Interface of QUDA with our measurement code: tuning and benchmarks of the staggered solvers

Tuesday, Jun 7, 2022

Assemblée Générale of the GDR QCD

Gen Wang

Centre de Physique Théorique

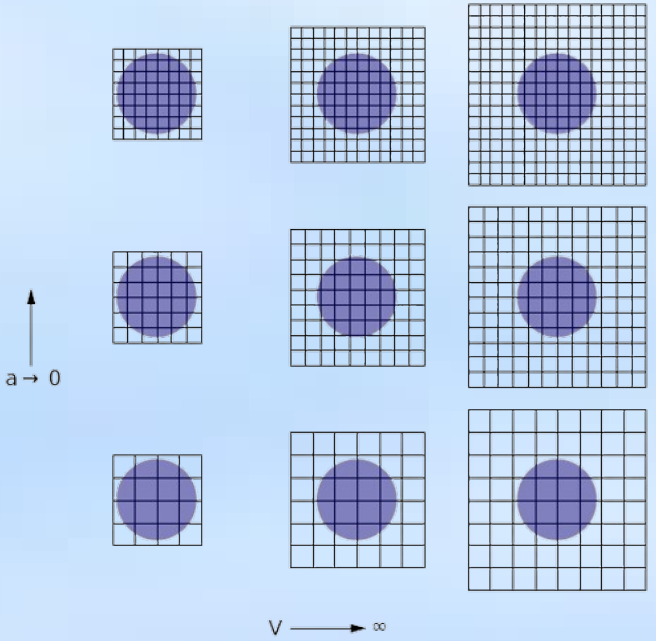
- Lattice QCD
- Interface with Quda
- Solver performance tuning
- Tools porting
- Conclusion

Lattice QCD

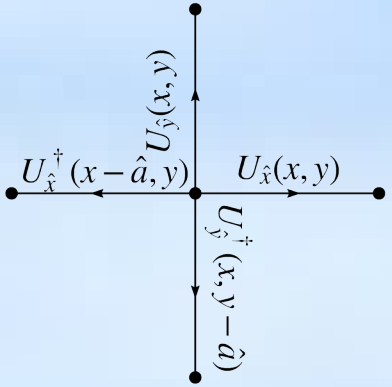
Path integral representation in Euclidean space

$$\langle \mathcal{O} \rangle = \frac{1}{Z} \int [dU] \prod_f \mathrm{Det}[\mathcal{D}_f] \mathcal{O}(U) e^{-S_G[U]}$$

Gauge configurations generated with Monte Carlo



$$\text{probability weight} \sim \prod_f \mathrm{Det}[\mathcal{D}_f] e^{-S_G[U]}$$



Operator measurement on each gauge configuration

$$\langle \mathcal{O} \rangle = \sum_{i=1}^{N} \frac{1}{N} \mathcal{O}(U_i) \quad \Longleftarrow D^{-1}(y, x_0)$$

Quark propagators under gauge links

Two-point functions

Most costly part

$D^{-1}(y, x_0)$

Conserved vector current

$J_\mu = (D_\mu \bar{q}) q$

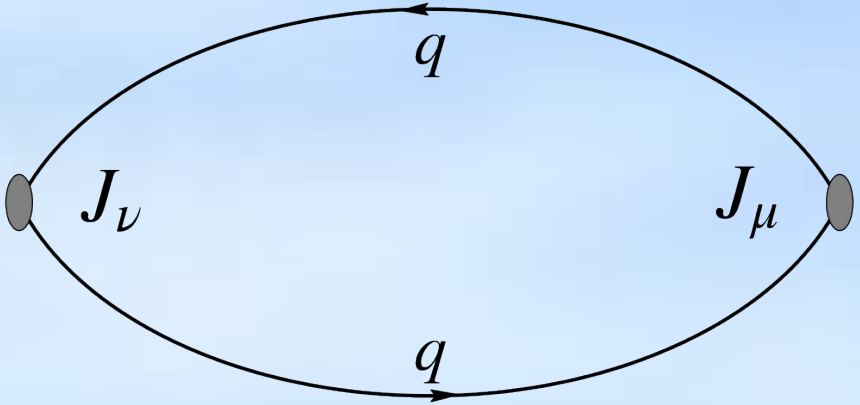
$$C(\vec{p}, \vec{q}, \mu, \nu) = \sum_{\vec{y}, \vec{z}} \langle J_\mu(\vec{z}, t_z) J_\nu(\vec{y}, t_y) \rangle \, e^{-i\vec{p}\cdot\vec{y}} e^{-i\vec{q}\cdot\vec{z}}$$

