Porting Non-equilibrium Green’s functions to GPUs

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

Daniele Soccedato, *University of Rome ‘Tor Vergata’*

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

Collaboration with NVIDIA Application Lab at Jülich
Non-Equilibrium Green’s Functions

- Many-Body Non-equilibrium Quantum Systems
- *libNEG*F library (FEM/TB/DFT Hamiltonians)
- Computationally intensive (dense matrix complex algebra)
- Objective of **EoCoE**: NEG to eXa-Scale (neXtGf)
System partitioning into layers (PLs)

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

Contacts

Interactions

\[
\begin{align*}
B(E, k) &= \left[ ES(k) - H(k) + \Sigma_{c}^{r}(E, k) + \Sigma_{\phi}^{r}(E, k) \right] \\
B(E, k)G^{r}(E, k) &= I \\
G^{n}(E, k) &= G^{r}(E, k)\left[ \Sigma_{c}^{n}(E, k) + \Sigma_{\phi}^{n}(E, k) \right]G^{r\dagger}(E, k) 
\end{align*}
\]

(Dyson)

(Keldish)
Iterative algorithm (Schur-complement)

\[ \mathbf{g}_L = \left[ \mathbf{E} \mathbf{S}_{LL} - \mathbf{H}_{LL} - \mathbf{H}_{LL+1} \mathbf{g}_{L+1}^U \mathbf{H}_{L+1L} \right]^{-1} \]

\[ \mathbf{G}_{L+1}^r = -\mathbf{g}_{L+1}^U \mathbf{H}_{L+1L} \mathbf{G}_{Lc}^r \]

Linear Algebra: Matrix Inversions and MxM multiplications

- Modern Fortran
- Linear Libraries (MKL BLAS/LAPACK)
- GPU port path: OpenACC & CUDA
- CUBLAS/CUSOLVER
Parallelization

Green’s functions are \( k \)- and \( E \)- dependent.

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

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, $\Sigma^r_C$

Iterative renormalization algorithm

\[
\begin{align*}
&\text{do } i = 1, \text{nmax} \\
&\quad g_i = A_i^{-1} \\
&\quad A_{i+1}^s = A_{i+1}^s - B_i g_i C_i \\
&\quad A_i = A_i - B_i g_i C_i - C_i g_i B_i \\
&\quad B_{i+1} = A_i g_i A_i \\
&\quad C_{i+1} = C_i g_i C_i \\
&\text{end do} \\
&g^s = A_i \xrightarrow{i \to \infty} \\
\end{align*}
\]

→ converge to the Surface GF and

\[
\Sigma^r_c = \tau_{Dc} g^s (E) \tau_{Dc}^\dagger
\]

→ OpenACC implementation

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

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

* Increase GF parameter $\delta \times 2$ (5.0e-4) due to precision loss FP32 $\rightarrow$ TF32

Peak performance on CGEMM:

**FLOPS = 2N^3 \times 4/Time**

<table>
<thead>
<tr>
<th>Size</th>
<th>M.Size</th>
<th>Tens.-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8 nm</td>
<td>5832</td>
<td>60 Tflops**</td>
</tr>
<tr>
<td>6.5 nm</td>
<td>10368</td>
<td>91 Tflops**</td>
</tr>
<tr>
<td>9.2 nm</td>
<td>20736</td>
<td>72 Tflops**</td>
</tr>
</tbody>
</table>
NSIGHT tool

Insight into one iteration

- **copyin**
- **getrf**
- **getrs**
- **cgemm**
- **cgemm**
- **cgemm**
- **cgemm**
- **cgemm**
- **cgemm**
- **cgemm**

**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 \( \sim N^2 \)

Computations go as \( \sim N^3 \)

\[ \Rightarrow \quad \text{winning for large matrices} \]
Implementation Details

Derived-type datastructures to hold matrices

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

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 copyFromGPU

We have not used recent features like ‘deepcopy’

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

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

\begin{verbatim}
!$acc data create(LU, work, pivot, err1, err2) 
!$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
istat=cusolverDnCgetrs(hcusolver, CUBLAS_OP_N, n ,n, LU, n, pivot, Ainv, n, err2)
\end{verbatim}

MxM multiplications \((C = AB)\)

\begin{verbatim}
!$acc enter data present(A,B,C) 
!$acc host_data use_device(A, B, C) 

istat = cublasCgemm(hcublas, ‘N’, ‘N’, m, n, k, alpha, A, m, B, k, beta, C, m)
\end{verbatim}
Porting the code

Iterative algorithm: example of gpu code structure

\[ G_{i,i-1}^r = g^D_{i,i-1} H_{i,i-1} G_{i-1,i-1}^r \]

Example:

```c
 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

![Graph showing speedup of GPU vs CPU](graph.png)
Double precision speedup
Ampere 100 vs Intel Xeon Platinum 8168 (24 cores)

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

![Graph showing Speedup of GPU vs CPU](image)
Towards eXaScale

- **T0**
  - 50 Tbps
  - k,E grid
  - k,E grid
  - Improved SMP of inversion

- **T0+1y**
  - 932 Tbps
  - Reduced Memory
  - Single precision

- **T0+2y**
  - 466 Tbps
  - k,E grid
  - TF32 precision

- **T0+3y**
  - 270,000 Tbps
  - 2.5 nm
  - GPU TensorCores

- **Peak (Mean)**
  - 270,000 (50,000)

- **Tflops/s**
  - 50 (5)

- **9.2 nm**
  - 932 (184)

- **4.8 nm**
  - 466 (92)

- **2.5 nm**
  - 50 (5)
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.

```fortran
use iso_c_binding

type c_DNS
  complex(sp), allocatable :: val(:,:)
  type(c_ptr) :: d_addr
end type c_DNS

A%d_addr

extern "C" int cu_createMat(void **d_A, int bc)
{
  err = cudaMalloc(d_A, bc);
}
```

- The final code improves porting.
Conclusions

• We have ported the whole *libNEGF* 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)

![Graph showing CPU Node-level MPI vs openMP (Xeon Platinum 48 cores)](image)

Intel MKL
1800x1800 matrices
Interaction Self-Energies

\[ \Sigma_{L\mu,L\nu}(k_t, E) = \sum_{q_t \in BZ} I(k_t, q_t) \left| z_{L\mu} - z_{L\nu} \right| \left[ (N_q + 1)G_{L\mu,L\nu}^<(q_t, E + \omega_q) \right. \\
\left. + N_q G_{L\mu,L\nu}^<(q_t, E - \omega_q) \right] \]

Implementation with cuTENSOR

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