Porting Non-equilibrium Green's functions to GPUs

Alessandro Pecchia, *ISMN-CNR, Rome, Italy*

Daniele Soccodato, *University of Rome 'Tor Vergata'*

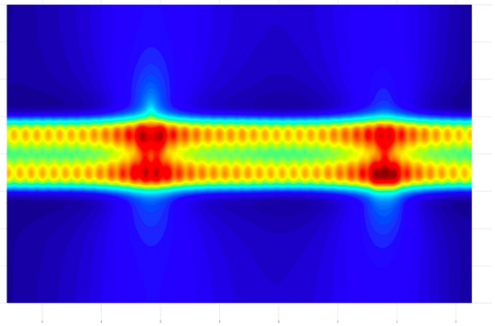
Sebastian Achilles, Edoardo Di Napoli , *Julich Supercomputing Centre*

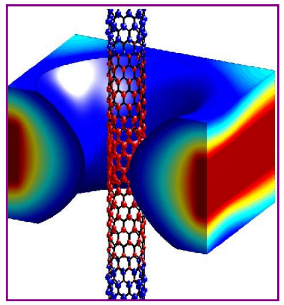
Collaboration with NVIDIA Application Lab at Jülich

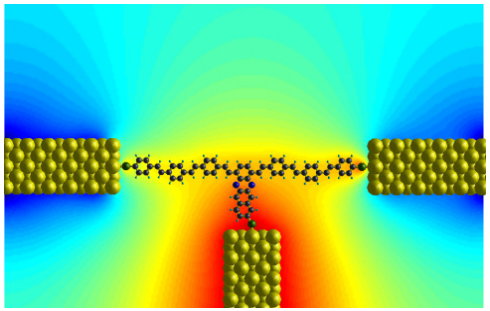
libNEGF

neXtGf

Non-Equilibrium Green's Functions

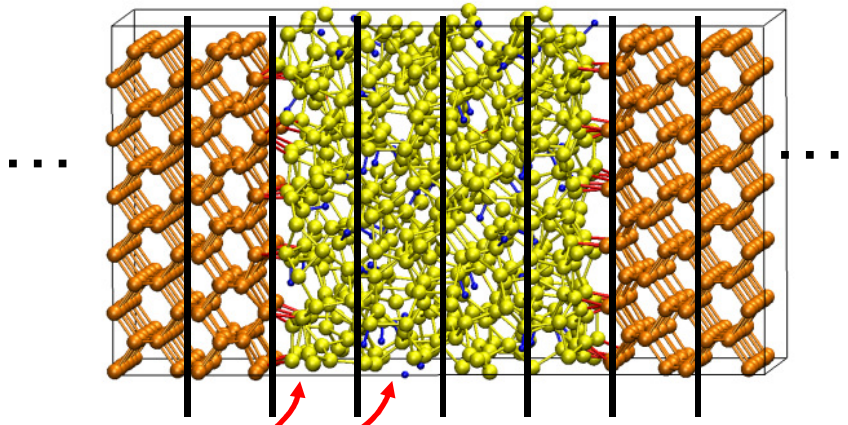






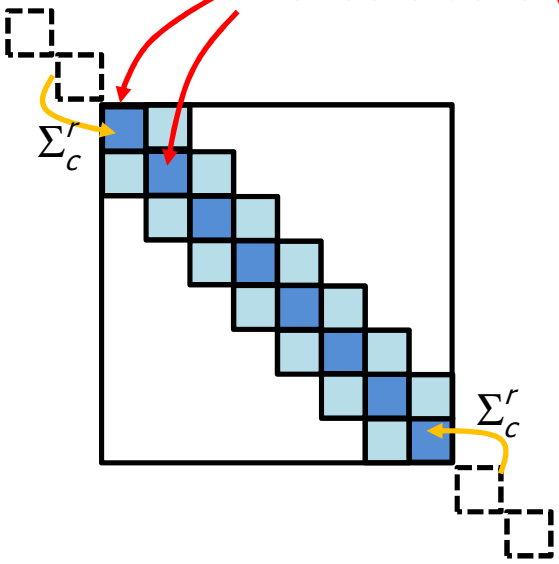
- Many-Body Non-equilibrium Quantum Systems
- *lib*NEGF library (FEM/TB/DFT Hamiltonians)
- Computationally intensive (dense matrix cmplx algebra)
- Objective of **EoCoE**: NEGF to eXa-Scale (neXtGf)

*lib*NEGF



System partitioning into layers (PLs)