$(\vec{y}, t_y)$

Only finite #points

Euclidean correlation functions in momentum space

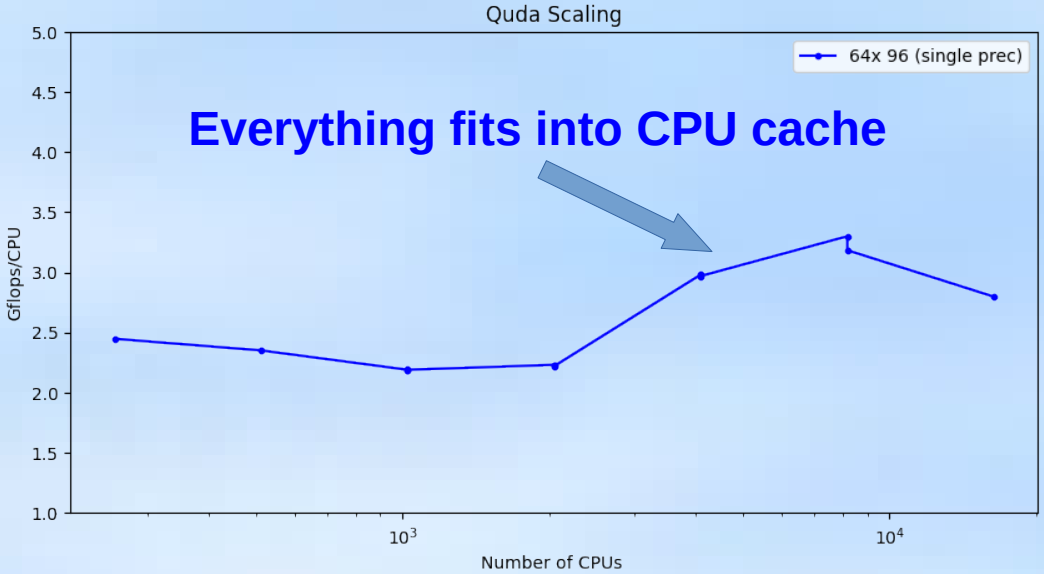
Fourier transform of >300 momenta



- Spectrum study
- HVP part of Muon gm2
- LBL part of Muon gm2
- Parameters for form factors
- ...

Current code status

48x96	solver	Momenta apply	Contraction
Cost (1728 CPUs, sec)	230	12	10



CPU code performance

A GPU solver for quark propagators might be enough to port the code

Quda library

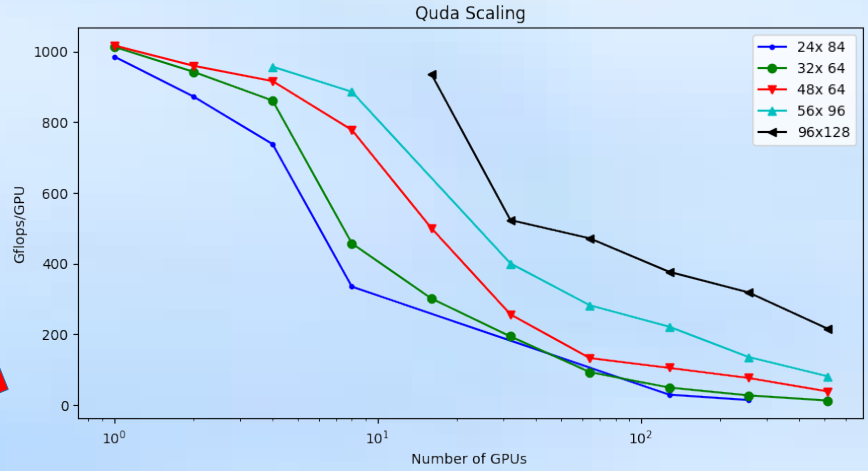
M. A. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi
http://lattice.github.io/quda/

