Porting VASP to GPU using OpenACC: exploiting the asynchronous execution model

Martijn Marsman, Stefan Maintz, Alexey Romanenko, Markus Wetzstein, and Georg Kresse

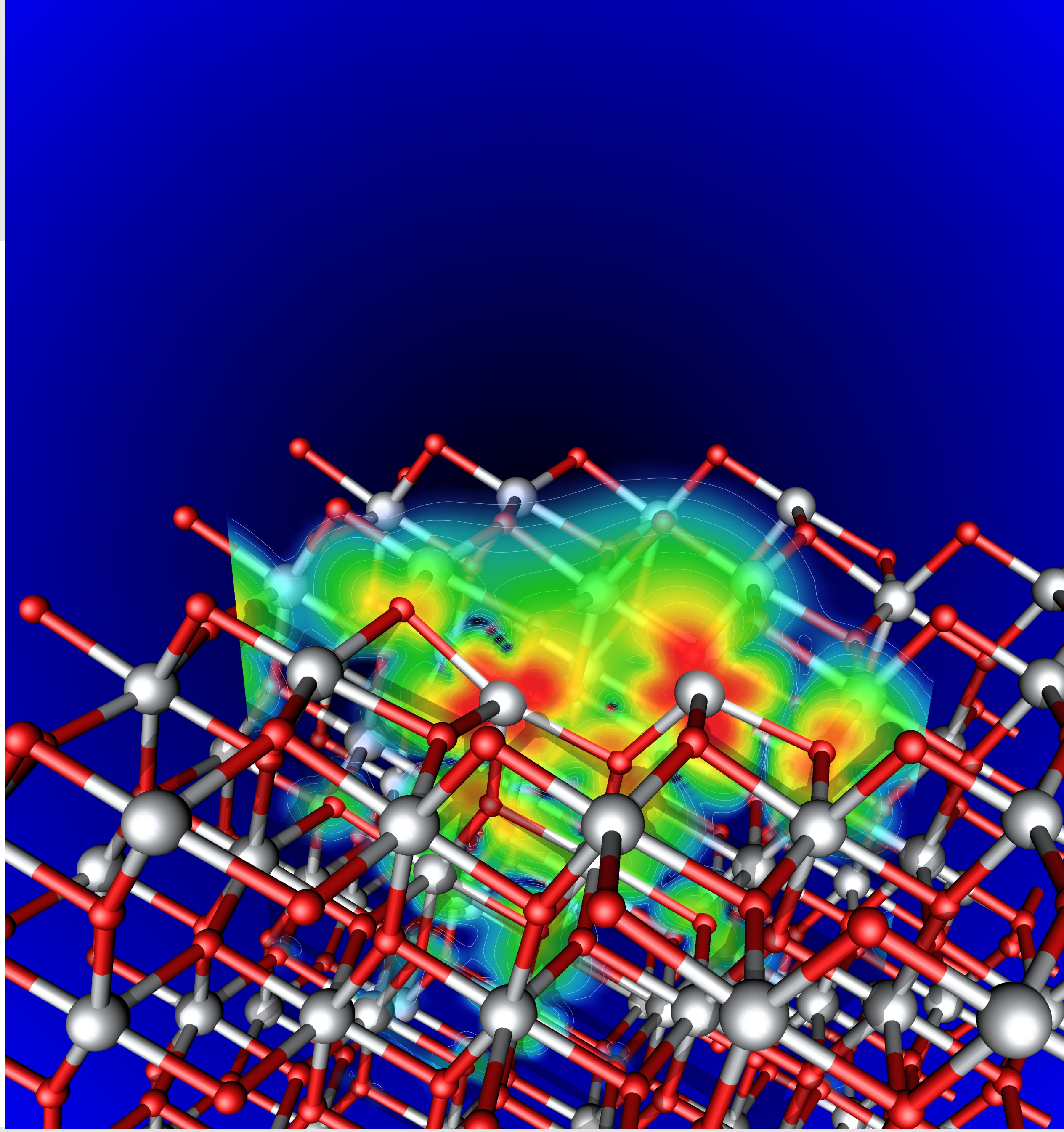
OpenACC Annual Meeting, Aug. 31st 2020

The Vienna Ab-initio Simulation Package: VASP

Electronic structure from first principles:

$$H\psi = E\psi$$

- Approximations:
  - Density Functional Theory (DFT)
  - Hartree-Fock/DFT-HF hybrid functionals
  - Random-Phase-Approximation (GW, ACFDT)
- 3500+ licensed academic and industrial groups world wide.
- 10k+ publications in 2015 (Google Scholar), and rising.
- Developed in the group of Prof. G. Kresse at the University Vienna and the VASP Software GmbH.



VASP: Computational Characteristics

VASP does:

- Lots of “smallish” FFTs: (e.g. 100$\times$100$\times$100)
- Matrix-Matrix multiplication (DGEMM and ZGEMM)
- Matrix diagonalization: $\mathcal{O}(N^3)$ ($N \approx$ #-of-electrons)
- All-2-all communication

Using (on CPU):

- fftw3d (or fftw-wrappers to mkl-ffts)
- LAPACK BLAS3 (mkl, OpenBLAS)
- scaLAPACK (or ELPA)
- MPI (OpenMPI, impi, …) [+ OpenMP]

VASP is pretty well characterized by the SPECfp2006 benchmark

VASP on GPU

- VASP has organically grown over more than 25 years
  (450k+ lines of Fortran 77/**90**/2003/2008/… code)
- Previous VASP5.4.4 release: some features were ported with CUDA C
  (DFT and hybrid functionals)
- Current VASP6.1.X releases: re-ported to GPU using OpenACC
- The OpenACC port is more complete already than the CUDA port
  (Gamma-only version, support for reciprocal space projectors, …)

Porting VASP to GPU using OpenACC

- Compiler-directive based: single source, readability, maintainability, …
- cuFFT, cuBLAS, cuSOLVER, CUDA aware MPI and NCCL
- Some dedicated kernel versions: e.g. batching FFTs, loop re-ordering
- "Manual" deep copies of derived types (nested and/or with pointer members)
- Multiple MPI ranks sharing a GPU (using MPS)
- Use the OpenACC asynchronous execution infrastructure

VASP: The main task

- Solve *N* eigenvalue equations (Kohn-Sham, Roothaan, Quasiparticle, …)

$$\Big[-\frac{1}{2}\Delta + V(\mathbf{r})\Big]\psi_i(\mathbf{r}) = \epsilon_i\psi_i(\mathbf{r}), \quad i = 1, .., N$$

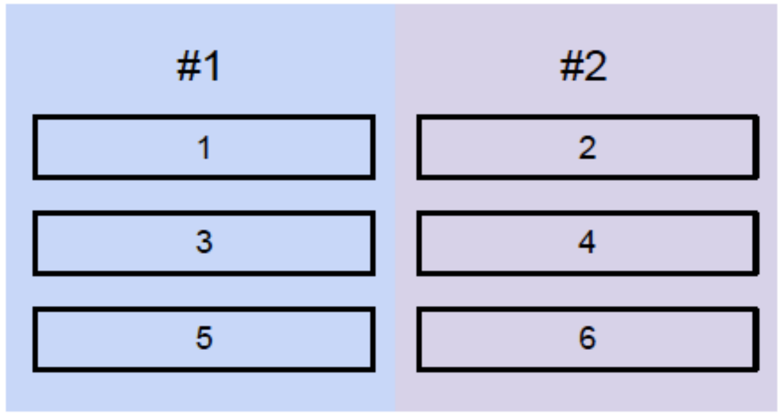
  for the “one-electron orbitals” $\psi_i(\mathbf{r})$, and “one-electron energies” ϵ_i.