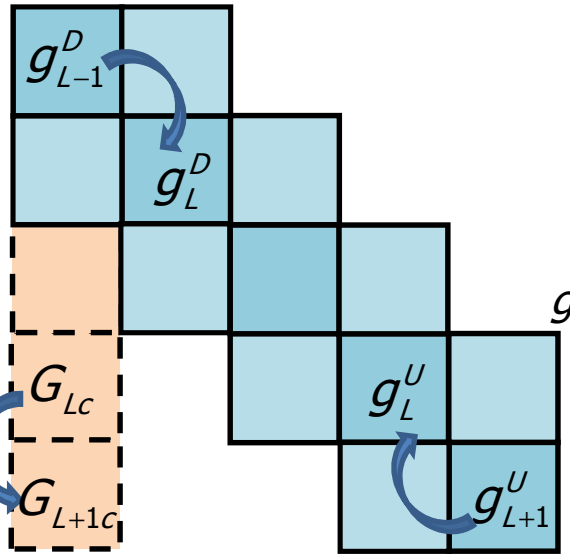
Dense blocks (the size depends on the system cross-section)



Contacts Interactions

$$\begin{cases} B(E,k) = \left[ES(k) - H(k) + \Sigma_c^r(E,k) + \Sigma_\phi^r(E,k) \right] \\ B(E,k)G^r(E,k) = I \quad \text{(Dyson)} \\ G^n(E,k) = G^r(E,k)\left[\Sigma_c^n(E,k) + \Sigma_\phi^n(E,k) \right] G^{r\dagger}(E,k) \quad \text{(Keldish)} \end{cases}$$

Iterative algorithm (Schur-complement)



$$g^U_L = \left[ES_{LL} - H_{LL} - H_{LL+1} g^U_{L+1} H_{L+1L} \right]^{-1}$$

$$G^r_{L+1c} = -g^U_{L+1} H_{L+1L} G^r_{Lc}$$

Linear Algebra: Matrix Inversions and MxM multiplications

CPU

- Modern Fortran
- Linear Libraries (MKL BLAS/LAPACK)

GPU

New

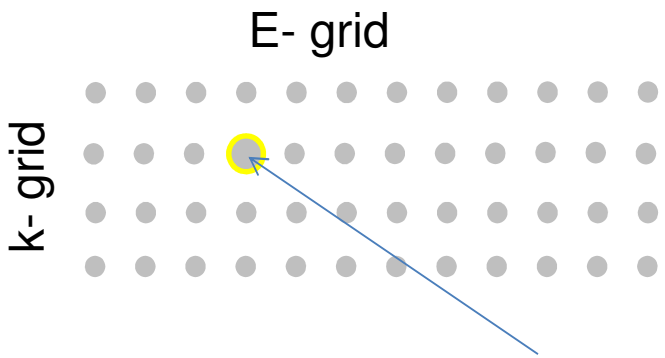
- GPU port path: OpenACC & CUDA
- CUBLAS/CUSOLVER

Parallelization

Green's functions are k- and E- dependent.

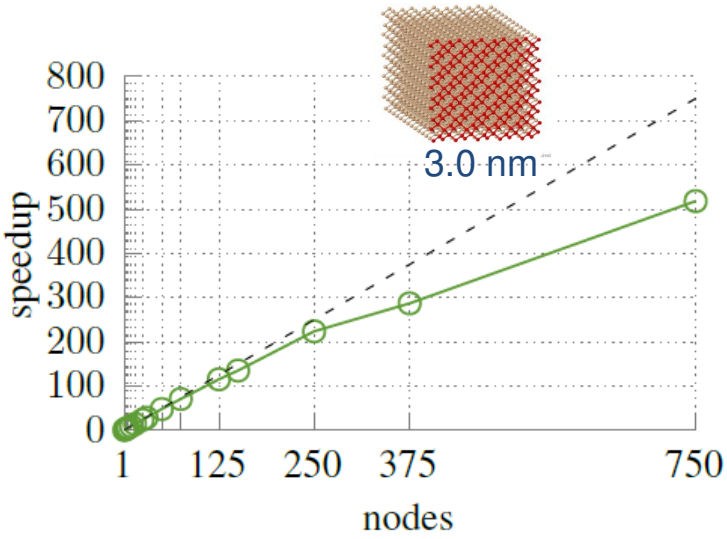
$$G^r(E,k) = \left[ES(k) - H(k) + \Sigma_c^r(E,k) + \Sigma_\phi^r(E,k) \right]^{-1}$$

Distribution of E and k over an MPI 2D cartesian grid



Each MPI task computes $G^r(E,k)$

MPI/openMP hybrid scheme



Strong scaling: 3 nm Si system.
4 tasks per node, 12 threads per task.

