HPC-PHYS 勉強会
2025/Sep./3 @ RIKEN-CCS

芯のないモンテカルロ殻模型による原子核の第一原理計算とGPUコード開発

筑波大学　計算科学研究センター
清水　則孝

富岳NEXTに向けた原子核第一原理計算コードの開発

- 第一原理計算：陽子と中性子（まとめて核子と呼ぶ）を構成要素とした原子核を量子多体計算で解く
  - 殻模型計算 (配位混合計算、CI計算 とも呼ばれる)
  - 空間を調和振動子基底（or さらに洗練された基底）で展開して、無限大に広い空間へ外挿し、エネルギー値を得る。

- 富岳NEXT → NVIDIA社との協業
  - NVIDIA社のGPU (H100)向けの開発
  - PEGASUSスパコン(筑波大CCS)、MIYABI-Gスパコン(JCAHPC)でコード開発を進めている

Miyabi supercomputer

78.8 PFLOPS

NVIDIA H100 GPU x 1,120

筑波大CCS-東大ITC
最先端共同HPC基盤施設
（JCAHPC）

Pegasus supercomputer

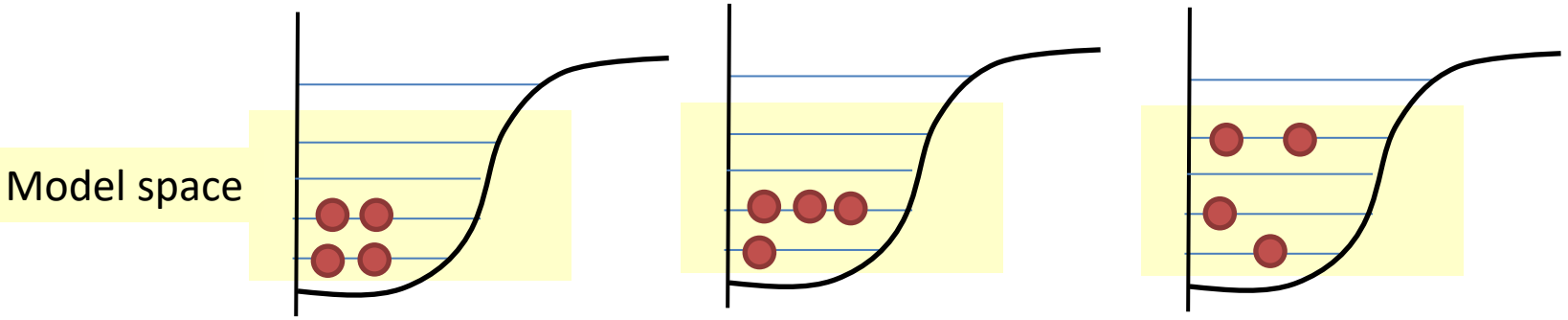
6.5PFLOPS
NVIDIA H100 GPU x 120
2TB persistent memory

筑波大学計算科学研究センター

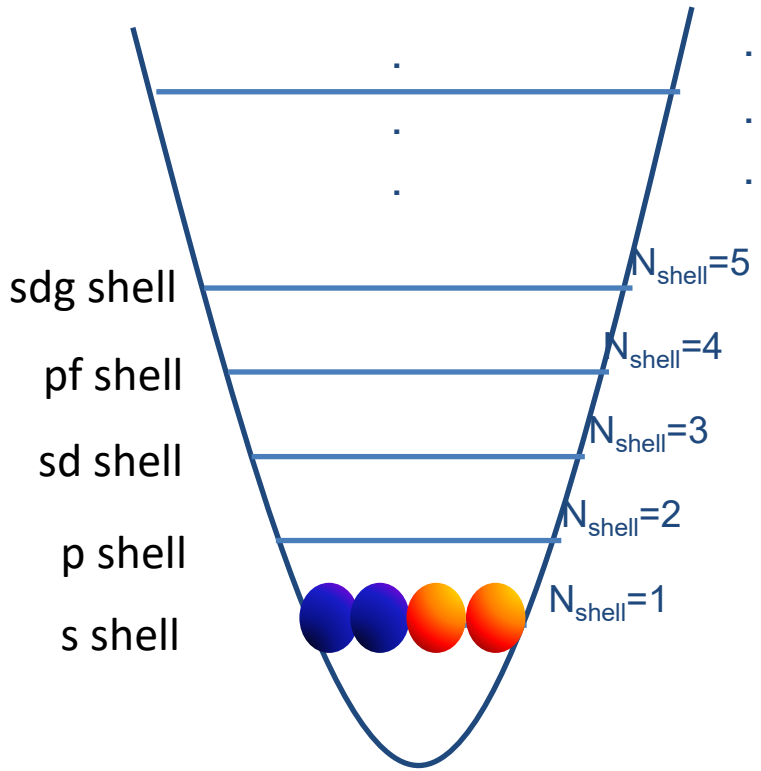
No-core shell-model calculations (FCI approach)

All nucleons are active particles in a certain “model space”.

Solve time-independent Schrodinger’s equation



$$|\Psi\rangle = v_1|m_1\rangle + v_2|m_2\rangle + v_3|m_3\rangle + \cdots$$



Model space with N_{shell} truncation

Solve Schrodinger’s Equation

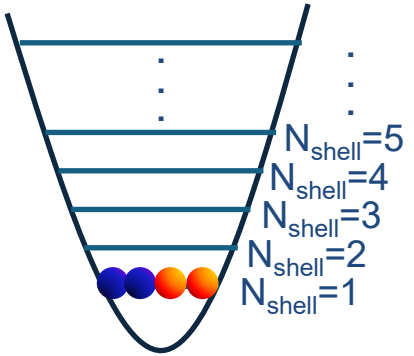
$$H|\Psi\rangle = E|\Psi\rangle \Rightarrow \sum_{m'}\langle m|H|m'\rangle\, v_{m'} = E v_m \qquad |\Psi\rangle = \sum_m v_m|m\rangle$$

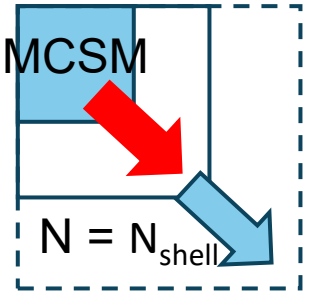
Pros: full many-body correlations inside the model space are included.

Cons: huge dimension of the Hamiltonian matrix.