- Orbitals are expanded in a plane wave basis set (*i.e.,* store the Fourier coefficients, $\psi_i(\mathbf{G})$)

- Eigenvalue equation is solved by repeated application of the Hamiltonian to the orbitals (Krylov methods)

- (Default) distribution of work and data: over “orbitals”

2 MPI-ranks



Many small kernels …

```
...
do i = 1, n
   call work1( psi(i), .. )
enddo
```

```
do j = 1, m
   c(j) = a(j) + b(j)
   ...
enddo
```

- A lot of relatively small kernels

```
call work_batch( psi(1:n), .. )
```

- Some work is ”batched” (often to maximize the performance of BLAS2/3 calls inside)

```
do i = 1, n
   call work2( psi(i), .. )
   call work3( psi(i), .. )
enddo
...
```

- Steps inside loop intend to respect cache coherency on “Xeon-like” hardware

Many small kernels …

```
...
do i = 1, n
   call work1( psi(i), .. )
enddo
```

```
!$acc parallel loop
do j = 1, m
   c(j) = a(j) + b(j)
   ...
enddo
```

- A lot of relatively small kernels

```
call work_batch_acc( psi(1:n), .. )
```

- In case of the “batched” work it often pays off to write a specific OpenACC version of the original routine

```
do i = 1, n
   call work2( psi(i), .. )
   call work3( psi(i), .. )
enddo
...
```

- Steps inside loop intend to respect cache coherency on “Xeon-like” hardware

Launch latency

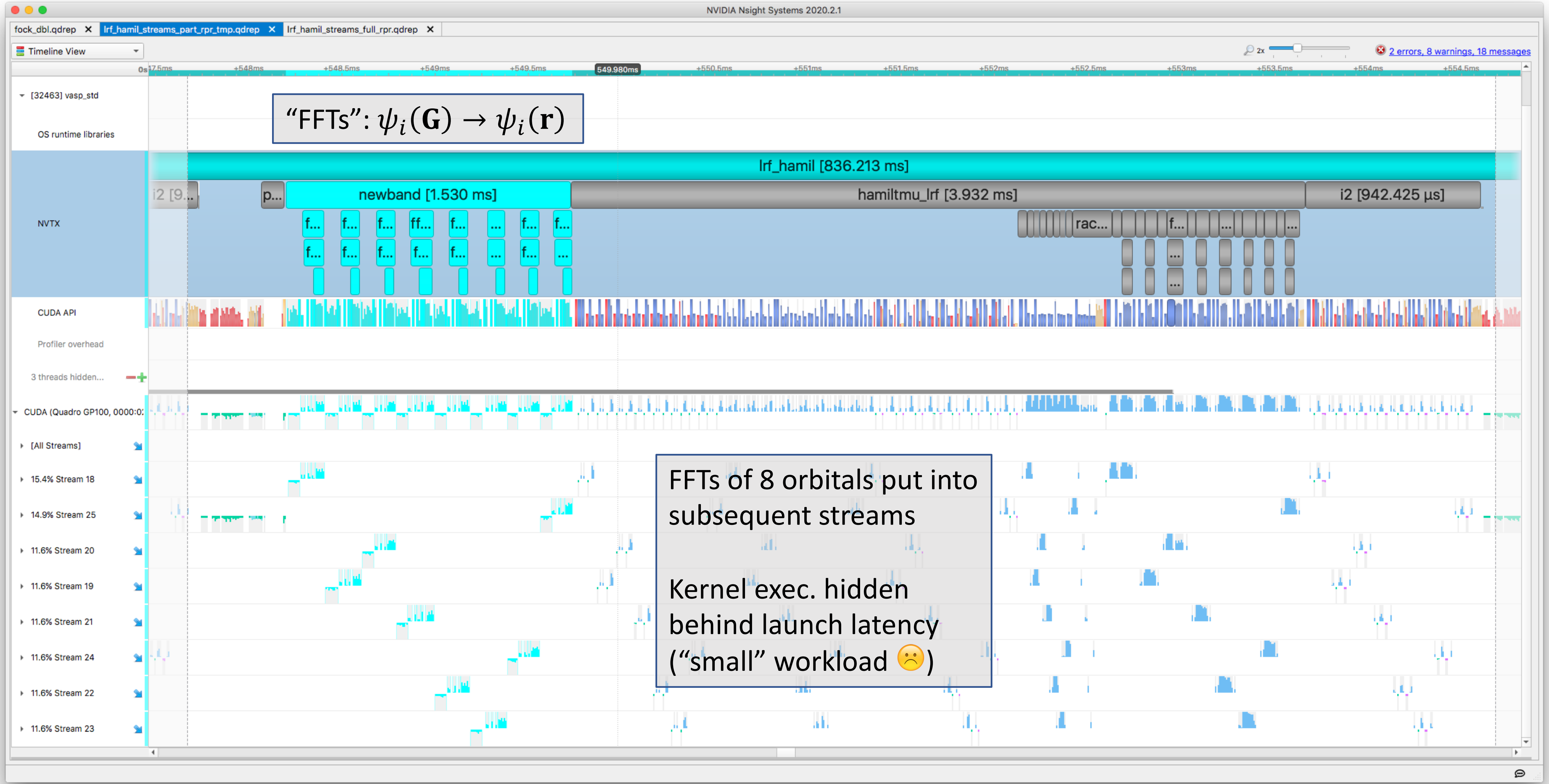
```
do i = 1, n
   queue = i
   call work1( psi(i), .. )
enddo
!$acc wait
call work_batch_acc( psi(1:n), .. )

do i = 1, n
   queue = i
   call work2( psi(i), .. )
   call work3( psi(i), .. )
enddo
```

```
!$acc parallel loop async(queue)
do j = 1, m
   c(j) = a(j) + b(j)
   ...
enddo
```

Try to hide launch latency by submitting independent kernels into subsequent asynchronous execution queues.

Since the CPU is not blocked by a running GPU kernel it can proceed to enqueue the next:

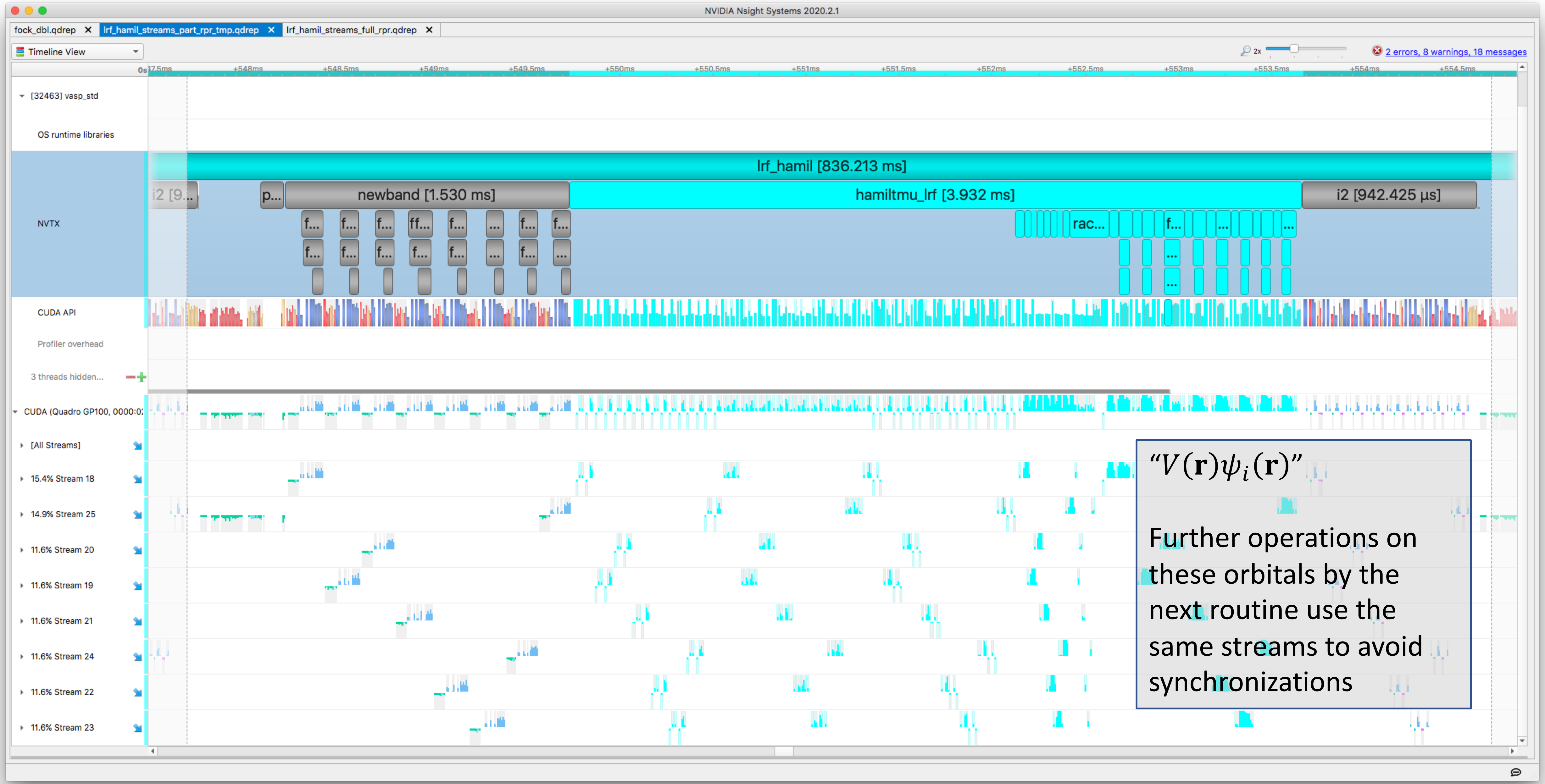
- Avoids unnecessary costly synchronizations
- Hides CPU launch latency behind kernel execution
- … or vice versa (when kernel runtime is small)



“FFTs”: $\psi_i(\mathbf{G}) \rightarrow \psi_i(\mathbf{r})$

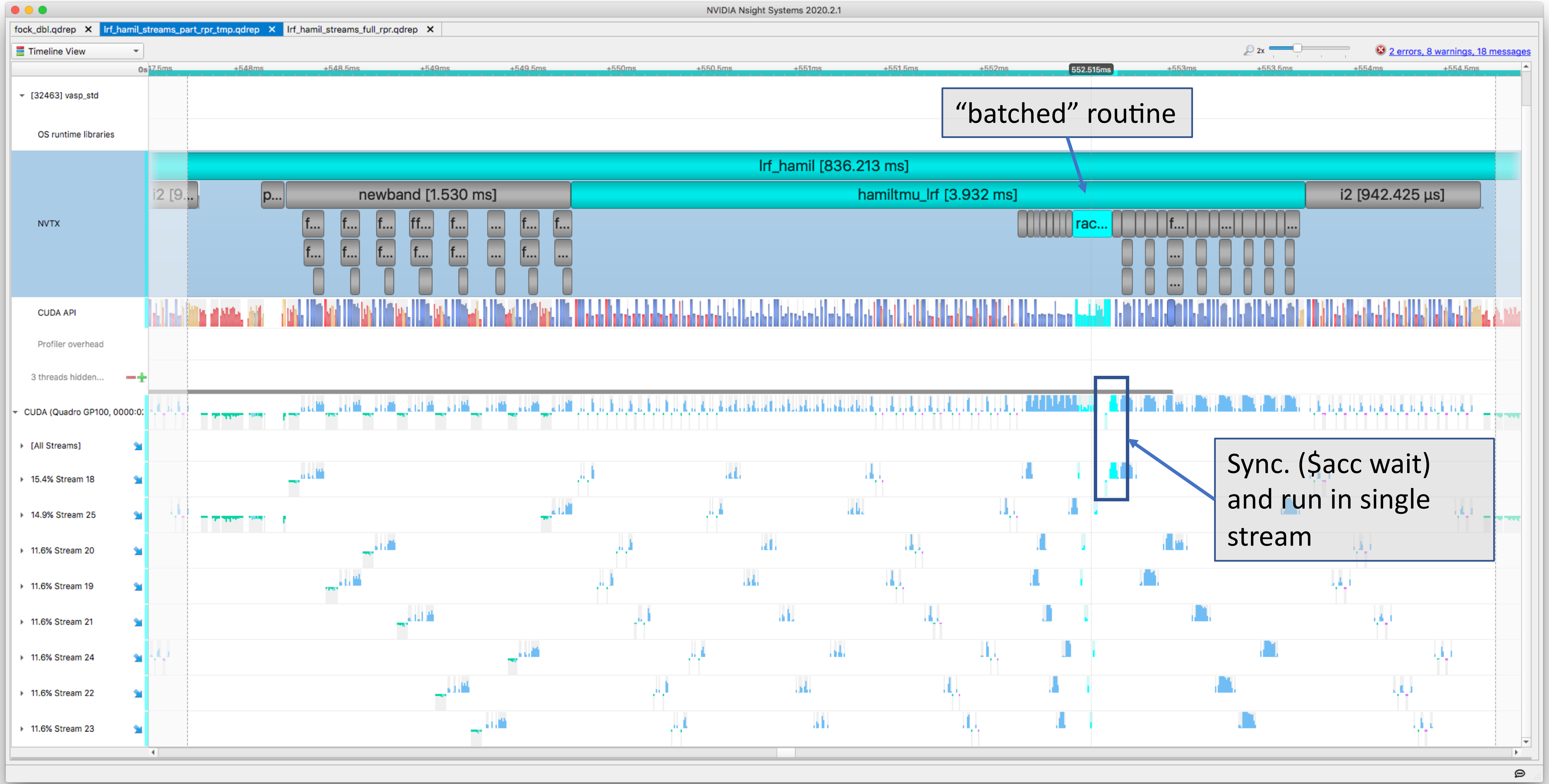
FFTs of 8 orbitals put into subsequent streams

