = 24
9 + 18 = 27
Capture range ended in the application.
Generating '/raid/pbs.72121.pbs111/nsys-report-0f7d.qdstrm'
[1/1] [0%                          ] report4.nsys-repProcessing events...
[1/1] [==========================100%] report4.nsys-rep
Generated:
    /home/users/adm/sup/malikm/cpp/report4.nsys-rep
malikm@a2ap-dgx036:~/cpp$
```

nsys profile --trace=cuda,osrt --capture-range=cudaProfilerApi ./vectadd 10000000

- **nsys profile** — switch
- **--trace=cuda,osrt --capture-range=cudaProfilerApi** — Options/flags for the switch
- **./vectadd** — application
- **10000000** — Options/parameters for the application

nsys profile: Important Flags

Important flags of profile switch:

--trace

Trace option is used to trace the usage of various API's in the workload. Examples include: cuda, nvtx, osrt, cudnn, cusparse, openacc, openmp, osrt, mpi
Example: **--trace=cuda,nvtx,osrt**

- Selecting "cuda" will cause the profiler to capture the time-ranges spent on memory movements from host to device, vice versa, and on the CUDA kernels.
- The option NVTX is covered separately in later sides.
- The option "osrt" causes the profiler to capture the usage of OS's system functions.
- If the option "mpi" is used, it will result in the capturing of the time spent on MPI communications. By default, OpenMPI implementation for MPI is assumed.
- Similarly the OpenMP API calls also can be captured by specifying "openmp", but this requires compilers with OpenMP version 5.0 or later.

nsys profile: Important Flags

--capture-range

- The “capture-range” flag is used to specify a window in the code for profiling the API calls of CUDA or NVTX. This will result in a report containing the timeline of our interest. The options for this flag can be: none, cudaProfilerApi, nvtx
  Example: **--capture-range=cudaProfilerApi,nvtx**

- However, note that appropriate “include” directive need to be present in the source code. (Please refer to the source code provided in the previous two slides)

- The option, --capture-range=cudaProfilerApi tells Nsight Systems to profile the CUDA API calls specified only between the calls to cudaProfileStart() and cudaProfileStop() functions.

--delay=600

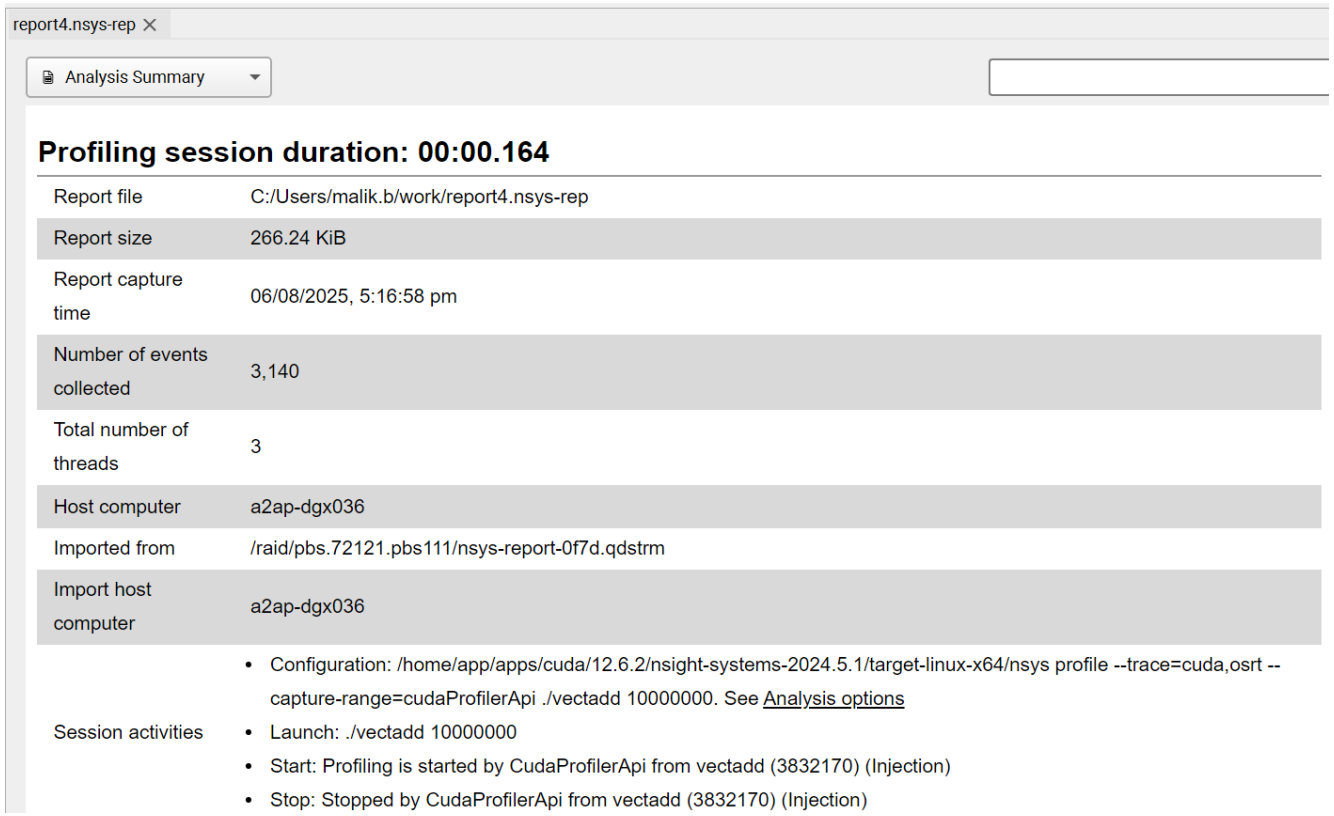
- Skip the first 600 seconds

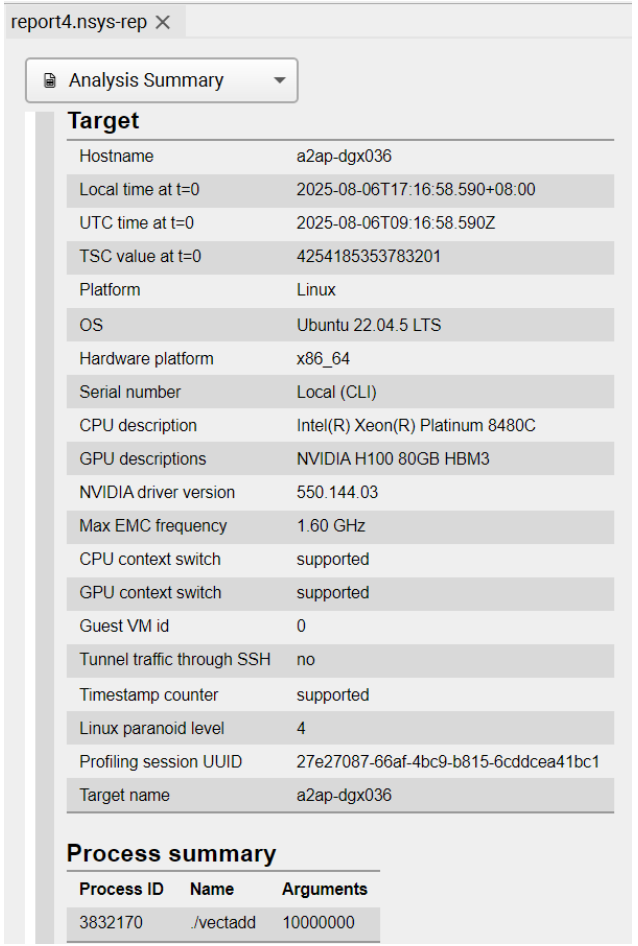
--duration=600

- Profile for only 600 seconds

Analysing on GUI: Analysis Summary

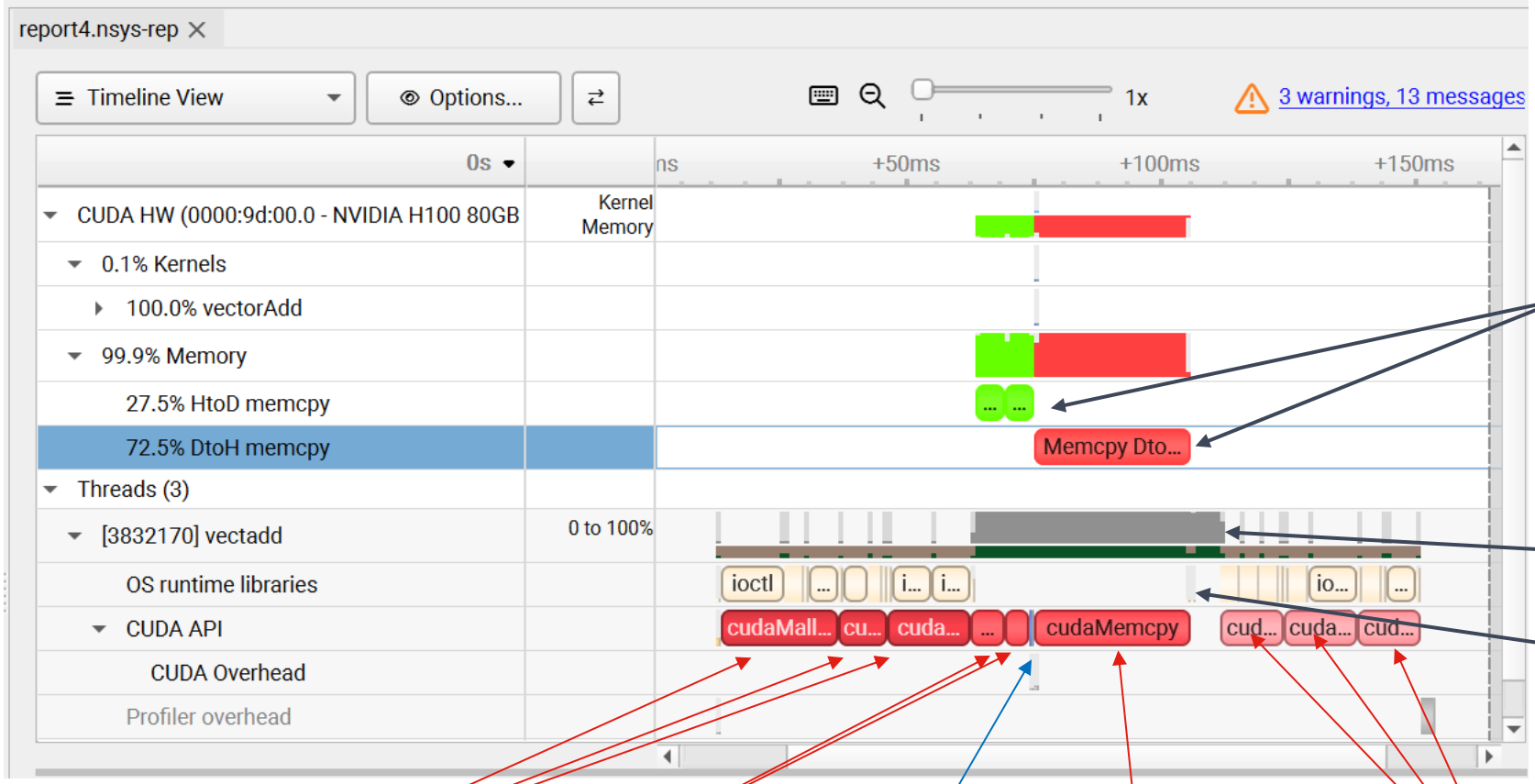
Open the generated report **“report4.nsys-rep”** in the GUI that you have installed as mentioned in slide 4.





Apart from the above information, the “analysis summary” also reports on the hardware such as CPU, GPU, NIC and on the environment variables

Analysing on GUI: Timeline View of Vectoradd



Hovering the mouse over these will show the memory throughputs and size

CPU Activity

Printing on the screen

Three memory allocations on the GPU

Two CPU-to-GPU memory copy

CUDA Kernel

GPU-to-CPU memory copy

Deallocation of the GPU memory

Hover the mouse to get more information on each timeline items

Analysing on GUI: Diagnostic Summary

report4.nsys-rep

Diagnostics Summary

Messages

	Source	Process ID	Time	Description
ℹ	Daemon	3832170	-00:04.482	Process was launched by the profiler, see /raid/pbs.72121.pbs111/nvidia/nsight_systems/quadd_session_103832141/streams/pid_3832170_stdout.log and stderr.log for program output
ℹ	Injection	3832170	-00:02.637	Common injection library initialized successfully.
ℹ	Injection	3832170	-00:02.526	OS runtime libraries injection initialized successfully.
ℹ	Injection	3832170	-00:02.391	Buffers holding CUDA trace data will be flushed on CudaProfilerStop() call. See --flush-on-cudaprofilerstop to control this behavior.
ℹ	Injection	3832170	-00:01.643	Loaded CUPTI library: /home/app/apps/cuda/12.6.2/nsight-systems-2024.5.1/target-linux-x64/libcupti.so.12.6
ℹ	Injection	3832170	-00:01.359	Enabling trace for device graph launch
ℹ	Injection	3832170	-00:01.359	CUDA injection initialized successfully.
ℹ	Analysis		00:00.000	Profiling has started.
ℹ	Injection	3832170	00:00.014	Profiling started due to cudaProfilerStart API invoked in the application.
ℹ	Analysis		00:00.164	Profiling has stopped.
ℹ	Injection	3832170	00:00.167	Number of CUPTI events produced: 28, CUPTI buffers: 50.
⚠	Analysis		00:00.466	Scheduling information is absent. The thread activity is deduced based on OS runtime libraries traces. This is inaccurate and does not take into account asynchronous interrupts and exception faults.
⚠	Analysis	3832170	00:00.472	Not all CUDA events might have been collected.
ℹ	Analysis	3832170	00:00.472	Number of CUDA events collected: 15.
⚠	Analysis	3832170	00:00.472	Not all OS runtime libraries events might have been collected.
ℹ	Analysis	3832170	00:00.472	Number of OS runtime libraries events collected: 34.

The code has run for 164 milliseconds. The warning messages shows inability to capture certain events due to the absence of root permission to profile CPU and GPU context switches.

Analysis on Command Line Using “nsys stats”

```
malikm@a2ap-dgx033:~/cpp$ nsys stats -r cuda_api_sum,cuda_gpu_kern_sum,cuda_gpu_mem_time_sum,cuda_gpu_mem_size_sum report4.nsys-rep

NOTICE: Existing SQLite export found: report4.sqlite
        It is assumed file was previously exported from: report4.nsys-rep
        Consider using --force-export=true if needed.

Processing [report4.sqlite] with [/home/app/apps/cuda/12.6.2/nsight-systems-2024.5.1/host-linux-x64/reports/cuda_api_sum.py]...

 ** CUDA API Summary (cuda_api_sum):

 Time (%)  Total Time (ns)  Num Calls   Avg (ns)    Med (ns)   Min (ns)  Max (ns)  StdDev (ns)       Name
 --------  ---------------  ---------  ----------  ----------  --------  --------  -----------  ----------------
     38.1         49804670          3  16601556.7  16743238.0   9671974  23389458    6859839.4  cudaMalloc
     32.2         42153218          3  14051072.7   6692761.0   5001390  30459067   14234882.9  cudaMemcpy
     29.6         38671166          3  12890388.7  11860837.0  11842014  14968315    1799561.6  cudaFree
      0.2           218920          1    218920.0    218920.0    218920    218920          0.0  cudaLaunchKernel
      0.0             4743          1      4743.0      4743.0      4743      4743          0.0  cuProfilerStart

Processing [report4.sqlite] with [/home/app/apps/cuda/12.6.2/nsight-systems-2024.5.1/host-linux-x64/reports/cuda_gpu_kern_sum.py]...

 ** CUDA GPU Kernel Summary (cuda_gpu_kern_sum):

 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)                          Name
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  ----------------------------------------------------
    100.0            44160          1   44160.0   44160.0     44160     44160          0.0  vectorAdd(const float *, const float *, float *, int)

Processing [report4.sqlite] with [/home/app/apps/cuda/12.6.2/nsight-systems-2024.5.1/host-linux-x64/reports/cuda_gpu_mem_time_sum.py]...

 ** CUDA GPU MemOps Summary (by Time) (cuda_gpu_mem_time_sum):

 Time (%)  Total Time (ns)  Count   Avg (ns)    Med (ns)   Min (ns)  Max (ns)  StdDev (ns)           Operation
 --------  ---------------  -----  ----------  ----------  --------  --------  -----------  ----------------------------
     72.5         29584047      1  29584047.0  29584047.0  29584047  29584047          0.0  [CUDA memcpy Device-to-Host]
     27.5         11219214      2   5609607.0   5609607.0   4851192   6368022    1072560.8  [CUDA memcpy Host-to-Device]

Processing [report4.sqlite] with [/home/app/apps/cuda/12.6.2/nsight-systems-2024.5.1/host-linux-x64/reports/cuda_gpu_mem_size_sum.py]...