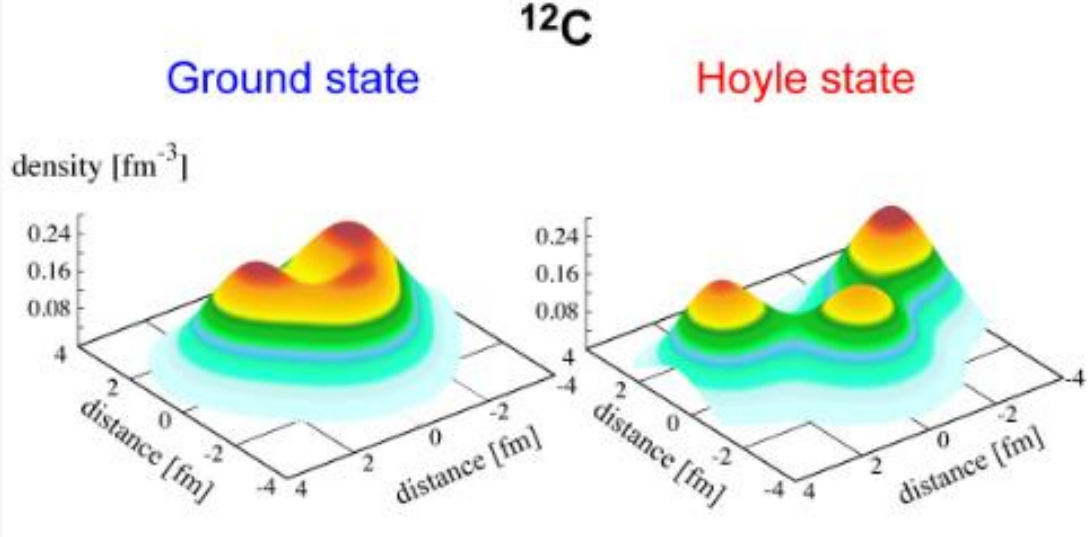
no-core Monte Carlo Shell Model (no-core MCSM)

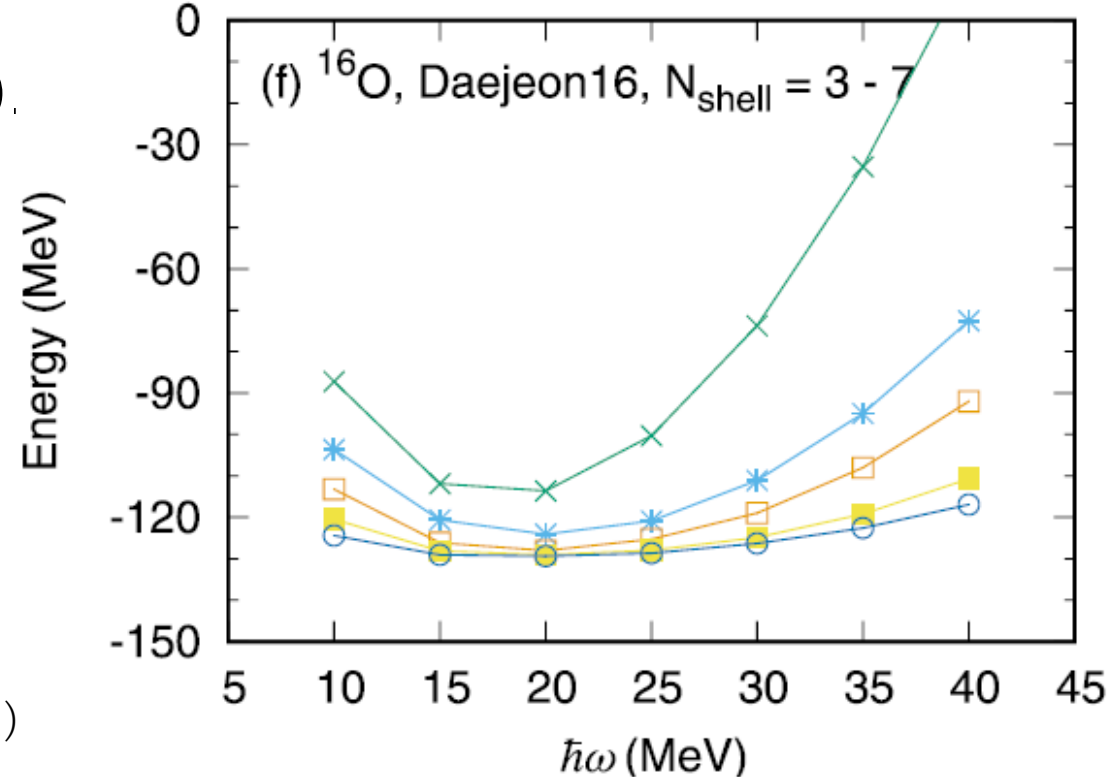
- Full configuration interaction (FCI) approach
- Harmonic Oscillator basis, Nshell=7 and extrapolation to Nshell→ ∞
- light nuclei up to ^{20}Ne
- Daejeon16 / JISP16 interaction (soft 2-body interaction with minimizing 3-body-force effect)





