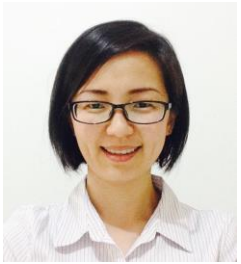
Large-Scale Materials Science Codes Porting Strategies on GPU Architectures, the BerkeleyGW Case Study

Mauro Del Ben

April 20, 2022

The Team

Mauro Del Ben



Charlene Yang

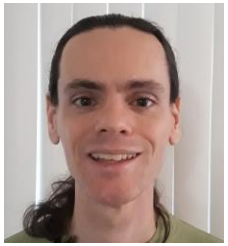
Zhenglu Li

Felipe H. da Jornada

Steven G. Louie

Jack Deslippe

William Huhn



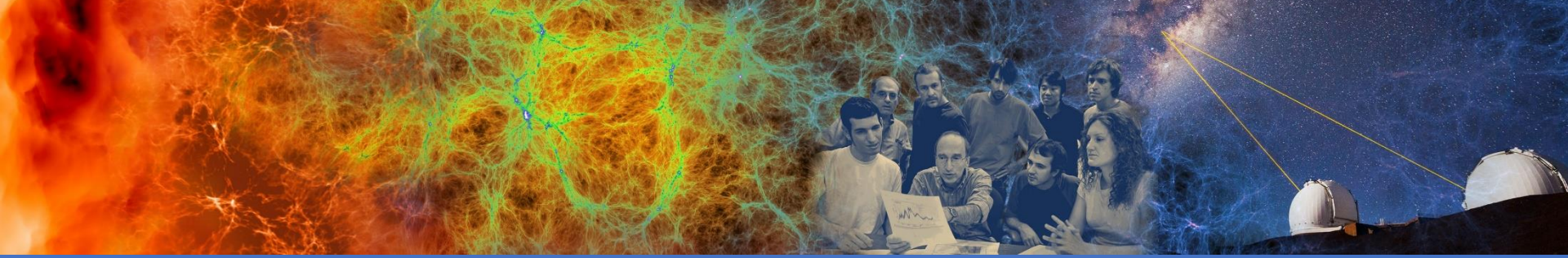
Phillip Thomas

"Accelerating Large-Scale Excited-State GW Calculations on Leadership Class HPC Systems" *in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20 No.4 pp.1 (2020)

ACM Gordon-Bell Finalist

Outline

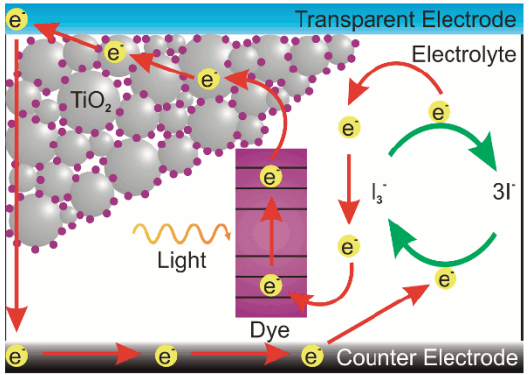
1. Motivation: Importance and Challenges
2. Performance portability strategy and implementation
3. Performance Results
4. Summary and Outlook



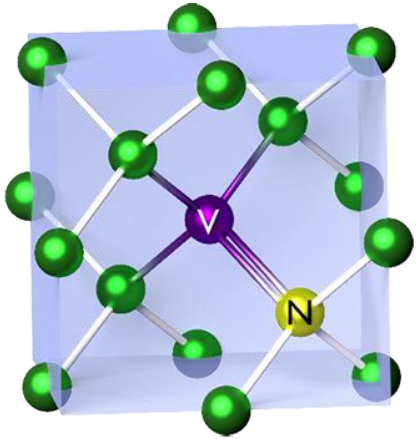
Motivation and Background

Material Science/Chemistry on the Path to Exascale

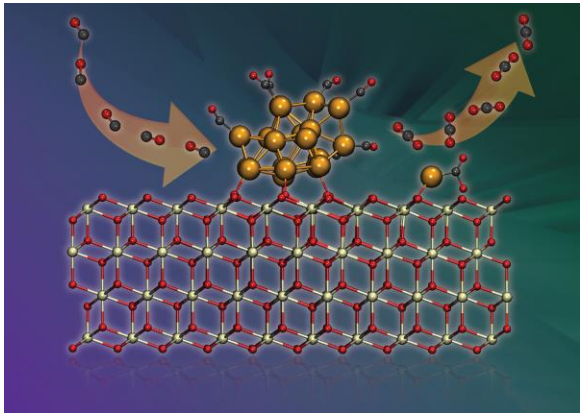
Grätzel cells: Oxide/Organic Interfaces



Cheap, reliable and sustainable **photovoltaics**



Defects in crystals: **qubits/quantum computers**
https://www.nist.gov/programs-projects/diamond-nv-center-magnetometry