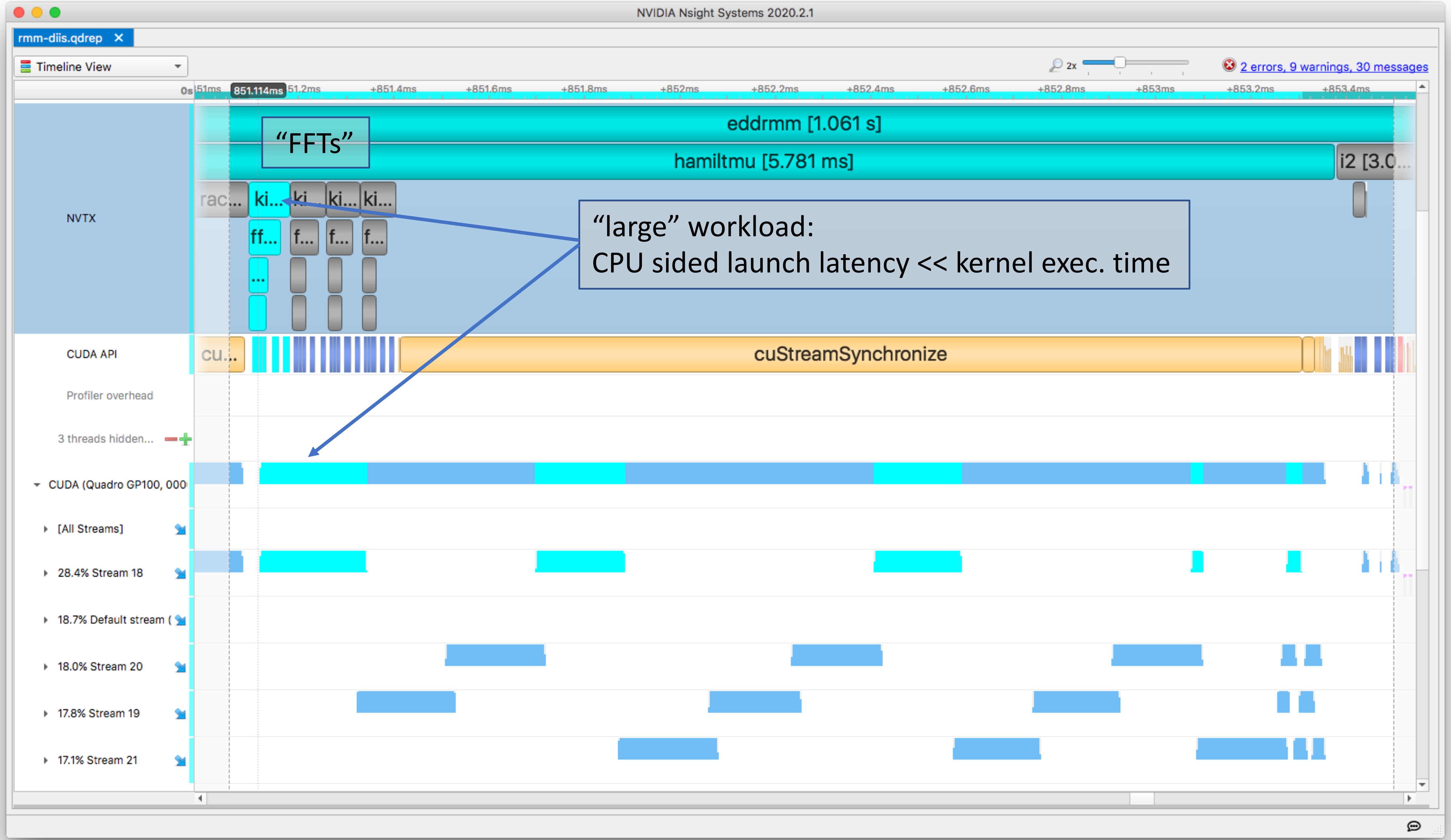
Kernel exec. hidden behind launch latency (“small” workload ☹)



"$V(\mathbf{r})\psi_i(\mathbf{r})$"

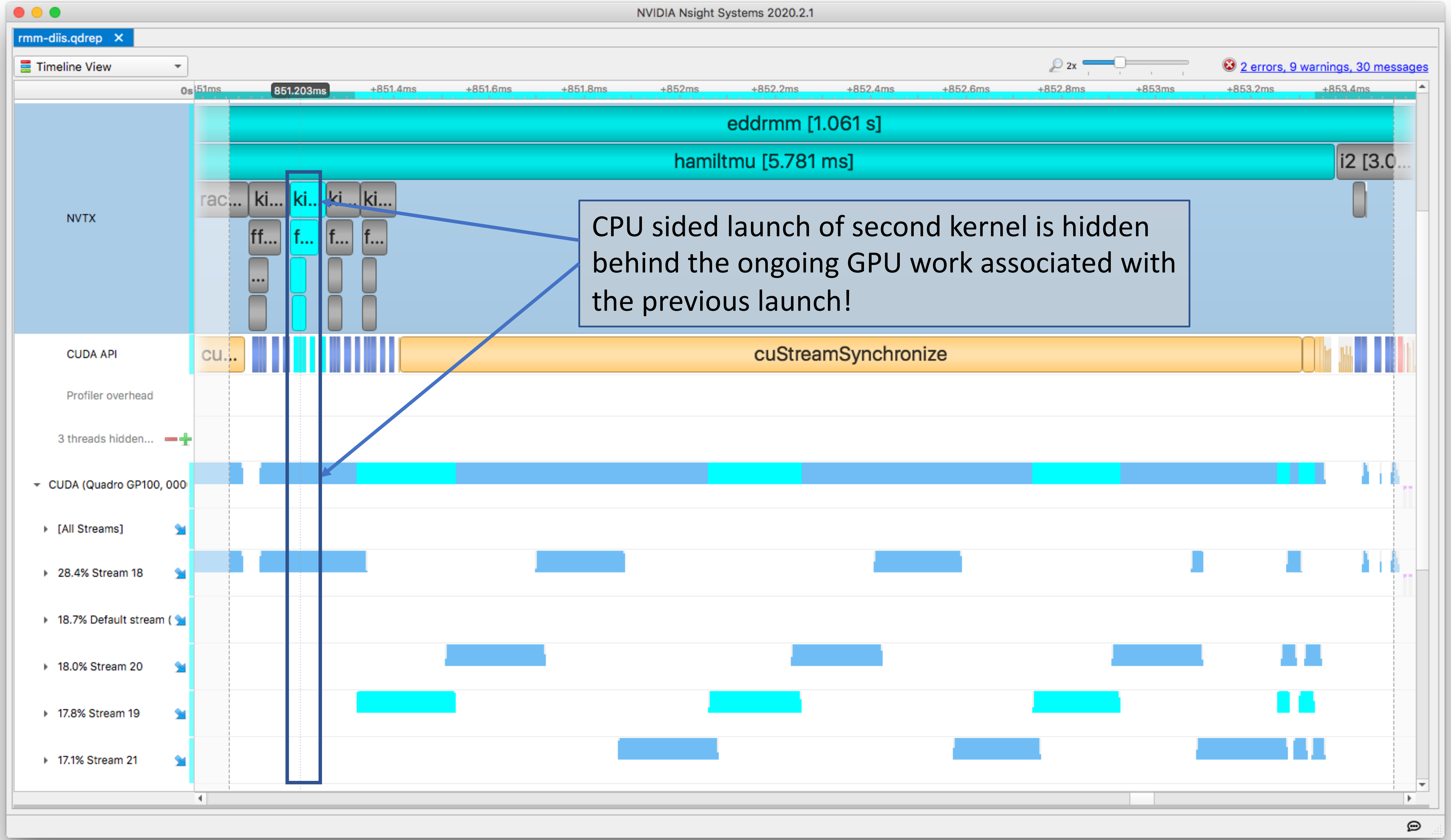
Further operations on these orbitals by the next routine use the same streams to avoid synchronizations


NVIDIA Nsight Systems 2020.2.1
fock_dbl.qdrep
lrf_hamil_streams_part_rpr_tmp.qdrep
lrf_hamil_streams_full_rpr.qdrep
Timeline View
2 errors, 8 warnings, 18 messages
552.515ms
[32463] vasp_std
OS runtime libraries
NVTX
CUDA API
Profiler overhead
3 threads hidden...
CUDA (Quadro GP100, 0000:0
[All Streams]
15.4% Stream 18
14.9% Stream 25
11.6% Stream 20
11.6% Stream 19
11.6% Stream 21
11.6% Stream 24
11.6% Stream 22
11.6% Stream 23
lrf_hamil [836.213 ms]
newband [1.530 ms]
hamiltmu_lrf [3.932 ms]
i2 [942.425 µs]
rac...
"batched" routine
Sync. ($acc wait) and run in single stream