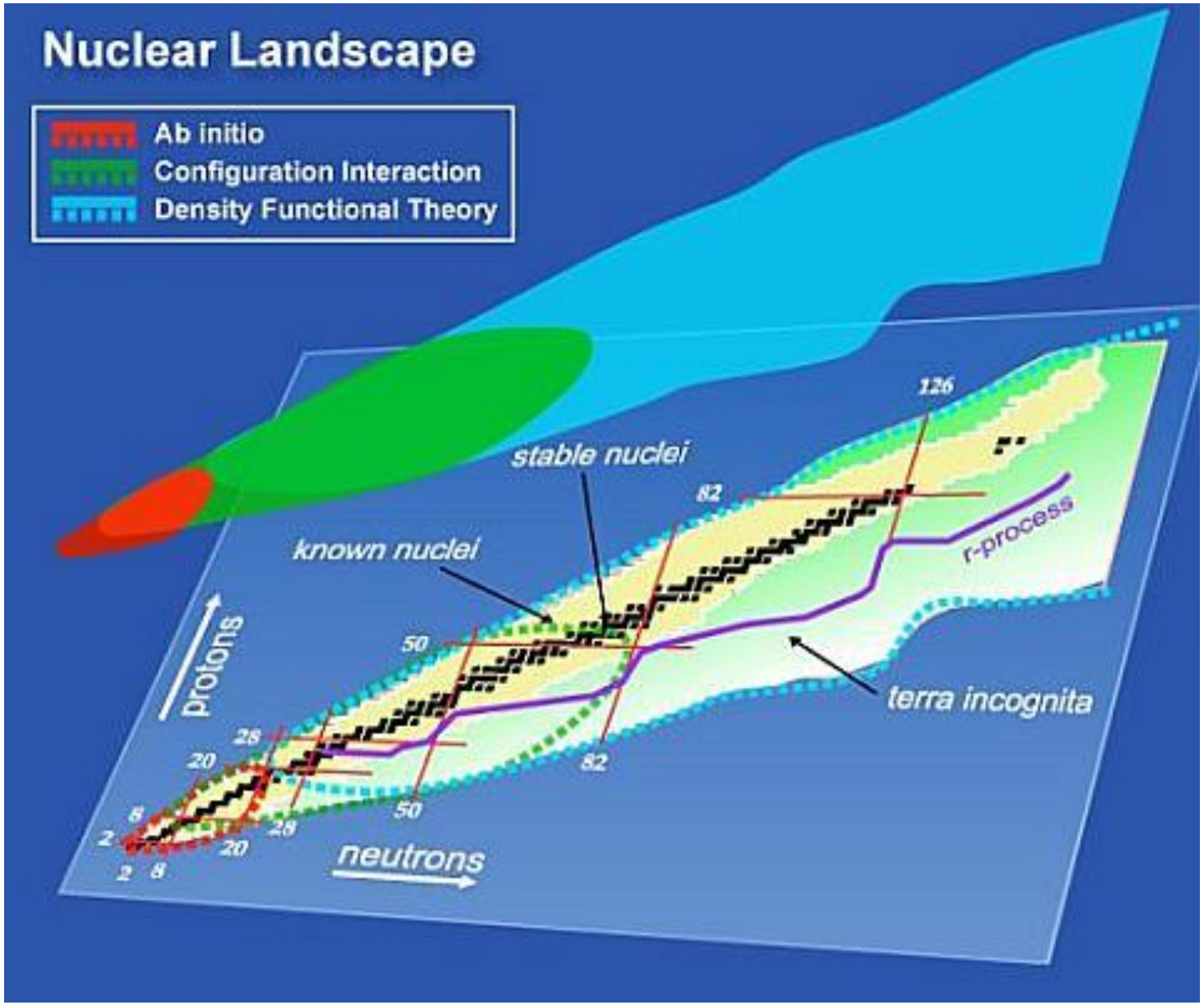
N = ∞





T. Abe, P. Maris *et al.,* Phys. Rev. C 104, 054315 (2021)
T. Otsuka, T. Abe *et al.,* Nat. Comm. 13, 2234 (2022)

核図表



芯を仮定しない殻模型計算は、最も信頼できる原子核の第一原理計算のひとつ。

この手法の適用領域は、これまで酸素16までに限られる。

これをカルシウム40までのsd殻核領域に広げたい。

Ref. UNEDF SciDAC collaboration

GPU対応コード実装方針

- 生のcudaコード開発はちょっと手が出ない

- nvfortran (24.9) + OpenACC を用いて実装
- 行列演算は可能な限りcuBLAS, 特にbatched GEMM を用いて高速化を図る

- PegasusとMiyabi-Gにてベンチマークテスト、プロダクションラン

モンテカルロ殻模型(Monte Carlo Shell Model)

- MCSMは、変分計算のひとつ（量子モンテカルロではない）
- 変形スレーター行列式（ハートリーフォックに用いられる形）を角運動量射影、パリティ射影、多基底を重ね合わせし、射影後のエネルギーを変分して最小化するようにDを定める。
- 基底数を増やしていくことで、「厳密解」に近づいていく。

MCSM 波動関数

J-projection

Parity projection

$$|\Psi\rangle = \sum_{i=1}^{N_b} \sum_{K=-J}^{J} f_{iK} P^J_{MK} P^\pi |\phi(D^{(i)})\rangle$$

$$|\phi(D)\rangle = \prod_k a_k^\dagger |-\rangle$$

$$a_k^\dagger = \sum_i D_{ik} c_i^\dagger$$

Superposition of multi basis

Slater determinants
form of Hartree-Fock wave function

エネルギー期待値$\langle\Psi|H|\Psi\rangle$を最小化するようにスレータ行列式 $|\phi(D^i)\rangle$ を求める。
N_b は 100~300程度

一般化固有値問題 (Hill-Wheeler equation)

$$\sum_{jK} H^J_{MKij} f_{jK} = E \sum_{jK} N^J_{MKij} f_{jK}$$

$$H^J_{MKij} = \langle \phi_i | H P^I_{MK} | \phi_j \rangle$$
$$N^J_{MKij} = \langle \phi_i | P^I_{MK} | \phi_j \rangle$$

変分計算なので、繰り返し解く。
ボトルネックはハミルトニアン行列要素$\langle \phi_i | H P^I_{MK} | \phi_j \rangle$の計算

ハミルトニアン行列要素とノルム行列要素

Slater determinant : $|\phi(D)\rangle = \prod_k a_k^\dagger |-\rangle \qquad a_k^\dagger = \sum_i D_{ik} c_i^\dagger$

Hamiltonian: $H = \sum_{l_1 l_2} t_{l_1 l_2} c_{l_1}^\dagger c_{l_2} + \frac{1}{4} \sum_{l_1 l_2 l_3 l_4} \bar{v}_{l_1 l_2, l_3 l_4} c_{l_1}^\dagger c_{l_2}^\dagger c_{l_4} c_{l_3}$

$$\left\langle \phi' \middle| H P_{MK}^I \middle| \phi \right\rangle = \sum_\Omega W_\Omega^{IMK} \mathcal{N}(\Omega) \left(\sum_{l_1 l_2} t_{l_1 l_2} \rho_{l_2 l_1}^{(\Omega)} + \frac{1}{2} \sum_{l_1 l_2 l_3 l_4} \rho_{l_3 l_1}^{(\Omega)} \bar{v}_{l_1 l_2, l_3 l_4} \rho_{l_4 l_2}^{(\Omega)} \right)$$

$$\mathcal{N}(\Omega) = \langle \phi' | R(\Omega) | \phi \rangle = \det\left(D_\Omega^\dagger D' \right) \qquad D_\Omega = R(\Omega) D$$

$$\rho_{kl}^{(\Omega)} = \frac{\left\langle \phi' \middle| c_l^\dagger c_k R(\Omega) \middle| \phi \right\rangle}{\langle \phi' | R(\Omega) | \phi \rangle} = D' \left(D_\Omega^\dagger D' \right)^{-1} D_\Omega^\dagger$$

