QUANTUM THEORY CENTER
SDU

OPENQCD ON GPU

Antonio Rago
Quantum Theory Center, University of Southern Denmark

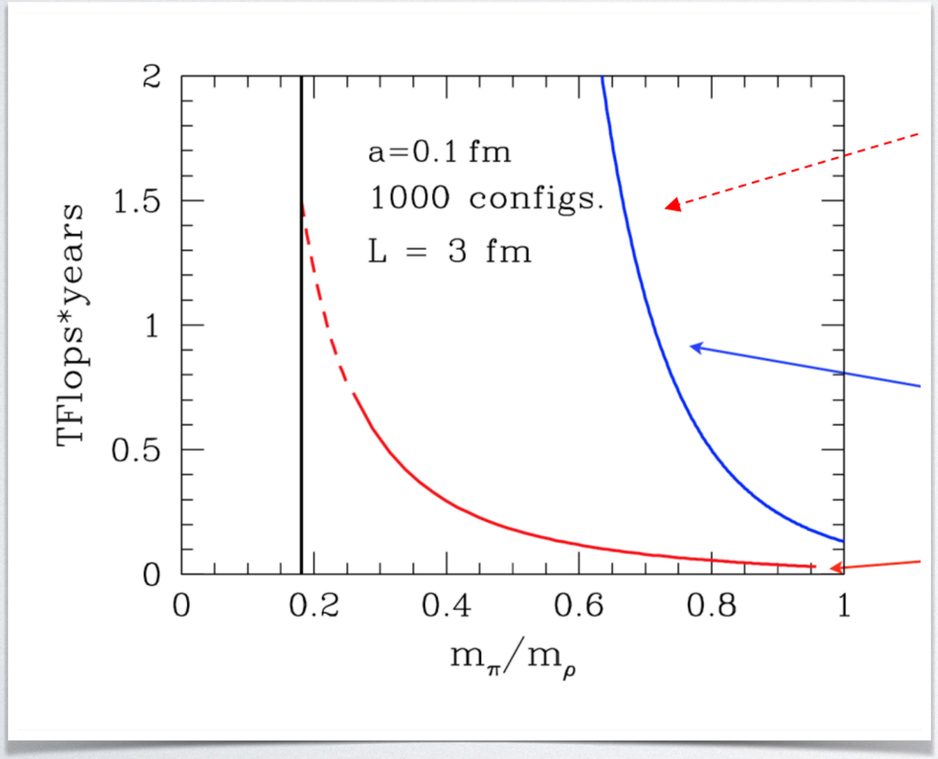
LATTICE 2024
LIVERPOOL

work done in collaboration with **Stefan Schaefer**

Why openQCD?

openQCD is a simulation suite for lattice QCD, featuring an efficient implementation of the $\mathcal{O}(a)$ improved Wilson-Dirac fermion operator.

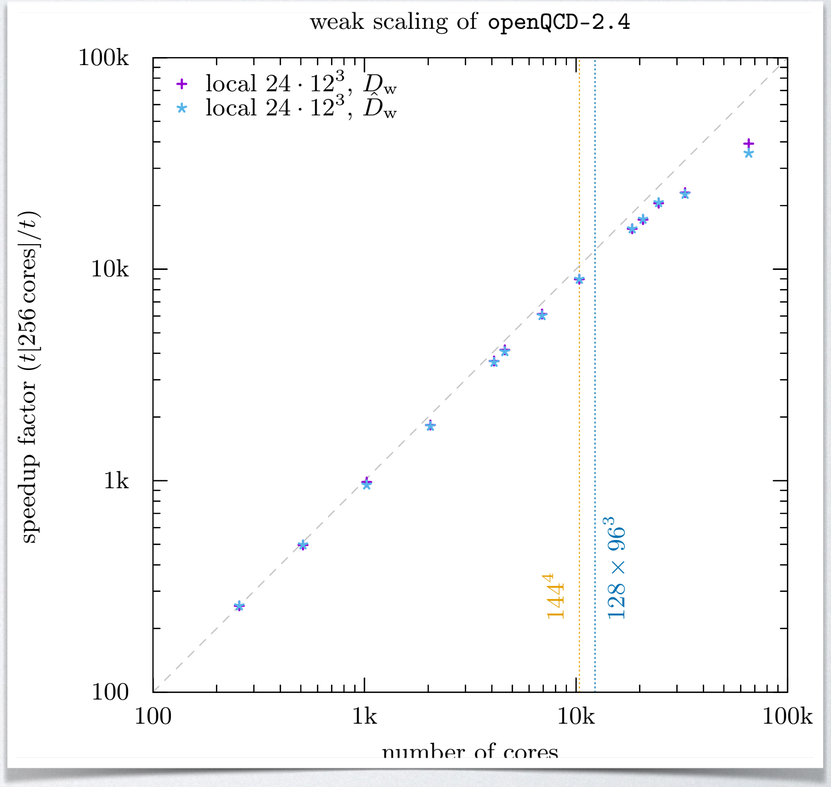
Pivotal to the performance properties of the code is the locally deflated solver.



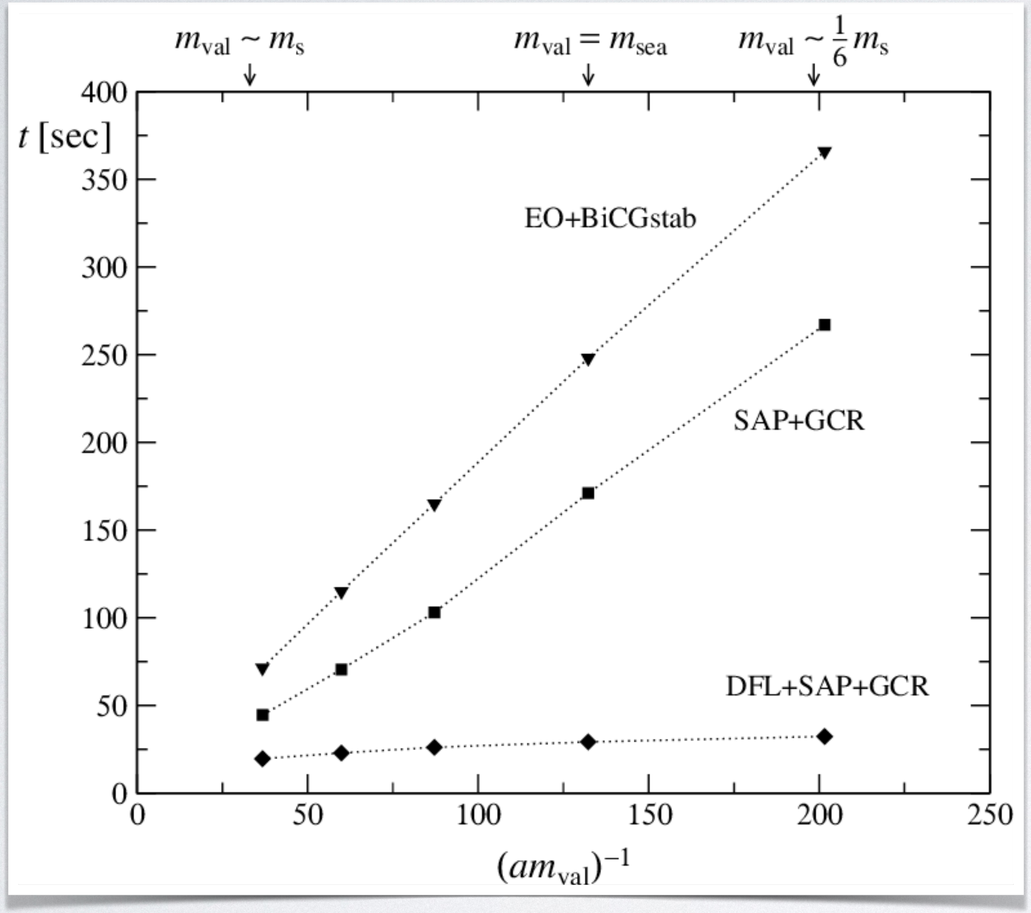
The Berlin Wall (2001)

