CUDA GRAPHS IN GROMACS

Alan Gray, NVIDIA

HiHAT Seminar, 19th January 2021

INTRODUCTION

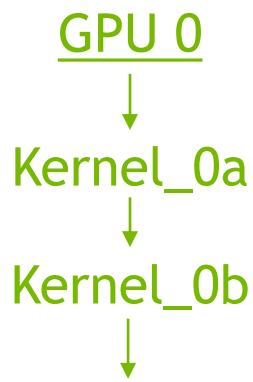
- GROMACS performance can be sensitive to launch overheads, for cases where we have a relatively small number of atoms per GPU.
- Use of CUDA Graphs allows multiple GPU activities to be launched as a single graph.
- I will describe a prototype implementation of graphs in GROMACS, for both single and multi GPU (supporting those cases which allow full GPU offloading)
  - Simple proxy code to demonstrate methods
  - GROMACS performance results and discussion

SIMPLE PROXY CODE TO DEMONSTRATE

```
#pragma omp parallel
    {
        int tid=omp_get_thread_num();
        if(tid==0){cudaSetDevice(0);
            cudaStreamBeginCapture(stream0,cudaStreamCaptureModeGlobal);
            cudaEventRecord(forkEvent,stream0); // Fork stream1
            #pragma omp barrier
            kernel_0a<<<1,1,0,stream0>>>();
            kernel_0b<<<1,1,0,stream0>>>();
            #pragma omp barrier
            cudaStreamWaitEvent(stream0,joinEvent,0); // Join stream1
            cudaStreamEndCapture(stream0,&graph);
            cudaGraphInstantiate(&instance,graph,NULL,NULL,0);
            cudaGraphLaunch(instance,stream0); //launch of graph
            cudaGraphLaunch(instance,stream0); //repeat launch of graph
        }
        if(tid==1){cudaSetDevice(1);
            #pragma omp barrier
            cudaStreamWaitEvent(stream1,forkEvent,0);
            kernel_1<<<1,1,0,stream1>>>();
            cudaEventRecord(joinEvent,stream1);
            #pragma omp barrier
        }
    }
```

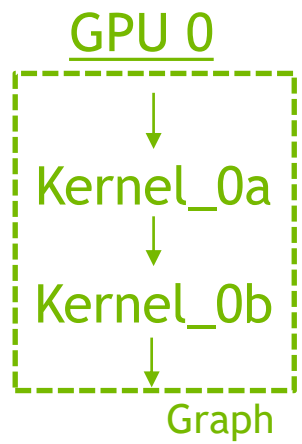
I will describe this step-by-step in following slides

```
#pragma omp parallel
    {
        int tid=omp_get_thread_num();
        if(tid==0){cudaSetDevice(0);
            cudaStreamBeginCapture(stream0,cudaStreamCaptureModeGlobal);
            cudaEventRecord(forkEvent,stream0); // Fork stream1
            #pragma omp barrier
            kernel_0a<<<1,1,0,stream0>>>();
            kernel_0b<<<1,1,0,stream0>>>();
            #pragma omp barrier
            cudaStreamWaitEvent(stream0,joinEvent,0); // Join stream1
            cudaStreamEndCapture(stream0,&graph);
            cudaGraphInstantiate(&instance,graph,NULL,NULL,0);
            cudaGraphLaunch(instance,stream0); //launch of graph
            cudaGraphLaunch(instance,stream0); //repeat launch of graph
        }
        if(tid==1){cudaSetDevice(1);
            #pragma omp barrier
            cudaStreamWaitEvent(stream1,forkEvent,0);
            kernel_1<<<1,1,0,stream1>>>();
            cudaEventRecord(joinEvent,stream1);
            #pragma omp barrier
        }
    }
```



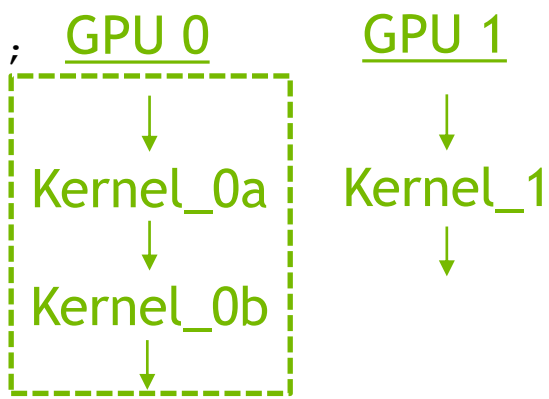

```
POD$ ./compile_and_run.sh
Hello from kernel 0a
Hello from kernel 0b
```

```
#pragma omp parallel
    {
        int tid=omp_get_thread_num();
        if(tid==0){cudaSetDevice(0);
            cudaStreamBeginCapture(stream0,cudaStreamCaptureModeGlobal);
            cudaEventRecord(forkEvent,stream0); // Fork stream1
            #pragma omp barrier
            kernel_0a<<<1,1,0,stream0>>>();
            kernel_0b<<<1,1,0,stream0>>>();
            #pragma omp barrier
            cudaStreamWaitEvent(stream0,joinEvent,0); // Join stream1
            cudaStreamEndCapture(stream0,&graph);
            cudaGraphInstantiate(&instance,graph,NULL,NULL,0);
            cudaGraphLaunch(instance,stream0); //launch of graph
            cudaGraphLaunch(instance,stream0); //repeat launch of graph
        }
        if(tid==1){cudaSetDevice(1);
            #pragma omp barrier
            cudaStreamWaitEvent(stream1,forkEvent,0);
            kernel_1<<<1,1,0,stream1>>>();
            cudaEventRecord(joinEvent,stream1);
            #pragma omp barrier
        }
    }
```