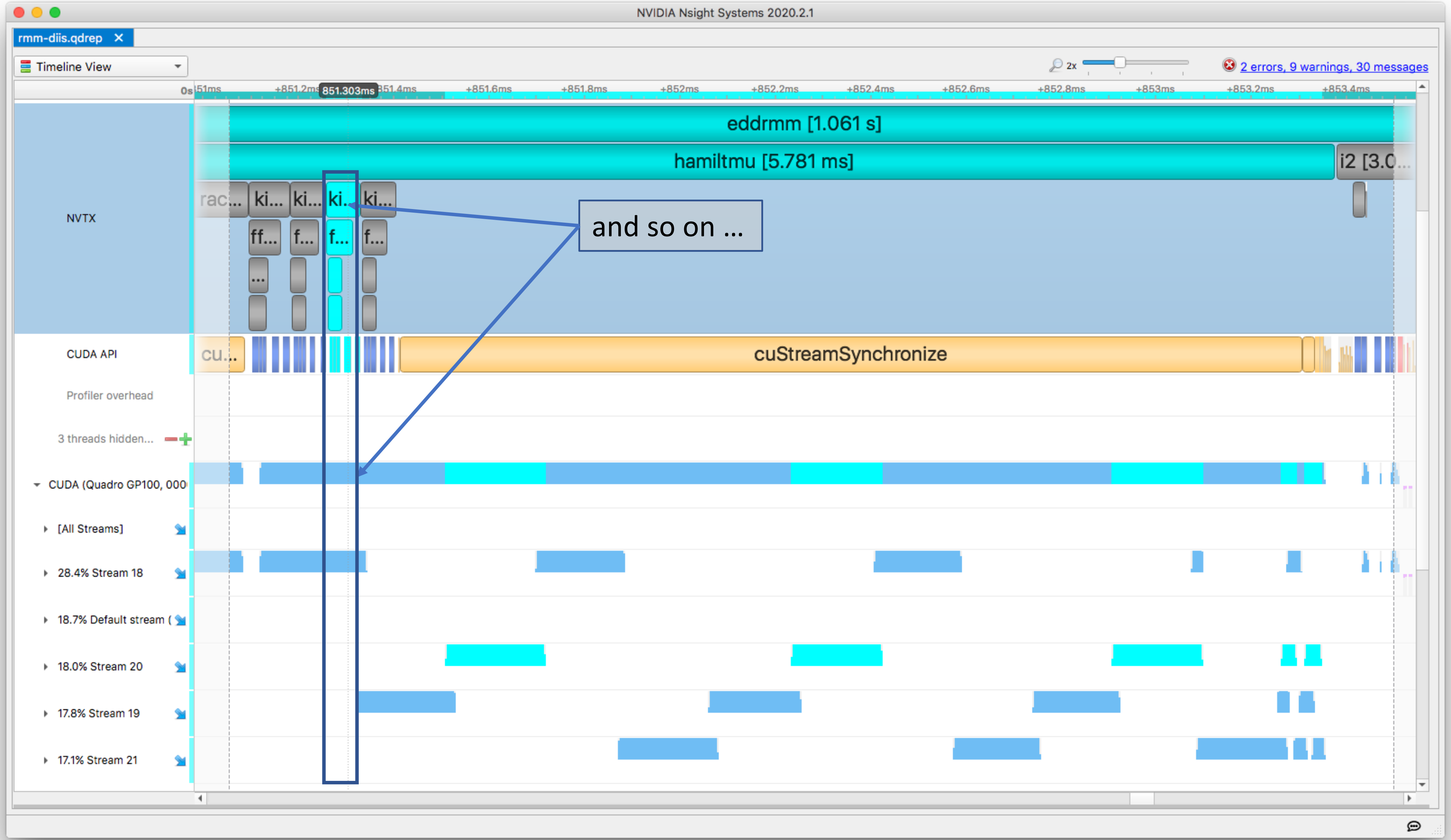


"FFTs"

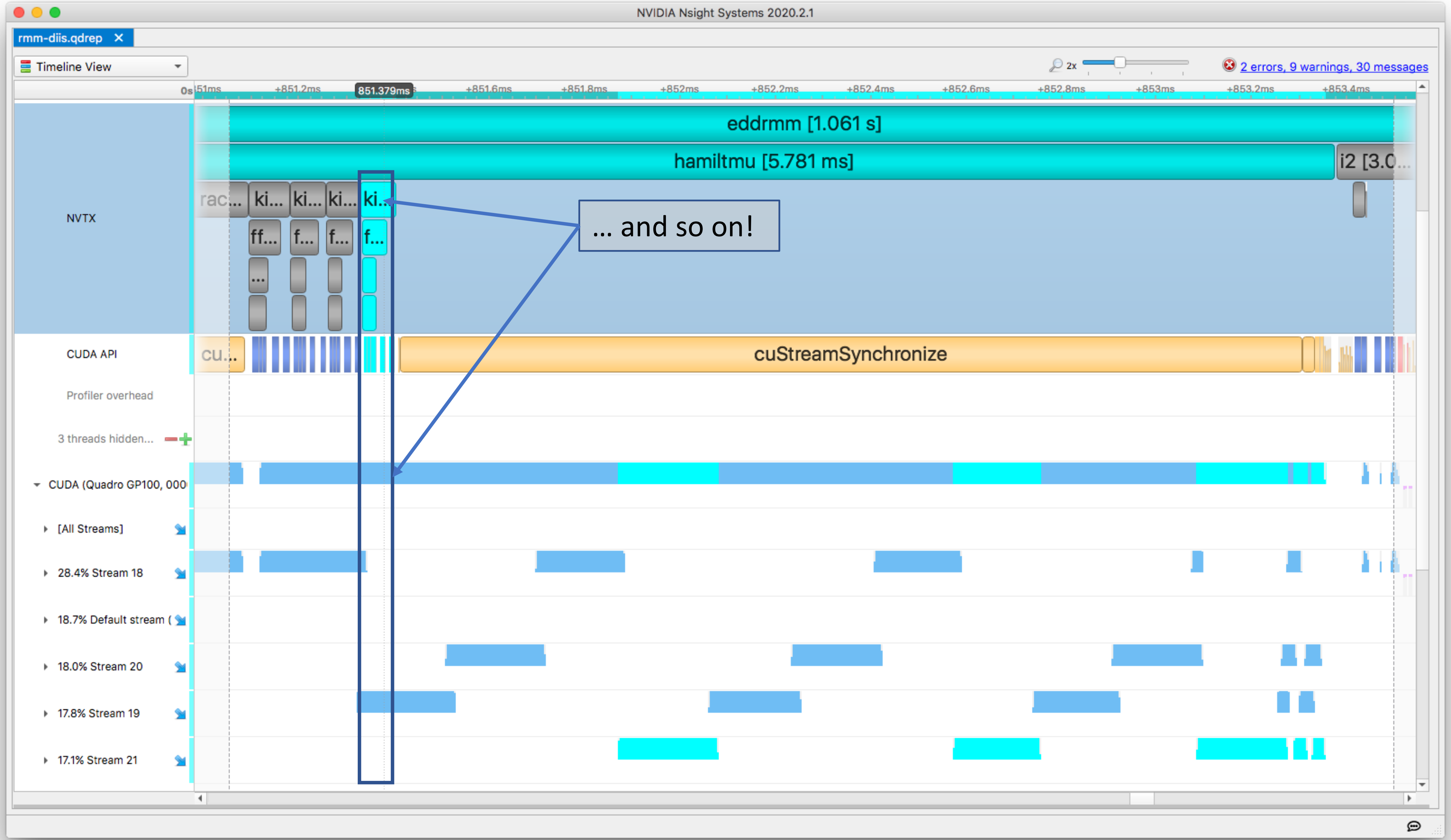
"large" workload:
CPU sided launch latency << kernel exec. time



CPU sided launch of second kernel is hidden behind the ongoing GPU work associated with the previous launch!