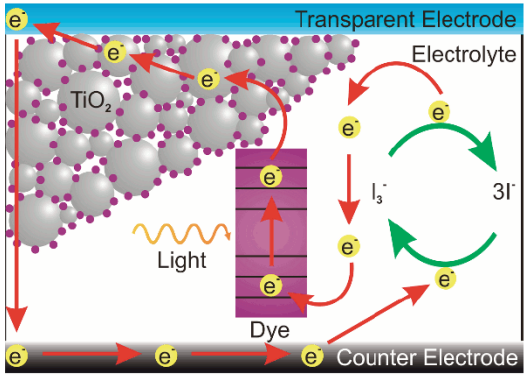


Chemical reaction at interfaces: **Catalysis**

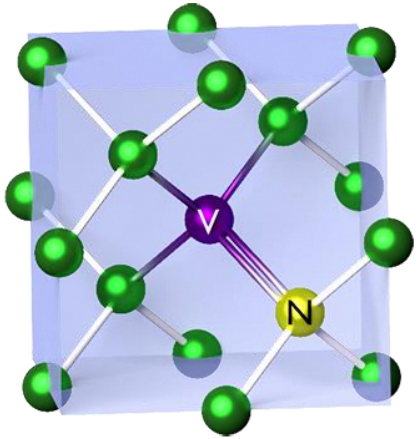
Mat. Sci & Chem apps, such as VASP, Quantum ESPRESSO, QMCPACK, NWchem, BerkeleyGW, CP2K, etc… **heavily use HPC facilities**

Material Science/Chemistry on the Path to Exascale

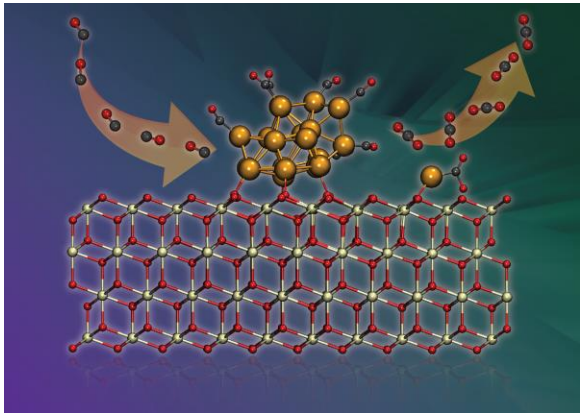
Grätzel cells: Oxide/Organic Interfaces



Cheap, reliable and sustainable **photovoltaics**



Defects in crystals: **qubits/quantum computers**
https://www.nist.gov/programs-projects/diamond-nv-center-magnetometry



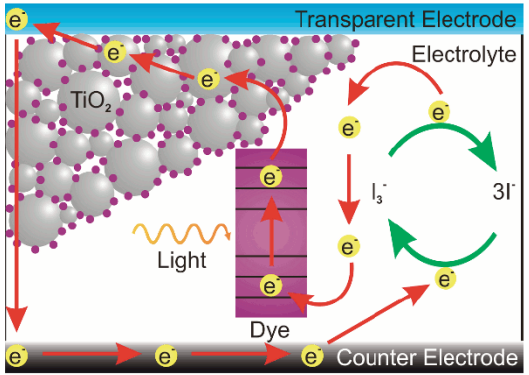
Chemical reaction at interfaces: **Catalysis**

Used to **study and understand the fundamental electronic properties of materials**: *necessary to design the components of novel devices*

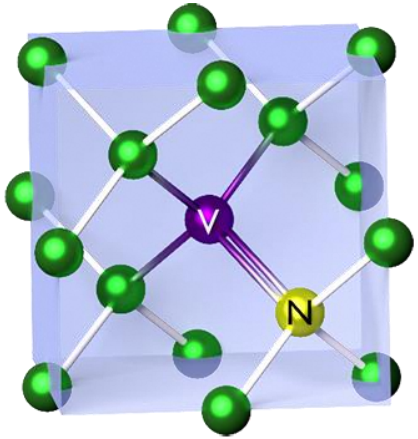
- Applications: Quantum Computers, Batteries, Photovoltaics, Catalysis, etc...