Contact Self-Energies, Σ_c^r

Iterative renormalization algorithm

do i = 1, nmax

$$g_i = A_i^{-1}$$

$$A_{i+1}^s = A_{i+1}^s - B_i g_i C_i$$

$$A_{i+1} = A_i - B_i g_i C_i - C_i g_i B_i$$

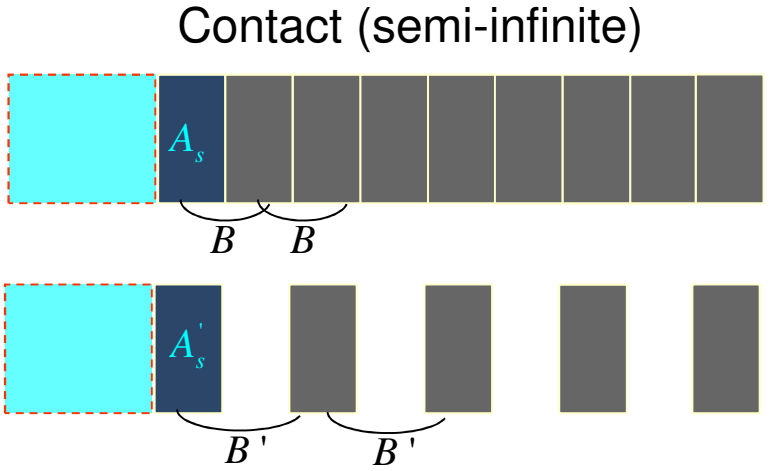
$$B_{i+1} = A_i g_i A_i$$

$$C_{i+1} = C_i g_i C_i$$

end do

$$g^s = A_{i\to\infty}^s$$

$$\begin{array}{|c|c|}\hline A_0 & B_0 \\ \hline C_0 & A_0 \\ \hline \end{array}$$



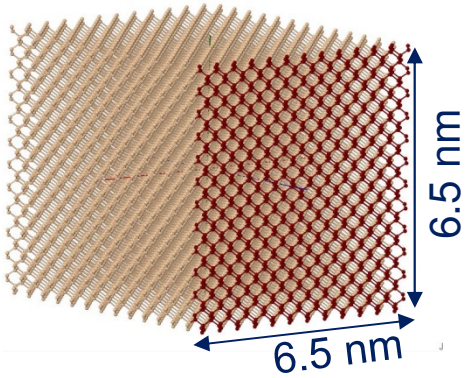
$\rightarrow$ converge to the Surface GF and $\Sigma_c^r = \tau_{Dc} g_s^r(E) \tau_{Dc}^\dagger$

$\rightarrow$ OpenACC implementation

M P Lopez Sancho, J M Lopez Sancho and J Rubio, J. Phys. F: Met. Phys. 15 851 (1985)

GPU vs CPU

Ampere 100 vs Intel Xeon Platinum 8168 (24 cores)



Profiling of the **decimation** algorithm (complex-single-precision) openACC implementation nvfortran 21.3

Size	Mat Size	CPU (threaded)	GPU	Tens.-C	Speedup
4.8 nm	5832	119 s	1.4 s	0.42 s	x30 - x300
6.5 nm	10368	450 s	4.0 s	1.19 s*	x100 - x400
9.2 nm	20736	3600 s	30.0 s	8.2 s*	x120 - x440

* Increase GF parameter δ x2 (5.0e-4) due to precision loss FP32$\rightarrow$ TF32