Ukawa (2001)

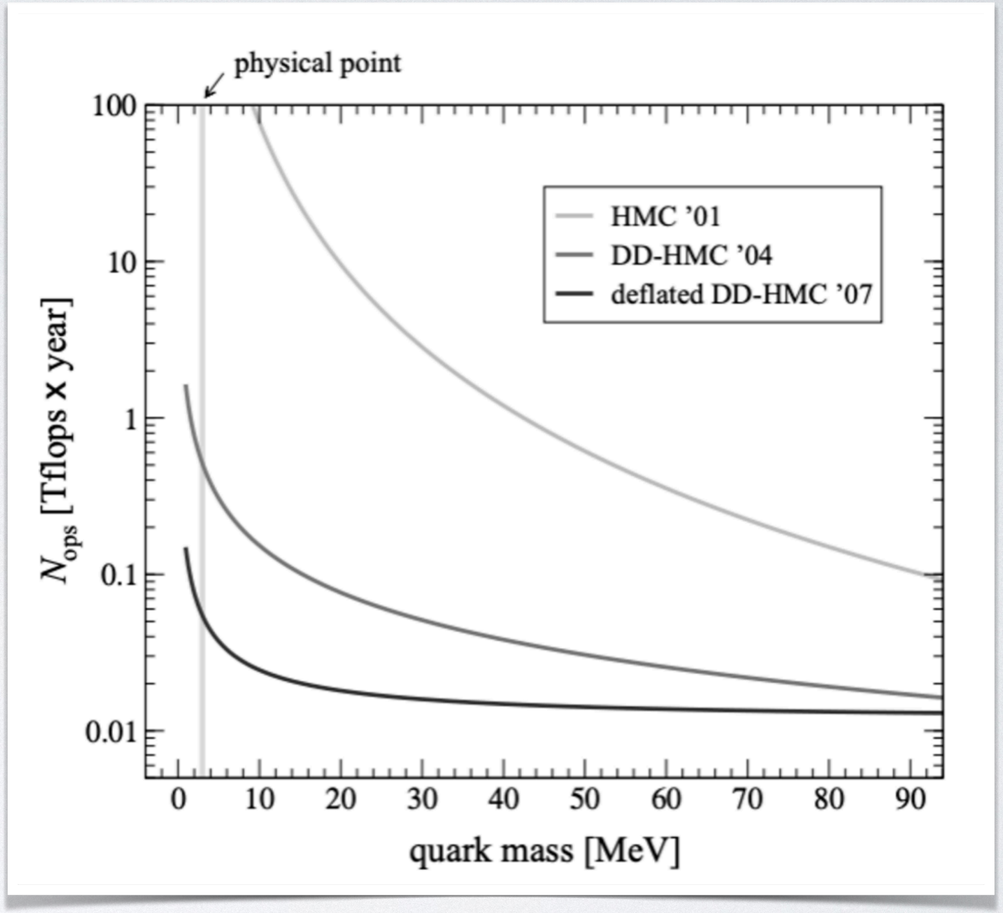
Luscher (2004)



Why openQCD?



[JHEP0712:011,2007]

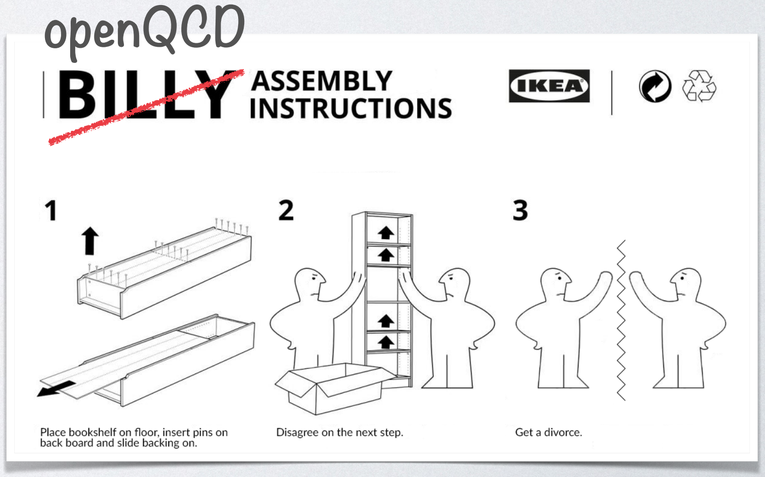


Domain Decomposition in HMC (2003-2004)