- Lattice QCD
- Interface with Quda
  - Quda features
  - Interface
  - Technical problems
- Solver performance tuning
- Tools porting
- Conclusion

Quda features

M. A. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi

- Kernel Autotuning
- Supports latest algorithms development and most actions (Clover, twisted mass, **Staggered fermion**, Domain wall)
- Large user community and frequent updates



Staggered fermion solver performance

- Special program model
- More sophisticated memory management
- Avoid communication if possible
- Combine more jobs for each kernel call
- ……

$$\text{A100 GPU} \rightarrow \frac{40\,\text{Tflops}}{160\,\text{GB}} \quad (4\,\text{GPUs})$$

$$\text{KNL} \rightarrow \frac{3.0\,\text{Tflops}}{96\,\text{GB}} \quad (64\,\text{CPUs})$$

Interface

Current CPU code written in plain “C”

- C++ functions (Quda used mostly)could not be called directly from C in most cases
- No support of class/structure functions
- No support of templates

Naïve work around

C side

c.h

```
void do_quda_inv(void* qinv, void* res, void* src, double mass);
```

utils.h

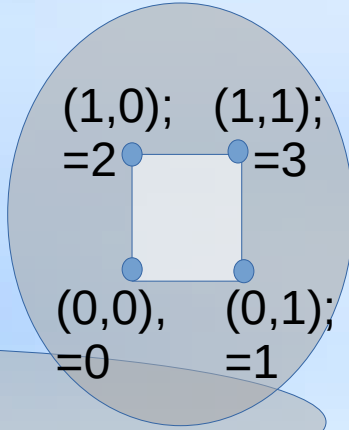
```
extern "C" void do_quda_inv(void* qinv, void* res, void* src,double mass,)
{((quda_inverter*) qinv)->do_inv(res, src, mass, err, niter);}
```

utils.cpp #include "qlat_utils.h"

```
……..qinv.do_inv(…)…
```

nvcc -c utils.cpp -o utils.o

C++ side



- Layout of MPI and data
- Phase convention for stagged fermions

- Lattice QCD
- Interface with Quda
- Solver performance tuning
  - Eigensystem usage
  - Matrix prod on GPU
- Tools porting
- Conclusion

Solver performance tuning

- Direct conjugate gradient (iterative solver)

Critical slowing down

$$D\vec{x} = \vec{y} \qquad \approx 1 - \frac{2}{k(D)} \qquad k(D) \equiv \frac{|\lambda_{max}|}{|\lambda_{min}| \approx m_q}$$

- Multi-grid

$$D(x) \quad \Leftrightarrow \quad P(\hat{x}, x),\ Q(x, \hat{x}) \quad \Leftrightarrow \quad D(\hat{x})$$

Preserve lowest eigenstate Reduce problem dim

Projective multigrid method for propagators in lattice gauge theory
Richard C. Brower, Claudio Rebbi, Ettore Vicari, Phys. Rev. D 43, 1965

- **Deflation**

$$\vec{x}_L = \sum_{\lambda_i=0}^{n\sim1000} \frac{1}{\lambda_i} v_i v_i^\dagger, \quad Dv_i = \lambda_i v_i$$