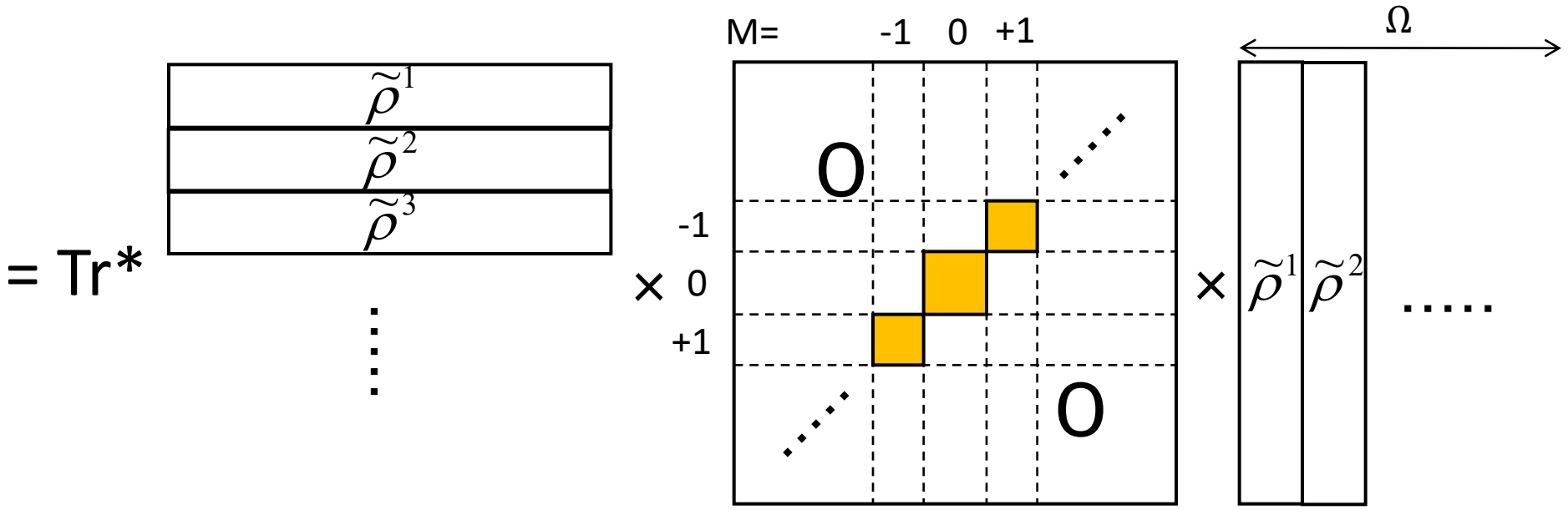
多数の小次元密行列(倍精度)の行列積、行列式、逆行列演算の組み合わせ

$\Omega \sim 60{,}000$

Efficient algorithm for evaluating Hamiltonian matrix elements in the MCSM

$\bar{v}_{l_1 l_2 l_3 l_4} = 0$ unless $m_{l_1} + m_{l_2} = m_{l_3} + m_{l_4}$

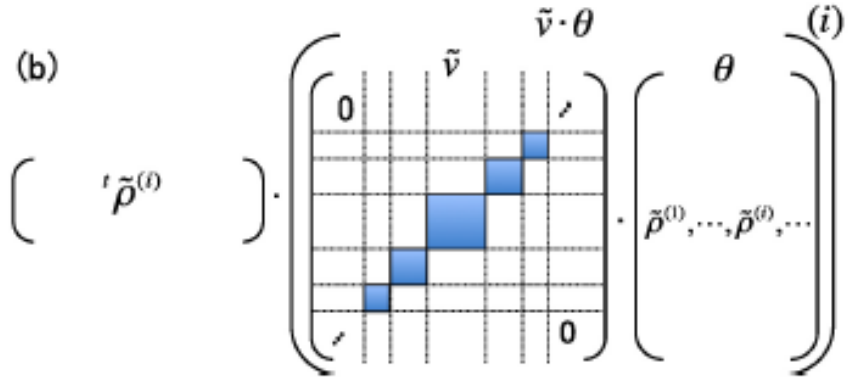
$$\langle \phi' | VP^I_{MK} | \phi \rangle = \sum_{\Omega} W^{IMK}_{\Omega} \mathcal{N}(\Omega) \rho^{(\Omega)}_{(l_3 l_1)} \frac{1}{2} \sum_{(l_4, l_2)} \bar{v}_{(l_1 l_3),(l_4 l_2)} \rho^{(\Omega)}_{(l_4 l_2)}$$



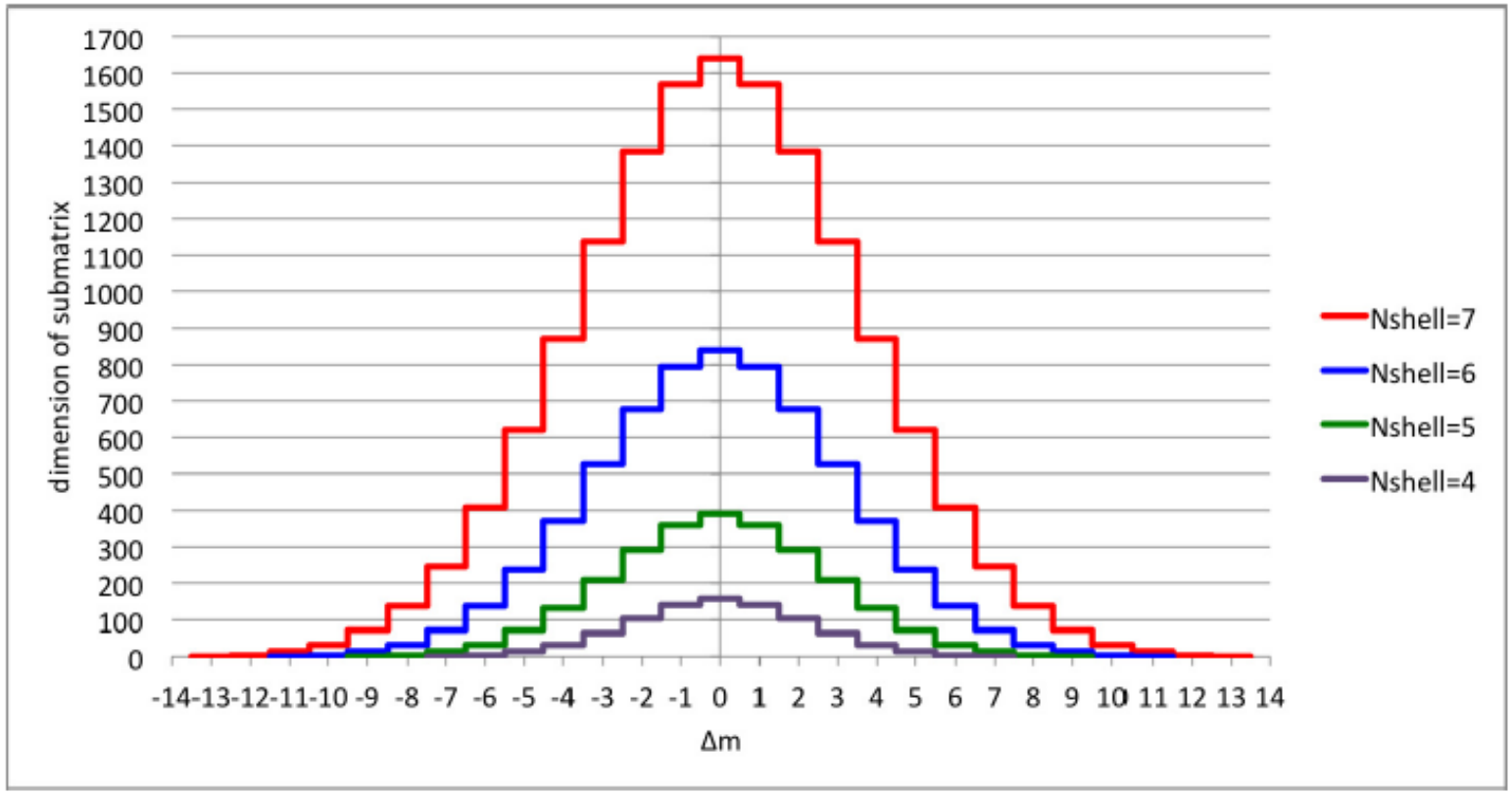
Products of a sparse matrix and a vector are replaced by the product of dense matrices