NVIDIA Nsight Systems 2020.2.1
rmm-diis.qdrep
Timeline View
2x
2 errors, 9 warnings, 30 messages
851.303ms
eddrmm [1.061 s]
hamiltmu [5.781 ms]
i2 [3.0...
rac...
ki...
ff...
f...
cu...
cuStreamSynchronize
NVTX
CUDA API
Profiler overhead
3 threads hidden...
CUDA (Quadro GP100, 000
[All Streams]
28.4% Stream 18
18.7% Default stream (
18.0% Stream 20
17.8% Stream 19
17.1% Stream 21

and so on ...


NVIDIA Nsight Systems 2020.2.1
rmm-diis.qdrep
Timeline View
2x
2 errors, 9 warnings, 30 messages
0s
851.379ms
+851.2ms
+851.6ms
+851.8ms
+852ms
+852.2ms
+852.4ms
+852.6ms
+852.8ms
+853ms
+853.2ms
+853.4ms
eddrmm [1.061 s]
hamiltmu [5.781 ms]
i2 [3.0...
rac...
ki...
ki...
ki...
ki...
ff...
f...
f...
f...
...
NVTX
CUDA API
cu...
cuStreamSynchronize
Profiler overhead
3 threads hidden...
CUDA (Quadro GP100, 000
[All Streams]
28.4% Stream 18
18.7% Default stream (
18.0% Stream 20
17.8% Stream 19
17.1% Stream 21

... and so on!

Launch latency

```
do i = 1, n
   queue = i
   call work1( psi(i), .. )
enddo
!$acc wait
call work_batch_acc( psi(1:n), .. )

do i = 1, n
   queue = i
   call work2( psi(i), .. )
   call work3( psi(i), .. )
enddo
```

```
!$acc parallel loop async(queue)
do j = 1, m
   c(j) = a(j) + b(j)
   ...
enddo
```

- Hide launch latency by submitting independent kernels into subsequent asynchronous execution queues:

Can yield a performance gain of 20-30% for our standard electronic minimization algorithms! (RMM-DIIS and blocked Davidson)

… often kernels hide behind launch overhead …

OpenACC + OpenMP

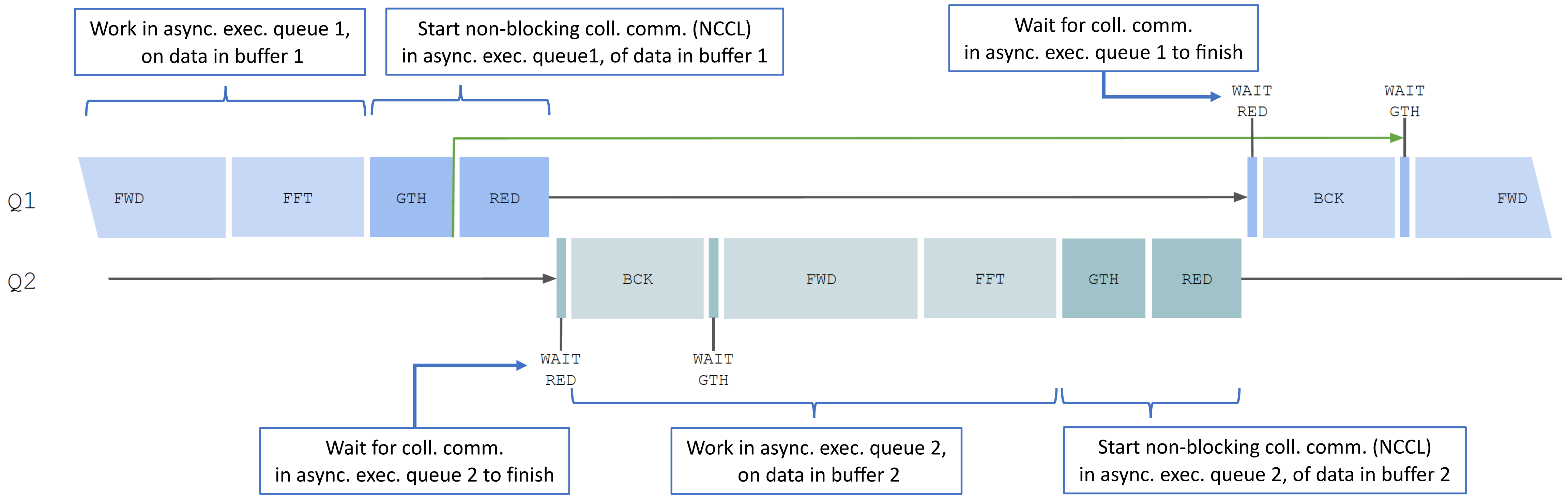
- Another idea: hide the launch latency by means of OpenMP (“concurrent” kernel launches)

```
!$omp parallel do
do i = 1, n
   queue = i
   call work1( psi(i), .. )
enddo
!$omp end parallel do
```

```
!$acc parallel loop async(queue)
do j = 1, m
   c(j) = a(j) + b(j)
   ...
enddo
```

Unfortunately this does not work (yet): the current CUDA drivers serialize the kernel launches inside the OpenMP parallelized loop ...