Greatly reduce the number of iterations

Large Memory needed to obtain these vectors

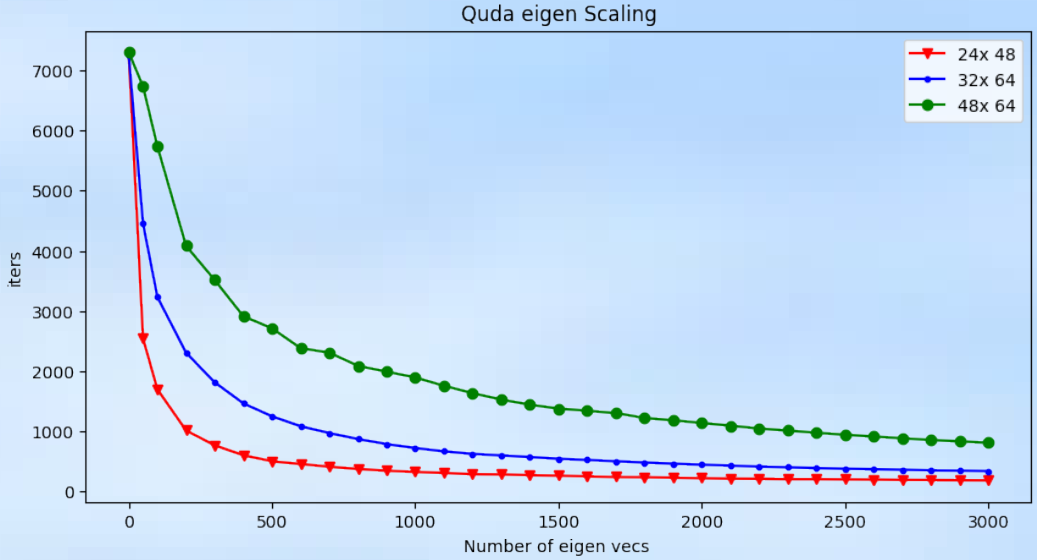
Solver performance tuning

Eigensystem solving under even-odd preconditioning

$$D = D_0 + m = \begin{pmatrix} m & D_{eo} \\ D_{oe} & m \end{pmatrix}, \quad D_{eo} = -D_{oe}^{\dagger}$$

$$\hat{D} = -D_{eo}D_{oe} + m^2, \quad \hat{D}^{\dagger} = \hat{D}$$

Scaling as with # of eigenvectors

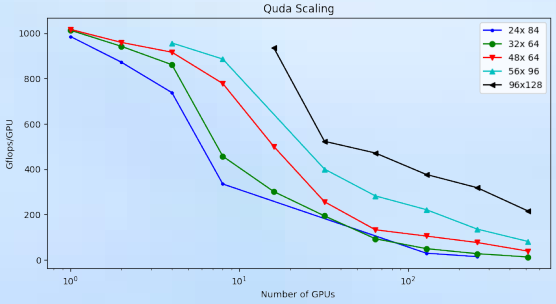


The # of eigenvectors scale as the volume of La = physical volume

Eigensystem usage

- Eigensystem solved from Krylov subspaces

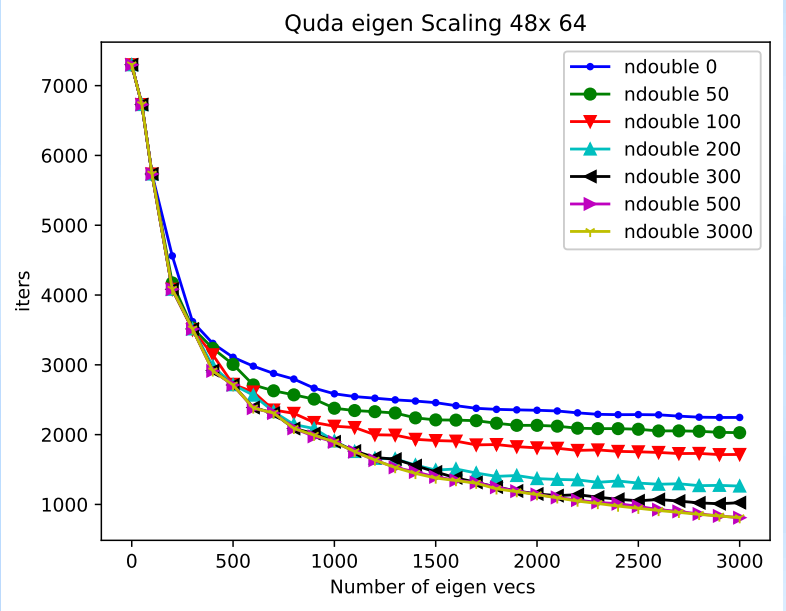
$$b, \quad \hat{D}b, \quad \hat{D}^2 b, \quad \hat{D}^3 b, \quad \cdots$$