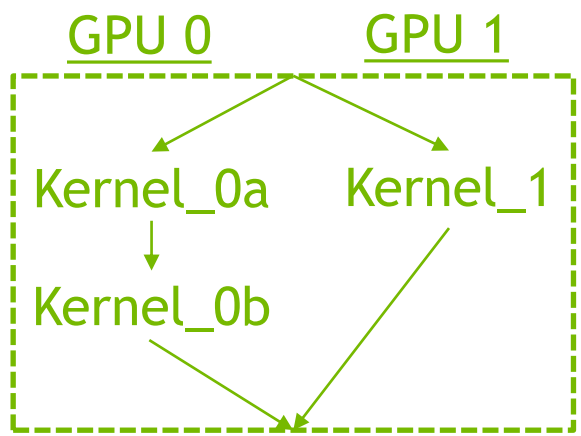

```
POD$ ./compile_and_run.sh
Hello from kernel 0a
Hello from kernel 0b
Hello from kernel 0a
Hello from kernel 0b
```

```
#pragma omp parallel
    {
        int tid=omp_get_thread_num();
        if(tid==0){cudaSetDevice(0);
            cudaStreamBeginCapture(stream0,cudaStreamCaptureModeGlobal);
            cudaEventRecord(forkEvent,stream0); // Fork stream1
            #pragma omp barrier
            kernel_0a<<<1,1,0,stream0>>>();
            kernel_0b<<<1,1,0,stream0>>>();
            #pragma omp barrier
            cudaStreamWaitEvent(stream0,joinEvent,0); // Join stream1
            cudaStreamEndCapture(stream0,&graph);
            cudaGraphInstantiate(&instance,graph,NULL,NULL,0);
            cudaGraphLaunch(instance,stream0); //launch of graph
            cudaGraphLaunch(instance,stream0); //repeat launch of graph
        }
        if(tid==1){cudaSetDevice(1);
            #pragma omp barrier
            cudaStreamWaitEvent(stream1,forkEvent,0);
            kernel_1<<<1,1,0,stream1>>>();
            cudaEventRecord(joinEvent,stream1);
            #pragma omp barrier
        }
    }
```




```
POD$ ./compile_and_run.sh
Hello from kernel 0a
Hello from kernel 0b
Hello from kernel 0a
Hello from kernel 0b
Hello from kernel 1
```

```
#pragma omp parallel
    {
        int tid=omp_get_thread_num();
        if(tid==0){cudaSetDevice(0);
            cudaStreamBeginCapture(stream0,cudaStreamCaptureModeGlobal);
            cudaEventRecord(forkEvent,stream0); // Fork stream1
            #pragma omp barrier
            kernel_0a<<<1,1,0,stream0>>>();
            kernel_0b<<<1,1,0,stream0>>>();
            #pragma omp barrier
            cudaStreamWaitEvent(stream0,joinEvent,0); // Join stream1
            cudaStreamEndCapture(stream0,&graph);
            cudaGraphInstantiate(&instance,graph,NULL,NULL,0);
            cudaGraphLaunch(instance,stream0); //launch of graph
            cudaGraphLaunch(instance,stream0); //repeat launch of graph
        }
        if(tid==1){cudaSetDevice(1);
            #pragma omp barrier
            cudaStreamWaitEvent(stream1,forkEvent,0);
            kernel_1<<<1,1,0,stream1>>>();
            cudaEventRecord(joinEvent,stream1);
            #pragma omp barrier
        }
    }
```