Deflated DD HMC in 2007

Develop C++/CUDA modules to replicate openQCD functions on the GPU

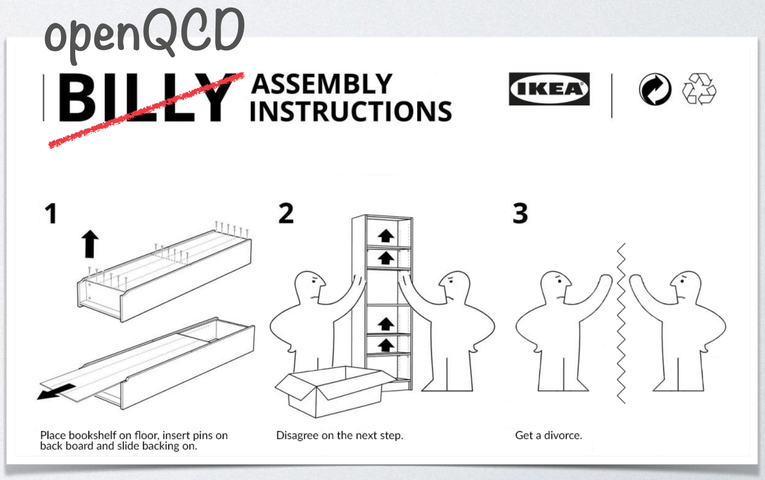
- Module Development:
  - Create a comprehensive set of C++/CUDA modules mapping the CPU (relevant) functions to GPU.
    Ensure these modules accurately reproduce the functionality of openQCD on the GPU.
- Data Structure Translation:
  - Maintain consistent and coherent translation between GPU and CPU data structures.
    Ensure seamless data flow and compatibility (of fields).
- Function Naming Conventions:
  - Adopt a clear and consistent naming convention.
    Example: dfl_subspace in CPU becomes dfl_subspace_gpu in GPU.
- GPU only:
  - Implement all the computational part of inversion cycle on the GPU.
    Future Goals: Extend this to include HMC and SMD processes.



A pragmatic approach

- Redesigned Data Structures:
  Allow full memory coalesced access in GPU data structures and memory access.
- Code Readability:
  Prioritised code readability and usability.
  Avoided optimising non-critical routines (e.g., not all data structures have full memory coalescence).
- CUDA Grid Mapping:
  Mapped CUDA grid of blocks to geometrical blocks for efficient computation.
  - Block Reductions:
    Implemented block reductions using CUB library for optimised performance.
- Database Duplication:
  Duplicated the openQCD database for GPU use.
  field status is propagated between CPU and GPU databases.

- Quadruple Precision Reduction:
  Implemented quadruple precision reduction to ensure computational accuracy.
- Random Number Generation:
  Integrated directly on the GPU robust random number generation for simulations.
- Validation:
  Guaranteed at field level for each openQCD test



The test ground

Hardware Configuration of the JUWELS Booster Module

- **936 compute nodes**
  - 2× AMD EPYC Rome 7402 CPU, 2× 24 cores, 2.8 GHz
  - Simultaneous Multithreading
  - 512 GB DDR4, 3200 MHz
  - 4× NVIDIA A100 GPU, 40 GB HBM2e
  - 4× InfiniBand HDR (Connect-X6)
  - Diskless

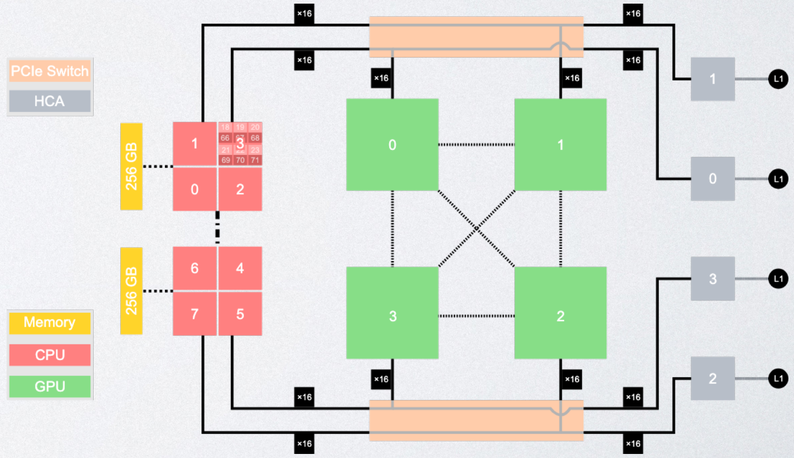
	GPU0	GPU1	GPU2	GPU3	NIC0	NIC1	NIC2	NIC3	CPU Affinity	NUMA Affinity	GPU NUMA ID
GPU0	X	NV4	NV4	NV4	PIX	SYS	SYS	SYS	18-23,66-71	3	N/A
GPU1	NV4	X	NV4	NV4	SYS	PIX	SYS	SYS	6-11,54-59	1	N/A
GPU2	NV4	NV4	X	NV4	SYS	SYS	PIX	SYS	42-47,90-95	7	N/A
GPU3	NV4	NV4	NV4	X	SYS	SYS	SYS	PIX	30-35,78-83	5	N/A
NIC0	PIX	SYS	SYS	SYS	X	SYS	SYS	SYS			
NIC1	SYS	PIX	SYS	SYS	SYS	X	SYS	SYS			
NIC2	SYS	SYS	PIX	SYS	SYS	SYS	X	SYS			
NIC3	SYS	SYS	SYS	PIX	SYS	SYS	SYS	X			

Legend:

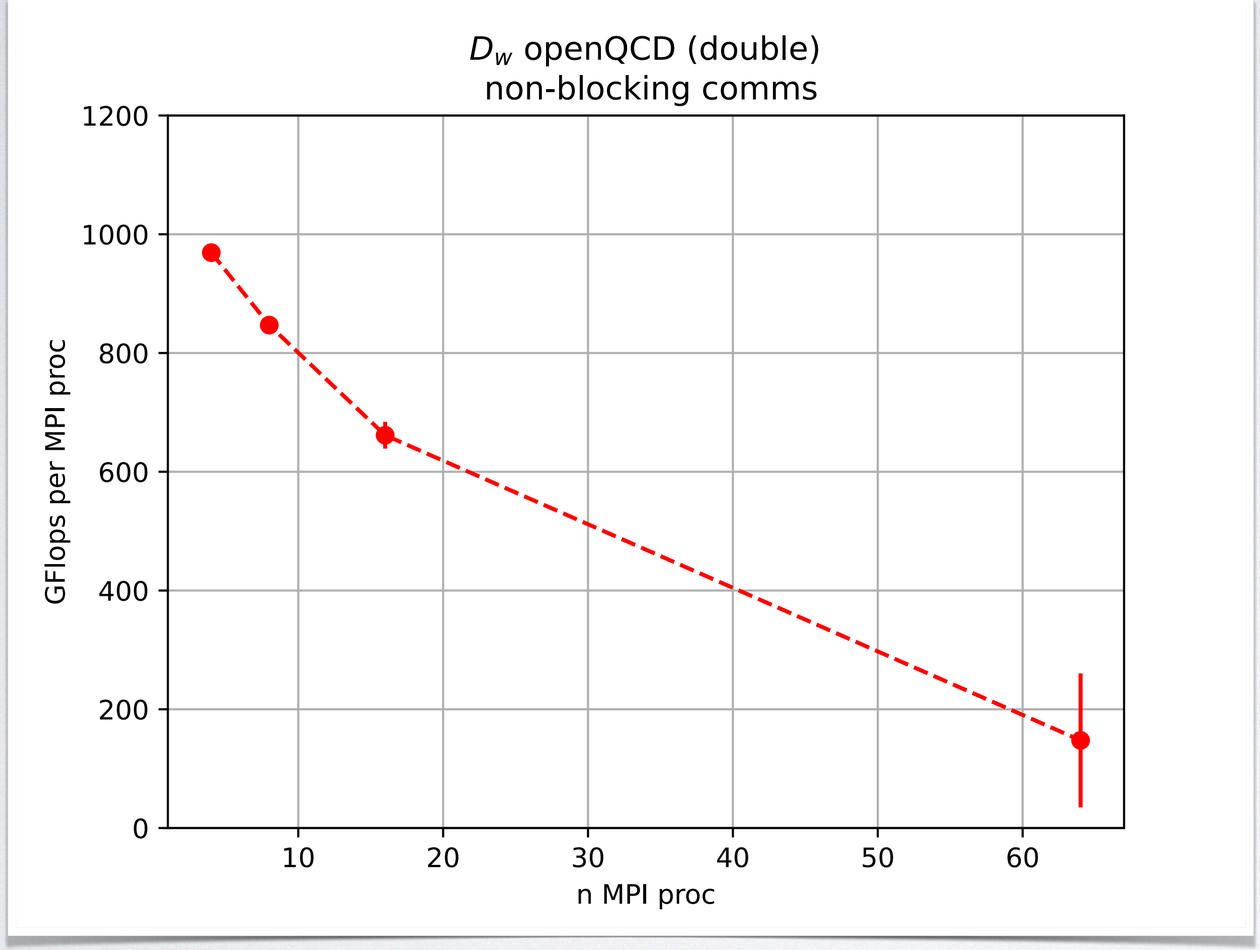
  X   = Self
  SYS = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX = Connection traversing at most a single PCIe bridge
  NV# = Connection traversing a bonded set of # NVLinks

NIC Legend:

  NIC0: mlx5_0
  NIC1: mlx5_1
  NIC2: mlx5_2
  NIC3: mlx5_3

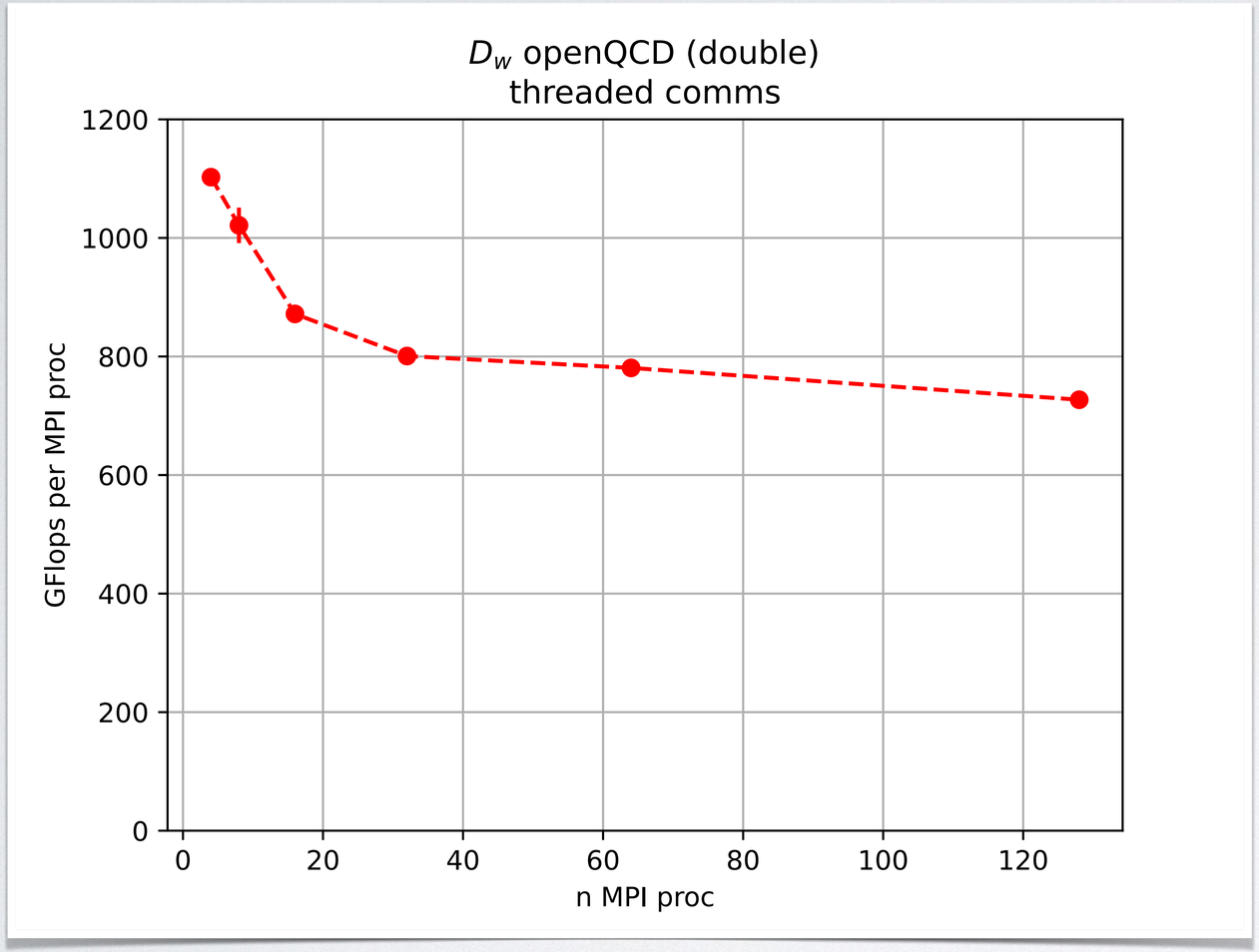


A first test: weak scaling D_W for a local 24^4

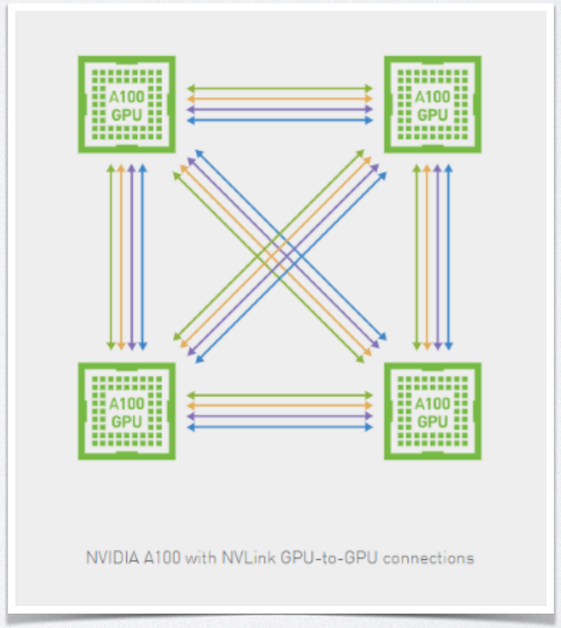


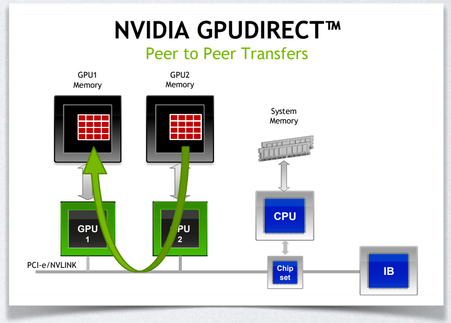
- No un-coalesced access
- EO memory arrangement for every field
- We opted for a index computation "in flight", no lookup tables.
  Block sizes defined a compile time
- Can achieve 1Tflops per A100 card

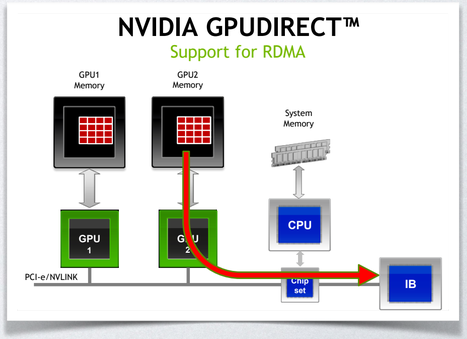
A first test: weak scaling D_W for a local 24^4



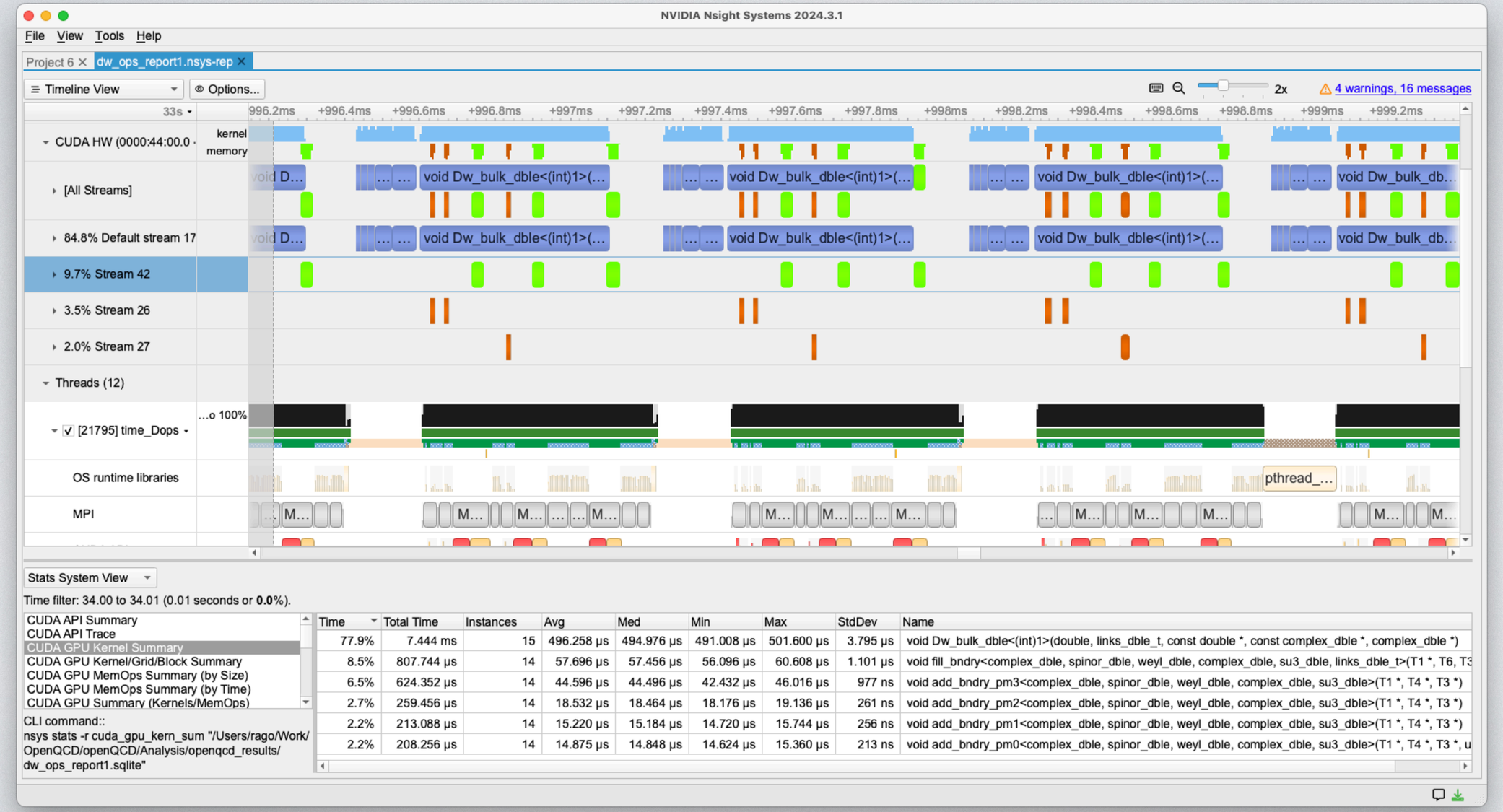
- Centralised the MPI communications and support locking and non locking strategy at compile time.
- MPI is often erratic:
  non-locking comms over p-threads
- Topology awareness guarantees a sensible speedup thanks to NVlink