Hiding collective communication using NCCL

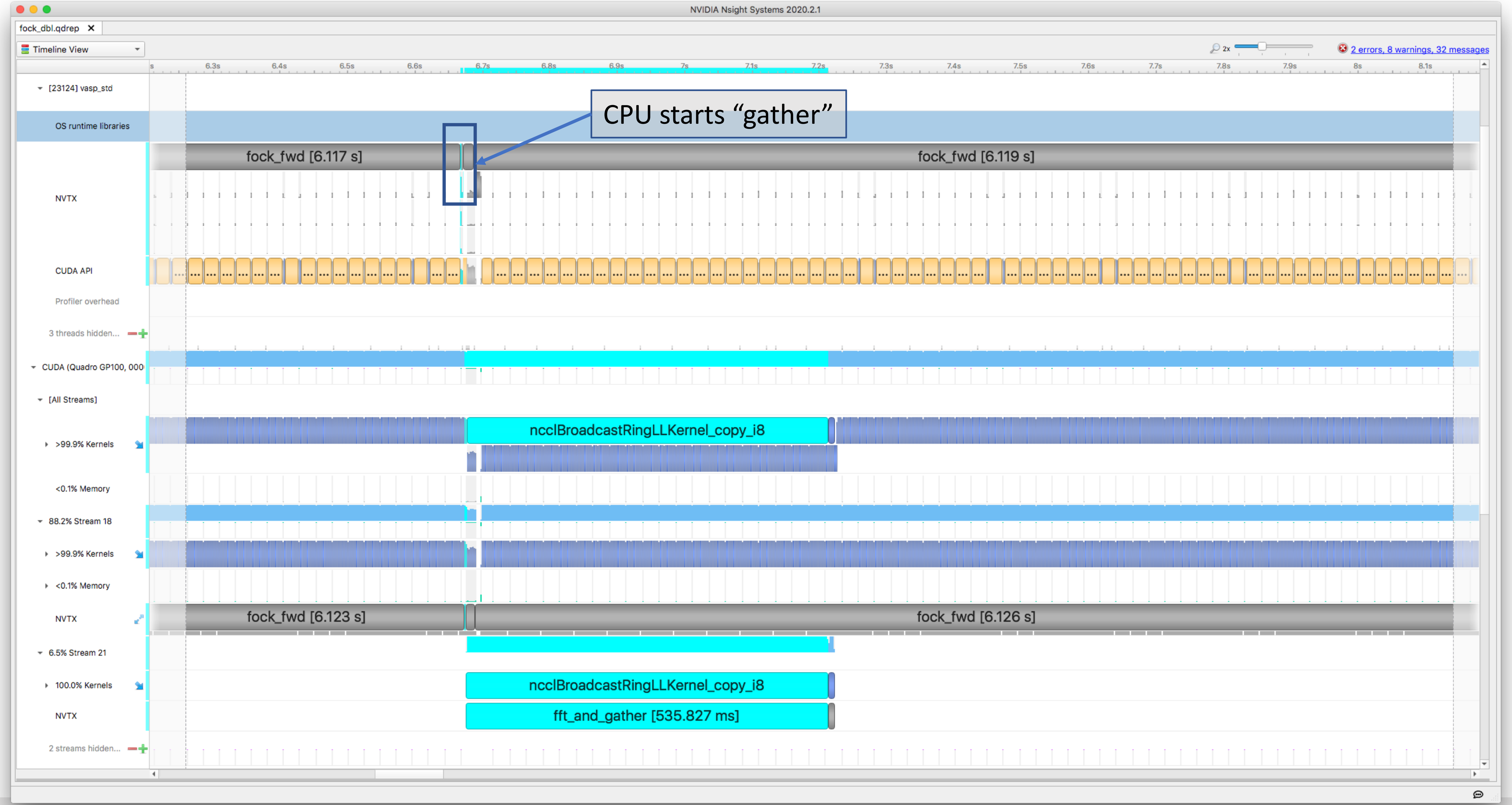

Work in async. exec. queue 1, on data in buffer 1
Start non-blocking coll. comm. (NCCL) in async. exec. queue1, of data in buffer 1
Wait for coll. comm. in async. exec. queue 1 to finish
WAIT RED
WAIT GTH
Q1
FWD
FFT
GTH
RED
BCK
FWD
Q2
BCK
FWD
FFT
GTH
RED
WAIT RED
WAIT GTH
Wait for coll. comm. in async. exec. queue 2 to finish
Work in async. exec. queue 2, on data in buffer 2
Start non-blocking coll. comm. (NCCL) in async. exec. queue 2, of data in buffer 2

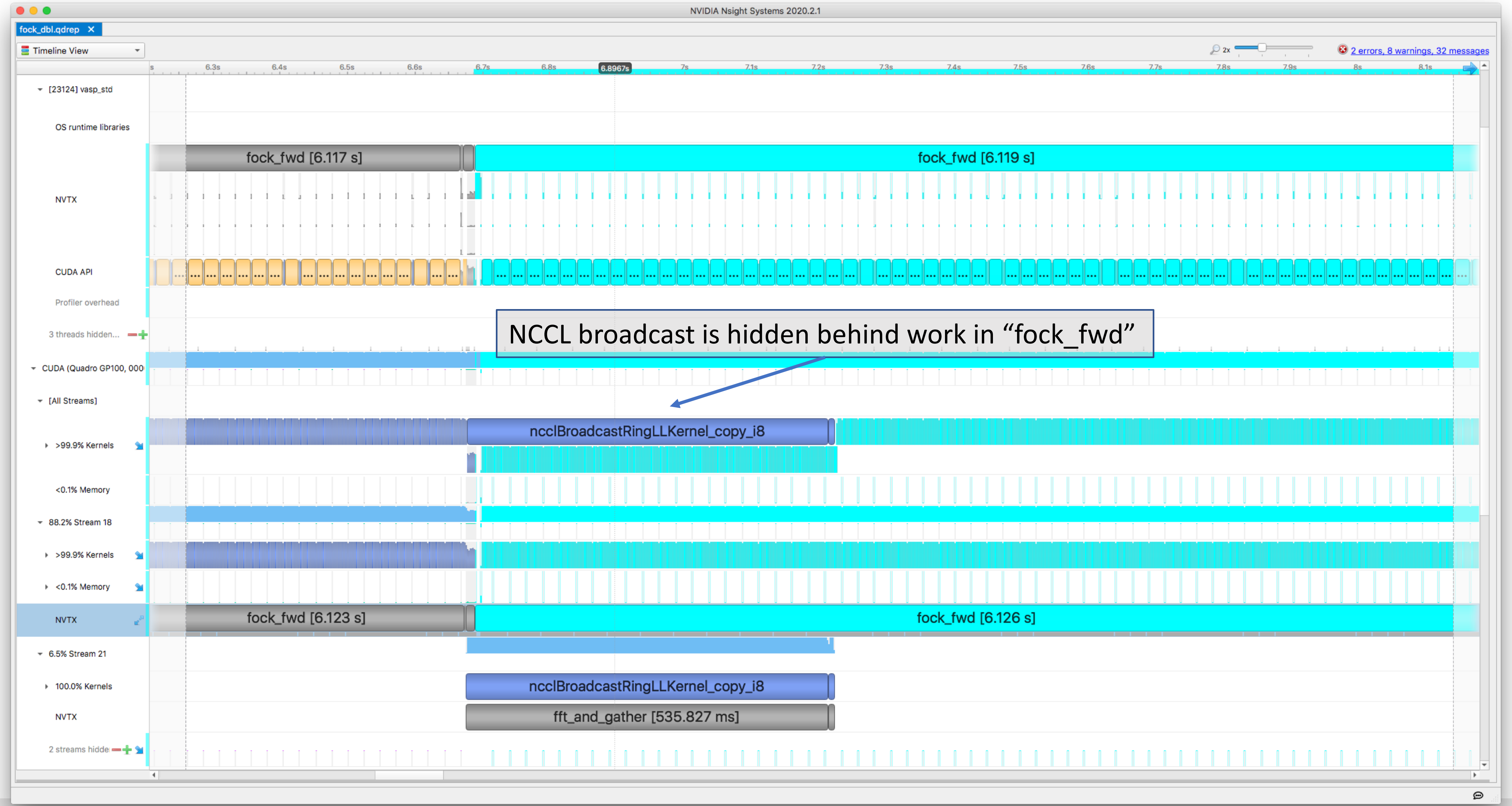
Hiding collective communication using NCCL

```
gather:


   call MPI_ibcast( ...,
                    irank, MPI_Comm,
                    buf%request,
                    ...
                  )


wait-gather:

   call MPI_wait( buf%request, ... )
```

```
gather:

   ncclRes = ncclGroupStart()

   ncclRes = ncclBcast( ...,
                        irank, NCCL_Comm,
                        acc_get_cuda_stream(buf%queue)
                      )

   ncclRes = ncclGroupEnd()


wait-gather:

   !$acc wait(buf%queue)
```

Straightforward: NCCL calls pretty much “drop-in” for non-blocking MPI calls!


NVIDIA Nsight Systems 2020.2.1
fock_dbl.qdrep
Timeline View
2 errors, 8 warnings, 32 messages
[23124] vasp_std
OS runtime libraries
NVTX
CUDA API
Profiler overhead
3 threads hidden...
CUDA (Quadro GP100, 000
[All Streams]
>99.9% Kernels
<0.1% Memory
88.2% Stream 18
>99.9% Kernels
<0.1% Memory
NVTX
6.5% Stream 21
100.0% Kernels
NVTX
2 streams hidden...
fock_fwd [6.117 s]
fock_fwd [6.119 s]
ncclBroadcastRingLLKernel_copy_i8
fock_fwd [6.123 s]
fock_fwd [6.126 s]
ncclBroadcastRingLLKernel_copy_i8
fft_and_gather [535.827 ms]
CPU starts “gather”