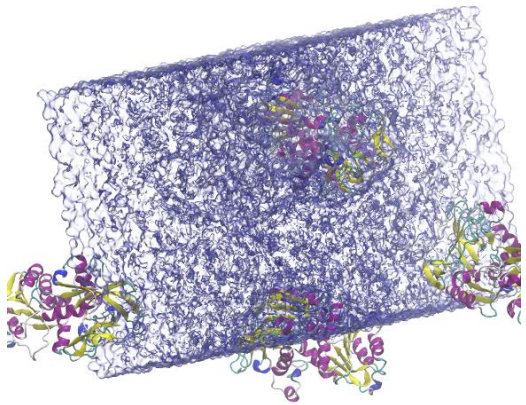



```
POD$ ./compile_and_run.sh
Hello from kernel 0a
Hello from kernel 0b
Hello from kernel 0a
Hello from kernel 0b
Hello from kernel 1
Hello from kernel 1
```

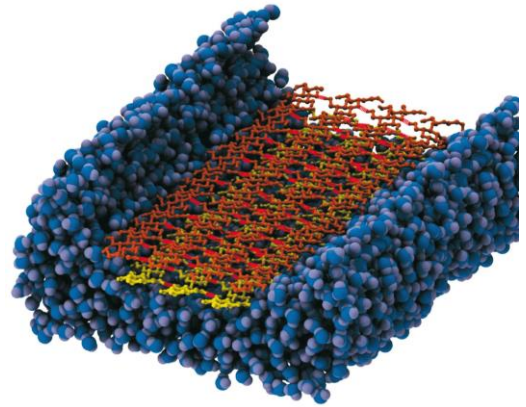
Simple Proxy Code	GROMACS
OpenMP	Thread-MPI (pthreads)
Events in shared memory space	Events passed via thread-MPI calls
2 tasks, one of which controls graph	N PP tasks + 1 PME task. One of the PP tasks controls graph.
Few kernels, single stream per task	Many more kernels, multiple streams per task
No communication	Communication via P2P CUDA memory copies, with synchronization controlled via events (with events shared via thread-MPI).
1 graph launched twice, no changes to graph.	Many graph launches, one per simulation timestep. Different graphs on odd and even steps. Only use graph on steps that don't have any synchronous activities (~70% of steps for ADHD). Graph changes every neighbour search/domain decomposition step.