- Normally will require triple or double the memory requirement
  - ---- Solve the eigen vectors in advance and save them
  - ---- 1000 vectors need 160 GB memory > 2 node

- Mixed precision eigensystem
  - 300 double precision vectors
  - 700 single precision vectors

- Eigensytem compression?
  - Additional reduce of precision of 700 vectors



arXiv: 1710.06884 M. A. Clark, Chulwoo Jung, and Christoph Lehner

Solver performance tuning

48x96	solver	Momenta apply	Contraction
Cost (4 GPUs, sec)	97.26	15143.36	175.74

$$C(\vec{p},\vec{q},\mu,\nu) = \sum_{\vec{y},\vec{z}} \langle J_\mu(\vec{z},t_z) J_\nu(\vec{y},t_y)\rangle \, e^{-i\vec{p}\cdot\vec{y}} e^{-i\vec{q}\cdot\vec{z}}$$

Expensive

For each y, boost the FT into matrix product

$$f(\vec{z},t)e^{-i\vec{q}\cdot\vec{z}} \quad \Longrightarrow \quad F(nt,V) \times Q(V,nq)$$

~0.16 GB ~0.5 GB

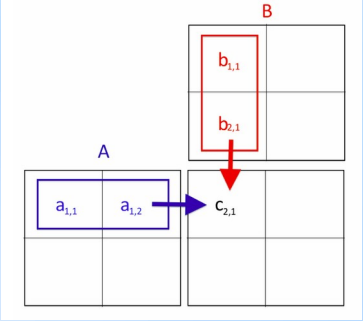
Friendly to GPU due to large computation with low memory usage

Matrix product on GPU

$$C(m,n) = A(m,r) \times B(r,n)$$

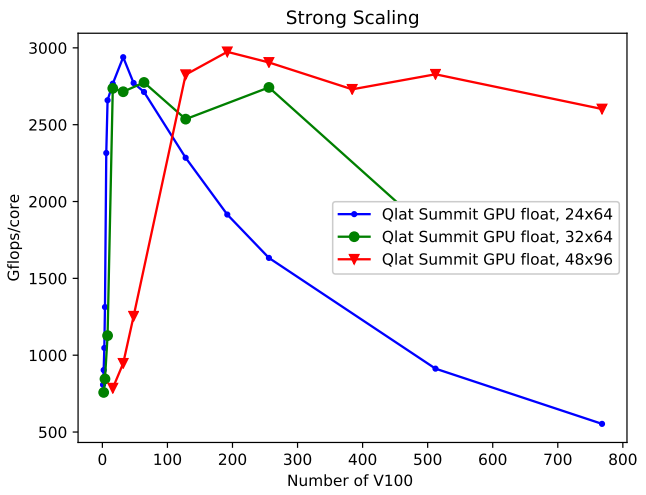
m ~ 100
r ~ 1e7
n ~ 300

- Order of ~1000 computation units



Each unit control one (m, n)

- ~32 threads within each unit
  - (dm, dn) data load to shared memory to increase memory bandwidth usage
- Use template to reduce usage of “**if**” and dynamical “**for**” loop



Dimension r on different node

- https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA/
- https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf
- https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