Matrix product is performed by DGEMM subroutine in BLAS level 3

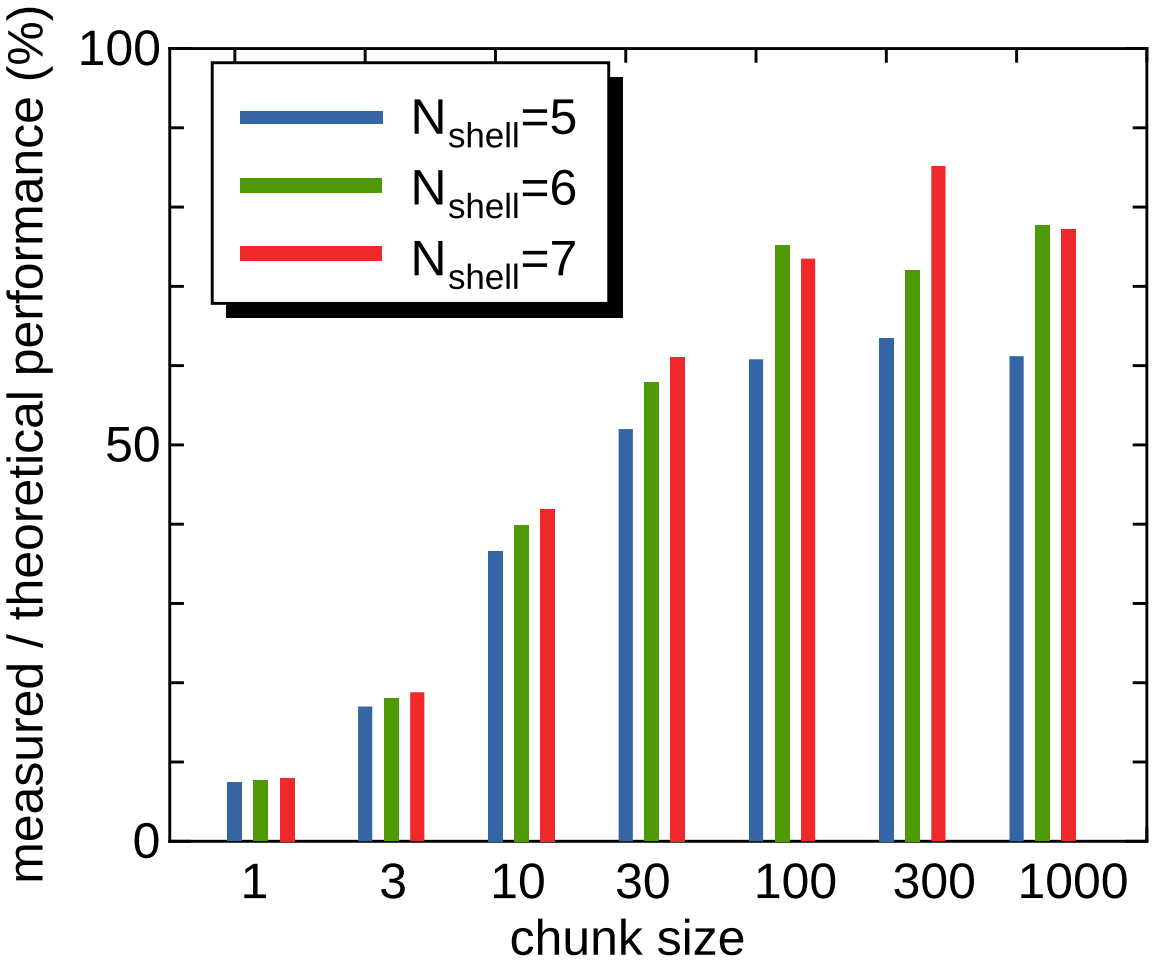
DGEMM/BLAS is always highly tuned for LINPACK Benchmark



$\bar{v}_{(l_1 l_3),(l_4 l_2)}$に含まれるブロック行列の次元数



Code tuning for Fugaku computer (bottle neck)



OpenACC

Insert “directive” for parallel computation on GPU like OpenMP.

In many case, the first trial would deaccelerate the code.
Many trials and errors to suppress the data transfer between CPU and GPU is required.

https://hpcworld.jp/archive/SPG/Pgi/OpenACC/004.html

cuBLAS

- BLAS library for Nvidia GPU
- product of dense matrices

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Memory access : $O(n^2)$
Floating-point operations : $O(n^3)$

small n : memory intensive
large n : computation intensive
⇒ tuned library for full performance!

Strided Batched GEMM performance

Many matrix products are performed in parallel.