BENCHMARKS

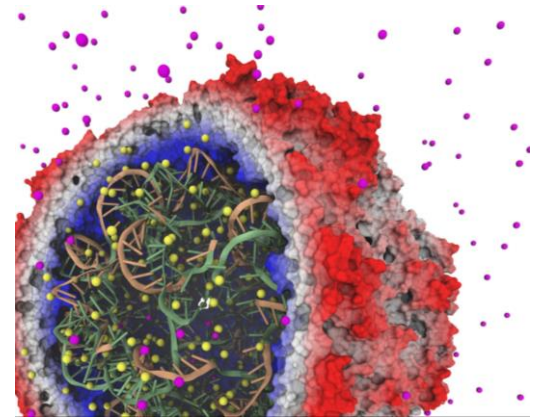
RNASE
~13K atoms



ADH Dodec
~100K atoms



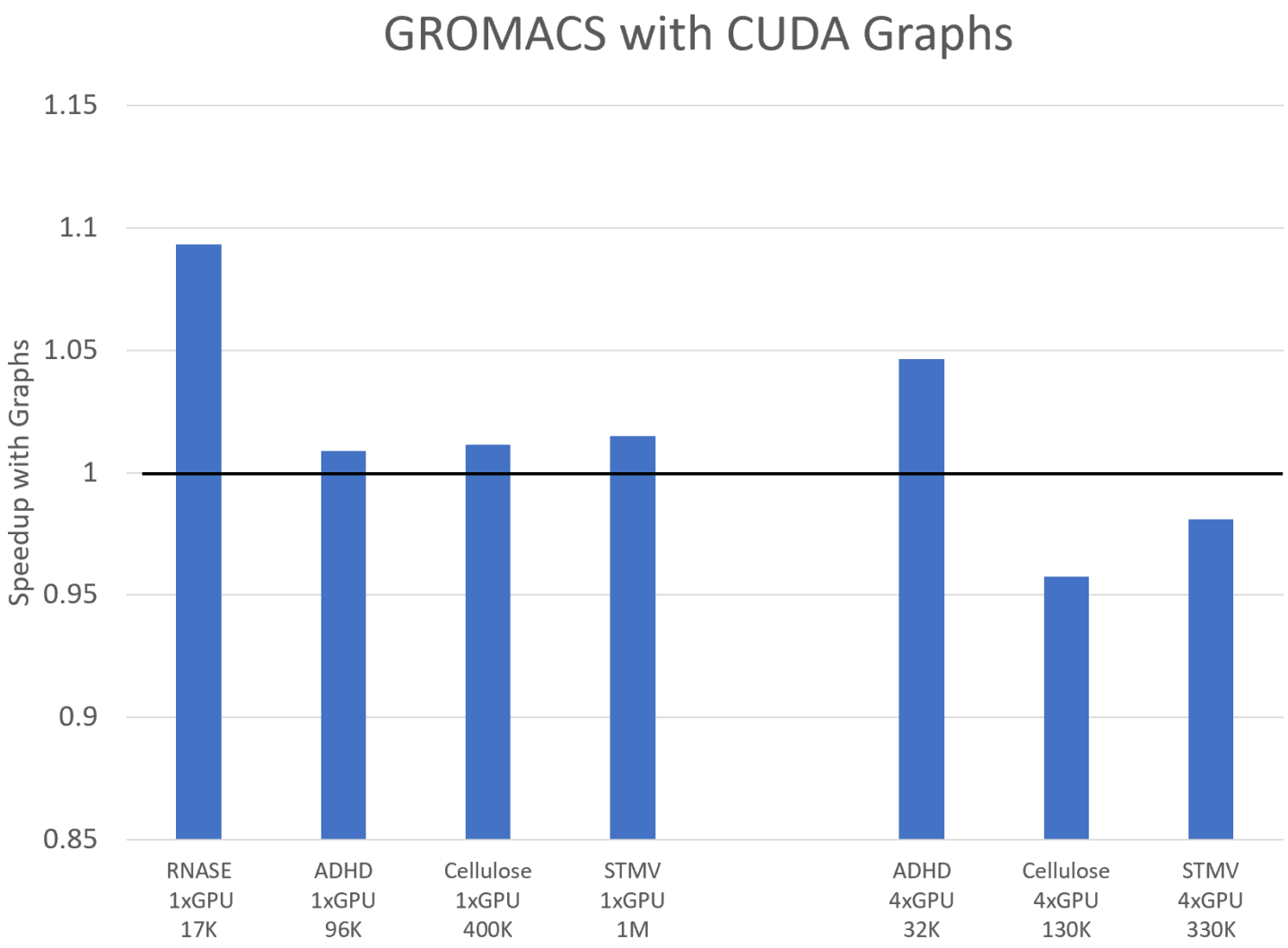
Cellulose
~400K atoms



STMV
~1M atoms

- All allow full GPU offloading of all calculations in regular simulation timesteps
- Graphs expected to have most benefit for small cases sensitive to launch overheads
- Will show results for single-GPU and 4xGPU (atoms divided between the 3 PP GPUs, and other GPU doing full PME)

PERFORMANCE: GRAPHS VS STREAMS



GRAPH RE-INSTANTIATION