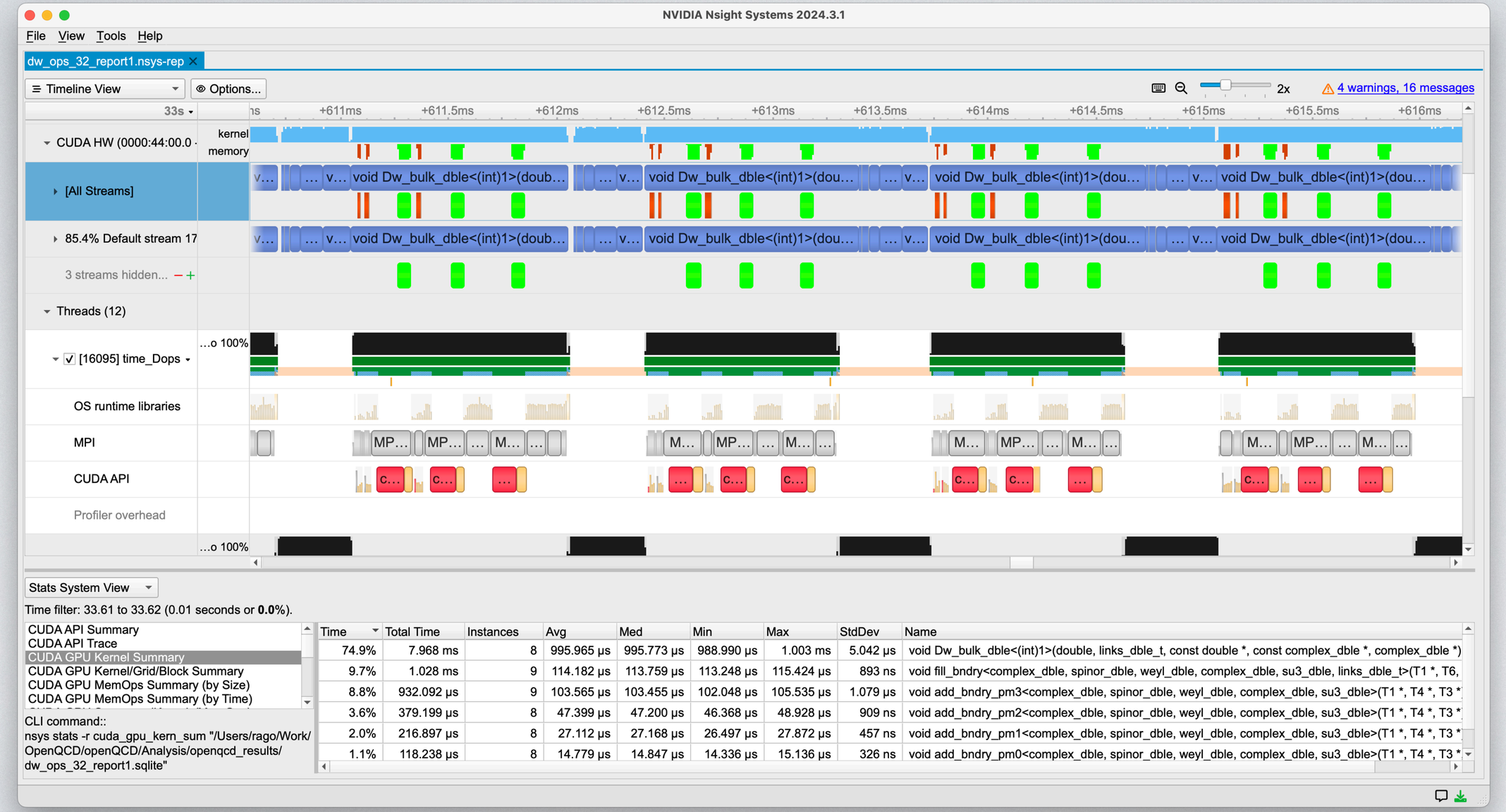


A first test: weak scaling D_W for a local 24^4



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
77.9%	7.444 ms	15	496.258 µs	494.976 µs	491.008 µs	501.600 µs	3.795 µs	void Dw_bulk_dble<(int)1>(double, links_dble_t, const double *, const complex_dble *, complex_dble *)
8.5%	807.744 µs	14	57.696 µs	57.456 µs	56.096 µs	60.608 µs	1.101 µs	void fill_bndry<complex_dble, spinor_dble, weyl_dble, complex_dble, su3_dble, links_dble_t>(T1 *, T6, T3
6.5%	624.352 µs	14	44.596 µs	44.496 µs	42.432 µs	46.016 µs	977 ns	void add_bndry_pm3<complex_dble, spinor_dble, weyl_dble, complex_dble, su3_dble>(T1 *, T4 *, T3 *)
2.7%	259.456 µs	14	18.532 µs	18.464 µs	18.176 µs	19.136 µs	261 ns	void add_bndry_pm2<complex_dble, spinor_dble, weyl_dble, complex_dble, su3_dble>(T1 *, T4 *, T3 *)
2.2%	213.088 µs	14	15.220 µs	15.184 µs	14.720 µs	15.744 µs	256 ns	void add_bndry_pm1<complex_dble, spinor_dble, weyl_dble, complex_dble, su3_dble>(T1 *, T4 *, T3 *)
2.2%	208.256 µs	14	14.875 µs	14.848 µs	14.624 µs	15.360 µs	213 ns	void add_bndry_pm0<complex_dble, spinor_dble, weyl_dble, complex_dble, su3_dble>(T1 *, T4 *, T3 *, u