$$C_{ij}^{(k)} = \sum_{k=1}^{n} A_{ik}^{(k)} B_{kj}^{(k)}$$

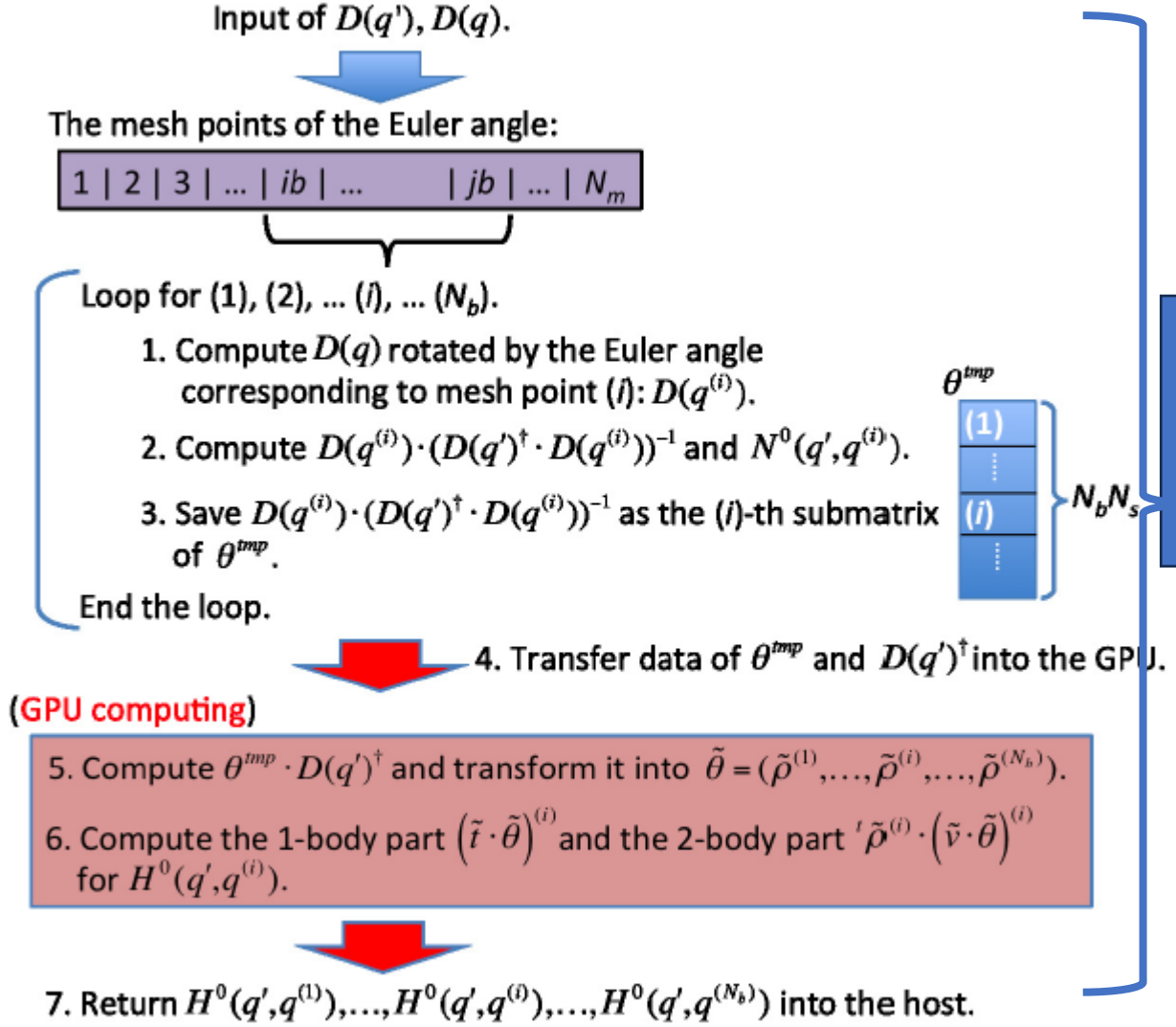
比較的小次元の密行列積でも性能が出せる。

多数の行列の逆行列を求めるサブルーチンも用意されている。

https://developer.nvidia.com/blog/cublas-strided-batched-matrix-multiply/

Togashi 2014

T. Togashi et al., Proc. Comp. Sci, 29, 1711 (2014)



N. B. no GPU library for the matrix inverse was available in 2014.

Present case

Transfer D and D' to GPU
$|\phi\rangle, |\phi'\rangle$

The whole computation of the projected matrix elements is computed on GPGPU using OpenACC + cuBLAS.

7. Return $H^0(q',q^{(1)}),\ldots,H^0(q',q^{(i)}),\ldots,H^0(q',q^{(N_b)})$ into the host.

$$\langle\phi'|HP^I_{MK}|\phi\rangle, \langle\phi'|P^I_{MK}|\phi\rangle,$$

Data transfer between GPU and CPU is minimized as far as possible.

allocate memory on GPU

data transfer

cuBLAS strided batched

OpenACC directive

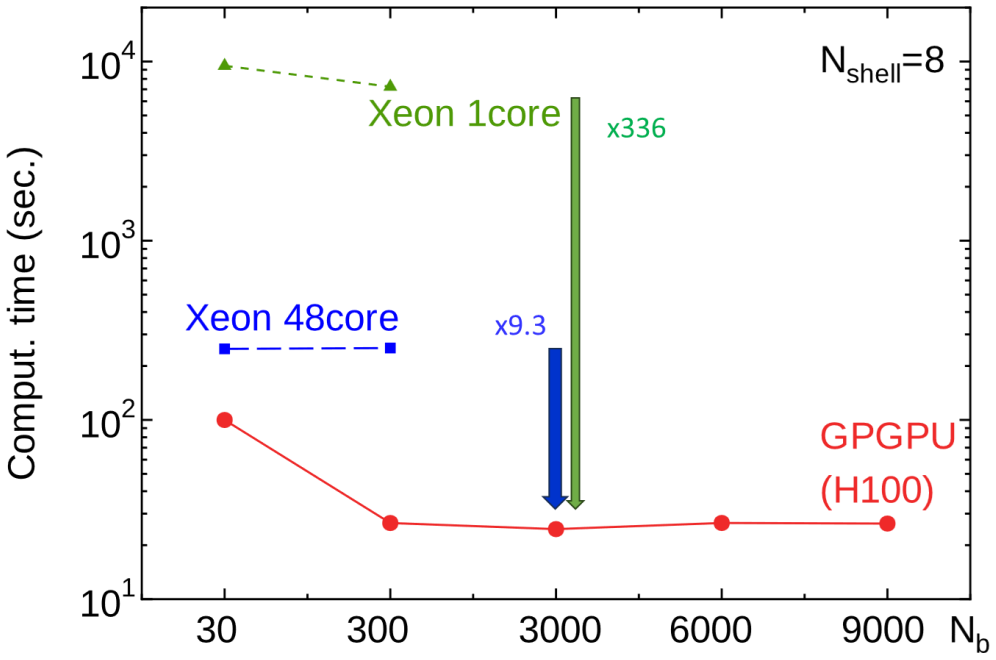
```
module gpu_tools
  use cudafor
  use cublas
  use iso_c_binding
  integer, allocatable, device :: abcp_task_d(:,:)
  complex(8), allocatable, device :: exp_ja_d(:,:), exp_jc_d(:,:)
  ...


  allocate( abcp_task_d( size(abcp_task_in,1), size(abcp_task_in,2) ) )
  abcp_task_d(:,:) = abcp_task_in(:,:)
  ...

  istat = cublasCreate(h_d)

  istat = cuBlasZgemmStridedBatched( h_d, cublas_op_n, cublas_op_n, nf, ns, nf, c1,&
       lwfdrwf1_d,            nf,        nf*nf, &
       lwfd1_d,               nf,        0,     cz, &
       tmp1_d,                nf,        nf*ns, mb )
  ...
  ...
  !$acc kernels deviceptr(r_d, a_d, ipvt_d)
  do k = 1, nbatch
     r_d(k) = (1d0, 0d0)
     do i = 1, n
        r_d(k) = r_d(k) * a_d(i, i, k)
        if (ipvt_d(i, k) /= i) r_d(k) = - r_d(k)
     end do
  end do
  !$acc end kernels
  ...

  istat = cublasDestroy(h_d)
```

additional ~800 lines for GPGPU

+ many “trials and errors” for tuning

MCSM code developments for GPU (bottle neck)

$$\langle \phi | H P^I_{MK} P^\pi | \phi' \rangle$$



tuned with OpenACC + cuBLAS library (batched BLAS)

performance / peak = 35.8 % (estimation)

Pegasus supercomputer
6.5PFLOPS @CCS U. Tsukuba
NVIDIA H100 PCIe 51TFLOPS

Miyabi-G supercomputer
78.8PFLOPS @Tsukuba-Tokyo JCAHPC
NVIDIA H100 NVL 66.9TFLOPS

The code development is almost completed. Ready for the production run on Miyabi-G.