 ** CUDA GPU MemOps Summary (by Size) (cuda_gpu_mem_size_sum):

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)           Operation
 ----------  -----  --------  --------  --------  --------  -----------  ----------------------------
     80.000      2    40.000    40.000    40.000    40.000        0.000  [CUDA memcpy Host-to-Device]
     40.000      1    40.000    40.000    40.000    40.000        0.000  [CUDA memcpy Device-to-Host]

malikm@a2ap-dgx033:~/cpp$
```

Profiling Vector Addition: Some Reflections

Some observations from the past four slides:

From the Analysis Summary view, we get:

- **Host info:** hostname and specs of host, GPU device, GPU driver and NIC.
- **User info:** login id and home folder
- **Executable and profiler options:** The full command line options used to generate this report.
- **Paths:** List of modules loaded, and the paths used for executables and libraries are captured. Useful for debugging.
- **Scheduler info:** Path to the working directory and the temporary local storage provided by the scheduler,
- **IB/Ethernet info:** The info on NIC contains whether it is IB or Ethernet. This will help in setting the environment variables for NVSHMEM and NCCL HCA list which should contain only IB HCA's for GPU Direct RDMA communications.

Profiling Vector Addition: Some Reflections (cont…)

From the Timeline View:

- **Memory vs kernel:** The memory allocation, copying from host to device, vice versa, and deallocation of the GPU memory consumes time far greater than the time required for kernel execution.

- **H2D vs D2H throughput:** Copying same size buffer from host to device is several times faster than device to host. This is because we use synchronous copying which requires CPU to take part in a role to receive the data from device (GPU). Asynchronous copying could help speeding this up.

- **Kernel time:** The time taken by kernel is negligible, implying the Nsight Compute is not needed for this.

- **Memory management has no overhead on CPU:** The memory allocation and deallocation operations are independent of CPU. This is evident from the fact that the CPU is fully utilized for OSRT functions during these timeframes.

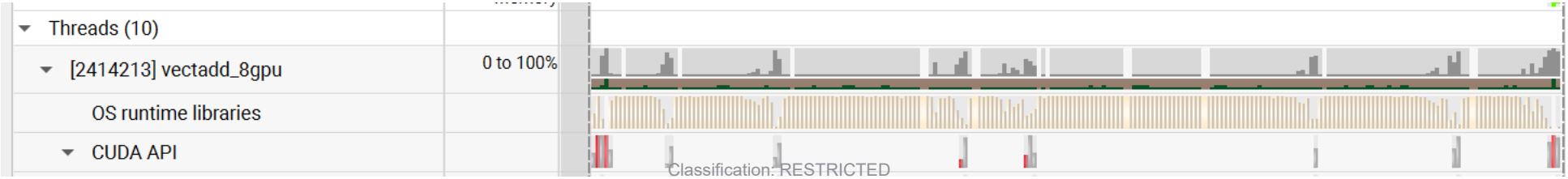
From the stats:

- **Sorted by sum:** It can be noted from the slide 13 that the stats have been collected for each type of CUDA activity, for example, cudaMalloc, and sorted by the summed value over each distinct activities.

Single Thread Multi GPU Single Node Vector Addition

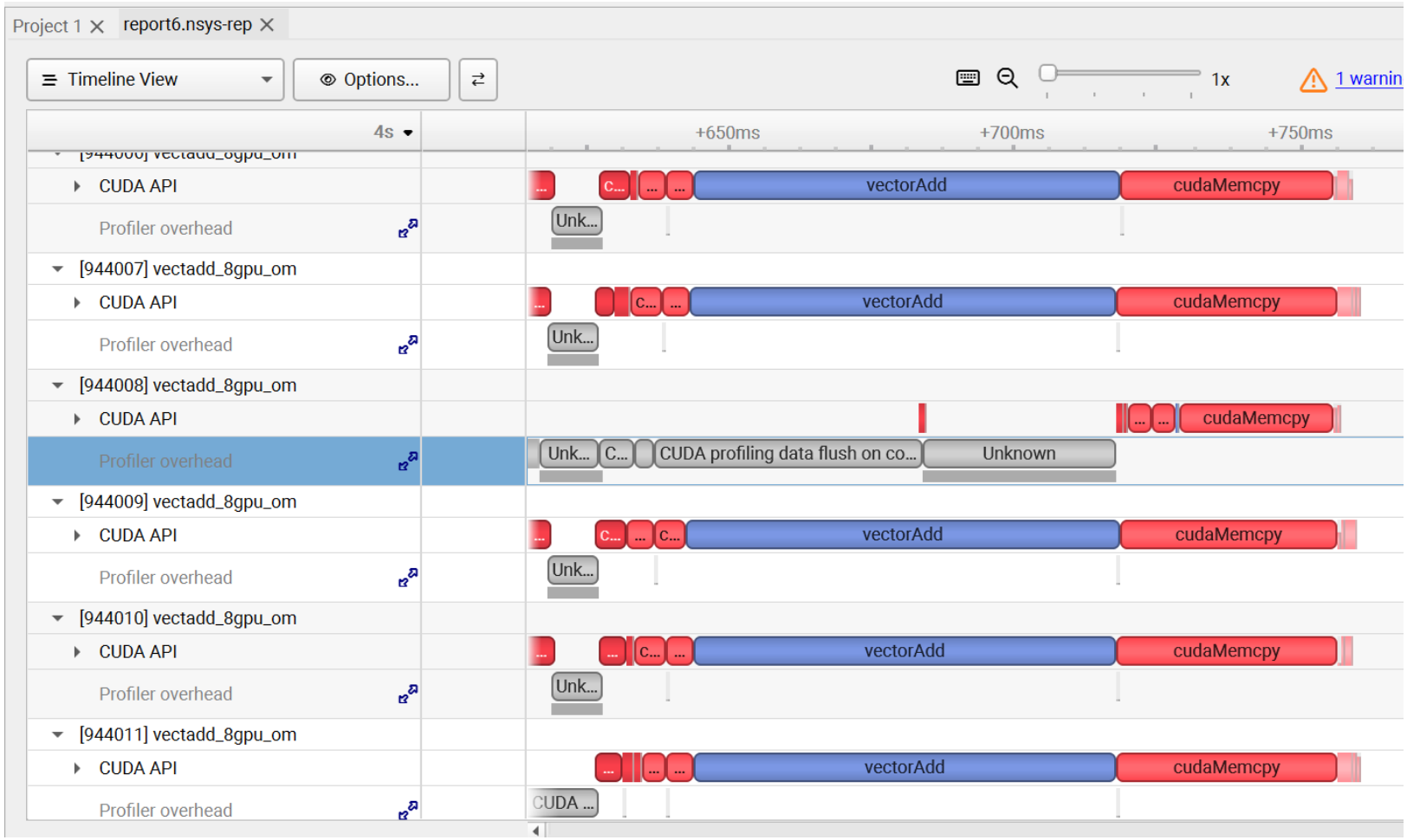
```
malikm@a2ap-login01:~/cpp$ cat vectoradd_8_gpu.cu
#include <iostream>
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

__global__ void vectorAdd(const float *a, const float *b, float *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // Compute thread index
    if (i < n) {
        c[i] = a[i] + b[i]; // Perform addition
    }
}

int main(int argc, char* argv[]) {
    int n = std::stoi(argv[1]); size_t size = n * sizeof(float);
    int deviceCount;
    cudaGetDeviceCount(&deviceCount); // Get the number of available GPUs

    float *h_a, *h_b, *h_c; h_a = new float[n]; h_b = new float[n]; h_c = new float[n];

    for (int i = 0; i < n; i++) {
        h_a[i] = i; h_b[i] = i * 2;
    }

    cudaProfilerStart();
    for (int idev = 0; idev < deviceCount; ++idev) {
        cudaProfilerStart();
        cudaSetDevice(idev); // Set the current device
        float *d_a, *d_b, *d_c;
        cudaMalloc((void**)&d_a, size); cudaMalloc((void**)&d_b, size);
        cudaMalloc((void**)&d_c, size);
        cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

        int blockSize = 512; int gridSize = (n + blockSize - 1) / blockSize;
        vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
        cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

        for (int i = 0; i < 10; i++) {
            std::cout << h_a[i] << " + " << h_b[i] << " = " << h_c[i] << std::endl;
        }
        std::cout << deviceCount << std::endl;

        cudaFree(d_a); cudaFree(d_b);  cudaFree(d_c);
    }
    cudaProfilerStop();
    delete[] h_a; delete[] h_b; delete[] h_c;
}
malikm@a2ap-login01:~/cpp$
```

- The code for single GPU vector-addition has been modified as on the left. This was run using 8 GPU's.
- The profiled result on timeline is shown at the bottom of this slide. In the row for "CUDA API", 8 bursts have been shown, each for each GPU. They are the replicas (almost) of the single GPU case.
- It should be noted that each device's memory operation and kernel execution runs serially one device after the other. This is because, the CUDA operations on each device are carried out using single thread.
- If the "for" loop over the devices (variable "idev" in the code on the left) is parallelized using OpenMP, each CUDA device will be handled in parallel.



Multi-Thread Multi GPU Single Node Vector Addition

The same code in the last slide, after parallelizing the for-loop over CUDA devices using openMP by adding the line **#pragma omp parallel for**, and compiling with enabling OpenMP, gives the following profile:



While profiling, **--trace=cuda** flag/option was used. ("osrt" was skipped).

nvcc compilation needs **–Xcompiler –fopenmp** flags

The code needs the line: **#include <omp.h>** in the preamble

Matrix Multiplication with and without Shared Memory

Shared memory:

- A portion of L1 cache can be used as shared memory. These memories are located in each SM -- fast accessible than global memory. These are called “shared” since they can be shared by all threads in the block. (*Refer to slide 8 for a diagrammatic view*)
- If a chunk of a memory is being frequently used by several threads in a block of an SM, it is advisable to store them in the shared memory area of the SM.

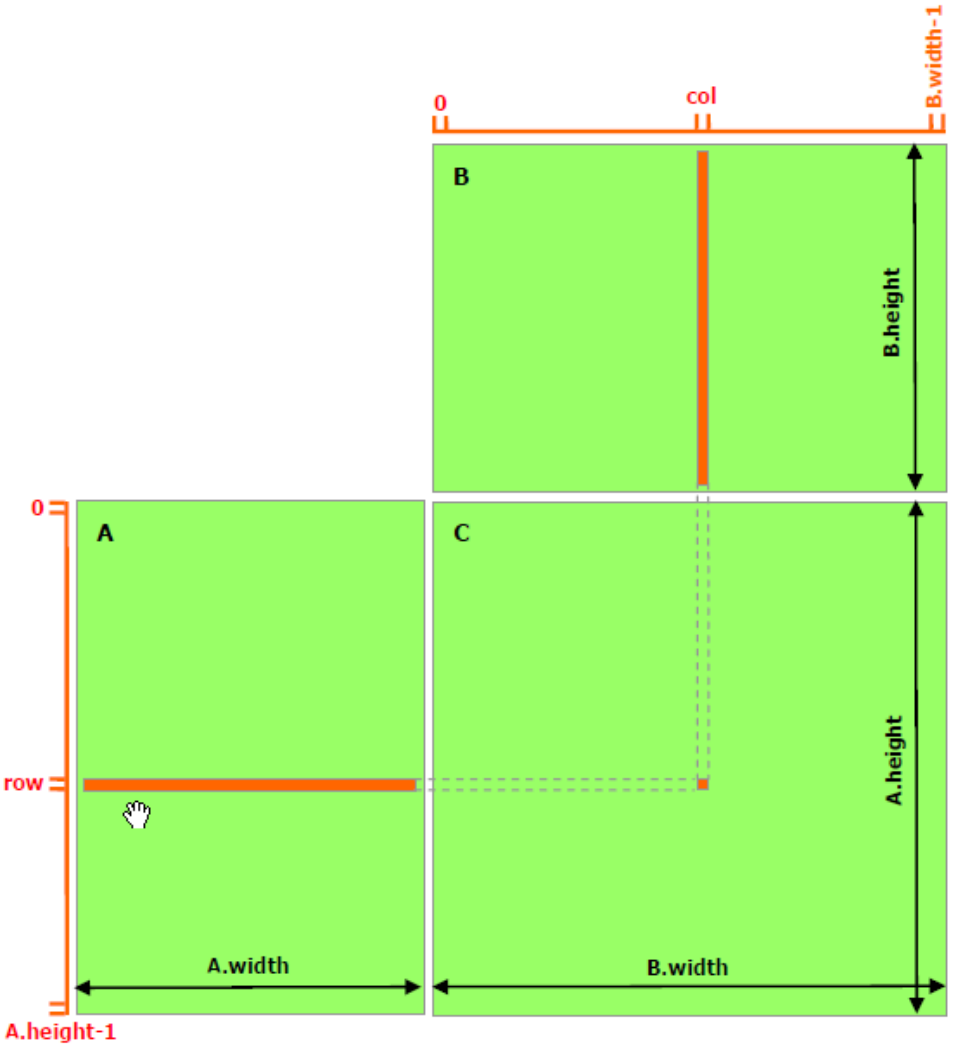
Two version of matrix multiplication code with and without shared memory:

- We have two versions of the matrix multiplication shown in the next slides.
- Their comparison is first shown with respect to the overall run time below.

```
malikm@a2ap-dgx034:~/cpp$ nvcc -gencode arch=compute_90,code=sm_90 -o matrixmult_gpu matrixmult.cu
malikm@a2ap-dgx034:~/cpp$ time ./matrixmult_gpu
The first 10 elements of C(=AB) are:
32768, 32768, 32768, 32768, 32768, 32768, 32768, 32768, 32768, 32768

real    0m24.726s
user    0m16.011s
sys     0m6.459s
malikm@a2ap-dgx034:~/cpp$
malikm@a2ap-dgx034:~/cpp$
malikm@a2ap-dgx034:~/cpp$ nvcc -gencode arch=compute_90,code=sm_90 -o matrixmult_gpu_shared matrixmult_shared.cu
malikm@a2ap-dgx034:~/cpp$ time ./matrixmult_gpu_shared
The first 10 elements of C(=AB) are:
32768, 32768, 32768, 32768, 32768, 32768, 32768, 32768, 32768, 32768

real    0m19.716s
user    0m11.568s
sys     0m6.136s
malikm@a2ap-dgx034:~/cpp$
```

Matrix Multiplication without and with Shared Memory

Without shared memory:



With shared memory:

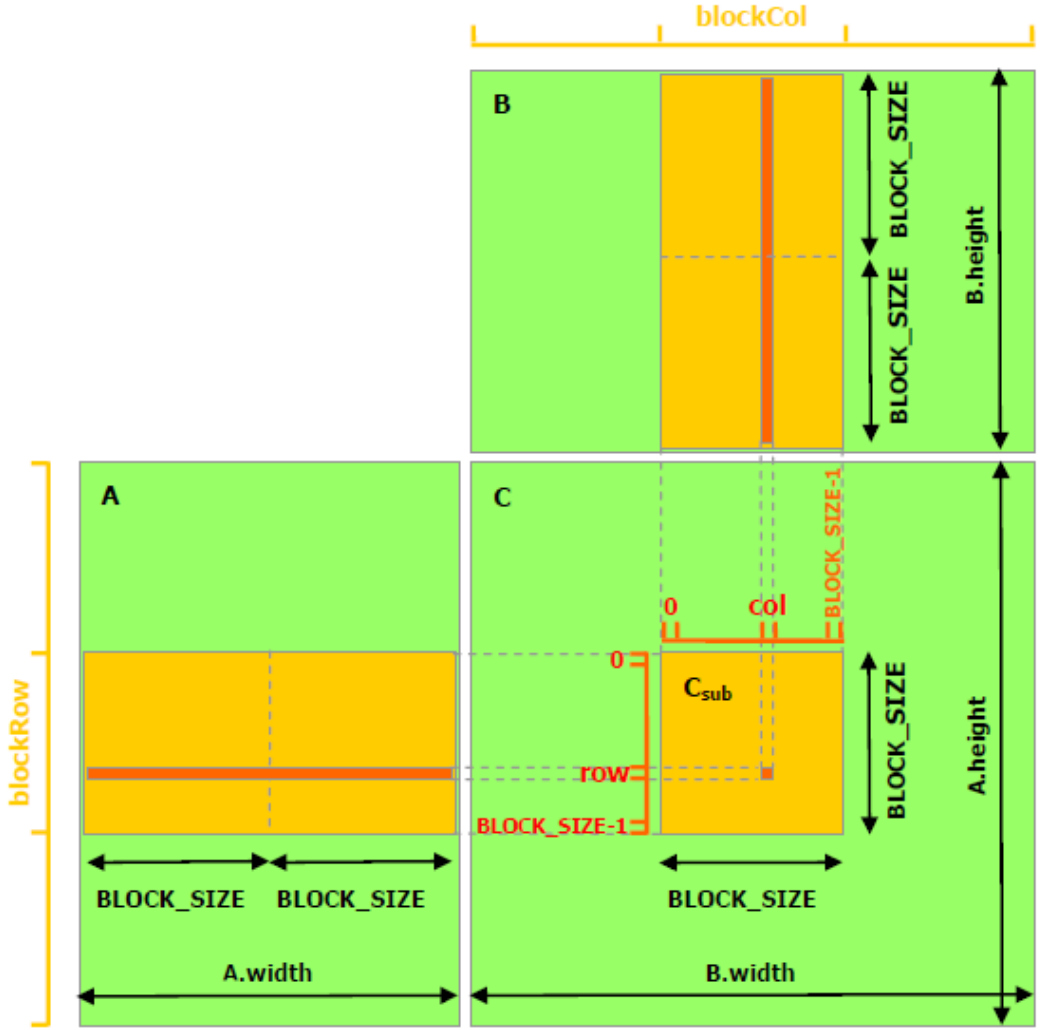


Image credit: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Matrix Multiplication without and with Shared Memory

Without shared memory:

- All matrices live in global memory
- Every element of the matrix C is computed on a thread.
- The entire row of A and the column of B shown in red are accessed from global memory by the thread for the red square in C.
- Accessing from global memory is expensive since it requires more clock cycles than accessing from shared memory in the L1 cache

With shared memory:

- All matrices live in global memory
- The orange regions live in the shared memory of the block.
- Since each block is scheduled with in an SM, copying this band of A and B to the shared memory location of the SM speeds up the calculation.
- The shared memory is specified by **__shared__** specifier.

```
Matrix Bsub = GetSubMatrix(B, m, blockCol);
// Shared memory used to store Asub and Bsub respectively
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
// Load Asub and Bsub from device memory to shared memory
```

The full codes can be in the files **matrixmult.cu** and **matrixmult_shared.cu**

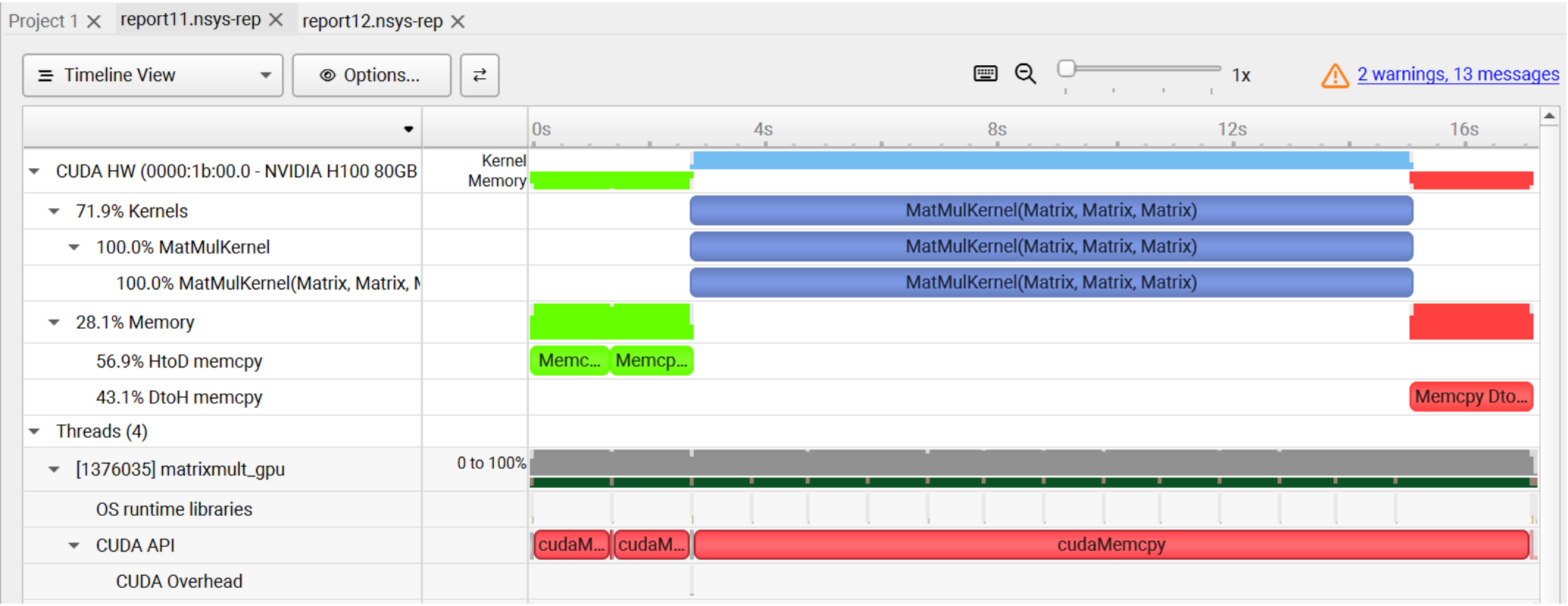
Matrix Multiplication with and without Shared Memory

After inserting CUDA-profiler API in the code, we compile and profile:

```
malikm@a2ap-dgx034:~/cpp$ nvcc -gencode arch=compute_90,code=sm_90 -o matrixmult_gpu matrixmult.cu
malikm@a2ap-dgx034:~/cpp$ nsys profile --trace=cuda,osrt --capture-range=cudaProfilerApi ./matrixmult_gpu
WARNING: CPU IP/backtrace sampling not supported, disabling.
Try the 'nsys status --environment' command to learn more.

WARNING: CPU context switch tracing not supported, disabling.
Try the 'nsys status --environment' command to learn more.