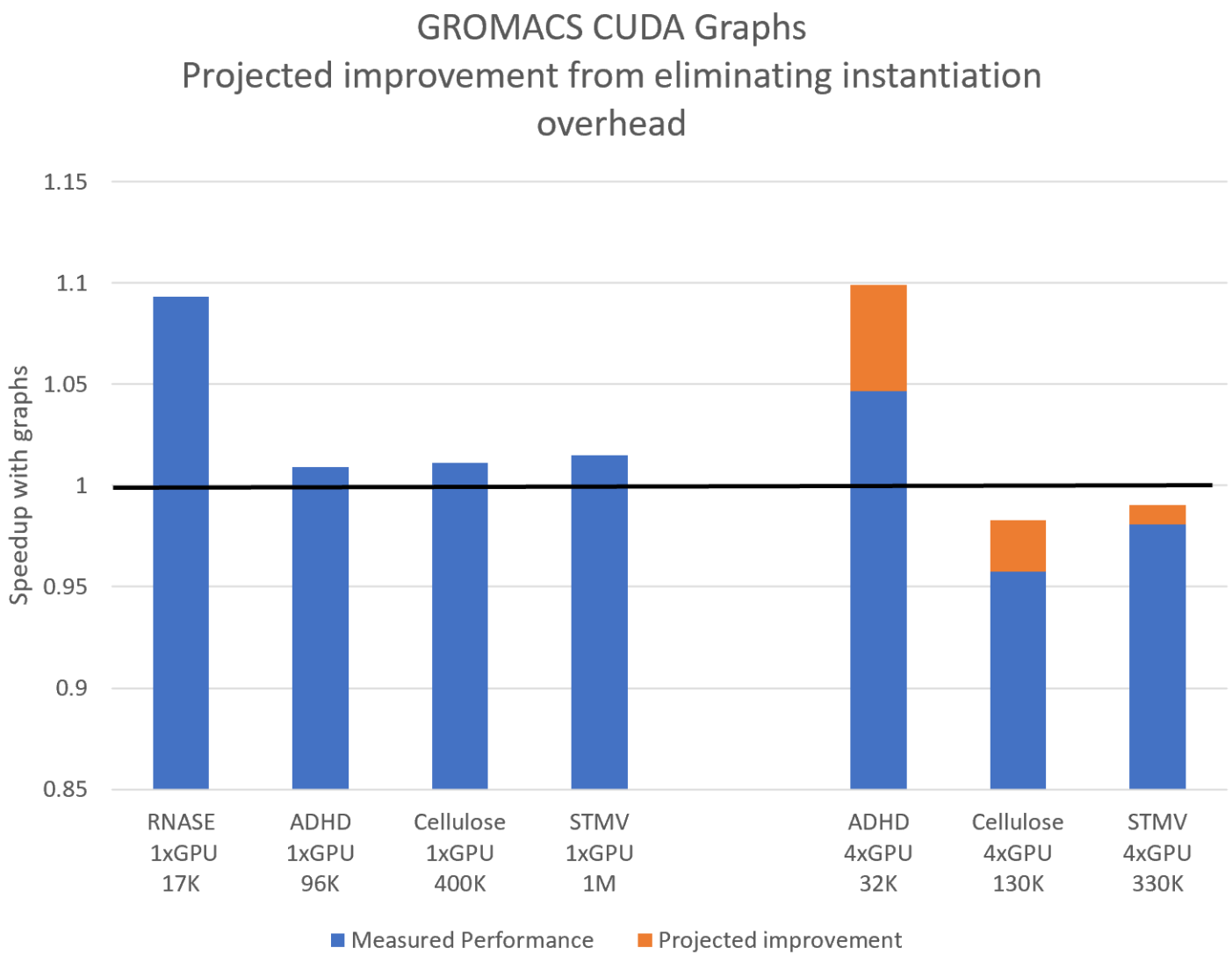
- Graph parameter changes every neighbour search/ domain decomposition (NS/DD) step.
- Can re-record and instantiate, but (while recording is cheap) instantiation is a significant overhead.
- Single GPU: successfully use `cudaGraphExecUpdate` every NS which has insignificant overhead, and avoid re-instantiation

```
__host__cudaError_t cudaGraphExecUpdate ( cudaGraphExec_t hGraphExec, cudaGraph_t hGraph,
cudaGraphNode_t* hErrorNode_out, cudaGraphExecUpdateResult* updateResult_out )
```