Performance

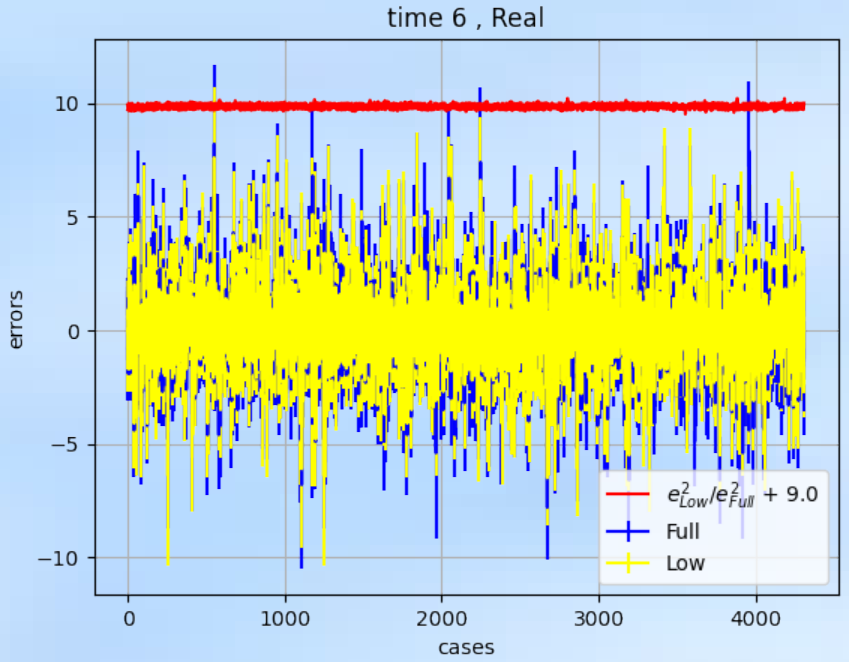
48x96	solver	Momenta apply	Contraction
Cost (4 GPUs, sec)	97.26	259.04	175.74
Cost (1728 CPUs, sec)	230	12	10

Table of timing using mixed precision eigensystem and matrix product

	CPUh	GPUh
Charge	8966 / 170 = 50	39

- Gain more by using GPU?
  - ---Reduce momenta apply and contraction time
- Detect where the noise come from

$$D^{-1}(z|y) = \sum_{\lambda_i} \frac{1}{\lambda_i} v_i(z) v_i^{\dagger}(y) + D_H^{-1}(z|y)$$



Volume averaging (all-to-All low modes)

$$C(\vec{p},\vec{q},\mu,\nu) = \sum_{\vec{y},\vec{z}} \langle J_\mu(\vec{z},t_z) J_\nu(\vec{y},t_y)\rangle \, e^{-i\vec{p}\cdot\vec{y}} e^{-i\vec{q}\cdot\vec{z}}$$

Split propagator into high-mode and low-mode part

$$D^{-1}(y|x) = \sum_i \lambda_i v_i(y) v^\dagger(x) + D_H^{-1}(y|x)$$

H. Neff, N. Eicker, T. Lippert, J. W. Negele, and K. Schilling, Phys. Rev. D 64, 114509 (2001)

Focus on low-mode and sum over all positions y (huge noise reduction)

$$C_L(\vec{p},\vec{q},\mu,\nu) = \sum_{\vec{y},\vec{z}} e^{-i\vec{p}\cdot\vec{y}} e^{-i\vec{q}\cdot\vec{z}} \mathrm{Tr}\left[\left(\sum_i \lambda_i v_i^\mu(y) v_i^\dagger(z)\right)\left(\sum_j \lambda_j v_i(z) v_j^{\dagger,\nu}(y)\right)\right]$$

Vector shifts

$$= \sum_{i,j} \lambda_i \lambda_j \left(\sum_{\vec{y}} e^{-i\vec{p}\cdot\vec{y}} \mathrm{Tr}\left[v_i^\mu(y) v_j^{\dagger,\nu}(y)\right]\right)\left(\sum_{\vec{z}} e^{-i\vec{q}\cdot\vec{z}} \mathrm{Tr}\left[v_i(z) v_i^\dagger(z)\right]\right)$$