Capture range started in the application.
Capture range ended in the application.
Generating '/tmp/nsys-report-afa4.qdstrm'
[1/1] [0%                          ] report11.nsys-repProcessing events...
[1/1] [==========================100%] report11.nsys-rep
Generated:
    /home/users/adm/sup/malikm/cpp/report11.nsys-rep
malikm@a2ap-dgx034:~/cpp$ nvcc -gencode arch=compute_90,code=sm_90 -o matrixmult_gpu_shared matrixmult_shared.cu
malikm@a2ap-dgx034:~/cpp$ nsys profile --trace=cuda,osrt --capture-range=cudaProfilerApi ./matrixmult_gpu_shared
WARNING: CPU IP/backtrace sampling not supported, disabling.
Try the 'nsys status --environment' command to learn more.

WARNING: CPU context switch tracing not supported, disabling.
Try the 'nsys status --environment' command to learn more.

Capture range started in the application.
Capture range ended in the application.
Generating '/tmp/nsys-report-dbff.qdstrm'
[1/1] [0%                          ] report12.nsys-repProcessing events...
[1/1] [==========================100%] report12.nsys-rep
Generated:
    /home/users/adm/sup/malikm/cpp/report12.nsys-rep
malikm@a2ap-dgx034:~/cpp$
```

Matrix Multiplication without Shared Memory

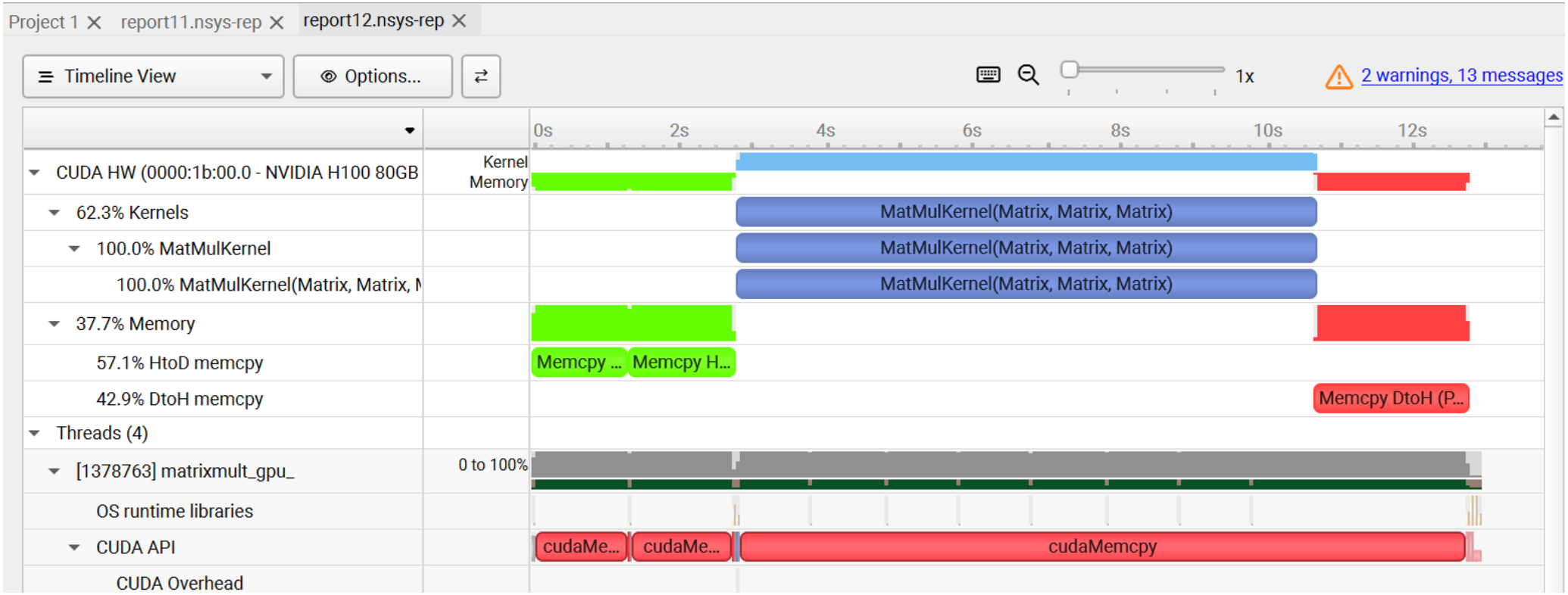
Visualizing the profile report obtained when not using the shared memory



The matrix dimensions have been set as **32768 x 32768** for each matrix.
The kernel roughly takes **12.2 seconds** when not using the shared memory

Matrix Multiplication with Shared Memory

Visualizing the profile report obtained when using the shared memory



For the same problem, the kernel roughly takes only **8 seconds** when using the shared memory with a speed up of 35%

Pinned vs Pageable Memories

Pinned Memory

- The host memory is page-locked (i.e., persistent in RAM).
- It makes the **asynchronous** memory operations possible.
- The CUDA functions **cudaMallocHost()** and **cudaHostAlloc()** are used to allocate the pinned memory.

Pageable Memory

- The content of host memory can be frequently swapped to storage devices.
- Not fast when compared to pinned memory.
- Asynchronous memory operations are not possible with this memory.
- This is the default memory allocated using the "***new***" keyword of C++.

Example: Let us consider the two versions of codes for similar operation shown in the next slide. Their runtime vastly differ as shown below:

```
malikm@a2ap-dgx034:~/cpp$ nvcc -gencode arch=compute_90,code=sm_90 -o pageable_mem_sync_cpy  pageable_mem_sync_cpy.cu
malikm@a2ap-dgx034:~/cpp$ nvcc -gencode arch=compute_90,code=sm_90 -o pinned_mem_async_cpy pinned_mem_async_cpy.cu
malikm@a2ap-dgx034:~/cpp$ time ./pageable_mem_sync_cpy

real    0m26.874s
user    0m10.071s
sys     0m14.718s
malikm@a2ap-dgx034:~/cpp$ time ./pinned_mem_async_cpy

real    0m14.228s
user    0m3.666s
sys     0m9.414s
malikm@a2ap-dgx034:~/cpp$
```

Pinned vs Pageable Memories

Code for pageable memory and synchronous copy between host and device:

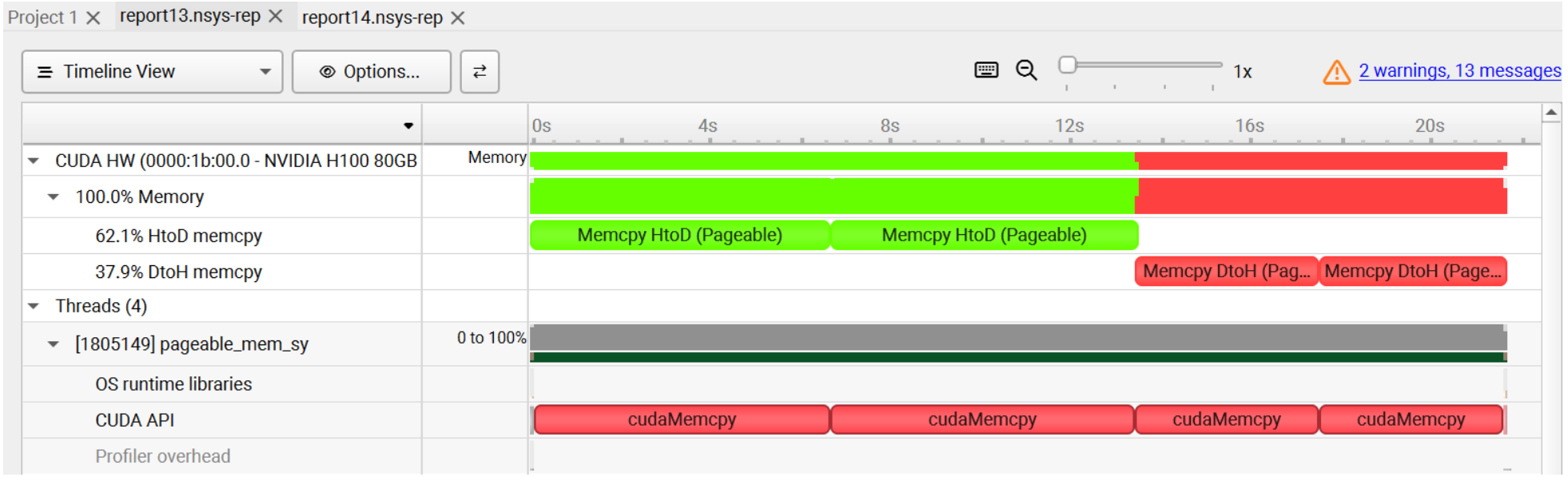
```
malikm@a2ap-login02:~/cpp$ cat pageable_mem_sync_cpy.cu
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

int main(int argc, char* argv[]) {
    size_t n = 3000000000; size_t size = n * sizeof(float);
    float *h_a, *h_b; h_a = new float[n]; h_b = new float[n];
    for (int i = 0; i < n; i++) h_a[i] = h_b[i] = i;

    float *d_a, *d_b;

    cudaProfilerStart();
    cudaMalloc((void**)&d_a, size); cudaMalloc((void**)&d_b, size);
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
    cudaMemcpy(h_a, d_a, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_b, d_b, size, cudaMemcpyDeviceToHost);
    cudaFree(d_a); cudaFree(d_b);
    cudaProfilerStop();

    delete[] h_a; delete[] h_b;
}
malikm@a2ap-login02:~/cpp$
```

Code for pinned memory and asynchronous copy between host and device:

```
malikm@a2ap-login02:~/cpp$ cat pinned_mem_async_cpy.cu
#include <cuda_runtime.h>
#include <cuda_profiler_api.h>

int main(int argc, char* argv[]) {
    size_t n = 3000000000; size_t size = n * sizeof(float);

    float *h_a, *h_b;
    cudaMallocHost(&h_a, size); cudaMallocHost(&h_b, size);
    for (int i = 0; i < n; i++) h_a[i] = h_b[i] = i;

    float *d_a, *d_b;

    cudaStream_t stream[2];
    for (int i = 0; i < 2; ++i) cudaStreamCreate(&stream[i]);

    cudaProfilerStart();
    cudaMalloc((void**)&d_a, size); cudaMalloc((void**)&d_b, size);
    cudaMemcpyAsync(d_a, h_a, size, cudaMemcpyHostToDevice, stream[0]);
    cudaMemcpyAsync(d_b, h_b, size, cudaMemcpyHostToDevice, stream[1]);
    cudaDeviceSynchronize();
    cudaMemcpyAsync(h_a, d_a, size, cudaMemcpyDeviceToHost, stream[0]);
    cudaMemcpyAsync(h_b, d_b, size, cudaMemcpyDeviceToHost, stream[1]);
    cudaDeviceSynchronize();
    cudaFree(d_a); cudaFree(d_b);
    cudaProfilerStop();

    cudaFreeHost(h_a); cudaFreeHost(h_b);
}
malikm@a2ap-login02:~/cpp$
```

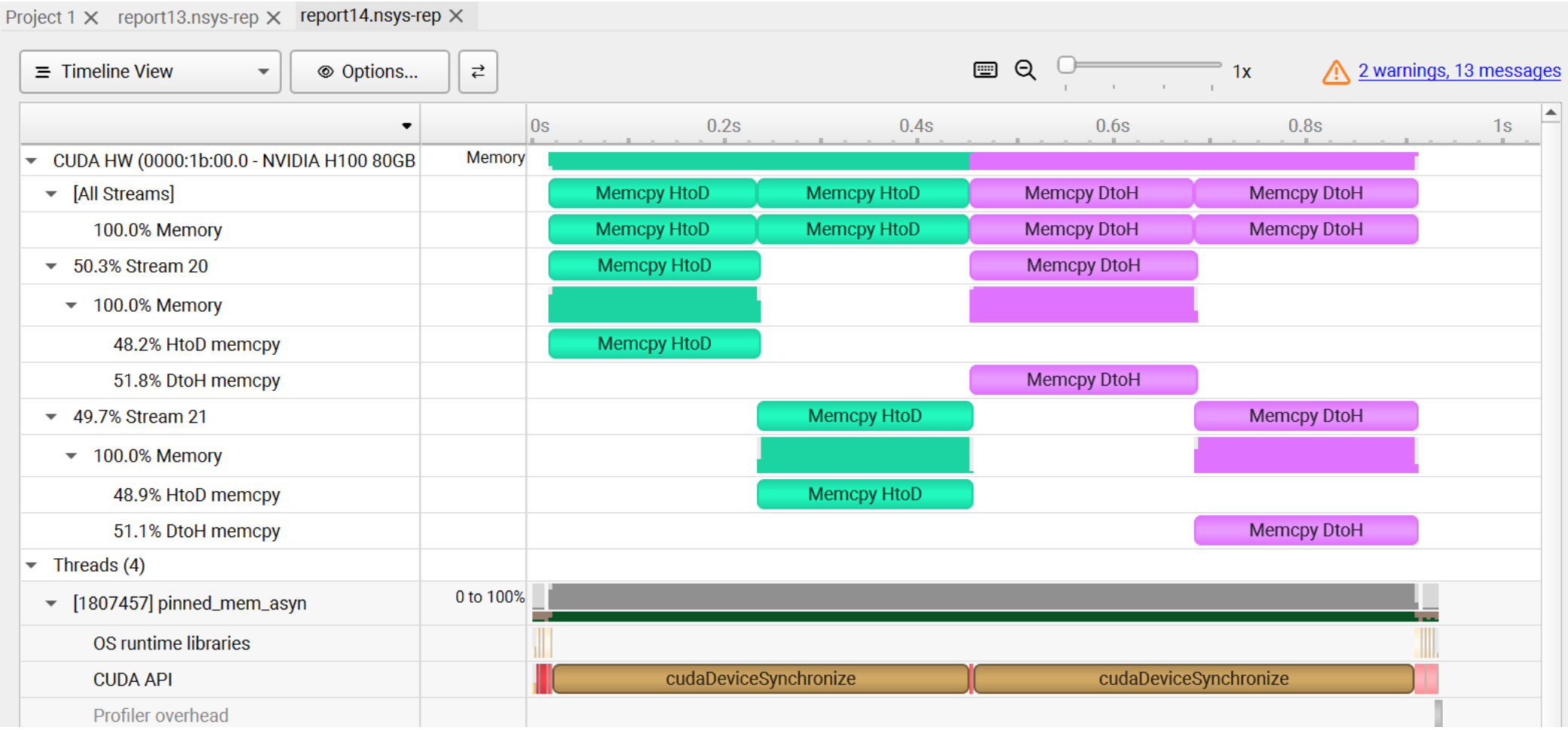
We will profile these two codes and present the timeline in the next two slides.

Pageable Memory and Synchronous Copy



As can be seen, all operations take place serially for more than **20 seconds**.

Pinned Memory and Asynchronous Copy



As can be seen, all operations take place in less than **1 second**.

National Supercomputing Centre

3. Profiling MPI Codes

MPI Profiling

Nsight System support two implementations of MPI:

- MPICH
- OpenMPI (Used mostly in ASPIRE2A PLUS)

"nsys profile" command's flags for MPI:

- **- - trace = mpi** This flag ensure that MPI's API calls are profiled.
- **- - mpi_impl = openmpi | mpich** This flag with "mpich" value is mandatory if using MPICH. By default, it is assumed to be OpenMPI

Some important MPI Calls that can be profiled:

```
MPI_Init[_thread], MPI_Finalize
MPI_Send, MPI_{B,S,R}send, MPI_Recv, MPI_Mrecv
MPI_Sendrecv[_replace]

MPI_Barrier, MPI_Bcast
MPI_Scatter[v], MPI_Gather[v]
MPI_Allgather[v], MPI_Alltoall[{v,w}]
MPI_Allreduce, MPI_Reduce[_{scatter,scatter_block,local}]
MPI_Scan, MPI_Exscan

MPI_Isend, MPI_I{b,s,r}send, MPI_I[m]recv
MPI_{Send,Bsend,Ssend,Rsend,Recv}_init
MPI_Start[all]
MPI_Ibarrier, MPI_Ibcast
MPI_Iscatter[v], MPI_Igather[v]
MPI_Iallgather[v], MPI_Ialltoall[{v,w}]
MPI_Iallreduce, MPI_Ireduce[{scatter,scatter_block}]
MPI_I[ex]scan
MPI_Wait[{all,any,some}]
```

Sample MPI Code: Matrix Multiplication

```
malikm@a2ap-dgx037:~/cpp$ cat matmult_mpi_4_nodes.cpp
// Author: Malik M Barakathullah
// Created: 10 Sept 2025
// Description: MPI example for matrix multiplication using four domains

#include <iostream>
#include <Eigen/Dense>
#include <mpi.h>
#include <cstdlib>
using std::cin;
using std::cout;
using std::endl;
using namespace Eigen;

int main(int argc, char ** argv)
{
  int myrank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Status status;

  setNbThreads(2);

  const int rows = 3000;
  const int cols = rows;
  const int half_rows = rows/2;
  const int half_cols = cols/2;

  if (myrank == 0) {
    MatrixXcd M = MatrixXcd::Zero(rows, cols);

    dcomplex c{1.2,1.0};
    M += MatrixXcd::Constant(rows, cols, c);

    MatrixXcd N = 2*MatrixXcd::Identity(rows,cols);
    MatrixXcd P(rows, cols);
    MatrixXcd P_mpi(rows, cols);
    // P = M*N;
    // if (rows < 20) {
    //   cout << "MxN computed without MPI" << endl;
    //   cout << P << endl;
    // }

    int sizes[2]       = {rows, cols};

    int subsizes_12[2] = {half_rows, half_cols}; int starts_12[2]   = {0, half_cols};
    int subsizes_22[2] = {half_rows, half_cols}; int starts_22[2]   = {half_rows, half_cols};
    int subsizes_21[2] = {half_rows, half_cols}; int starts_21[2]   = {half_rows, 0};

    int subsizes_row1[2] = {half_rows, cols}; int starts_row1[2]   = {0, 0};
    int subsizes_row2[2] = {half_rows, cols}; int starts_row2[2]   = {half_rows, 0};
    int subsizes_col1[2] = {rows, half_cols}; int starts_col1[2]   = {0, 0};
    int subsizes_col2[2] = {rows, half_cols}; int starts_col2[2]   = {0, half_cols};

    MPI_Datatype type_12;
    MPI_Type_create_subarray(2, sizes, subsizes_12, starts_12,
                             MPI_ORDER_FORTRAN, MPI_C_DOUBLE_COMPLEX, &type_12);
    MPI_Type_commit(&type_12);
    MPI_Datatype type_22;
    MPI_Type_create_subarray(2, sizes, subsizes_22, starts_22,
                             MPI_ORDER_FORTRAN, MPI_C_DOUBLE_COMPLEX, &type_22);
    MPI_Type_commit(&type_22);
    MPI_Datatype type_21;
    MPI_Type_create_subarray(2, sizes, subsizes_21, starts_21,
                             MPI_ORDER_FORTRAN, MPI_C_DOUBLE_COMPLEX, &type_21);
    MPI_Type_commit(&type_21);

    MPI_Datatype type_row1;
    MPI_Type_create_subarray(2, sizes, subsizes_row1, starts_row1,
                             MPI_ORDER_FORTRAN, MPI_C_DOUBLE_COMPLEX, &type_row1);
    MPI_Type_commit(&type_row1);
    MPI_Datatype type_row2;
    MPI_Type_create_subarray(2, sizes, subsizes_row2, starts_row2,
                             MPI_ORDER_FORTRAN, MPI_C_DOUBLE_COMPLEX, &type_row2);
    MPI_Type_commit(&type_row2);
    MPI_Datatype type_col1;
    MPI_Type_create_subarray(2, sizes, subsizes_col1, starts_col1,
                             MPI_ORDER_FORTRAN, MPI_C_DOUBLE_COMPLEX, &type_col1);
    MPI_Type_commit(&type_col1);
    MPI_Datatype type_col2;
    MPI_Type_create_subarray(2, sizes, subsizes_col2, starts_col2,
                             MPI_ORDER_FORTRAN, MPI_C_DOUBLE_COMPLEX, &type_col2);
    MPI_Type_commit(&type_col2);