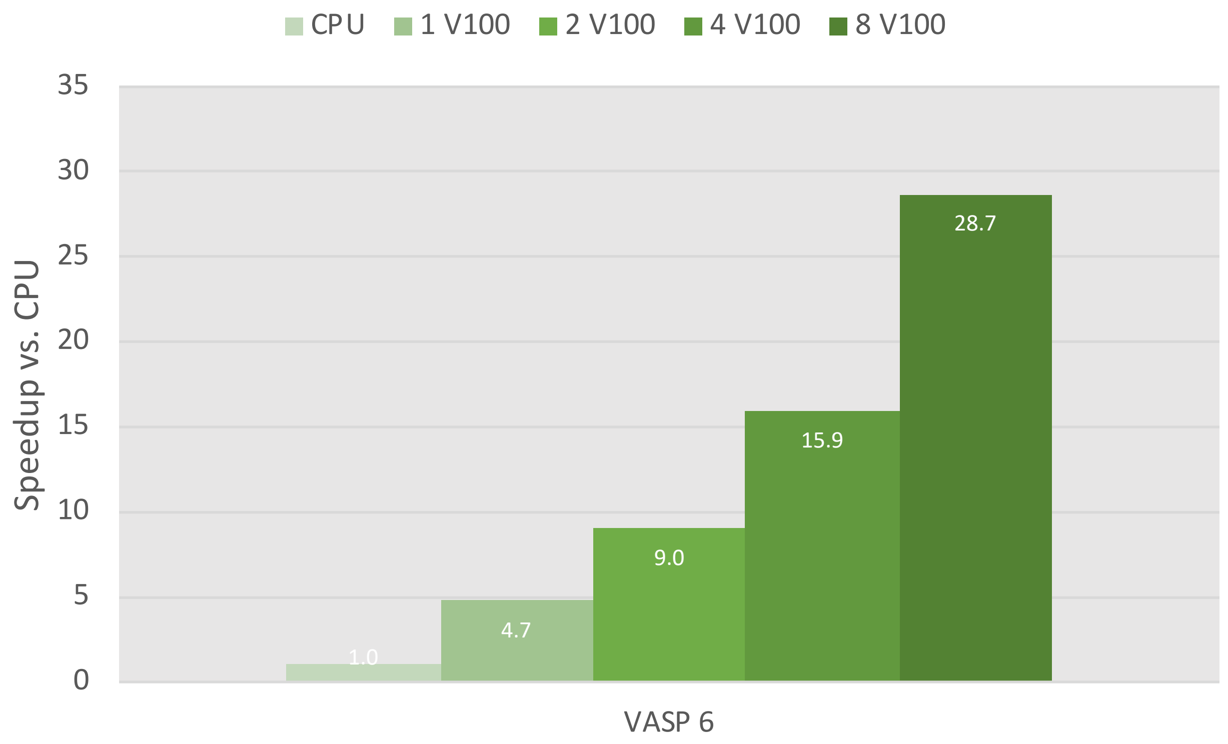


NCCL broadcast is hidden behind work in “fock_fwd”

VASP on GPU benchmarks

"Si256_VJT_HSE06"

- Vacancy in Si ($\Omega \cong 5200$ Å^3)
- 255 Si atoms (1020 e−)
- DFT/HF-hybrid functional
- Conjugate gradient
- Batched FFTs
- Explicit overlay of computation and communication using non-blocking collectives (NCCL)

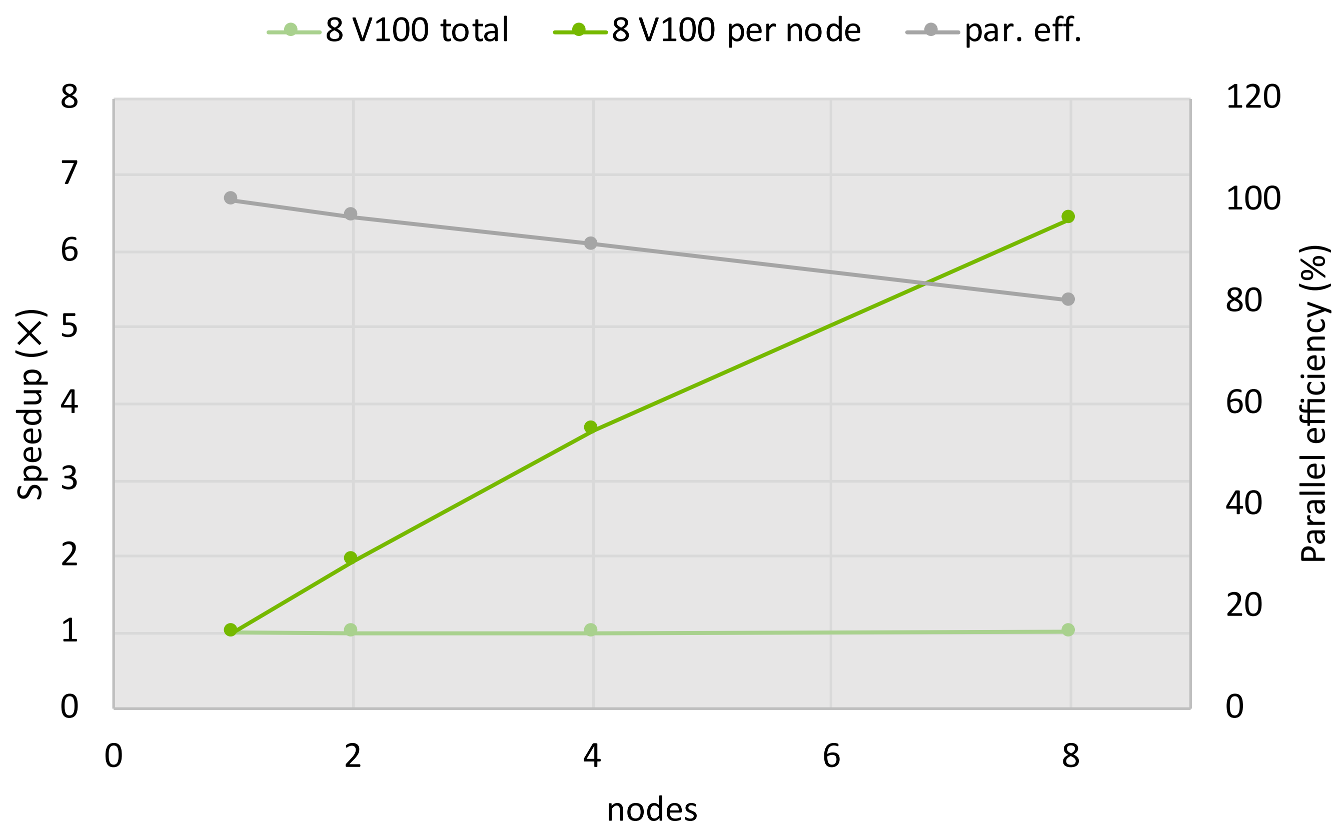


- CPU: 2✕ E5-2698 v4 @ 2.20 GHz: 40 physical cores

VASP on GPU: multi-node behaviour

"Si256_VJT_HSE06"

- Vacancy in Si ($\Omega \cong 5200$ Å^3)
- 255 Si atoms (1020 e−)
- DFT/HF-hybrid functional
- Conjugate gradient
- Batched FFTs
- Explicit overlay of computation and communication using non-blocking collectives (NCCL)



- Node: DGX1 feat. 8✕ NVIDIA V100-SXM2-16GB
- Interconnect: Mellanox ConnectX-6 Infiniband with HDR200 cards

Conclusions

“Many small kernels”

- Submit (identical) independent kernels in subsequent async. exec. queues:
  - + Hide CPU-sided launch effort behind kernel execution
  - – … or vice versa (small workloads)
  - ? OpenMP + OpenACC …
- Batching small kernels in specific “OpenACC” routines would be better performance-wise, but more invasive

“Hiding collective communication”

- Use NCCL as a “drop-in” for non-blocking MPI collectives (bcast, reduce, …) and overlay comm. and comp. by putting them in their own async. exec. queues:
  - + Works really well (on- and across-node)
  - – Additional library dependency
  - – Ranks that communicate through NCCL may not share a GPU … (issue for small workloads)
  - ? NVIDIA hardware is covered, what about other GPUs …

THE END

Special thanks to Stefan Maintz, Alexey Romanenko, Andreas Hehn, and Markus Wetzstein from NVIDIA and PGI!

And to Ani Anciaux-Sedrakian and Thomas Guignon at IFPEN!

And to you for listening!