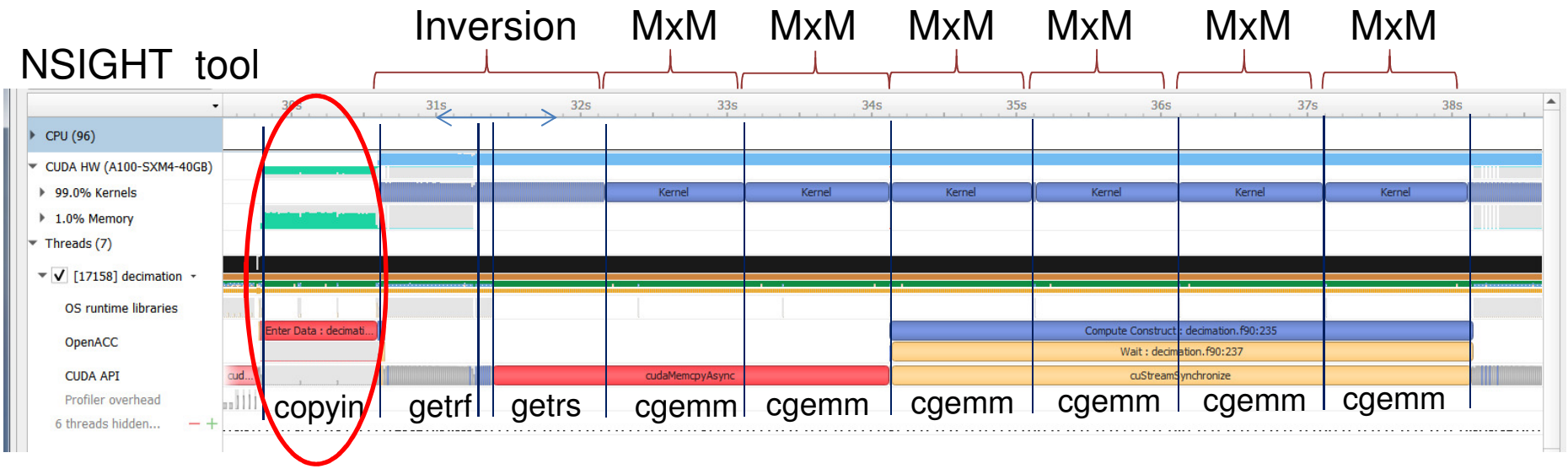
Peak performance on CGEMM:

**FLOPS = $2xN^3x4$/Time

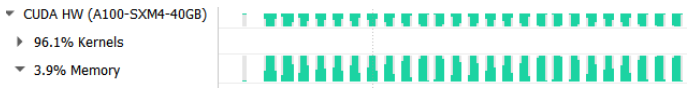
Size	M.Size	Tens.-C
4.8 nm	5832	60 Tflops**
6.5 nm	10368	91 Tflops**
9.2 nm	20736	72 Tflops**

Insight into one iteration

NSIGHT tool



Fast CPU-to-GPU copy ~13 GiB/s (PCIe 4.0 x16 supports up to 32 GiB/s)



16MiB chunks. Throughput: 20-25 GiB/s

Data copies go as ~N^2

Computations go as ~N^3

$\Rightarrow$ winning for large matrices

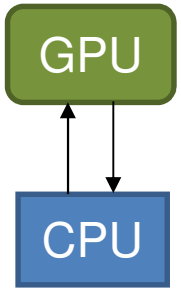
Implementation Details

```
type  z_DNS
   complex(dp), allocatable :: val(:,:)
end type
```

Derived-type datastructures to hold matrices

Keep control over data movements: explicit copyin/copyout

```
subroutine copyToGPU(A)
   type(z_DNS), intent(in) :: A
   !$acc enter data copyin(A,A%val)
 end subroutine copyToGPU
```




```
subroutine copyFromGPU(A)
   type(z_DNS), intent(in) :: A
   !$acc host update(A%val)
 end subroutine copyFormGPU
```

We have not used recent features like ‘deepcopy’

Also we do not use ‘managed’ memory (automatic migration)

Linear Algebra calls

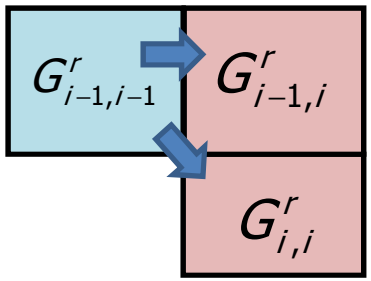
Matrix Inversion $(A \rightarrow LU \rightarrow LUA_{inv} = I)$

```
!$acc data create(LU, work, pivot, err1, err2)
!$acc host_data use_device(A, Ainv, LU, work, pivot, err1, err2)
istat=cublasCcopy(hcublas, n*n, A, 1, LU, 1)
istat=cusolverDnCgetrf(hcusolver, n, n, LU, n, work, pivot, err1)
Ainv = Id  !OpenACC kernel
istat=cusolverDnCgetrs(hcusolver, CUBLAS_OP_N, n ,n, LU, n, pivot, Ainv, n, err2)
```

MxM multiplications $(C = AB)$

```
!$acc enter data present(A,B,C)
!$acc host_data use_device(A, B, C)
istat = cublasCgemm(hcublas, 'N', 'N', m, n, k, alpha, A, m, B, k, beta, C, m)
```

Porting the code



Iterative algorithm: example of gpu code structure

$$G^r_{i,i-1} = g^D_i H_{i,i-1} G^r_{i-1,i-1}$$

example:

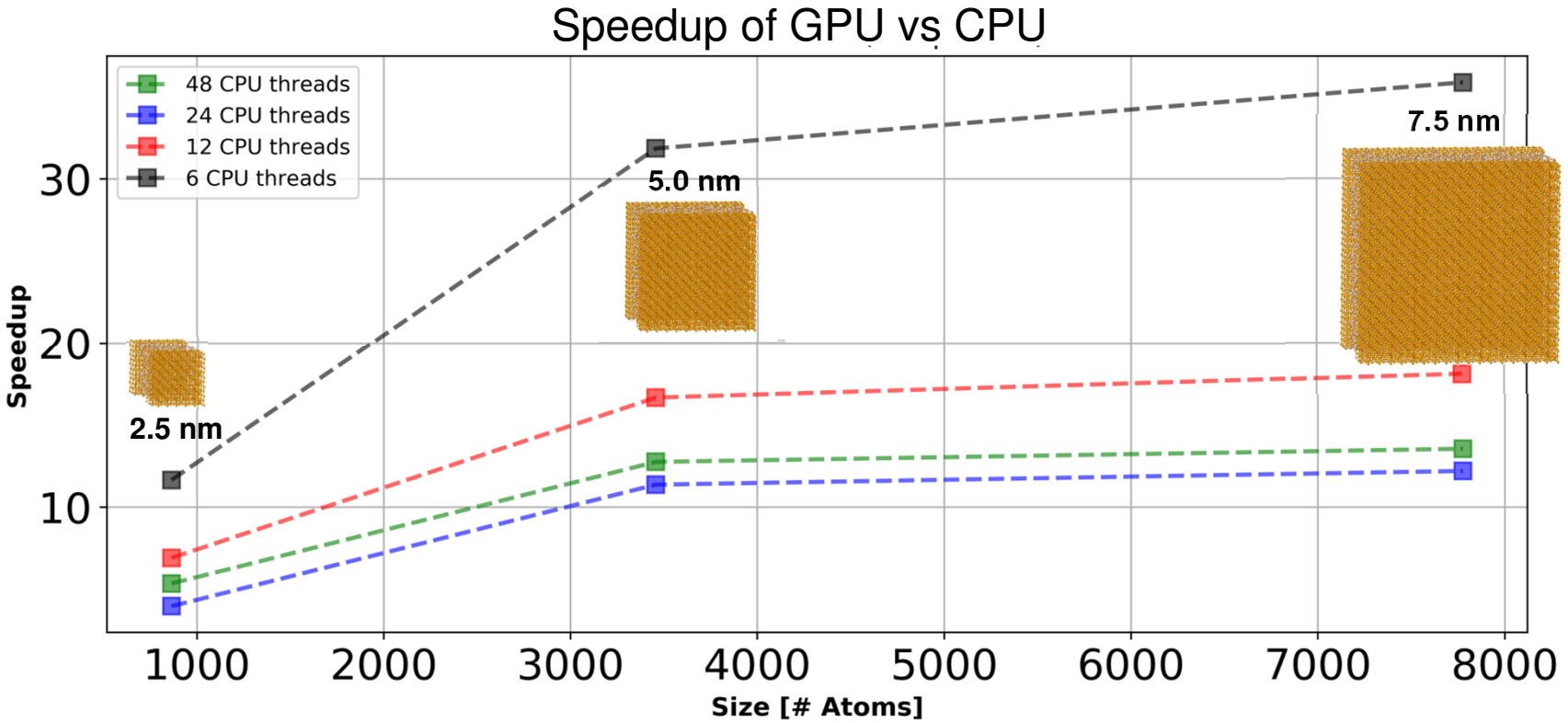
```
call createAll(work1, ·, ·)
call copyToGPU(ESH(i,i-1))
call matmul_gpu(hh, one, gD(i)%val, ESH(i,i-1)%val, zero, work1%val)
call deleteGPU(ESH(i,i-1))
call createAll(Gr(i,i-1), ·, ·)
call matmul_gpu(hh, mone, work1%val, Gr(i-1,i-1)%val, zero, Gr(i,i-1)%val)
call destroyAll(work1)
```

Explicit memory management

Double precision speedup

Ampere 100 vs Intel Xeon Platinum 8168 (24 cores)