    MPI_Send(&M(0,0), 1, type_row1, 1, 0, MPI_COMM_WORLD);
    MPI_Send(&N(0,0), 1, type_col2, 1, 1, MPI_COMM_WORLD);

    MPI_Send(&M(0,0), 1, type_row2, 2, 0, MPI_COMM_WORLD);
    MPI_Send(&N(0,0), 1, type_col1, 2, 1, MPI_COMM_WORLD);

    MPI_Send(&M(0,0), 1, type_row2, 3, 0, MPI_COMM_WORLD);
    MPI_Send(&N(0,0), 1, type_col2, 3, 1, MPI_COMM_WORLD);

    P_mpi.topLeftCorner(half_rows,half_cols)
              = M.topRows(half_rows) * N.leftCols(half_cols);

    MPI_Recv(&P_mpi(0,0), 1, type_12, 1, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(&P_mpi(0,0), 1, type_21, 2, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(&P_mpi(0,0), 1, type_22, 3, 0, MPI_COMM_WORLD, &status);

    MPI_Type_free(&type_12);
    MPI_Type_free(&type_21);
    MPI_Type_free(&type_22);

    if (rows < 20) {
      cout << "MxN computed with MPI" << endl;
      cout << P_mpi << endl;
    }

  }
  else if (myrank != 0) {
    MatrixXcd M = MatrixXcd::Zero(half_rows, cols);
    MatrixXcd N = MatrixXcd::Zero(rows, half_cols);
    MatrixXcd T  = MatrixXcd::Zero(half_rows, half_cols);

    MPI_Recv(&M(0,0), half_rows * cols, MPI_C_DOUBLE_COMPLEX, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(&N(0,0), rows * half_cols, MPI_C_DOUBLE_COMPLEX, 0, 1, MPI_COMM_WORLD, &status);
    T = M*N;
    MPI_Send(&T(0,0), half_rows * half_cols, MPI_C_DOUBLE_COMPLEX, 0, 0, MPI_COMM_WORLD);
  }

  //cout << "rank" << std::getenv("OMPI_COMM_WORLD_RANK");
  MPI_Finalize();
```

Sample MPI Code: Matrix Multiplication

The features in this code (required for understanding the profiling)

- Performs the Matrix multiplication $\boldsymbol{T} = \boldsymbol{MN}$
- Uses tiled domain decomposition
- Uses 4 domains for the complex double matrix $\boldsymbol{T}$
- Matrix $\boldsymbol{M}$ is split into two row blocks. Matrix $\boldsymbol{N}$ is split into two column blocks.
- Rank 0 sends two chunks of data to each of ranks 1,2, and 3 after completing the multiplication on its share of data chunk. The ranks 1,2 and 3 receives them first and send does the multiplication afterwards.
- Rank 0 received one chunk of data from each of ranks 1, 2 and 3
- Uses Eigen package for the Openmp implementation of matrix multiplication. We use two threads in the code.

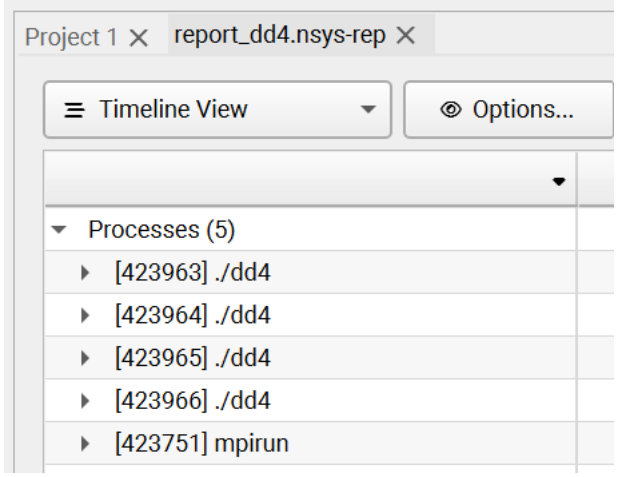
Single Node MPI Profiling

In a single-node profiling of multiple MPI processes, it is sufficient to lump the reports of all processes into a single file.

The "**nsys**" command precedes the **mpirun** command in this case

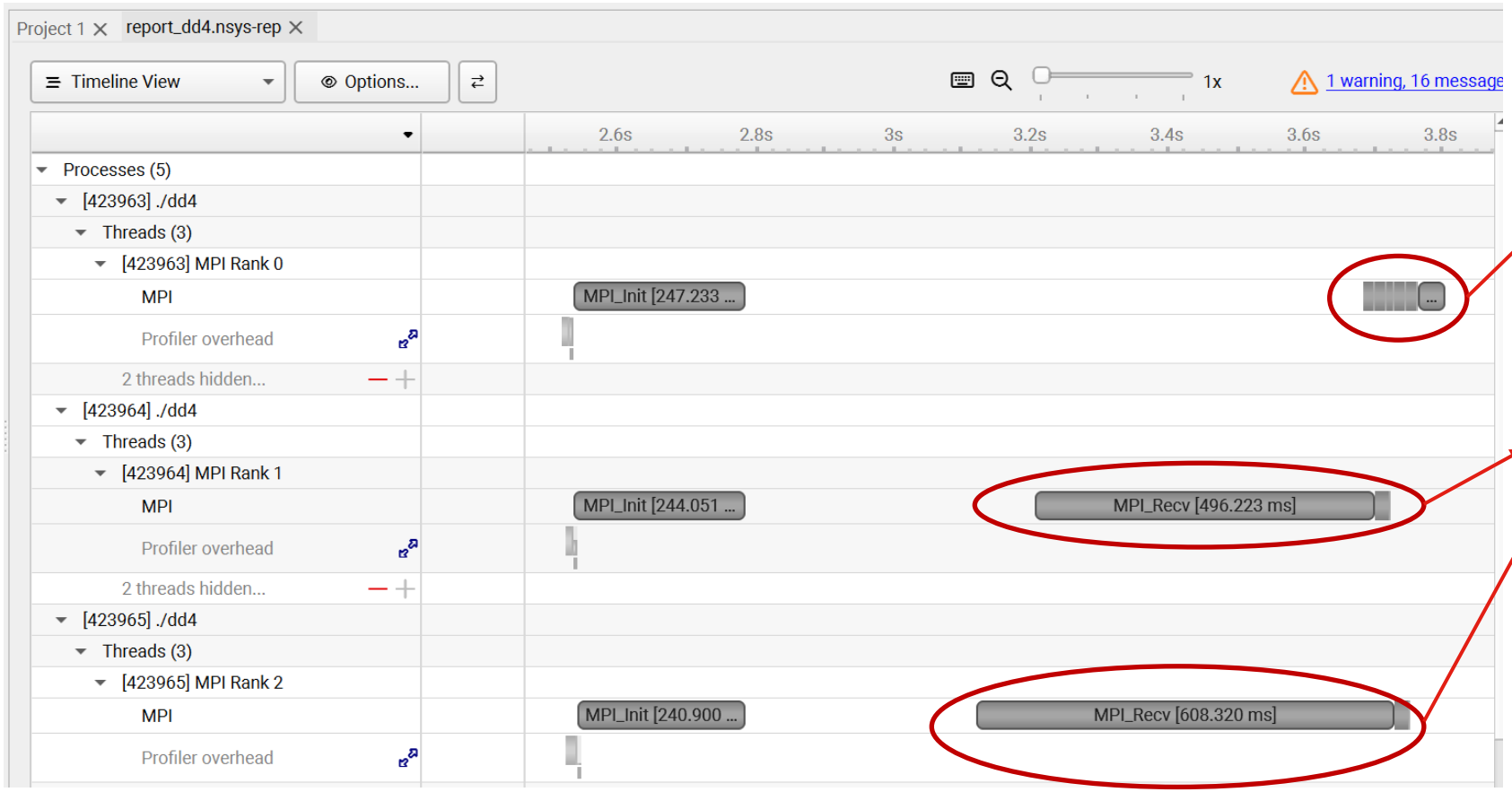
```
malikm@a2ap-dgx034:~/cpp$ mpic++ -fopenmp -o dd4 matmult_mpi_4_nodes.cpp -I./eigen
malikm@a2ap-dgx034:~/cpp$ nsys profile -o report_dd4 --trace=mpi  mpirun -np 4 ./dd4 2>/dev/null
Collecting data...
Generating '/tmp/nsys-report-7e75.qdstrm'
[1/1] [========================100%] report_dd4.nsys-rep
Generated:
    /home/users/adm/sup/malikm/cpp/report_dd4.nsys-rep
malikm@a2ap-dgx034:~/cpp$
```

Let us visualize the generated report_dd4.nsys-rep.



- The generated report_dd4.nsys-rep is shown on the left.
- Under each process, the MPI calls are shown in timeline view in the next slide.
- Each of MPI's timeline can also be viewed under events view by right-clicking.
- For simplicity **we did not capture CPU activity.**

Single Node MPI Profiling



Six MPI_Send() from Rank 0

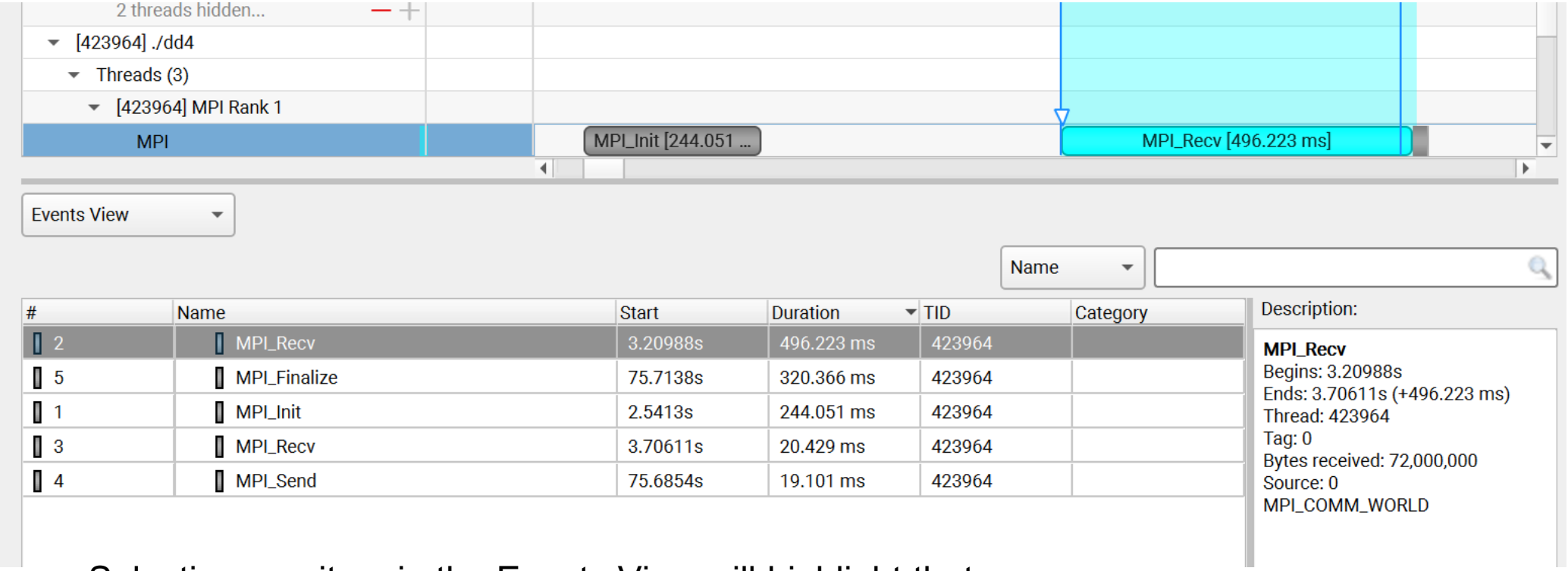
Two MPI_Recv() by other Ranks

- The Rank 0 sends the data late to other nodes since it does the multiplication on the data meant for itself first.

- The other ranks wait for the data from Rank 0, since it does multiplication only after receiving them.

Single Node MPI Profiling

You can right-click on the row for “MPI” to get the “**Events View**”.
The Events View of the MPI row of Rank-1 is shown below:



- Selecting one item in the Events View will highlight that item on the Timeline View as shown above.
- The Description field shows more information, the rank of the source, bytes received, and the message tag.

Note that this rank, Rank 1, has two MPI_Recv’s and one MPI_Send which are consistent with the source code shown 4 slides before.

Multi-Node MPI Profiling

When profiling multi-node MPI jobs

- Unlike in the case of single-node MPI profiling, the **mpirun** and its options should appear first in the command line followed by **nsys** and its options, which are then followed by the application (and its options, if any).

- The profile reports of the processes of different nodes cannot be written on a single file.

- Multiple report files for each of the MPI process can be created by using the environment variable **OMPI_COMM_WORLD_RANK** declared by the *mpirun* wrapper script of OpenMPI. This variable will have the rank of each MPI process as its value at each such process.

- This is done by attaching in the commandline a string "**%q{OMPI_COMM_WORLD_RANK}**" to the filename for the reports.

- In the case of MPICH the environment variable **PMI_RANK** is to be used.

Multi-Node MPI Profiling: Matrix Multiplication

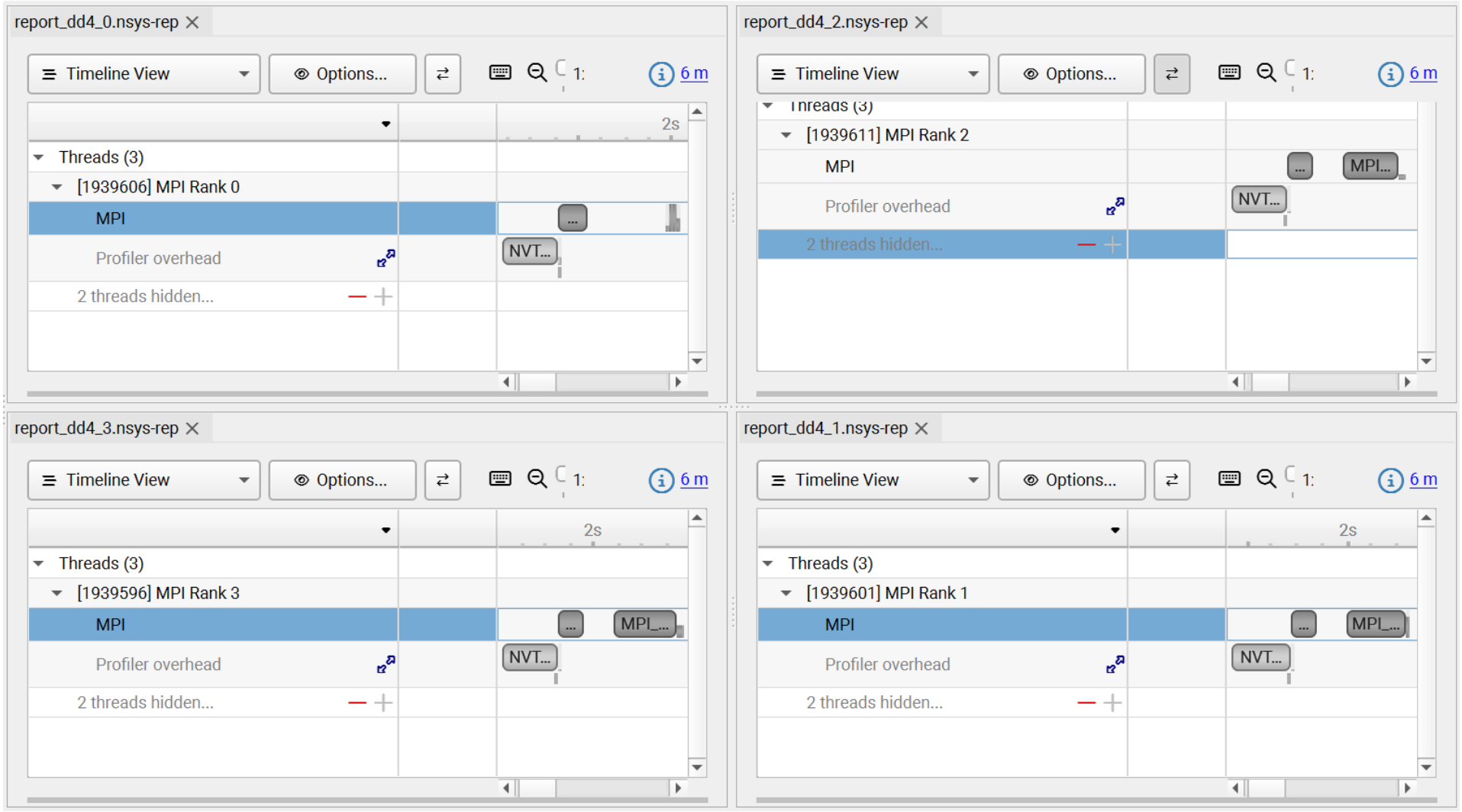
Let us profile the same code using this method on multiple nodes as shown below

```
malikm@a2ap-dgx034:~/cpp$ mpirun -np 4 nsys profile --trace=mpi -o report_dd4_%q{OMPI_COMM_WORLD_RANK} ./dd4 2>/dev/null
Collecting data...
Collecting data...
Collecting data...
Collecting data...
Generating '/tmp/nsys-report-3d71.qdstrm'
Generating '/tmp/nsys-report-d4c5.qdstrm'
Generating '/tmp/nsys-report-c82b.qdstrm'
Generating '/tmp/nsys-report-25cc.qdstrm'
[1/1] [========================100%] report_dd4_1.nsys-rep
Generated:
    /home/users/adm/sup/malikm/cpp/report_dd4_1.nsys-rep
[1/1] [========================100%] report_dd4_2.nsys-rep
[1/1] [========================100%] report_dd4_0.nsys-rep
Generated:
    /home/users/adm/sup/malikm/cpp/report_dd4_0.nsys-rep
Generated:
    /home/users/adm/sup/malikm/cpp/report_dd4_2.nsys-rep
[1/1] [========================100%] report_dd4_3.nsys-rep
Generated:
    /home/users/adm/sup/malikm/cpp/report_dd4_3.nsys-rep
```

As can be noted from this, four report files have been generated since we used 4 MPI process.

Multi-Node MPI Profiling: Matrix Multiplication

The generated multiple reports can be opened and rearranged by undocking them and then docking at the corners we prefer.



Profiling a Chosen MPI Process

If only a single or subset of MPI processes only need to be profiled, a wrapper bash script like the one below can be used:

```
malikm@a2ap-dgx034:~/cpp$ cat nsys_profile.sh
#!/bin/bash

# Use $PMI_RANK for MPICH and $SLURM_PROCID with srun.
if [ $OMPI_COMM_WORLD_RANK -eq 0 ]; then # 0 signifies rank 0
  nsys profile -e NSYS_MPI_STORE_TEAMS_PER_RANK=1 --trace=mpi "$@"
else
  "$@"
fi
malikm@a2ap-dgx034:~/cpp$ chmod u+x nsys_profile.sh
malikm@a2ap-dgx034:~/cpp$ ls -l nsys_profile.sh
-rwxrw-r-- 1 malikm malikm 212 Sep 11 11:02 nsys_profile.sh
malikm@a2ap-dgx034:~/cpp$ mpirun -np 4 ./nsys_profile.sh ./dd4 2>/dev/null
Collecting data...
Generating '/tmp/nsys-report-283b.qdstrm'
[1/1] [========================100%] report10.nsys-rep
Generated:
    /home/users/adm/sup/malikm/cpp/report10.nsys-rep
malikm@a2ap-dgx034:~/cpp$
```

Gives execute permission

In this case, the report10.nsys-rep would contain the profiling information for Rank-0 only.

National Supercomputing Centre

4. Profiling Using NVTX

NVTX: Introduction

- NVTX can register traces in the code for visualizing the instances of, and ranges between NVTX API calls.
- Requires the include statement: #include <nvtx3/nvtx3.hpp>
- In ASPIRE2A PLUS, the modules cuda/12.6.1 or cuda/12.6.2 is to be loaded to access NVTX3.
- The following path variable need to be set (despite the module above is loaded. These modules will be corrected later to make this take effect): export CPLUS_INCLUDE_PATH= /app/apps/cuda/12.6.2/nsight-systems-2024.5.1/target-linux-x64/nvtx/include/
- For C++, there is a wealth of NVTX API calls available to group and visualize the code.
- However, for beginners the NVTX functions such as Range and Marks will suffice.
- NVTX annotations are already in use in the opensource projects like Pytorch, so that it can be visualized when enabled while profiling

NVTX: Introduction

NVTX Ranges

- An NVTX Range annotates a range of continuous statements.
- Among the two types of ranges, namely unique and scoped, the scoped ranges are easier to handle with less profiling overhead.

Scoped Ranges

- It captures the time duration of the execution of a scope “{ contents between braces in the code}” where this range is declared.
- This can be declared by nvtx3::scoped_range r{"some_name"};
- In the case of functions, one can use, NVTX3_FUNC_RANGE() which takes the function name as the name of the range.

NVTX: Introduction

NVTX Markers

- A location in the code can be visualized by annotating with markers

```
nvtxRangePush("My Function"); // Start a range
// ... code to be profiled ...
nvtxRangePop(); // End the range
```

- Example: nvtxMark("Important Event Occurred");
- Useful for debugging.

Event Markers

- Nested ranges can be defined nvtxRangePush(“some name”); and nvtxRangePop(); (See the figure above for clarity).
- nvtxRangePop() ends the range started by the most recent nvtxRangePush(“ <a string> ”)
- In the case of functions, one can use, NVTX3_FUNC_RANGE().

NVTX Example: Matrix Multiplication

Let us consider the source code:

```
malikm@a2ap-login01:~/cpp$ cat matmult_nvtx.cpp
#include <iostream>
#include <Eigen/Dense>
#include <nvtx3/nvtx3.hpp>

using namespace Eigen;

auto mult_complex(int row_)
{
  NVTX3_FUNC_RANGE();  // Range around the whole function

  setNbThreads(2);
  MatrixXcd M = MatrixXcd::Random(row_,row_);
  MatrixXcd N = MatrixXcd::Random(row_,row_);
  MatrixXcd P(row_,row_);
  P = M*N;
  return P;
}

auto mult_real(int row_)
{
  NVTX3_FUNC_RANGE();  // Range around the whole function

  setNbThreads(2);
  MatrixXd M = MatrixXd::Random(row_,row_);
  MatrixXd N = MatrixXd::Random(row_,row_);
  MatrixXd P(row_,row_);
  P = M*N;
  return P;
}


int main(){
  int n_iter = 5;
  for (int i = 0; i < n_iter; i++) {
     nvtx3::scoped_range loop{"loop range"};  // Range for iter
     auto P = mult_complex(500);
     auto Q = mult_real(500);
  }
}
malikm@a2ap-login01:~/cpp$
```

In this example, the Eigen package is used for matrix multiplication only to minimize the length of the code for brevity, and to focus on the usage of NVTX API calls. The usage of NVTX in a CUDA code could be similar.

The features of this code

- The code performs five iterations
- At each iteration, it calls two functions: One of them performs multiplication on two complex double matrices.
- The other performs the same, but on a couple of real double matrices.
- Uses NVTX3_FUNC_RANGE() in each function. Therefore, these ranges will be named after their function names.
- Uses scoped range for each loop and names it “loop range”

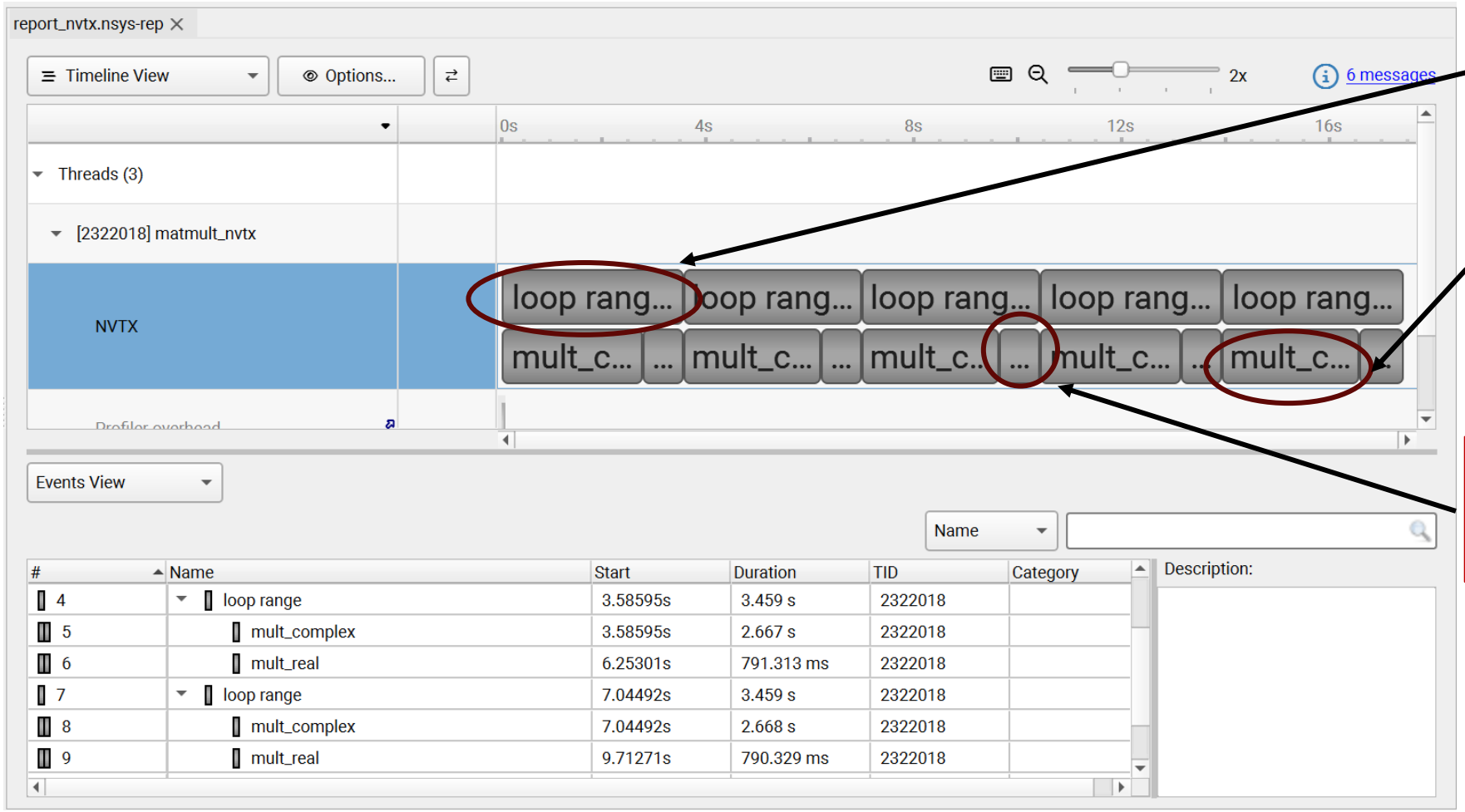
NVTX Example: Matrix Multiplication

Let us profile this code as shown below:

```
malikm@a2ap-dgx034:~/cpp$ module load cuda/12.6.2
malikm@a2ap-dgx034:~/cpp$ export CPLUS_INCLUDE_PATH=/app/apps/cuda/12.6.2/nsight-sy
stems-2024.5.1/target-linux-x64/nvtx/include/
malikm@a2ap-dgx034:~/cpp$ g++ -o matmult_nvtx matmult_nvtx.cpp  -I./eigen
malikm@a2ap-dgx034:~/cpp$ nsys profile --trace=nvtx -o report_nvtx ./matmult_nvtx 2
>/dev/null
Collecting data...
Generating '/tmp/nsys-report-aa02.qdstrm'
[1/1] [========================100%] nsys-report-35e8.nsys-rep
Generated:
    /tmp/nsys-report-35e8.nsys-rep
malikm@a2ap-dgx034:~/cpp$ rm report_nvtx.nsys-rep
malikm@a2ap-dgx034:~/cpp$
malikm@a2ap-dgx034:~/cpp$
malikm@a2ap-dgx034:~/cpp$
malikm@a2ap-dgx034:~/cpp$ module load cuda/12.6.2
malikm@a2ap-dgx034:~/cpp$ export CPLUS_INCLUDE_PATH=/app/apps/cuda/12.6.2/nsight-sy
stems-2024.5.1/target-linux-x64/nvtx/include/
malikm@a2ap-dgx034:~/cpp$ g++ -o matmult_nvtx matmult_nvtx.cpp  -I./eigen
malikm@a2ap-dgx034:~/cpp$ nsys profile --trace=nvtx -o report_nvtx ./matmult_nvtx 2
>/dev/null
Collecting data...
Generating '/tmp/nsys-report-234c.qdstrm'
[1/1] [========================100%] report_nvtx.nsys-rep
Generated:
    /home/users/adm/sup/malikm/cpp/report_nvtx.nsys-rep
malikm@a2ap-dgx034:~/cpp$
```

NVTX Example: Matrix multiplication

Upon visualizing this report, “report_nvtx.nsys-rep”, we see:



Iteration range

Complex double matrix multiplication range

Real double matrix multiplication

Event view: Complex double matrix takes 3.5 times more time than real double matrix for multiplication

National Supercomputing Centre

5. Profiling Python Codes Including Pytorch Modules

NVTX Python: Introduction

NVTX offers sufficient tools to profile a Python code. The reference URL: https://nvidia.github.io/NVTX/python/reference.html

```
import nvtx
```

Annotations

- **nvtx.Annotate** is a decorator for functions which allows them to be set with messages and colours that need to shown in the visualization tools such as Nsight Systems.

- Examples:

```
@nvtx.annotate(message="my_message", color="blue")
def my_func():
    pass
```

```
@nvtx.annotate()  # message defaults to "my_func"
def my_func():
    pass
```

- Can also be used as a context manager as in:

```
with nvtx.annotate(message="my_message", color="green"):
    pass
```

- This is similar to the scoped range for whole of a function's scope or other scopes that we saw earlier in the case of C++.

NVTX Python: Introduction

Domains and Categories

- **nvtx.Domain** is a major grouping of a section of the code by naming it with a name.
- It is a class that offers methods for annotating ranges and marking locations of the code with chosen names.
- Python's imported modules may be already using a domain with a name specific to that module in order to visualize them in the Nsight Systems.
- Domains can be created as: `my_domain = nvtx.get_domain("domain name")`
- The usage of the domains will be covered in the subsequent slides but using them when profiling causes less overhead.
- **Categories:** These are further groupings of the codes under a chosen domain. These are not classes unlike the domains but just labels for viewing them under different colours and names in the timespan of their domain in the Nsight system.

NVTX Python: Introduction

Markers

- **nvtx.mark** is used to mark any event for the visualization tool. Example:

```
nvtx.mark("Loss > Limit", color='blue', domain="n", category="c")
```

- **Push/pop Ranges:** These mark a range of the code region in a single thread by a name and colour. Example:

```
nvtx.push_range("parsing", color='blue', domain="domain", category="c")
 # code for parsing
nvtx.pop_range() # Ending the scope of the most recent push_range()
```

- **Start/end Ranges:** These are markers that can span several threads within its range.

```
nvtx.start_range("a name", color='blue', domain="d", category="c")
 # code
nvtx.stop_range() # Ending the scope of the most recent stop_range()
```

NVTX Python: Introduction

Markers with nvtx.Domain()

- **nvtx.Domain.mark()** is used to mark any event for the visualization tool. Example:

```
domain = nvtx.get_domain('my domain’)
attr = domain.get_event_attributes(color='red’)
attr.message = "a is zero now"
domain.mark(attr)
```

- **Nvtx.Domain()’s Push/pop Ranges:** Same as before, but for domains. Example:

```
domain = nvtx.Domain('my_domain’)
attributes = domain.get_event_attributes(
                                    message='range 1', color='blue’)
domain.push_range(attributes)
# The code snippet that we want to mark by this range
domain.pop_range()
```

- These methods under the Domain class will take less overhead. The counter parts in the previous slides are called **global markers**, while these methods are called **domain markers**

NVTX Python: Example: Matrix Arithmetic

Let us consider the code that uses only the global versions of NVTX markers:

```
malikm@a2ap-login01:~/python$ cat matmult.py
import nvtx
import numpy as np

@nvtx.annotate(color="cyan")
def matmult(A, B): return A @ B

@nvtx.annotate(color="magenta")
def element_mult(A, B): return A*B

@nvtx.annotate(color="orange")
def matadd(A, B): return A + B

if __name__ == "__main__":
    m, n, p = 500, 500, 500

    nvtx.push_range("Complex", color="blue", domain="Complex")
    A = np.random.rand(m, n) + 1j*np.random.rand(m, n)
    B = np.random.rand(n, p) + 1j*np.random.rand(n, p)

    nvtx.push_range("Mult",color="red", domain="Complex", category="Mult")
    C = matmult(A, B);  C1 = element_mult(A, B)
    nvtx.pop_range()

    nvtx.push_range("Add", color="green", domain="Complex", category="Add")
    D = matadd(A, B)
    nvtx.pop_range()
    nvtx.pop_range()

    nvtx.push_range("Real", color="purple", domain="Real")
    E = np.random.rand(m, n); F = np.random.rand(n, p)

    nvtx.push_range("Mult", color="red", domain="Real", category="Mult")
    G = matmult(E, F); G1 = element_mult(E, F)
    nvtx.pop_range()

    nvtx.push_range("Add", color="green", domain="Real", category="Add")
    H = matadd(E, F)
    nvtx.pop_range()
    nvtx.pop_range()
malikm@a2ap-login01:~/python$
```

This code has the following schema:

Domain: Complex
- Category: Mult
  - calls:
    - matmult()
    - element_mult()
- Category: Add
  - calls:
    - matadd()

Domain: Real
- Category: Mult
  - calls:
    - matmult()
    - element_mult()
- Category: Add
  - calls:
    - matadd()

NVTX Python: Example: Matrix Arithmetic

Features of this code required for understanding the profiling report:

- This code does arithmetic under two different datatypes:
  - (a) Complex double – Forms a domain
  - (b) Real double – The second domain

- Under each domain it performs two categories of arithmetic:
  - (a) Multiplication – First category
  - (b) Addition – Second category

- Under Multiplication category, it calls two functions:
  - (a) matmult() – performs matrix multiplication
  - (b) element_mult() – performs elementwise multiplication

- Under Addition category, it calls one function:
  - (a) matadd() – performs matrix addition

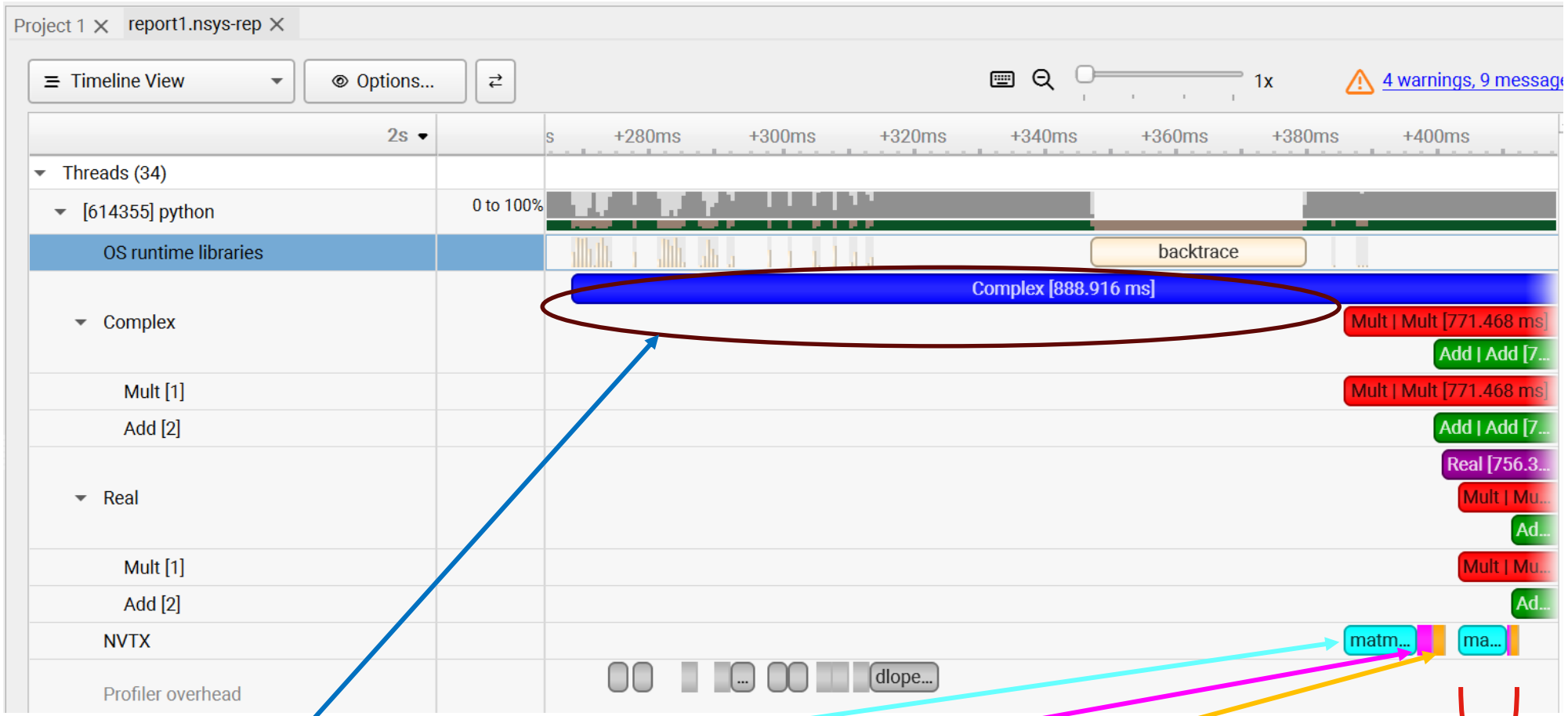
```
malikm@a2ap-dgx034:~/python$ nsys profile python matmult.py 2>/dev/null
Collecting data...
Generating '/tmp/nsys-report-3fec.qdstrm'
[1/1] [==========================100%] report1.nsys-rep
Generated:
    /home/users/adm/sup/malikm/python/report1.nsys-rep
malikm@a2ap-dgx034:~/python$
```

Profiling the code

The generated report is shown in the next slide

NVTX Python: Example: Matrix Arithmetic

Upon visualizing this report, “report1.nsys-rep”, we see:



The overhead to declare and assign complex double matrices

complex double matrix multiplication

complex double matrix element wise multiplication

complex double matrix addition

Similarly, for real double matrices

NVTX Python: Example: Matrix Arithmetic

Findings from the report

- The overhead in declaring and assigning complex matrices is enormous. Note that these are objects with memory allocations for both real and imaginary parts, and with all methods for complex arithmetic and functions sch as finding the real part, modulus, phase, etc.
- Due to this overhead the matrix multiplication on the complex double data starts quite late.
- Such overhead is absent in the case of real double data.
- The elementwise multiplication and addition almost take similar time duration.

NVTX Python: Pinned vs Pageable Memories

Let us consider the following example code to demonstrate the difference between pageable and pinned memory