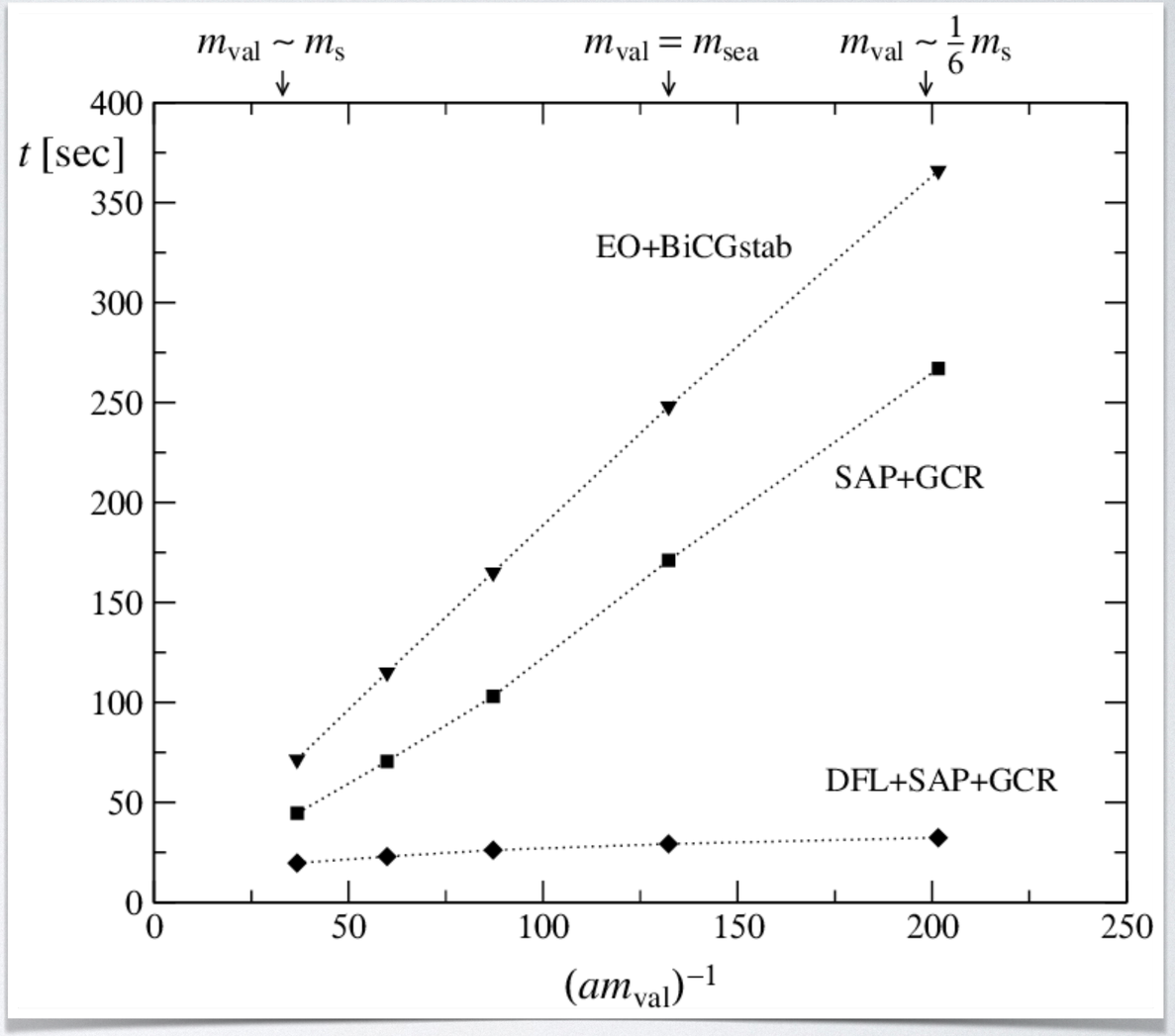
A first test: weak scaling D_W for a local 48×24^3



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
74.9%	7.968 ms	8	995.965 µs	995.773 µs	988.990 µs	1.003 ms	5.042 µs	void Dw_bulk_dble<(int)1>(double, links_dble_t, const double *, const complex_dble *, complex_dble *)
9.7%	1.028 ms	9	114.182 µs	113.759 µs	113.248 µs	115.424 µs	893 ns	void fill_bndry<complex_dble, spinor_dble, weyl_dble, complex_dble, su3_dble, links_dble_t>(T1 *, T6,
8.8%	932.092 µs	9	103.565 µs	103.455 µs	102.048 µs	105.535 µs	1.079 µs	void add_bndry_pm3<complex_dble, spinor_dble, weyl_dble, complex_dble, su3_dble>(T1 *, T4 *, T3 *
3.6%	379.199 µs	8	47.399 µs	47.200 µs	46.368 µs	48.928 µs	909 ns	void add_bndry_pm2<complex_dble, spinor_dble, weyl_dble, complex_dble, su3_dble>(T1 *, T4 *, T3 *
2.0%	216.897 µs	8	27.112 µs	27.168 µs	26.497 µs	27.872 µs	457 ns	void add_bndry_pm1<complex_dble, spinor_dble, weyl_dble, complex_dble, su3_dble>(T1 *, T4 *, T3 *
1.1%	118.238 µs	8	14.779 µs	14.847 µs	14.336 µs	15.136 µs	326 ns	void add_bndry_pm0<complex_dble, spinor_dble, weyl_dble, complex_dble, su3_dble>(T1 *, T4 *, T3 *

A more challenging case: SAP and DFL

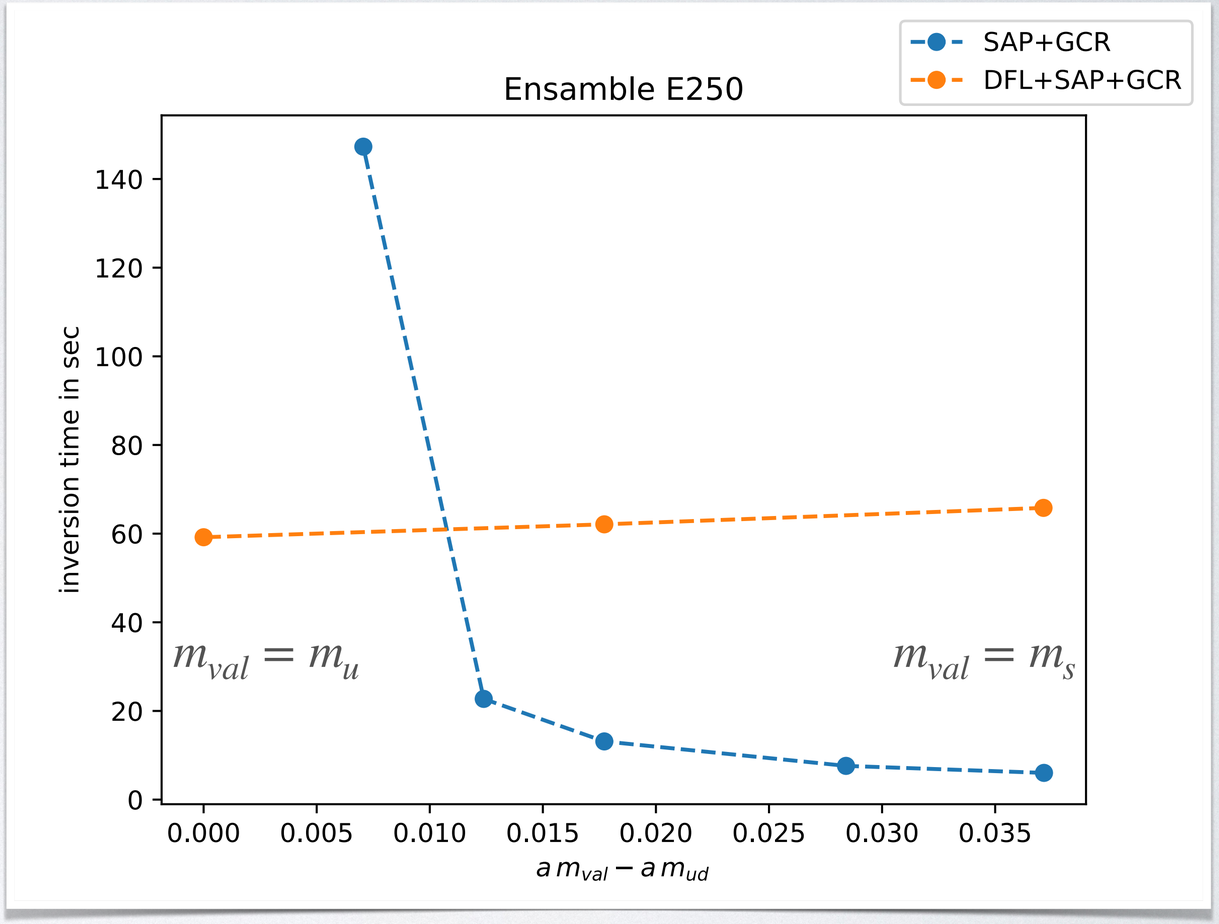
- SAP and Deflation are two inverters that take advantage of
  locality and mode coherence.
- Our strategy has been to focus on an efficient porting of the two routines.
  - block geometry allowing seamless use of different block sizes
  - Shared memory within the sap blocks
  - Block reduction through cubs thanks to the binding of the block indices to the CUDA grid