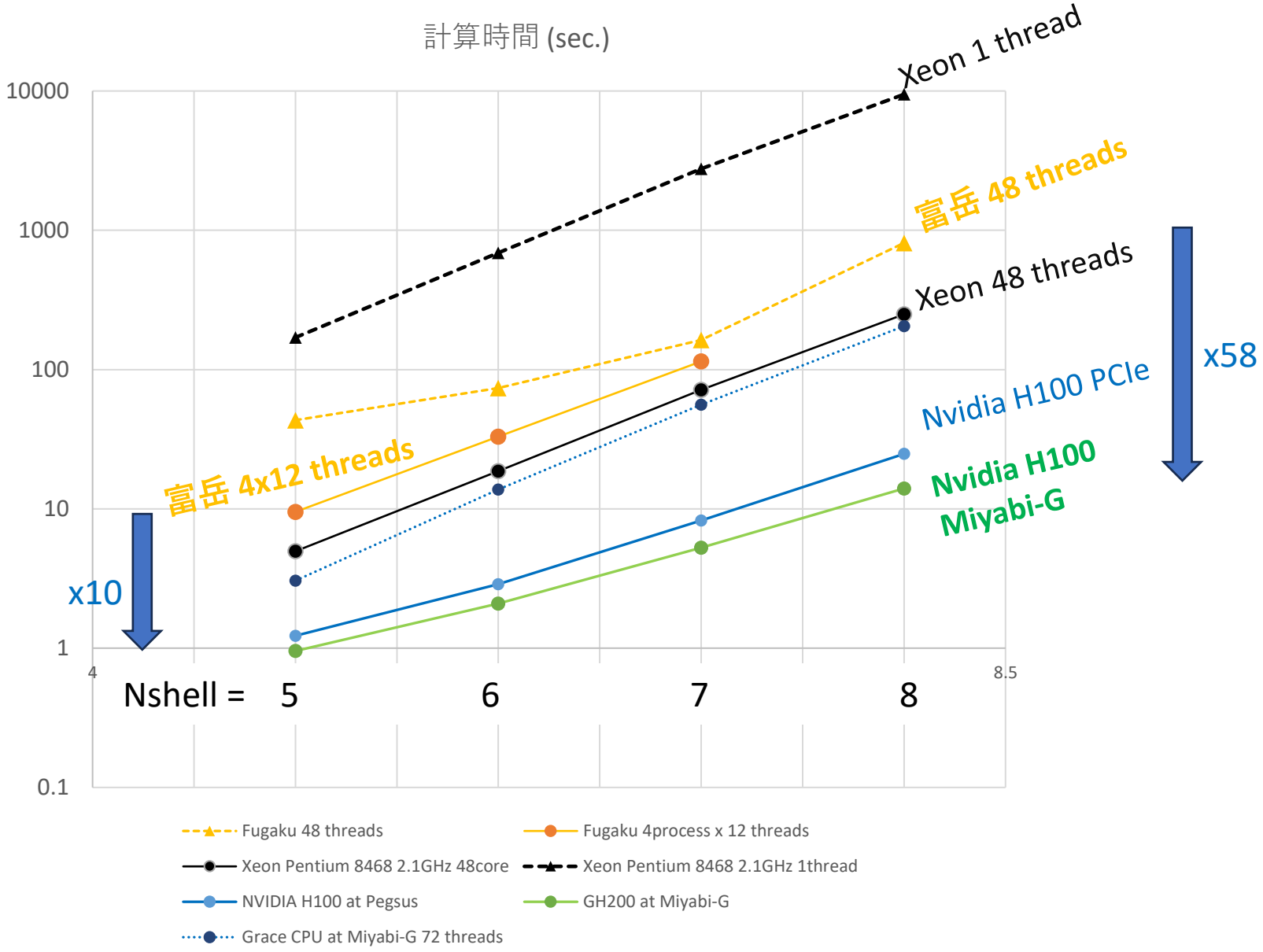
MCSM calc. with Nshell=8 becomes feasible!

“Natural orbital” technique and Chiral N3LO force (EM 1.8/2.0) with NO2B approximation were introduced by Miyagi-san.

ハミルトニアン行列要素計算時間のベンチマークテスト (1ノードあたり)

c.f. 理論ピーク性能

富岳 A64FX	3.1 TFLOPS
Xeon P8468	3.2 TFLOPS
Nvidia H100 PCIe	51.0 TFLOPS
H100 @ Miyabi-G	66.9 TFLOPS



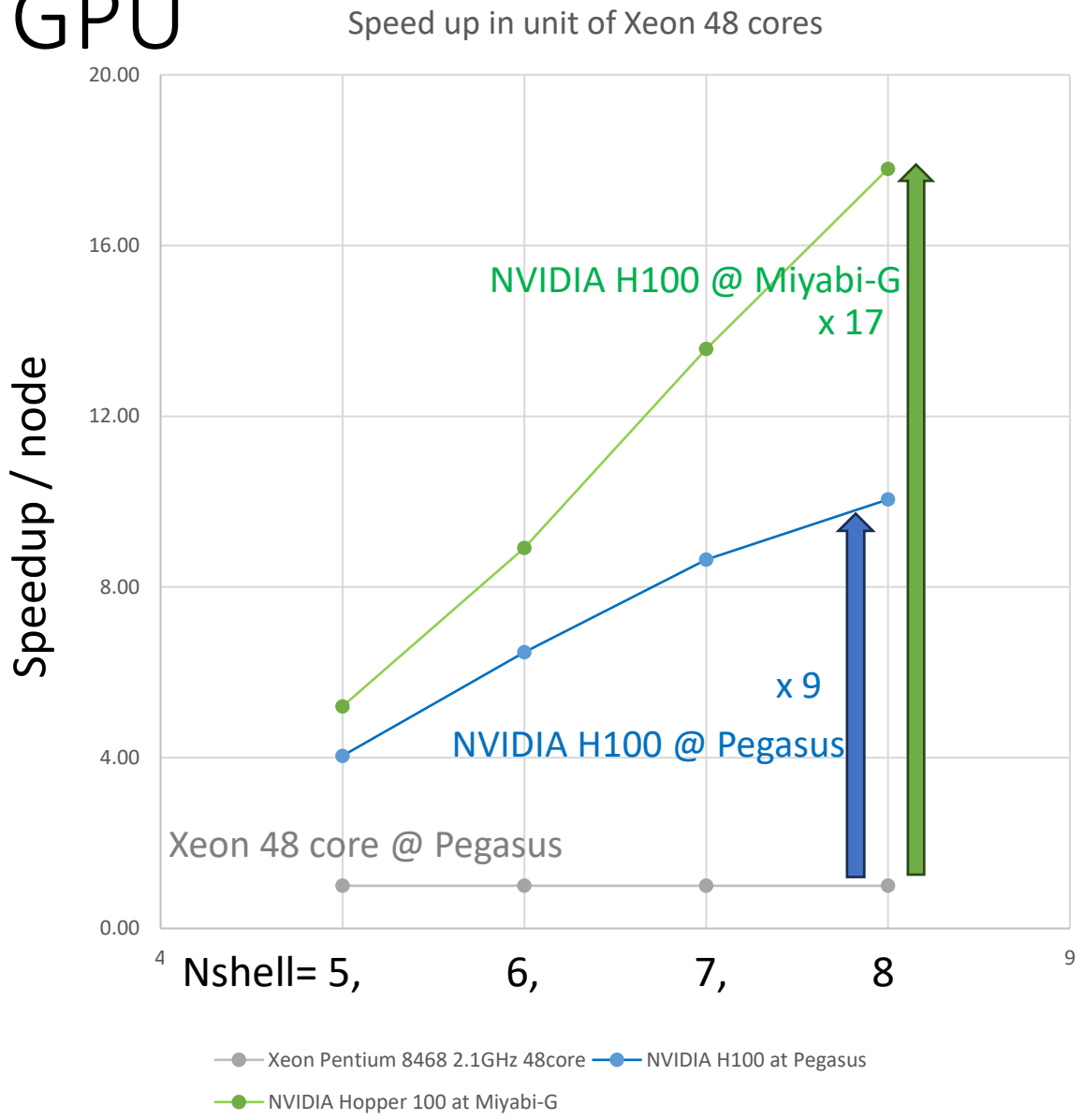
MCSM code developments for GPU

$$\langle\phi|HP^{I}_{MK}P^{\pi}|\phi'\rangle = \sum_{i=1}^{\sim 30{,}000} W_i\langle\phi|HR_i|\phi'\rangle$$