> Check whether an executable graph can be updated with a graph and perform the update if possible.

- Multi GPU: there still exists a limitation that prevents us from using this functionality when graph is (re-)captured across multiple CPU threads. Work planned to lift this restriction.

PROJECTED PERFORMANCE



FUTURE WORK

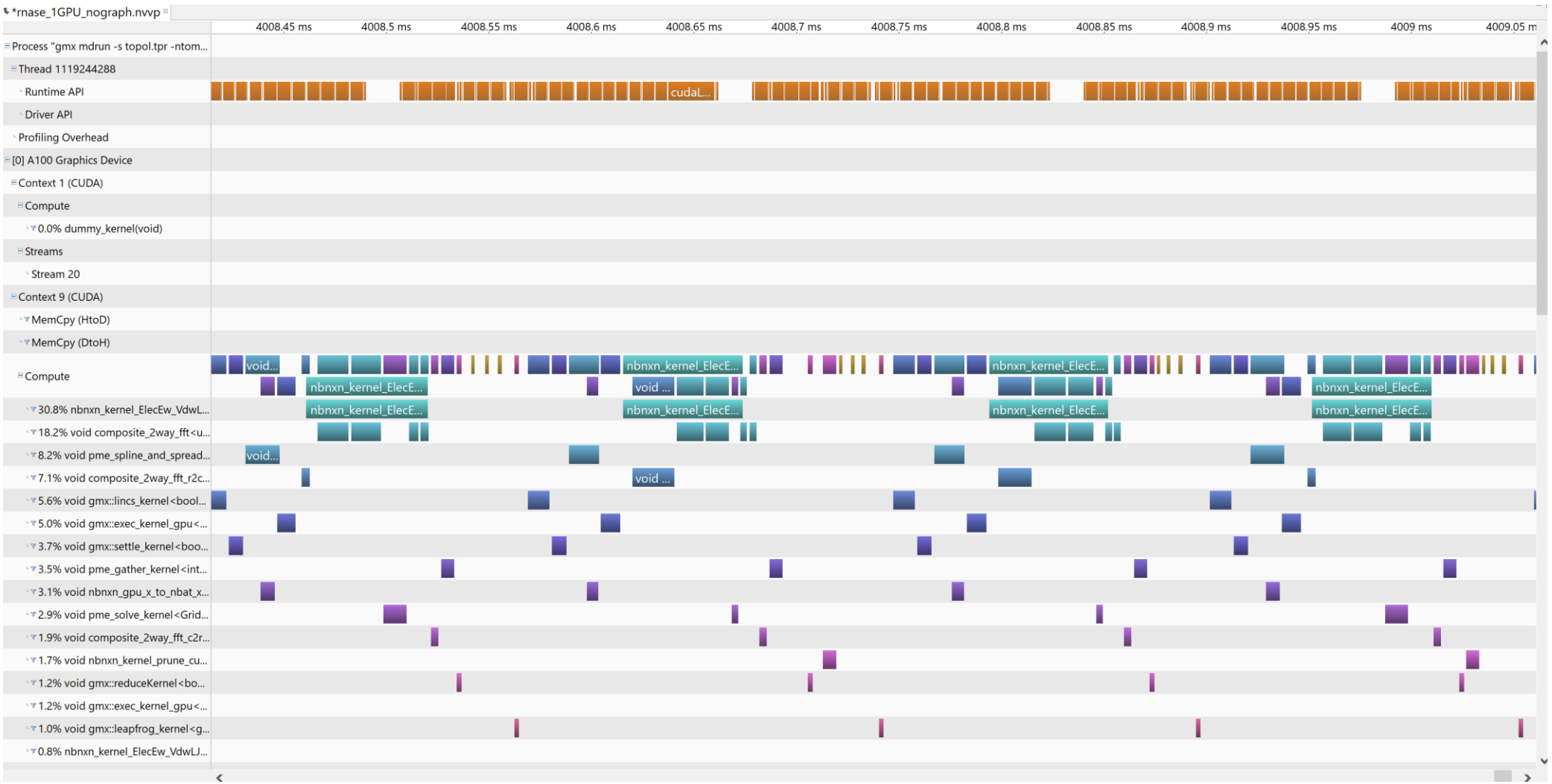
- Extend implementation to also support cases that have CPU involvement every timestep
- Options:
  - Restrict each graph to subset of each timestep
    - Relatively simple, but also restricts benefits of using graphs
  - Host nodes
    - Possibly most “natural” solution, but may have performance limitations
  - Regular CPU code combined with CUDA stream memory operations cuStreamWriteValue32/cuStreamWaitValue32
    - Support is evolving; will require some code refactoring