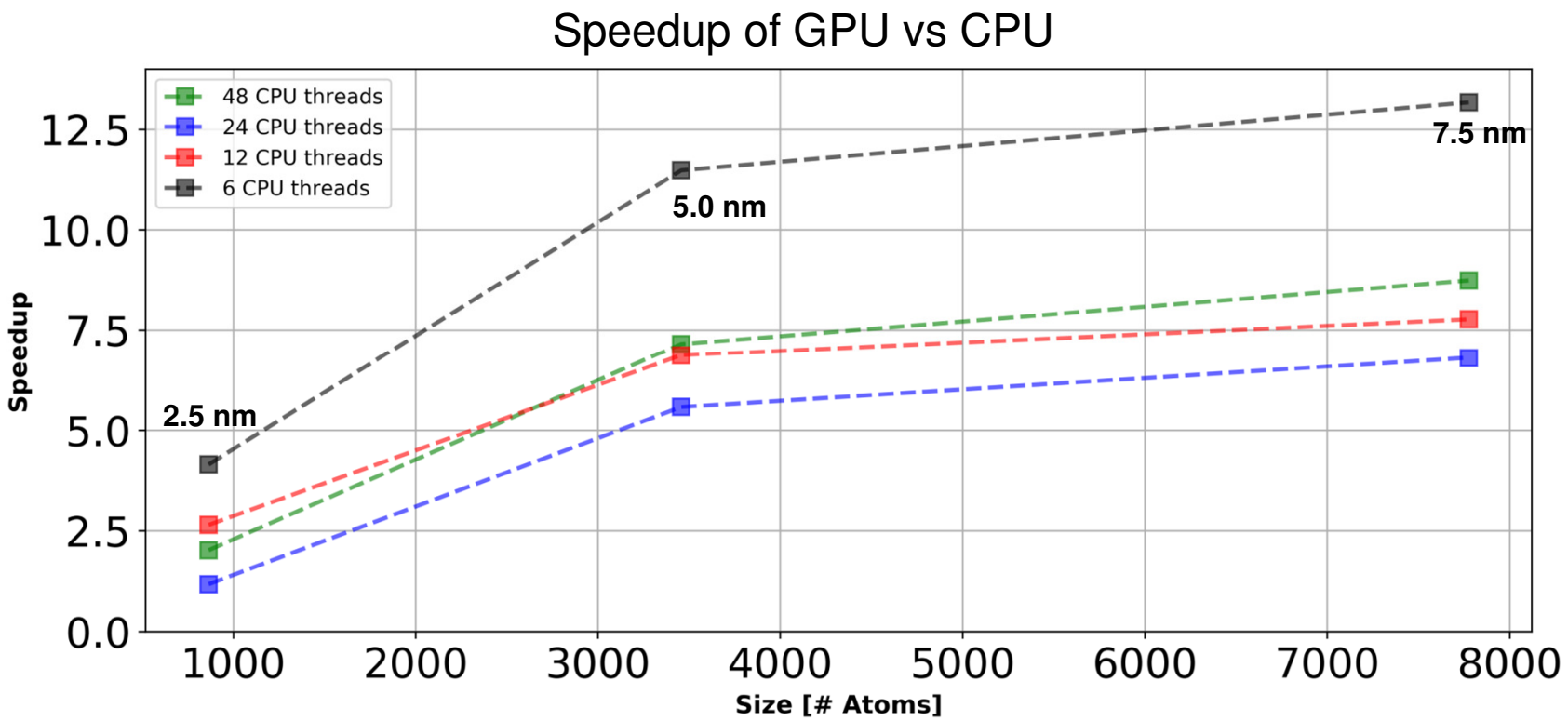
Timing of the **decimation** algorithm



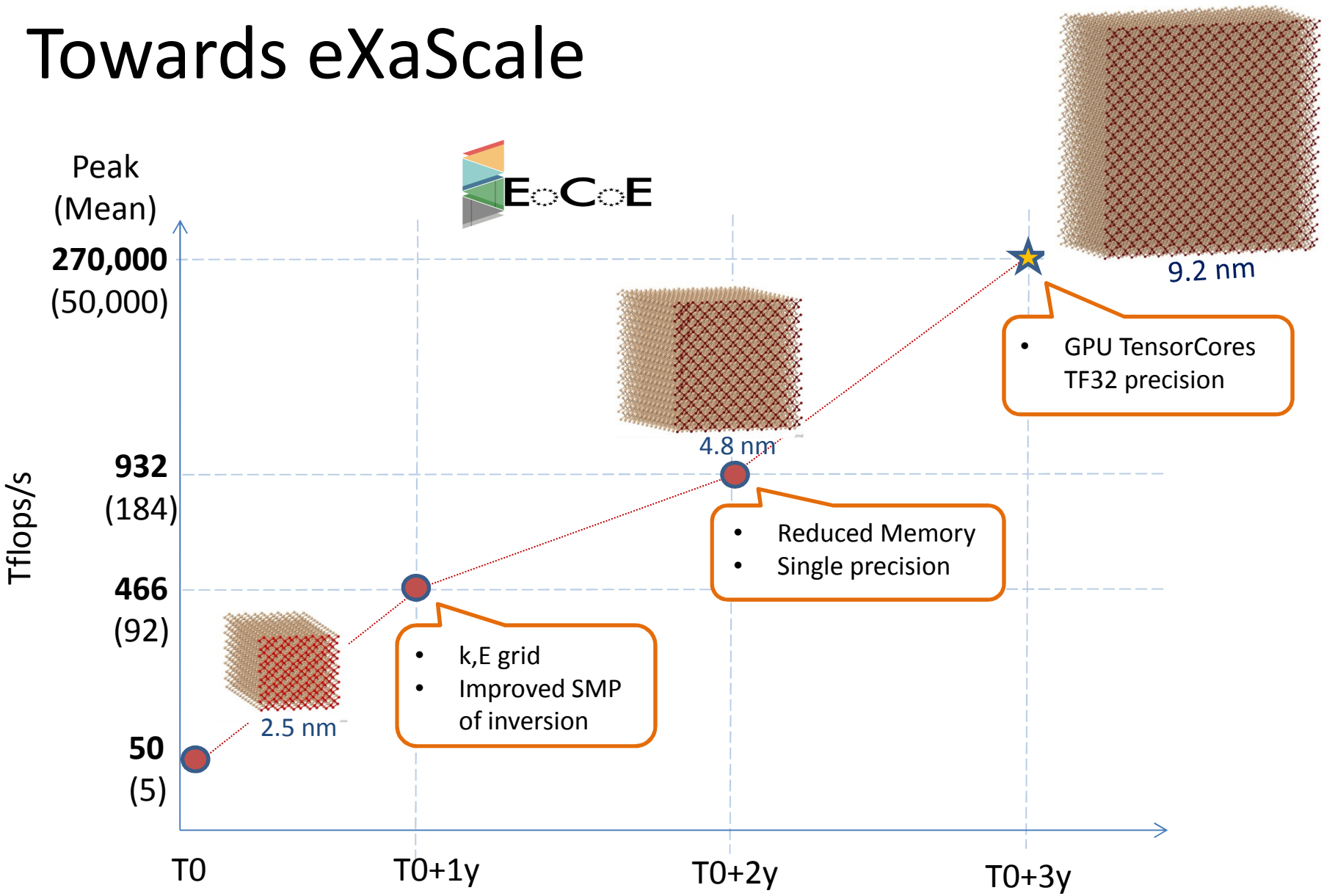
Double precision speedup

Ampere 100 vs Intel Xeon Platinum 8168 (24 cores)

Timing for a complete calculation of **ballistic Transmission**

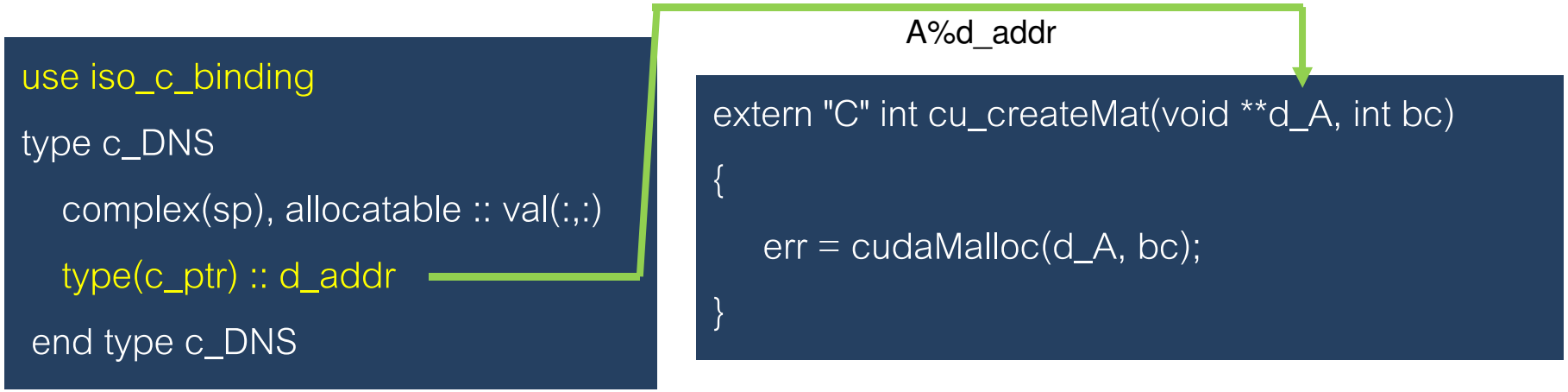


Towards eXaScale



CUDA Implementation

- OpenACC has several advantages in the development speed.
- NVIDIA Fortran compiler triggers a dependence chain (Fortran Modules) This can be problematic sometimes.
- CUDA offers a larger range of features, kernel flexibility and optimizations.