- tuned with OpenACC + cuBLAS library (batched BLAS)
- performance / peak ≃ 35 % (hot spot, estimation)

Pegasus supercomputer
6.5PFLOPS @CCS U. Tsukuba
NVIDIA H100 PCIe 51TFLOPS

Miyabi-G supercomputer
78.8PFLOPS @Tsukuba-Tokyo JCAHPC
NVIDIA H100 NVL 66.9TFLOPS



チューニング作業において個人的に悩んだ個所

```
!$acc kernels deviceptr(s, vrho_d, vyd_d)
!$acc loop independent
do kb = 1, mb                ! mb = 1000
   do i = 1, ld_vyd          ! ld_vyd = 115200 (Nshell=8)
      s(1,kb) = s(1,kb) + vrho_d(i,1,kb) * vyd_d(i,1,kb) &
        &               - vrho_d(i,2,kb) * vyd_d(i,2,kb)
      s(2,kb) = s(2,kb) + vrho_d(i,1,kb) * vyd_d(i,2,kb) &
        &               + vrho_d(i,2,kb) * vyd_d(i,1,kb)
   end do
end do
!$acc end kernels
```

多数回、複素数ベクトルの内積を求めている。（実部と虚部に分けて計算している）

$$s_k \leftarrow s_k + \sum_i v_{i,k} * y_{i,k}$$

Miyabi-G で 該当箇所一回あたり 0.54秒かかっており、とても遅い。
(ハミルトニアン行列要素計算時間の54%相当。)

試行錯誤の末に見つけた最適解

```
!$acc kernels deviceptr(s, vrho_d, vyd_d)
!$acc loop independent
do kb = 1, mb
   s(1,kb) = s(1,kb) +  sum( vrho_d(:,1,kb) * vyd_d(:,1,kb) &
     &                     - vrho_d(:,2,kb) * vyd_d(:,2,kb) )
   s(2,kb) = s(2,kb) +  sum( vrho_d(:,1,kb) * vyd_d(:,2,kb) &
     &                     + vrho_d(:,2,kb) * vyd_d(:,1,kb) )
end do
!$acc end kernels
```

ループ内部に sum() 関数を使うと、うまく内部も並列化してくれる。
0.54秒から、0.062秒に高速化 (8.7倍!)

複素数型に変換したり、batched BLASライブラリを使うとかえって遅くなる。
もっと速くできるという方は教えてください。（まだ計算時間全体の6%を占めている）

メモリー転送は 180GB/s 相当。 (NVIDIA H100 のメモリーバンド幅は 2TB/s)

別のチューニング例

```
do k = 1, n
  nmat_d(:,:) = nmat_d(:,:) + wl_d( :,:,k)
end do
```

$$n_{i,j} \leftarrow n_{i,j} + \sum_k w_{i,j,k}$$

```
zv1_d(:) = (1d0, 1d0)
call cublaszgemv( 'n', njbase**2, n, c1, wl_d, njbase**2, &
                  zv1_d, 1, c1, nmat_d, 1)
```

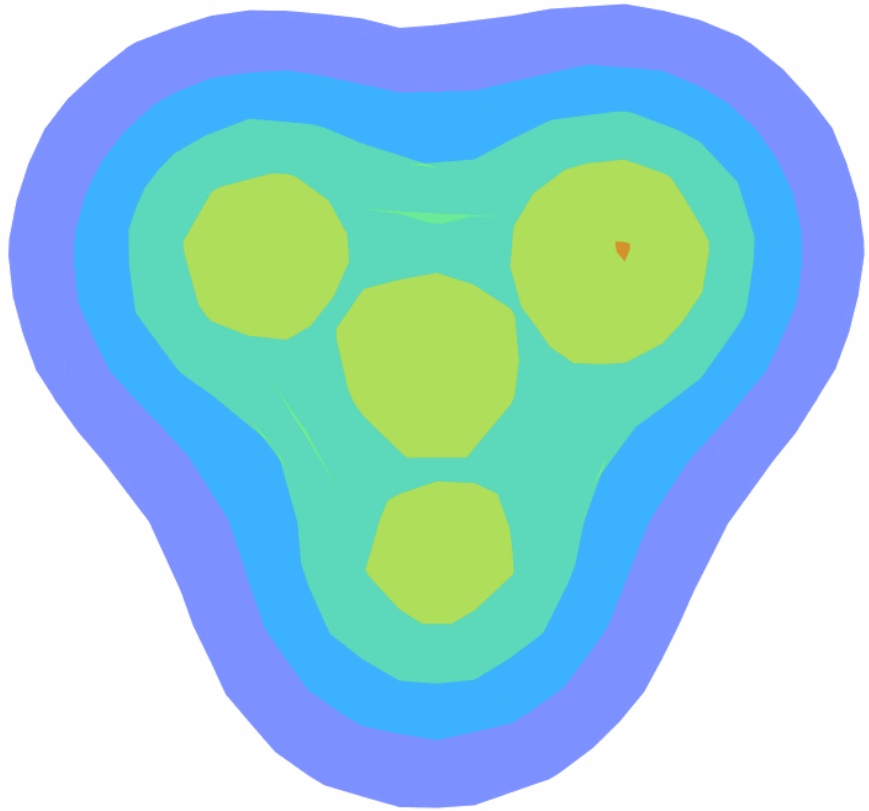
$$n_{i,j} \leftarrow n_{i,j} + \sum_k w_{i,j,k}\, v_k \qquad v_k = 1$$

要素１のベクトルを用意し、行列ベクトル積(cublaszgemv) を用いると高速化される

基底状態の密度分布

炭素12

酸素16



Summary

- 核子を構成要素とする原子核の第一原理計算に向けて、モンテカルロ殻模型法のGPU対応コード開発をおこなった。Nshell=8の計算を可能とした。
- 富岳NEXTでは、カイラル有効場理論による核力を用いて、Nshell=8、9での原子核の第一原理計算による研究が可能となると期待できる。
- cuBLASライブラリの利用、特にbatched GEMMライブラリが高速化に有用。batched 逆行列(getrf, getri)なども用意されており、充実してきた印象。
- ボトルネック部分のハミルトニアン行列要素計算を高速化し、Intel CPUに対して最大で17倍の高速化を達成した。

個人的な不満：（富岳ネクストで解決されていることを期待します！）

- OpenACCでは、手作業による試行錯誤のチューニングが必須。
- これまで計算時間を無視できていた他の物理量を計算する部分が、CPUでは計算できなくなるほど大きくなってきた。すべてのアルゴリズムをGPU化するのは大変。