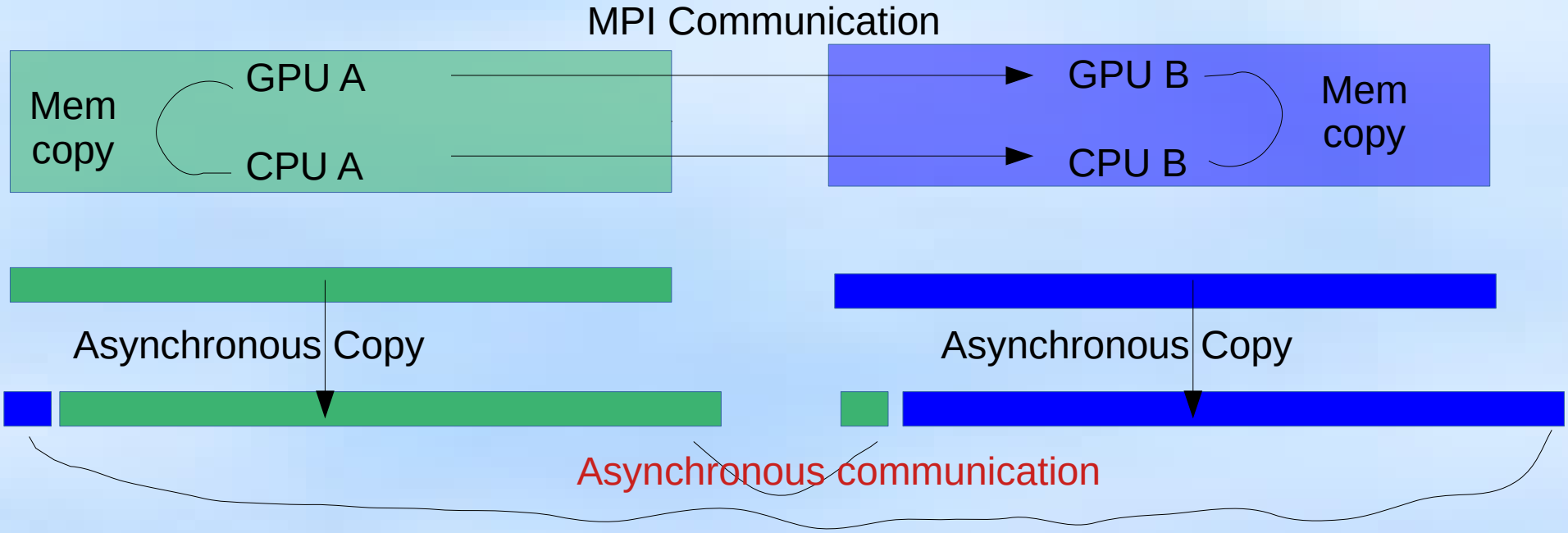
~2000 x 2000

Fast Fourier transform

Matrix product

- Lattice QCD
- Interface with Quda
- Solver performance tuning
- Tools porting
  - Vector shift
  - Fast Fourier transform (FFT)
- Conclusion

Vector shift



Vec shift performancde GB/s	2	4	6	8	16	32	64
GPU double	53.59	53.59	53.59	**27.17**	**26.04**	**11.03**	**17.61**
CPU double	13.76	13.76	9.39	9.91	9.36	8.92	8.89

Fast Fourier transform (FFT)

- cuFFT (https://docs.nvidia.com/cuda/cufft/index.html similar to FFTW on CPU)
  - Only supports single GPU FFT
  - Only supports 3D FFT

$$\sum_{\vec{y}} e^{-i\vec{p}\cdot\vec{y}} F_j(\vec{y}, t_y)$$

GPU A

L(**0,8**) x V(x,y,z) x T(0, 16), **3D**

↓

L(**0,4**) x V(x,y,z) x T(**0, 32**), **4D**

GPU B

L(**0,8**) x V(x,y,z) x T(**16, 32**), **3D**

↓

L(**4,8**) x V(x,y,z) x T(**0, 32**) , **4D**

GPU (Glops/V100)	1	2	4	6	8	16	32	64
3D single	**370.54**	**367.91**	**369.03**	**360.80**	**343.02**	316.03	277.13	95.27
4D single	287.65	134.11	62.43	40.31	23.89	9.69	--	--

Flops calculate Naïvely from V ln(V) X 8

Conclusions

- Porting of few parts of CPU code to GPU reached reasonable performance
- Basic tools porting such as matrix product and FFT have been done
- A complete code which shares the same algorithm between CPU and GPU is desired and will be reached in near future

Thank You