CONCLUSIONS

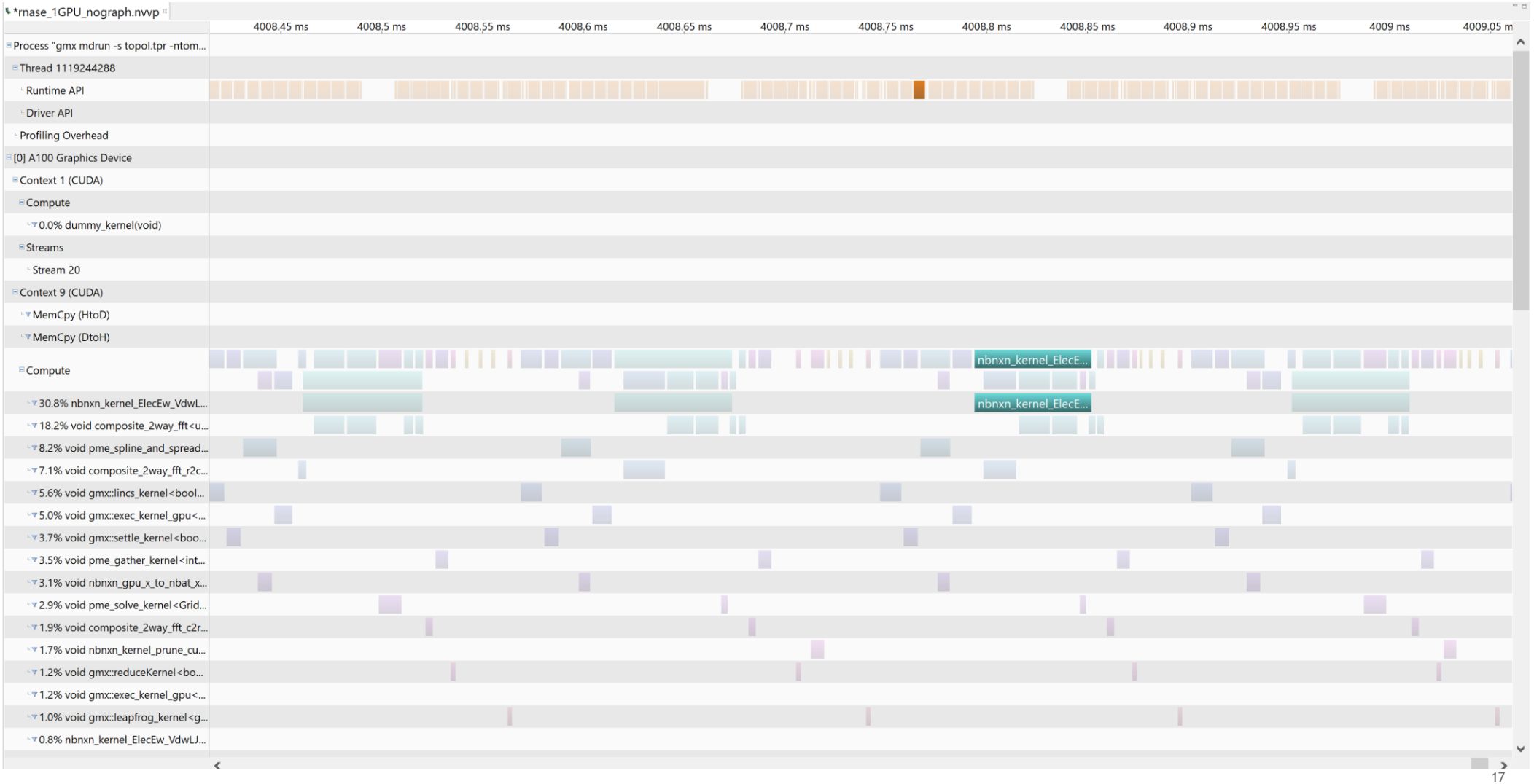
- Prototype implementation of CUDA graphs in GROMACS has been developed.
- Offers a benefit of up to 10% for small cases (tens of thousands of atoms per GPU).
- Further work required to remove instantiation overhead for multi-GPU cases.
- Investigations underway to extend to those cases which require CPU involvement every timestep.