```
malikm@a2ap-login02:~/python$ cat pinned_pageable.py
import nvtx
import torch
from torch.utils.data import DataLoader, TensorDataset

sample_size = 10000
feature_size = 1000000

data = torch.randn(sample_size, feature_size)
labels = torch.randint(0, 2, (sample_size,))


dataset = TensorDataset(data, labels)


dataloader_pinned = DataLoader(dataset, batch_size=10000, pin_memory=True)

nvtx.push_range("pinned", color="blue")
for batch_data, batch_labels in dataloader_pinned:
    gpu_data = batch_data.to("cuda", non_blocking=True)
    gpu_labels = batch_labels.to("cuda", non_blocking=True)
nvtx.pop_range()


dataloader_pageable = DataLoader(dataset, batch_size=10000, pin_memory=False)

nvtx.push_range("pageable", color="red")
for batch_data, batch_labels in dataloader_pageable:
    gpu_data = batch_data.to("cuda")
    gpu_labels = batch_labels.to("cuda")
nvtx.pop_range()

malikm@a2ap-login02:~/python$
```

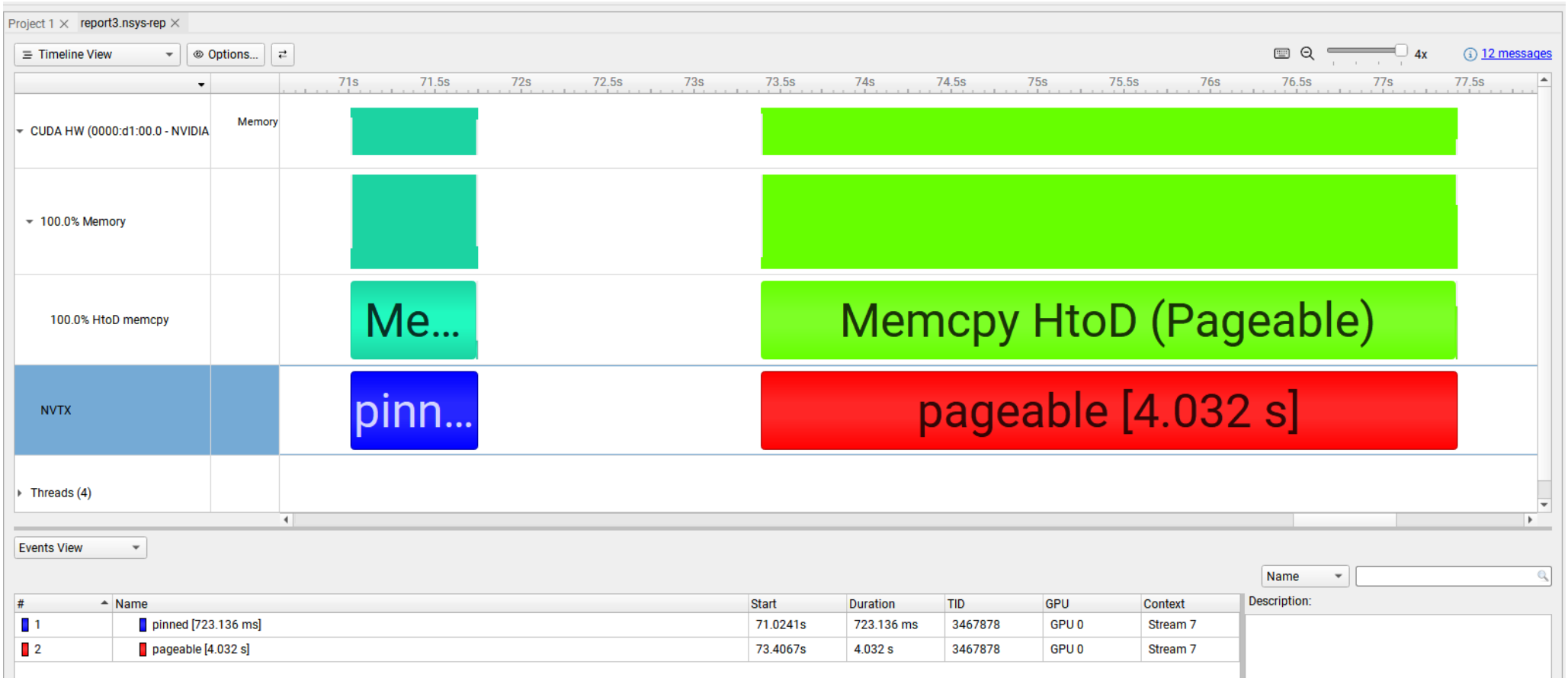
Pinned Memory H to D Copying

Pageable Memory H to D Copying

The profile is shown in the next page

NVTX Python: Pinned vs Pageable Memories

```
malikm@a2ap-dgx035:~/python$ nsys profile --trace=nvtx,cuda python pinned_pageable.py 2>/dev/null
Collecting data...
Generating '/raid/pbs.96793.pbs111/nsys-report-caca.qdstrm'
[1/1] [==========================100%] report3.nsys-rep
Generated:
    /home/users/adm/sup/malikm/python/report3.nsys-rep
```



As can be seen, the HtoD pinned memory copy taken **1/5th** of the time taken by pageable memory.

NVTX Python: Example: Automatic Annotation

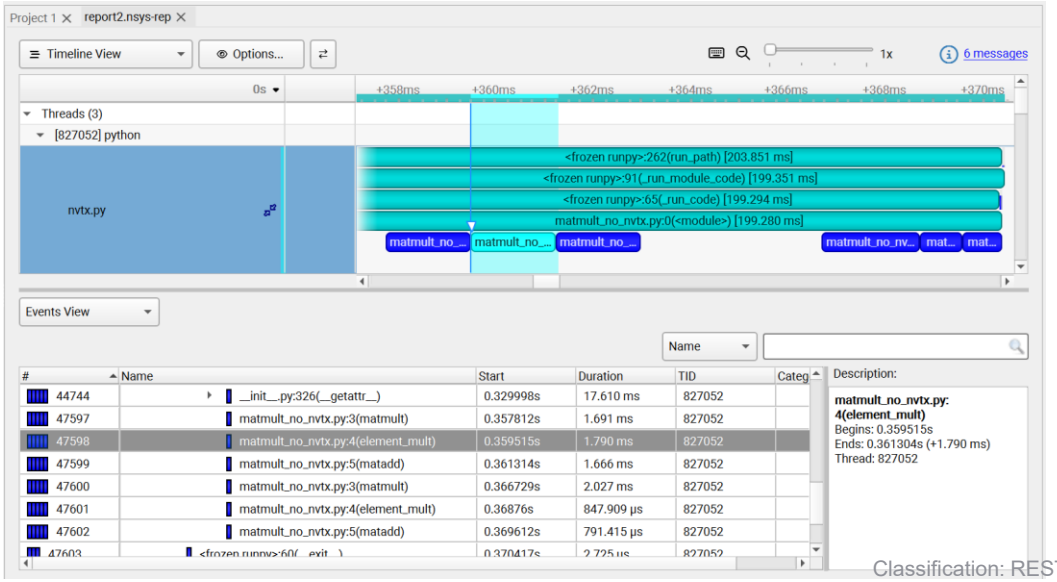
Automatic Annotations

- A code that has not been annotated as shown on right, can be profiled by annotating all functions automatically without changing the code.

- The automatic annotation and profiling of the code on the right is performed as below:

```
malikm@a2ap-dgx034:~/python$ nsys profile python -m nvtx matmult_no_nvtx.py 2>/dev/null
Collecting data...
Generating '/tmp/nsys-report-1717.qdstrm'
[1/1] [========================100%] report2.nsys-rep
Generated:
    /home/users/adm/sup/malikm/python/report2.nsys-rep
malikm@a2ap-dgx034:~/python$
```

```
malikm@a2ap-dgx034:~/python$ cat matmult_no_nvtx.py
import numpy as np

def matmult(A, B): return A @ B
def element_mult(A, B): return A*B
def matadd(A, B): return A + B

if __name__ == "__main__":
    m, n, p = 500, 500, 500

    A = np.random.rand(m, n) + 1j*np.random.rand(m, n)
    B = np.random.rand(n, p) + 1j*np.random.rand(n, p)
    C = matmult(A, B);  C1 = element_mult(A, B)
    D = matadd(A, B)
    E = np.random.rand(m, n); F = np.random.rand(n, p)
    G = matmult(E, F); G1 = element_mult(E, F)
    H = matadd(E, F)
malikm@a2ap-dgx034:~/python$
```



The profiled result is shown on the left. However, this is not giving a deep insight like we did in the previous slides. You can make minimum edit in the code with following lines to focus on a specific area of our code

```
pr = nvtx.Profile()
pr.enable()  # begin annotating function calls
# -- do something -- #
pr.disable()  # stop annotating function calls
```

Python: cProfile: Matrix Multiplication

cProfile is a Python core module for profiling general Python codes.
Let us consider the matrix multiplication code shown below:

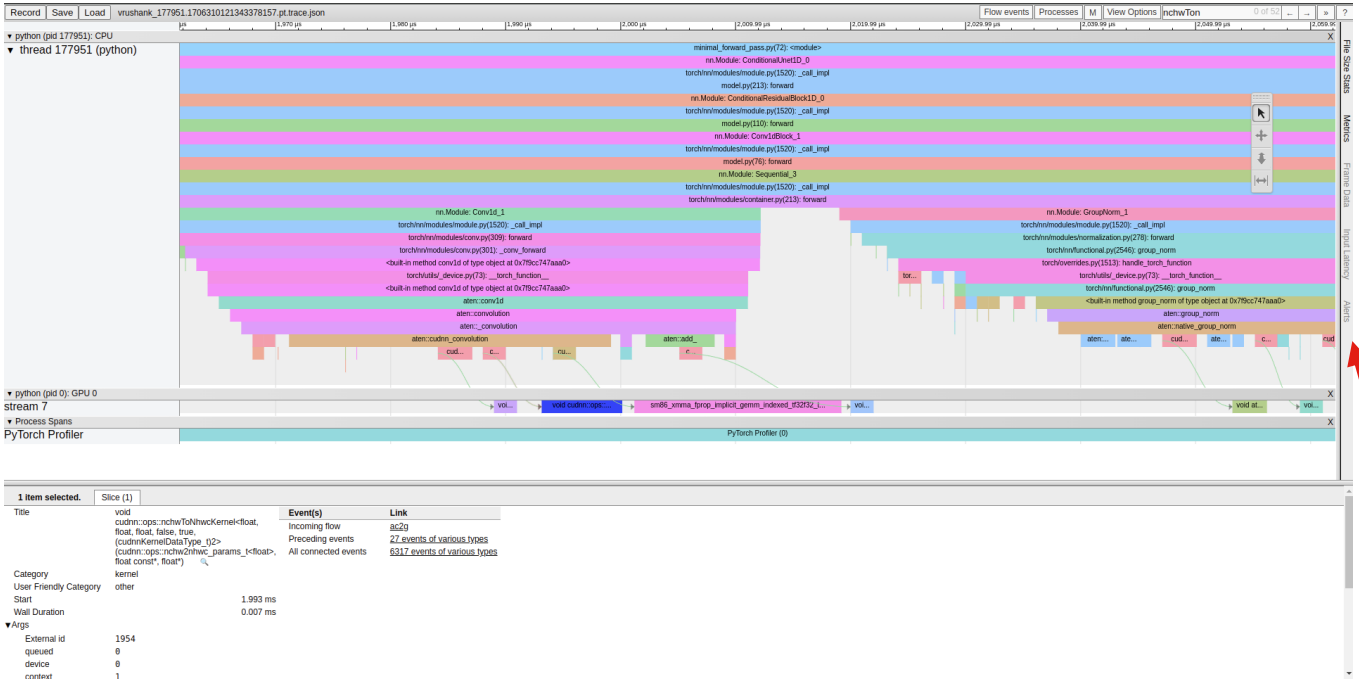
```
malikm@a2ap-dgx034:~/python$ cat matmult_no_nvtx.py
import numpy as np

def matmult(A, B): return A @ B
def element_mult(A, B): return A*B
def matadd(A, B): return A + B

if __name__ == "__main__":
    m, n, p = 4000, 4000, 4000

    A = np.random.rand(m, n) + 1j*np.random.rand(m, n)
    B = np.random.rand(n, p) + 1j*np.random.rand(n, p)
    C = matmult(A, B);  C1 = element_mult(A, B)
    D = matadd(A, B)
    E = np.random.rand(m, n); F = np.random.rand(n, p)
    G = matmult(E, F); G1 = element_mult(E, F)
    H = matadd(E, F)
```

Upon profiling the above code as shown in the screenshot, we see that the element-wise multiplications takes more time than the matrix multiplication. (Probing this surprising result is beyond the scope of this workshop)

```
malikm@a2ap-dgx034:~/python$ python -m cProfile matmult_no_nvtx.py
         100823 function calls (98530 primitive calls) in 1.348 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    103/1    0.000    0.000    1.348    1.348 {built-in method builtins.exec}
        1    0.685    0.685    1.348    1.348 matmult_no_nvtx.py:1(<module>)
        9    0.001    0.000    0.293    0.033 __init__.py:1(<module>)
        2    0.187    0.094    0.187    0.094 matmult_no_nvtx.py:4(element_mult)
        2    0.185    0.092    0.185    0.092 matmult_no_nvtx.py:5(matadd)
        2    0.169    0.085    0.169    0.085 matmult_no_nvtx.py:3(matmult)
```

Profiling Pytorch Models Using “torch.Profiler”

- **Pytorch** has inhouse profiling tools in the form of context managers.
- The results can be sorted in terms of “Self CPU time”, CPU time, CUDA time, etc.
- The results could also be viewed in Chrome browser



A sample visualisation in chrome browser

Profiling Pytorch Models: Iris Classification

Iris flower classification model: Let us consider a classification model on Iris dataset

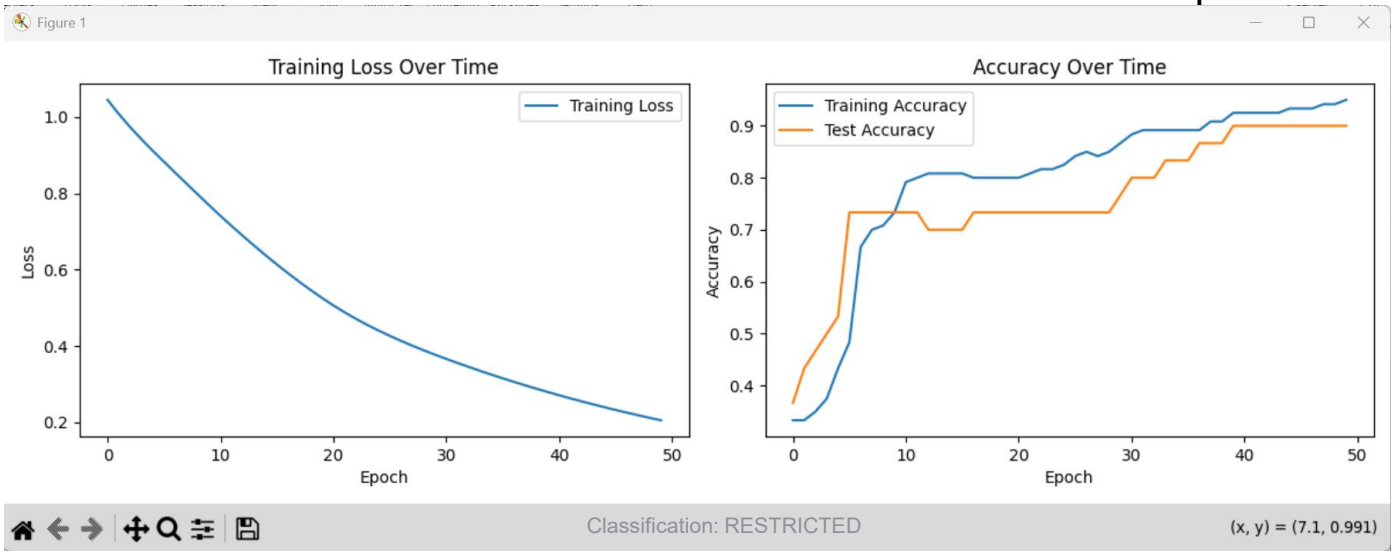
```
malikm@a2ap-login02:~/python$ cat iris.py
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

torch.manual_seed(42)

print("Loading Iris dataset...")
iris = load_iris(); X = iris.data; y = iris.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

X_train_tensor = torch.FloatTensor(X_train)
X_test_tensor = torch.FloatTensor(X_test)
y_train_tensor = torch.LongTensor(y_train)
y_test_tensor = torch.LongTensor(y_test)

class SimpleNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(SimpleNet, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(hidden_size, num_classes)
    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        return x

input_size = 4; hidden_size = 10; num_classes = 3
learning_rate = 0.01; num_epochs = 50

model = SimpleNet(input_size, hidden_size, num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

train_losses = []; train_accuracies = []; test_accuracies = []

print("Starting training...")
print(f"{'Epoch':<6} {'Train Loss':<12} {'Train Acc':<10} {'Test Acc':<10}")
print("-" * 40)

for epoch in range(num_epochs):
    outputs = model(X_train_tensor)
    loss = criterion(outputs, y_train_tensor)

    optimizer.zero_grad(); loss.backward(); optimizer.step()

    _, predicted = torch.max(outputs.data, 1)
    train_accuracy = (predicted == y_train_tensor).float().mean()

    with torch.no_grad():
        test_outputs = model(X_test_tensor)
        _, test_predicted = torch.max(test_outputs.data, 1)
        test_accuracy = (test_predicted == y_test_tensor).float().mean()

    train_losses.append(loss.item())
    train_accuracies.append(train_accuracy.item())
    test_accuracies.append(test_accuracy.item())
```

Profiling Pytorch Models: Iris Classification

```
    if (epoch + 1) % 10 == 0 or epoch == 0:
        print(f"{epoch + 1:<6} {loss.item():<12.4f} " + \
              f" {train_accuracy.item():<10.4f} {test_accuracy.item():<10.4f}")

print("\nTraining completed!")

model.eval()
with torch.no_grad():
    train_outputs = model(X_train_tensor)
    _, train_predicted = torch.max(train_outputs.data, 1)
    final_train_accuracy = (train_predicted == y_train_tensor).float().mean()

    test_outputs = model(X_test_tensor)
    _, test_predicted = torch.max(test_outputs.data, 1)
    final_test_accuracy = (test_predicted == y_test_tensor).float().mean()

print(f"\nFinal Results:")
print(f"Training Accuracy: {final_train_accuracy.item():.4f}")
print(f"Test Accuracy: {final_test_accuracy.item():.4f}")

print(f"\nSample predictions:")
print("True labels:", y_test[:10])
print("Predicted:  ", test_predicted[:10].numpy())
malikm@a2ap-login02:~/python$
```

The features of this code:

- Loads Iris flower dataset that has four features (X) and one target label (y).
- It has three classes, so 3 different y.
- Our model named “SimpleNet()” uses two fully connected layer, also known as Feedforward, linear or MLP layers.
- Uses Cross Entropy loss and Adam optimizer with learning rate 0.01.
- Trains for 50 epochs without batching



Profiling Pytorch Models: torch.Profiler

- Pytorch provides a class to profile a portion of the code in a context manager:

profiler.profile(with_stack=True, profile_memory=True)

Here, "*with_stack=True*" ensures referring back to the source code while reporting the profiling information.