- The final code improves porting.

Conclusions

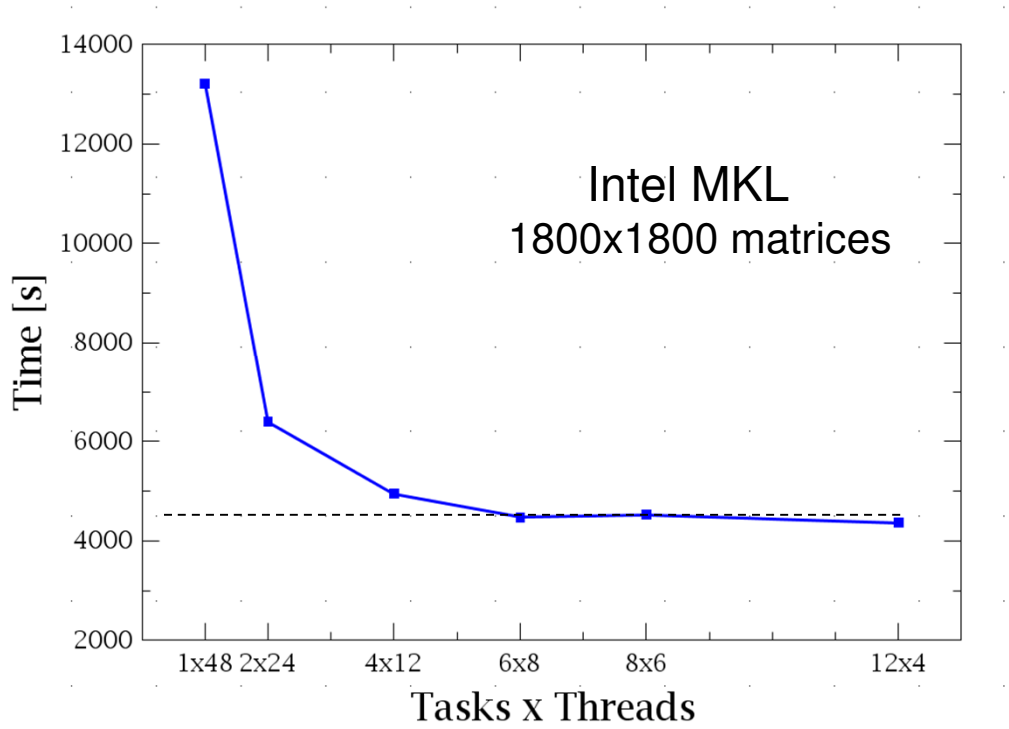
- We have ported the whole *lib*NEGF to GPU using OpenACC
- double-precision speed-up of up to 7.5 (large matrices)
- Miniapps in single precision exploiting tensor cores show much bigger speedups (100-400)
- We are currently fixing the library to recompile in sp

THANKS !

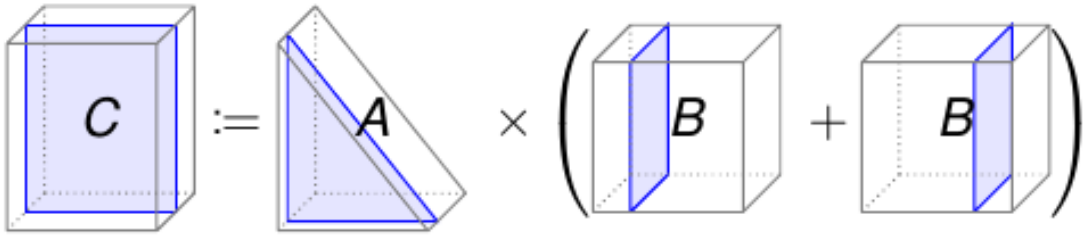
GPU porting strategy

Port to GPU linear algebra operations performed at single task level

CPU Node-level MPI vs openMP (Xeon Platinum 48 cores)



Interaction Self-Energies



$$\Sigma^{<}_{L\mu,L'\nu}(k_t, E) = \sum_{q_t \in BZ} I(k_t, q_t, \left|z_{L\mu} - z_{L'\nu}\right|)\Big[(N_q + 1)G^{<}_{L\mu,L'\nu}(q_t, E + \omega_q) + N_q G^{<}_{L\mu,L'\nu}(q_t, E - \omega_q)\Big]$$

Implementation with cuTENSOR

- Requires outer loop around cuTENSOR call
- Extra mops to decompress A
- 2x FLOPS to account for B + B
- → extra MOPS and extra FLOPS