EXTRA SLIDES

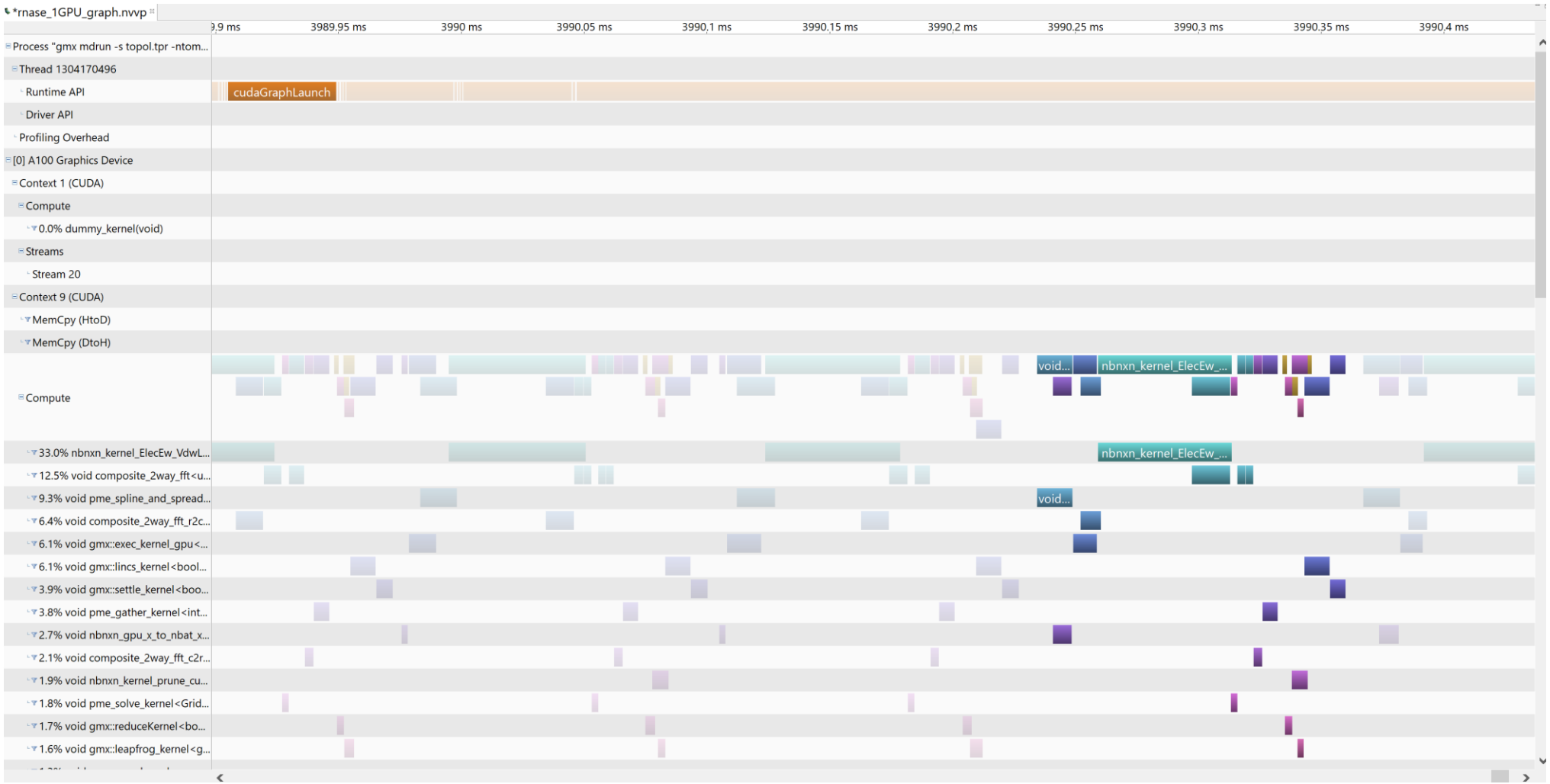
RNASE, 1 GPU, NO GRAPHS



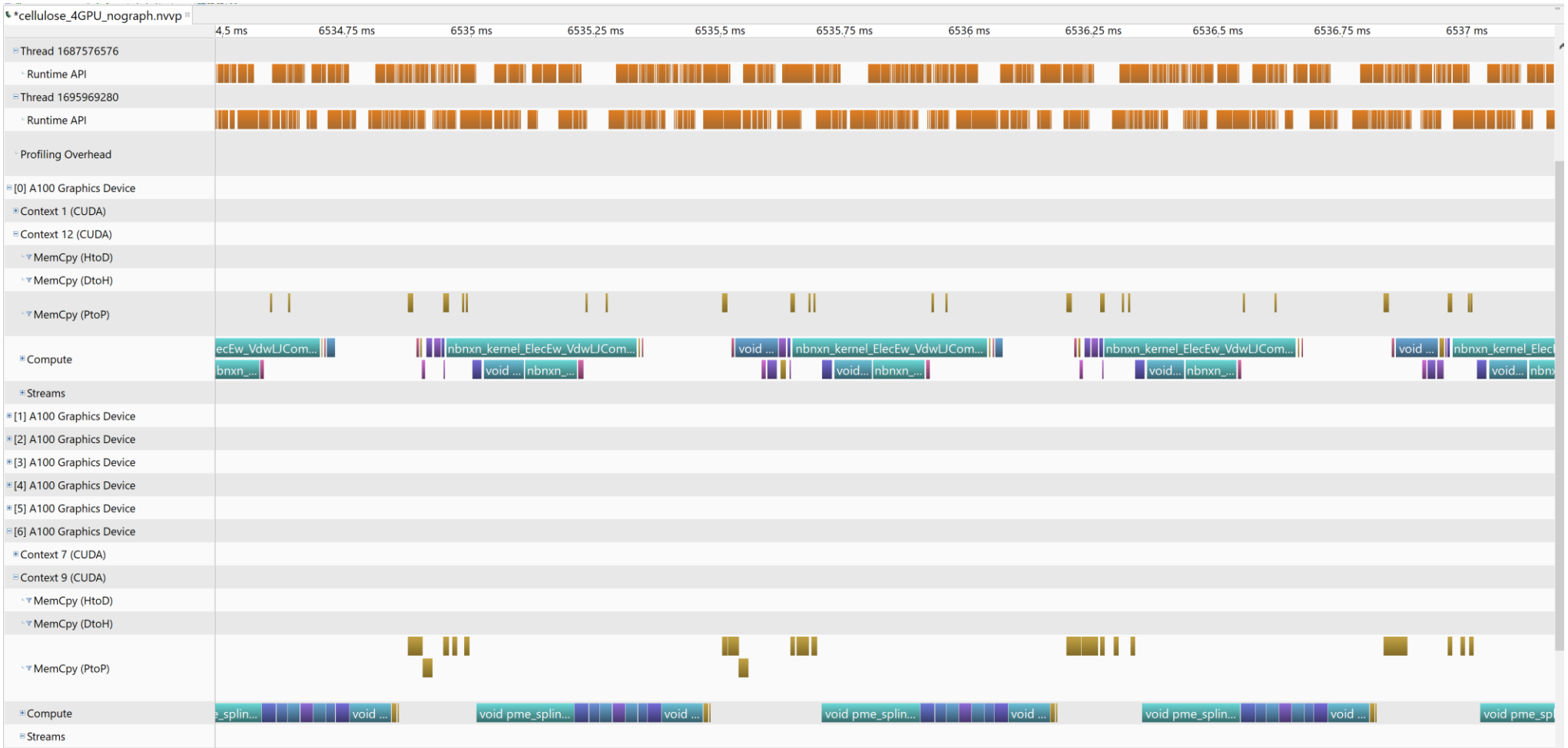
RNASE, 1 GPU, NO GRAPHS



RNASE, 1 GPU, GRAPHS



CELLULOSE, 4 GPU, NO GRAPHS



CELLULOSE, 4 GPU, GRAPHS