```
class torch.profiler.profile(*, activities=None, schedule=None, on_trace_ready=None,
record_shapes=False, profile_memory=False, with_stack=False, with_flops=False, with_modules=False,
experimental_config=None, execution_trace_observer=None, acc_events=False, use_cuda=None,
custom_trace_id_callback=None)
```

- In the above options, "use_cuda" is deprecated. The CUDA can be disabled by using "activities" option (see the documentation). It is enabled by default if GPU is available.

- The **torch.Profiler** also provides a way of annotating function calls with chosen names each in a different context manager.

Example: **profiler.record_function("a chosen name")**

Profiling Pytorch Models: Iris Classification

- Let us profile the code for Iris Classification shown earlier.
- The relevant snippets of the code where changes have been made are shown below.
- **The inserted lines have been boxed in cyan color.**

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import torch.autograd.profiler as profiler
```

```
class SimpleNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(SimpleNet, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        with profiler.record_function("Forward Pass"):
            x = self.layer1(x)
            x = self.relu(x)
            x = self.layer2(x)
        return x
```

```
with profiler.profile(with_stack=True, profile_memory=True) as prof:
    for epoch in range(num_epochs):
        outputs = model(X_train_tensor)
        loss = criterion(outputs, y_train_tensor)

        optimizer.zero_grad(); loss.backward(); optimizer.step()

        _, predicted = torch.max(outputs.data, 1)
        train_accuracy = (predicted == y_train_tensor).float().mean()

        with torch.no_grad():
            test_outputs = model(X_test_tensor)
            _, test_predicted = torch.max(test_outputs.data, 1)
            test_accuracy = (test_predicted == y_test_tensor).float().mean()

        train_losses.append(loss.item())
        train_accuracies.append(train_accuracy.item())
        test_accuracies.append(test_accuracy.item())

        if (epoch + 1) % 10 == 0 or epoch == 0:
            print(f"{epoch + 1:<6} {loss.item():<12.4f} " + \
              f" {train_accuracy.item():<10.4f} {test_accuracy.item():<10.4f}")

prof.export_chrome_trace("trace.json")

print(prof.key_averages(group_by_stack_n=5)
        .table(sort_by='cpu_time_total', row_limit=10))

print(prof.key_averages(group_by_stack_n=5)
        .table(sort_by='cuda_time_total', row_limit=10))
```

Profiling Pytorch Models: Iris Classification

The first three inserted lines have been introduced in the earlier slides. Now let us focus on the lines below.

```
prof.export_chrome_trace("trace.json")

print(prof.key_averages(group_by_stack_n=5)
        .table(sort_by='cpu_time_total', row_limit=10))

print(prof.key_averages(group_by_stack_n=5)
        .table(sort_by='cuda_time_total', row_limit=10))
```

- **Profiler.export_chrome_trace("trace_json"):** The method of the Profiler class helps to save the profile log in a json format suitable for view in chrome browser under **chrome://tracing**

- **Profiler.key_averages(group_by_stack_n=5):** This averages the respective "keys" (or the quantities that will be reported) over the last five entries of recent call stack.

- **Profiler.table(sort_by="cpu_time_total"):** The *table* method outputs the result in the form of a table. Here *cpu_time_total* refers to the CPU time taken by including the child processes that would have been stated by the process mentioned in the rows of the table. One can choose to sort only by the main process by the option self_cpu_time.

- **Profiler.table(sort_by="gpu_time_total"):** Same as the above but for the time consumed by the processes that used GPU.

The printed table on the screen while running the code and the visualization are presented in the next couple of slides.

Profiling Pytorch Models: Iris Classification

- The two tables printed on the screen: The top one sorts by total CPU time.
- The bottom one is after sorting by the total GPU time.

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg
Optimizer.step#Adam.step	25.58%	12.576ms	36.23%	17.814ms	356.274us
Forward Pass	10.40%	5.115ms	27.01%	13.278ms	132.785us
aten::linear	0.64%	316.928us	14.27%	7.018ms	35.089us
aten::addmm	9.17%	4.508ms	12.06%	5.928ms	29.638us
cudaLaunchKernel	9.81%	4.824ms	9.81%	4.824ms	2.744us
autograd::engine::evaluate_function: AddmmBackward0	1.19%	586.465us	8.09%	3.979ms	39.794us
AddmmBackward0	0.64%	317.063us	5.34%	2.627ms	26.268us
aten::item	0.58%	285.442us	4.97%	2.445ms	4.305us
aten::_local_scalar_dense	1.09%	535.294us	4.39%	2.160ms	3.803us
aten::mm	2.50%	1.227ms	3.91%	1.922ms	12.811us

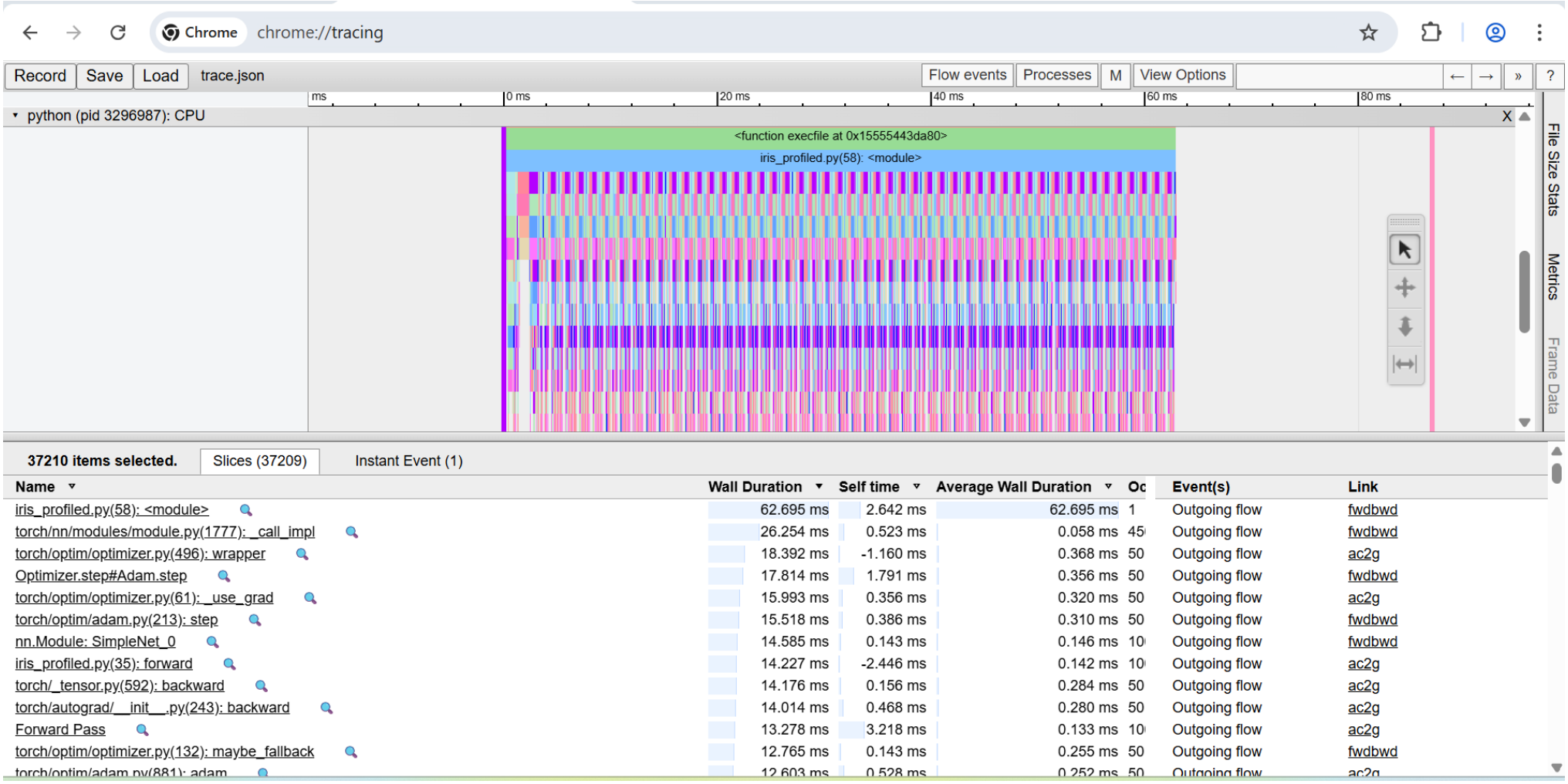
Self CPU time total: 49.168ms

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg
Forward Pass	10.40%	5.115ms	27.01%	13.278ms	132.785us
aten::linear	0.64%	316.928us	14.27%	7.018ms	35.089us
aten::t	1.46%	716.284us	2.62%	1.289ms	2.344us
aten::transpose	0.77%	377.069us	1.16%	572.690us	1.041us
aten::as_strided	0.67%	327.192us	0.67%	327.192us	0.385us
aten::addmm	9.17%	4.508ms	12.06%	5.928ms	29.638us
cudaLaunchKernelExC	0.66%	323.837us	0.66%	323.837us	3.238us
aten::relu	0.66%	322.418us	2.33%	1.145ms	11.453us
aten::clamp_min	1.11%	546.588us	1.67%	822.875us	8.229us
cudaLaunchKernel	9.81%	4.824ms	9.81%	4.824ms	2.744us

Self CPU time total: 49.168ms

Profiling Pytorch Models: Iris Classification

- Once loading the **trace.json** in **chrome://tracing** press the arrow button shown below.
- This will allow you to select the region in the timeline.
- Select the whole of the wavy-looking region which corresponds to the epochs. This will display the stats in a table below as shown.



Name	Wall Duration	Self time	Average Wall Duration	Oc	Event(s)	Link
iris_profiled.py(58): <module>	62.695 ms	2.642 ms	62.695 ms	1	Outgoing flow	fwdbwd
torch/nn/modules/module.py(1777): _call_impl	26.254 ms	0.523 ms	0.058 ms	45	Outgoing flow	fwdbwd
torch/optim/optimizer.py(496): wrapper	18.392 ms	-1.160 ms	0.368 ms	50	Outgoing flow	ac2g
Optimizer.step#Adam.step	17.814 ms	1.791 ms	0.356 ms	50	Outgoing flow	fwdbwd
torch/optim/optimizer.py(61): _use_grad	15.993 ms	0.356 ms	0.320 ms	50	Outgoing flow	ac2g
torch/optim/adam.py(213): step	15.518 ms	0.386 ms	0.310 ms	50	Outgoing flow	fwdbwd
nn.Module: SimpleNet_0	14.585 ms	0.143 ms	0.146 ms	10	Outgoing flow	fwdbwd
iris_profiled.py(35): forward	14.227 ms	-2.446 ms	0.142 ms	10	Outgoing flow	ac2g
torch/_tensor.py(592): backward	14.176 ms	0.156 ms	0.284 ms	50	Outgoing flow	ac2g
torch/autograd/__init__.py(243): backward	14.014 ms	0.468 ms	0.280 ms	50	Outgoing flow	ac2g
Forward Pass	13.278 ms	3.218 ms	0.133 ms	10	Outgoing flow	ac2g
torch/optim/optimizer.py(132): maybe_fallback	12.765 ms	0.143 ms	0.255 ms	50	Outgoing flow	fwdbwd
torch/optim/adam.py(881): adam	12.603 ms	0.528 ms	0.252 ms	50	Outgoing flow	ac2g

Profiling Pytorch Models: Using Nsight Systems

- The profiling by automatic NVTX annotations are also possible in the latest versions of Nsight Sytems (Versions 2025.1 and above).
- But these versions will be installed only in future in ASPIRE 2A PLUS after the December MOS 2025.
- Once installed change the CUDA version to the latest by loading the appropriate CUDA module.
- Then the pytorch code can be profiles as shown in the following screenshot:

```
malikm@a2ap-dgx034:~/python$ nsys profile -t nvtx --pytorch=autograd-nvtx python iris.py
unrecognised option '--pytorch=autograd-nvtx'

usage: nsys profile [<args>] [application] [<application args>]
Try 'nsys profile --help' for more information.
```

Profiling Torchrun Process: Using Nsight Systems

- We will cover Torchrun in “Advanced Workshop on Parallel Computing Models”.
- A specific NCCL rank of the process group can be profiled as shown in the screenshot below:

```
$ cat run.py

import subprocess
import sys
import os local_rank = int(os.environ["LOCAL_RANK"])

args = sys.argv[1:]
args_string = ' '.join(args)

#Define the command to execute
print(f"Profile local rank {local_rank} only")
if local_rank == 0:
  command = "nsys profile -t cuda,nvtx -o test_run python " + args_string
else:
  command = "python " + args_string

#Run the command
subprocess.run(command, shell=True)

$ torchrun --nnodes=1 --nproc-per-node=8 run.py target_python_script.py
```

The boxed command is similar to the one introduced earlier.

The target_python_script.py can have NVTX annotations that we saw earlier.

National Supercomputing Centre

6. Debugging using GDB, CUDA-GDB and PDB

Debugging with GDB

- The GNU Debugger (GDB) is a powerful tool to debug C and C++ codes.

- To use this tool, the compiler flag “-g” need to be passed at the time of compiling. This generates the executable informed with the line numbers of the source code to give a source level support. Example:

```
malikm@a2ap-login02:~/cpp$ g++ -g -o matmult_buggy matmult_buggy.cpp
malikm@a2ap-login02:~/cpp$
```

- Once the executable is generated with this flag, the GDB session can be started as below:

```
malikm@a2ap-login02:~/cpp$ gdb matmult_buggy
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from matmult_buggy...
(gdb)
```

Debugging with GDB

Once into the GDB session, there are a set of commands to navigate the source code while executing them line by line. Some of the most useful once are below:

- **b** or **break <line number>** – Introduction of breakpoints at a chosen line number. Instead of line number, one can also mention a function name to pause the execution before calling that function.
- **r** or **run <options to executable, if any>** - run the code from the start until the breakpoint.
- **delete** – deletes all the breakpoints. **delete <number>** deletes only that breakpoint.
- **info b** – lists the breakpoints with their number.
- **l or list <line number>** – Prints few lines from the source code around the specified line number.
- **p or print <variable name>** – Prints the value of the variable
- **s or step** – step into the next line by executing the current line. If next line is a function call, enter into it.
- **n or next** – step over the next line by executing the current line. If next line is a function call, execute the whole of the function without executing it.
- **u or until** – same as **n** but don't iterate over loops to next iteration.

GDB: Example with Matrix Multiplication

- Let us debug matmult_buggy.cpp shown below with a deliberately introduced bug.
- The line 11 is buggy. "**<=**" should actually be "**<**"

```
 1 #include <iostream>
 2 #include <cstdlib>
 3
 4 using namespace std;
 5
 6 void matrixMultiply(int** A, int** B, int** result,
 7                     int rowsA, int colsA, int colsB) {
 8     for (int i = 0; i < rowsA; i++) {
 9         for (int j = 0; j < colsB; j++) {
10             result[i][j] = 0;
11             for (int k = 0; k <= colsA; k++) {
12                 result[i][j] += A[i][k] * B[k][j];
13             }
14         }
15     }
16 }
17
18 void printMatrix(int** matrix, int rows, int cols) {
19     for (int i = 0; i < rows; i++) {
20         for (int j = 0; j < cols; j++) {
21             cout << matrix[i][j] << " ";
22         }
23         cout << endl;
24     }
25 }
26
27 int main() {
28     const int rowsA = 2;
29     const int colsA = 3;
30     const int colsB = 2;
31
32     int** A = new int*[rowsA];
```

```
33     for (int i = 0; i < rowsA; i++) {
34         A[i] = new int[colsA];
35         for (int j = 0; j < colsA; j++) {
36             A[i][j] = i * colsA + j + 1;
37         }
38     }
39
40     int** B = new int*[colsA]; // colsA = rowsB
41     for (int i = 0; i < colsA; i++) {
42         B[i] = new int[colsB];
43         for (int j = 0; j < colsB; j++) {
44             B[i][j] = i * colsB + j + 1;
45         }
46     }
47
48     // Create result matrix (should be 2x2)
49     int** result = new int*[rowsA];
50     for (int i = 0; i < rowsA; i++) {
51         result[i] = new int[colsB];
52     }
53
54     matrixMultiply(A, B, result, rowsA, colsA, colsB);
55
56     cout << "\nResult (2x2):" << endl;
57     printMatrix(result, rowsA, colsB);
58
59     for (int i = 0; i < rowsA; i++) delete[] A[i];
60     for (int i = 0; i < colsA; i++) delete[] B[i];
61     for (int i = 0; i < rowsA; i++) delete[] result[i];
62     delete[] A; delete[] B; delete[] result;
63 }
```

GDB: Example with Matrix Multiplication

Let us introduce a breakpoint at line number 11, just before the actual bug, and list the code around that number as shown. The run the code until that breakpoint.

```
(gdb) b 11
Breakpoint 1 at 0x1269: file matmult_buggy.cpp, line 11.
(gdb) l 11
6       void matrixMultiply(int** A, int** B, int** result,
7                           int rowsA, int colsA, int colsB) {
8           for (int i = 0; i < rowsA; i++) {
9               for (int j = 0; j < colsB; j++) {
10                  result[i][j] = 0;
11                  for (int k = 0; k <= colsA; k++) {
12                      result[i][j] += A[i][k] * B[k][j];
13                  }
14              }
15          }
(gdb) run
Starting program: /home/users/adm/sup/malikm/cpp/matmult_buggy
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, matrixMultiply (A=0x55555556aeb0, B=0x55555556af10, result=0x55555556af90, rowsA=2, colsA=3, colsB=2) at matmult_buggy.
cpp:11
11                  for (int k = 0; k <= colsA; k++) {
(gdb)
```

Then let us step through the code line-by-line using the command **s** (or **step**):

```
(gdb) s
12                      result[i][j] += A[i][k] * B[k][j];
(gdb) s
11                  for (int k = 0; k <= colsA; k++) {
(gdb) s
12                      result[i][j] += A[i][k] * B[k][j];
(gdb) s
11                  for (int k = 0; k <= colsA; k++) {
(gdb) s
12                      result[i][j] += A[i][k] * B[k][j];
(gdb) s
11                  for (int k = 0; k <= colsA; k++) {
(gdb) s
12                      result[i][j] += A[i][k] * B[k][j];
```

One can see that it is an iteration over k value.

GDB: Example with Matrix Multiplication

Then, buggy line is stepped over as shown. Once the error message is shown, one can inspect the variable values to find out the reason for the error.

```
(gdb) s

Program received signal SIGSEGV, Segmentation fault.
0x00005555555552e2 in matrixMultiply (A=0x55555556aeb0, B=0x55555556af10, result=0x55555556af90, rowsA=2, colsA=3, colsB=2) at matmul
t_buggy.cpp:12
12                      result[i][j] += A[i][k] * B[k][j];
(gdb) print k
$1 = 3
(gdb)
```

As shown in the screenshot:

- The number of columns of the matrix A, i.e., **colsA** is 3.
- This means that its index can run only from 0 to 2.
- However, the **k** has picked up a value of 3 which attempts to access the matrices A and B outside their boundaries in memory.
- This catches the bug. Changing "**<=**" to "**<**" solves it.

Though we have debugged, one can see that call stack by using the "bt" or backtrace as shown below, which would help identifying bugs in a nested call stacks:

```
(gdb) bt
#0  0x00005555555552e2 in matrixMultiply (A=0x55555556aeb0, B=0x55555556af10, result=0x55555556af90, rowsA=2, colsA=3, colsB=2)
    at matmult_buggy.cpp:12
#1  0x000055555555558d in main () at matmult_buggy.cpp:54
(gdb)
```

GDB: Debugging MPI Communication

- Debugging MPI Communication involves attached MPI processes to GDB.
- Consider the following source code:

```cpp
#include <mpi.h>
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Bug: Potential deadlock - process 0 sends but doesn't receive
    if (rank == 0) {
        int data = 42;
        cout << "Process 0 sending data: " << data << endl;
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        // Missing: MPI_Recv from process 1
    }
    else if (rank == 1) {
        int received_data;
        MPI_Recv(&received_data, 1, MPI_INT, 0, 0,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        cout << "Process 1 received: " << received_data << endl;
        // Process 1 tries to send back but process 0 isn't receiving
        int response = 100;
        MPI_Send(&response, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
```

- In this code 2 MPI processes are used.
- Rank 0 sends a data to Rank 1 but fails to receive the data sent from Rank 1.
- Then the Rank 1 calls “MPI_BARRIER()” in the end. The rule is that this function need to be called by all other processes. Since this is not called by the Rank 0, this will cause the program to hang.