A more challenging case: SAP and DFL

	TxL^3	M_π	M_K	a
E250	$192x96^3$	130 MeV	500 MeV	0.065 fm

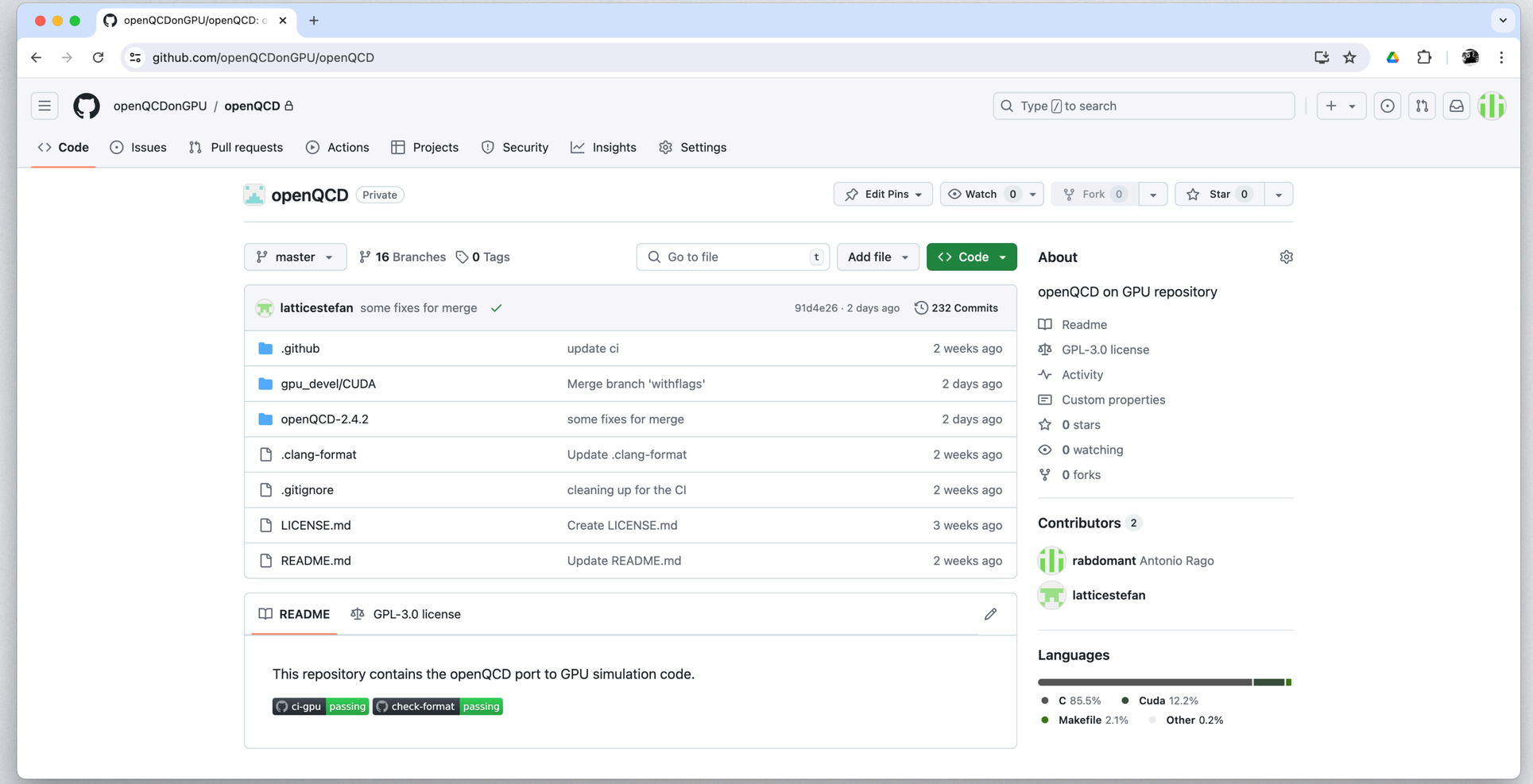
- ✓ SAP seems to be a pretty well tuned routine, with small communication overhead.
- ✓ On the lightest point the computational cost is equally shared between the SAP and the little-GCR.
- ☐ Kernel fusion could improve the little-GCR but the overall performance is not going to change drastically.



64 Juwels booster nodes

Modern coding standards

- git & Continuous integration
- Code coverage



Project status

Already implemented:

- ✓ SAP / DFL / GCR
- ✓ Only PBC (Periodic Boundary Conditions):
- ✓ Random Number Generator
- ✓ Quadruple Precision in Reductions
- ✓ AMD/HIP porting works and passes initial checks.

What is missing

- Open Boundary Conditions
- Exponential c_{sw}
- Independent/automatic single kernel tuning
- Second stage hmc/smd (although gauge and c_{sw} forces gauge already implemented)

Future plans and code release

Testing and Release Strategy:

- Conduct initial testing phase with a selected group of colleagues and collaborators
- Identify and fix any preliminary bugs
- Gather feedback on usability and performances
- Ensure stability and reliability before broader release

Second Stage: Public release

- Release the code to the public with an appropriate license upon initial publication
- Provide detailed documentation and user guides
- Include comprehensive results from testing and performance evaluations