GDB: Debugging MPI Communication

Let us compile the code and run it in the background. Because this program is buggy, the MPI communications will hang, thus allowing us time for attaching the MPI processes to GDB and debug.

After giving execution command, list the processes of this executable to find the process-ids:

```
malikm@a2ap-dgx034:~/cpp$ mpic++ -g -o mpi_debug mpi_debug.cpp
malikm@a2ap-dgx034:~/cpp$ mpirun -np 2 ./mpi_debug &
[1] 3478496
malikm@a2ap-dgx034:~/cpp$ Process 1 received: 42
Process 0 sending data: 42

malikm@a2ap-dgx034:~/cpp$ ps aux | grep mpi_debug
malikm   3478496  6.0  0.0 42673988 37748 pts/0  Sl   17:14   0:00 mpirun -np 2 ./mpi_debug
malikm   3478548  1.1  0.0 183716 17008 pts/0    Sl   17:14   0:00 ./mpi_debug
malikm   3478550 94.0  0.0 183720 17320 pts/0    Rl   17:14   0:10 ./mpi_debug
malikm   3478921  0.0  0.0   7012  2228 pts/0    S+   17:14   0:00 grep --color=auto mpi_debug
malikm@a2ap-dgx034:~/cpp$
```

We see that the processes 3478548 and 3478550 are the MPI processes.

GDB: Debugging MPI Communication

- Then attach the first process to the GDB and find out where the code hangs:

```
malikm@a2ap-dgx034:~/cpp$ gdb -p 3478548
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 3478548
[New LWP 3478553]
[New LWP 3478554]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1".
0x000015555502f7f8 in clock_nanosleep () from /usr/lib/x86_64-linux-gnu/libc.so.6
(gdb) where
#0  0x000015555502f7f8 in clock_nanosleep () from /usr/lib/x86_64-linux-gnu/libc.so.6
#1  0x0000155555034677 in nanosleep () from /usr/lib/x86_64-linux-gnu/libc.so.6
#2  0x0000155555065f2f in usleep () from /usr/lib/x86_64-linux-gnu/libc.so.6
#3  0x0000155555420c17 in ompi_mpi_finalize () from /usr/lib/x86_64-linux-gnu/libmpi.so.40
#4  0x000055555555d802 in main (argc=1, argv=0x7fffffffcba8) at mpi_debug.cpp:31
(gdb)
```

("where" is another name for "backtrace")

```
malikm@a2ap-dgx034:~/cpp$ gdb -p 3478550
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 3478550
[New LWP 3478551]
[New LWP 3478552]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1".
0x0000155554ec070f in opal_progress () from /usr/lib/x86_64-linux-gnu/libopen-pal.so.40
(gdb)
(gdb) where
#0  0x0000155554ec070f in opal_progress () from /usr/lib/x86_64-linux-gnu/libopen-pal.so.40
#1  0x00001555554110e5 in ompi_request_default_wait () from /usr/lib/x86_64-linux-gnu/libmpi.so.40
#2  0x000015555546cd13 in ompi_coll_base_barrier_intra_recursivedoubling () from /usr/lib/x86_64-li
#3  0x0000155555425482 in PMPI_Barrier () from /usr/lib/x86_64-linux-gnu/libmpi.so.40
#4  0x000055555555d7fd in main (argc=1, argv=0x7fffffffcba8) at mpi_debug.cpp:28
(gdb)
```

- The left screenshot corresponds to rank 0.
- As said by the last line, this rank hangs in the statement, MPI_FINALIZE() (Line 31 of the source code shown before). This means that it has finished its part and waiting for the other process also to reach this statement.
- But the Rank 1, shown on right, hangs in the line 28 of the source code, which is MPI_BARRIER() statement. Since this statement is not executed by Rank 0, Rank 1 waits for it to execute, causing the over all hanging.

Debugging with CUDA-GDB

Enabling debugging for both the host and device codes:

```
malikm@a2ap-dgx034:~/cpp$ nvcc -g -G -o vectadd_cudagdb -gencode arch=compute_90,code=sm_90 vectoradd.cu
malikm@a2ap-dgx034:~/cpp$
```

CUDA-GDB is similar to GDB with respect to debugging the host code.

- Debug specific thread in the GPU
- Focus on threads in a block
- Focus the debugging to specific kernels
- Print the values of variables in the GPU
- Set breakpoints in the kernel function.
- Check the GPU memory
- Print the values of arrays in GPU kernels
- Trace the stack of calls in GPU to different device kernels.
- Print instructions and registers

```
malikm@a2ap-dgx034:~/cpp$ cuda-gdb vectadd_cudagdb
NVIDIA (R) cuda-gdb 12.6
Portions Copyright (C) 2007-2024 NVIDIA Corporation
Based on GNU gdb 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.
This is free software: you are free to change and redist
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This CUDA-GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://forums.developer.nvidia.com/c/developer-tools/
Find the CUDA-GDB manual and other documentation resourc
    <https://docs.nvidia.com/cuda/cuda-gdb/index.html>.

For help, type "help".
Type "apropos word" to search for commands related to "
Reading symbols from vectadd_cudagdb...
(cuda-gdb)
(cuda-gdb)
(cuda-gdb)
```

Debugging with CUDA-GDB

The general workflow is similar to that of GDB.

- **b** or **break <line number>**
- **r** or **run <options to executable, if any>**
- **delete**
- **info b**
- **l or list <line number>**
- **p or print <variable name>**
- **s or step**
- **n or next**
- **u or until**

Apart from these operations, CUDA-GDB also the following important additional commands

- **set cuda break_on_launch application**
- Conditional: **break <kernel_func_name> if threadIdx.x == 15 and blockIdx.x==1**
- **cuda kernel <k> block <i,j,k> thread <l,m,n>**
- **info cuda devices; info cuda threads,** etc
- **set cuda memcheck on**
- Setting **autostepping** for precise error mesg.

```
(cuda-gdb) autostep vectoradd.cu:13 for 11 lines
Note: breakpoint 1 also set at pc 0x55555555fdfc.
Breakpoint 2 at 0x55555555fdfc: file /home/users/adm/sup/malikm/cpp/vectoradd.cu, line 13.
Created autostep of length 11 lines
(cuda-gdb) info autosteps
Num     Type           Disp Enb Address            What
1       autostep       keep y   0x000055555555fdfc in main at /home/users/adm/sup/malikm/cpp/vectoradd.cu:13 for 11 lines
2       autostep       keep y   0x000055555555fdfc in main at /home/users/adm/sup/malikm/cpp/vectoradd.cu:13 for 11 lines
(cuda-gdb)
```

Inspecting Vector Addition with CUDA-GDB

Let us consider the sample code below. It has been line-numbered for the easy of presentation.

```
 1 #include <iostream>
 2 #include <cuda_runtime.h>
 3 #include <cuda_profiler_api.h>
 4
 5 __global__ void vectorAdd(const float *a, const float *b, float *c, int n) {
 6     int i = blockIdx.x * blockDim.x + threadIdx.x; // Compute thread index
 7     if (i < n) {
 8         c[i] = a[i] + b[i]; // Perform addition
 9     }
10 }
11
12 int main(int argc, char* argv[]) {
13     int n = std::stoi(argv[1]); size_t size = n * sizeof(float);
14     float *h_a, *h_b, *h_c; h_a = new float[n]; h_b = new float[n]; h_c = new float[n];
15
16     for (int i = 0; i < n; i++) {
17         h_a[i] = i; h_b[i] = i * 2;
18     }
19
20     float *d_a, *d_b, *d_c;
21     cudaProfilerStart();
22     cudaMalloc((void**)&d_a, size); cudaMalloc((void**)&d_b, size); cudaMalloc((void**)&d_c, size);
23     cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
24     cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
25
26     int blockSize = 512; int gridSize = (n + blockSize - 1) / blockSize;
27     vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
28     cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
29
30     for (int i = 0; i < 10; i++) {
31         std::cout << h_a[i] << " + " << h_b[i] << " = " << h_c[i] << std::endl;
32     }
33
34     delete[] h_a; delete[] h_b; delete[] h_c; cudaFree(d_a); cudaFree(d_b);  cudaFree(d_c);
35     cudaProfilerStop();
36 }
```

Inspecting Vector Addition with CUDA-GDB

Let us start the cuda-gdb session on the vectadd_cudagdb executable as shown in three slides before and set the breakpoint before kernel launch.

```
(cuda-gdb) set cuda break_on_launch application
(cuda-gdb) run 100000000
Starting program: /home/users/adm/sup/malikm/cpp/vectadd_cudagdb 100000000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x15550bb66000 (LWP 1982010)]
[New Thread 0x15550ae83000 (LWP 1982011)]
[Detaching after fork from child process 1982012]
[New Thread 0x155508401000 (LWP 1982087)]
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device 0, sm 128, warp 0, lane 0]

CUDA thread hit application kernel entry function breakpoint, vectorAdd<<<(195313,1,1),(512,1,1)>>> (a=0x15529e000000,
    b=0x155286000000, c=0x15526e000000, n=100000000) at vectoradd.cu:6
6           int i = blockIdx.x * blockDim.x + threadIdx.x; // Compute thread index
(cuda-gdb) step
7           if (i < n) {
(cuda-gdb) step
8               c[i] = a[i] + b[i]; // Perform addition
(cuda-gdb) step
10      }
(cuda-gdb) print i
$1 = 0
(cuda-gdb) print c[0]
$2 = 0
(cuda-gdb)
```

The command line argument for the executable

- Here we have stepped into the kernel and have inspected the running index and the resultant vector after the first iteration.
- The line number of the breakpoint is shown as 6

Inspecting Vector Addition with CUDA-GDB

Let us restart the session and set breakpoint at line 9. Then inspect the value of " **i** "

```
(cuda-gdb) break vectoradd.cu:9
Breakpoint 1 at 0xc43a: file /home/users/adm/sup/malikm/cpp/vectoradd.cu, line 10.
(cuda-gdb) l 9
4
5       __global__ void vectorAdd(const float *a, const float *b, float *c, int n) {
6           int i = blockIdx.x * blockDim.x + threadIdx.x; // Compute thread index
7           if (i < n) {
8               c[i] = a[i] + b[i]; // Perform addition
9           }
10      }
11
12      int main(int argc, char* argv[]) {
13          int n = std::stoi(argv[1]); size_t size = n * sizeof(float);
(cuda-gdb) run 100000000
Starting program: /home/users/adm/sup/malikm/cpp/vectadd_cudagdb 100000000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x15550bb66000 (LWP 3666826)]
[New Thread 0x15550ae83000 (LWP 3666831)]
[Detaching after fork from child process 3666832]
[New Thread 0x155508401000 (LWP 3667313)]
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (32,0,0), device 0, sm 128, warp 1, lane 0]

CUDA thread hit Breakpoint 1.1, vectorAdd<<<(195313,1,1),(512,1,1)>>> (a=0x15529e000000, b=0x155286000000, c=0x15526e000000,
    n=100000000) at vectoradd.cu:10
10      }
(cuda-gdb) print i
$1 = 32
(cuda-gdb) cuda kernel 0 block(100,0,0) thread(100,0,0)
[Switching focus to CUDA kernel 0, grid 1, block (100,0,0), thread (100,0,0), device 0, sm 42, warp 3, lane 4]
10      }
(cuda-gdb) print i
$2 = 51300
(cuda-gdb)
```

The value of $i = 32$, since the computation takes by warp after warp.

In block(1,0,0) and thread(100,0,0), we get i = $blockIdx.x \times blockDim.x + threadIdx.x = 100 \times 512 + 100 = 51300$

Inspecting Vector Addition with CUDA-GDB

The GPU information can be obtained, while the code has been halted at the breakpoint, as follows:

```
(cuda-gdb) info cuda devices
  Dev PCI Bus/Dev ID                 Name Description SM Type SMs Warps/SM Lanes/Warp Max Regs/Lane                 Active SMs Mask
*   0        1b:00.0 NVIDIA H100 80GB HBM3   GH100GL-A   sm_90 132       64         32           256 0xffffffffffffffffffffffffffffffff
    1        43:00.0 NVIDIA H100 80GB HBM3   GH100GL-A   sm_90 132       64         32           256 0x00000000000000000000000000000000
    2        52:00.0 NVIDIA H100 80GB HBM3   GH100GL-A   sm_90 132       64         32           256 0x00000000000000000000000000000000
    3        61:00.0 NVIDIA H100 80GB HBM3   GH100GL-A   sm_90 132       64         32           256 0x00000000000000000000000000000000
    4        9d:00.0 NVIDIA H100 80GB HBM3   GH100GL-A   sm_90 132       64         32           256 0x00000000000000000000000000000000
    5        c3:00.0 NVIDIA H100 80GB HBM3   GH100GL-A   sm_90 132       64         32           256 0x00000000000000000000000000000000
    6        d1:00.0 NVIDIA H100 80GB HBM3   GH100GL-A   sm_90 132       64         32           256 0x00000000000000000000000000000000
    7        df:00.0 NVIDIA H100 80GB HBM3   GH100GL-A   sm_90 132       64         32           256 0x00000000000000000000000000000000
(cuda-gdb)
```

From this, we note the following:

- We have 8 H100 GPUs each with 80 GB high bandwidth memory.
- SM's are of the compute category 9.0 under the NVIDIA's categorization for CUDA capability.
- Each GPU has 132 SM's.
- Each SM can schedule 64 warps at a time, that is 2048 threads at a time.
- Each warp has 32 threads
- For each thread, there are 256 instructions in the register file.
- We are using only one GPU at the moment for the vector-addition program.

Inspecting Vector Addition with CUDA-GDB

We can get the call stack information from any of these equivalent commands : bt, backtrace, where, and info stack.

```
    7        d1:00.0 NVIDIA H100 80GB HBM3    GH100GL-A    sm_90 132       64         32           256 0x0000000000000000000000000000000
(cuda-gdb) bt
#0  vectorAdd<<<(195313,1,1),(512,1,1)>>> (a=0x15529e000000, b=0x155286000000, c=0x15526e000000, n=100000000) at vectoradd.cu:10
(cuda-gdb) where
#0  vectorAdd<<<(195313,1,1),(512,1,1)>>> (a=0x15529e000000, b=0x155286000000, c=0x15526e000000, n=100000000) at vectoradd.cu:10
(cuda-gdb) info stack
#0  vectorAdd<<<(195313,1,1),(512,1,1)>>> (a=0x15529e000000, b=0x155286000000, c=0x15526e000000, n=100000000) at vectoradd.cu:10
(cuda-gdb)
```

- Since there is only one kernel the output looks simpler in the above screenshot.
- The kernel calls other kernels, there will be a stack of their calls in the output.

Compute-sanitizer: Inspecting Vector Addition

Compute-sanitizer

This is a tool to check illegal memory accesses, stack overflow, race conditions and errors related to synchronization.

```
malikm@a2ap-dgx034:~/cpp$ compute-sanitizer --tool memcheck  ./vectadd_cudagdb 100000000
========= COMPUTE-SANITIZER
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
3 + 6 = 9
4 + 8 = 12
5 + 10 = 15
6 + 12 = 18
7 + 14 = 21
8 + 16 = 24
9 + 18 = 27
========= ERROR SUMMARY: 0 errors
malikm@a2ap-dgx034:~/cpp$
malikm@a2ap-dgx034:~/cpp$
malikm@a2ap-dgx034:~/cpp$
malikm@a2ap-dgx034:~/cpp$ compute-sanitizer --tool racecheck  ./vectadd_cudagdb 100000000
========= COMPUTE-SANITIZER
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
3 + 6 = 9
4 + 8 = 12
5 + 10 = 15
6 + 12 = 18
7 + 14 = 21
8 + 16 = 24
9 + 18 = 27
========= RACECHECK SUMMARY: 0 hazards displayed (0 errors, 0 warnings)
malikm@a2ap-dgx034:~/cpp$
```

For more info: https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html

For examples of using it on a couple of buggy codes, please refer to: https://developer.nvidia.com/blog/debugging-cuda-more-efficiently-with-nvidia-compute-sanitizer/

PDB: Debugging Python Codes

Python module PDB debugs codes in a manner GDB debugs C or C++ codes

All of the commands that we saw with GDB are applicable

```python
import numpy as np
import scipy as scp

A = np.array([[3, 2],
              [1, 4]])

B = np.array([[1, 2],
              [2, 4]])
I = np.array([[1, 0],
              [0, 1]])

def eigvals_1(A):
    return np.linalg.eigvals(A)
def eigvals_2(A):
    return np.linalg.eig(A)
def eigvals_3(A):
    return scp.linalg.eig(A,I)

eigenvalues_method_1 = eigvals_1(A)
print("Eigenvalues (using eigvals):", eigenvalues_method_1)

eigenvalues_method_2, eigenvectors = np.linalg.eig(A)
print("Eigenvalues (using eig):", eigenvalues_method_2)
print("Eigenvectors:", eigenvectors)


print("Eigenvalues from Scipy GEV", eigvals_3(A))
```

- For an example to play with, let us consider the code **eig_debug.py** shown on the left.
- This code does not have any bugs, but the next slides demonstrates that the workflow with PDB is same as that of GDB.
- In this program eigenvalues of a matrix A is found by three different algorithms, the last being the generalized eigenvalue method.

PDB: Debugging Python Codes

In the workflow we have shown, the debugging session is started as:

python –m pdb eig_debug.py

Then the rest of the way to navigate the source code line by line is same as we did in the case of gdb.

```
malikm@a2ap-login01:~/python$ python -m pdb eig_debug.py
> /home/users/adm/sup/malikm/python/eig_debug.py(1)<module>()
-> import numpy as np
(Pdb) b 13
Breakpoint 1 at /home/users/adm/sup/malikm/python/eig_debug.py:13
(Pdb) r
> /home/users/adm/sup/malikm/python/eig_debug.py(13)eigvals_1()
-> return np.linalg.eigvals(A)
(Pdb) s
--Call--
> /home/users/adm/sup/malikm/scratch/python/python-3.12.11/lib/python3.12/site-packages/numpy/linalg/_linalg.py(492)_unary_dispatcher
()
-> def _unary_dispatcher(a):
(Pdb) w
  /home/users/adm/sup/malikm/scratch/python/python-3.12.11/lib/python3.12/bdb.py(627)run()
-> exec(cmd, globals, locals)
  <string>(1)<module>()
  /home/users/adm/sup/malikm/python/eig_debug.py(19)<module>()
-> eigenvalues_method_1 = eigvals_1(A)
  /home/users/adm/sup/malikm/python/eig_debug.py(13)eigvals_1()
-> return np.linalg.eigvals(A)
> /home/users/adm/sup/malikm/scratch/python/python-3.12.11/lib/python3.12/site-packages/numpy/linalg/_linalg.py(492)_unary_dispatcher
()
-> def _unary_dispatcher(a):
(Pdb)
```

Contact Us

National Supercomputing Centre

Website : https://nscc.sg

Self Service Portal : https://help.nscc.sg/

Helpdesk : https://keris.service-now.com/csm

Email : help@nscc.sg

Contact : +65 6645 3412

National Supercomputing Centre

Email : help@nscc.sg

National Supercomputing Centre

Thank You

NSCC.SG