Material Science/Chemistry on the Path to Exascale

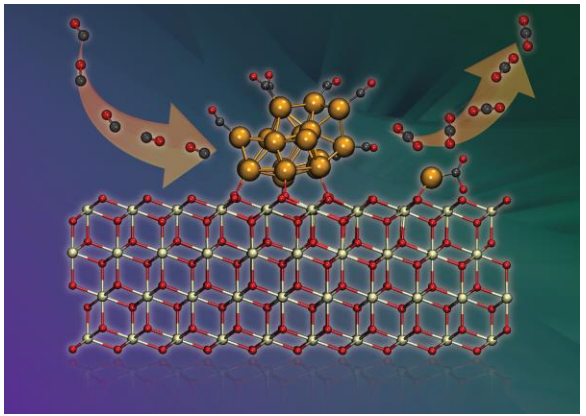
Grätzel cells: Oxide/Organic Interfaces



Cheap, reliable and sustainable **photovoltaics**



Defects in crystals: **qubits/quantum computers**
https://www.nist.gov/programs-projects/diamond-nv-center-magnetometry



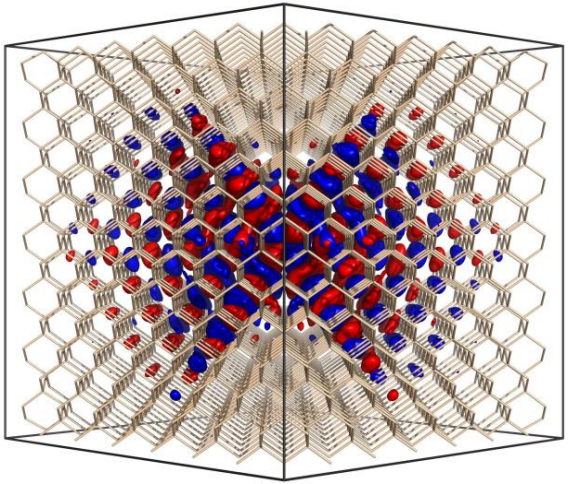
Chemical reaction at interfaces: **Catalysis**

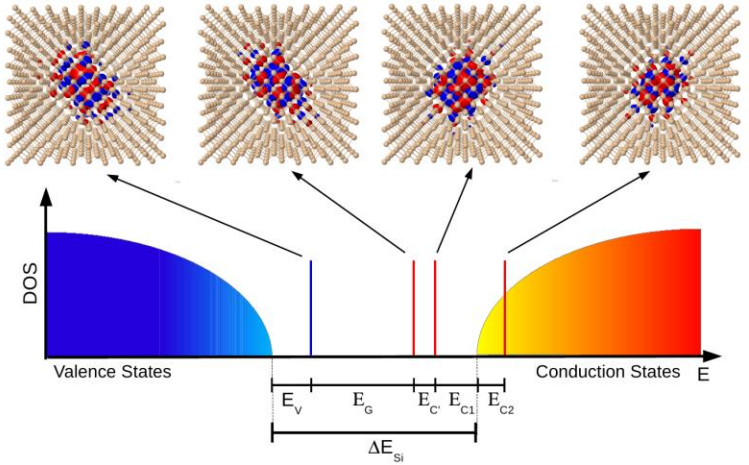
Density Functional Theory (DFT) the workhorse for over three decades

- Excellent compromise between accuracy and computational efficiency
- Ground state theory: often problematic for excited state phenomena

Excited State Properties of Complex Materials

Focus shift from ground to excited state properties





Example: Divacancy point defect in crystalline silicon, prototype of a solid-state Qubit

- Accuracy beyond DFT: **GW and GW+BSE**
- Unprecedented simulation sizes: **1000's of atoms**

The GW Method: State of the Art

The GW method represents one of the most effective and accurate approach to predict excited-state properties in a wide range of materials

Application of GW to thousands atoms systems still a challenge

Reduce time to solution and extend applicability:

- Develop methods to reduce prefactor and scaling with system size
- Improve performance of existing implementation

In this talk the various strategies employed to accelerate BerkeleyGW on GPU to achieve best performance will be discussed.

The BerkeleyGW Software Package

BerkeleyGW

A massively parallel software package to compute the electronic excited-state properties of materials via GW, Bethe-Salpeter equation (BSE) and beyond

The BerkeleyGW Software Package

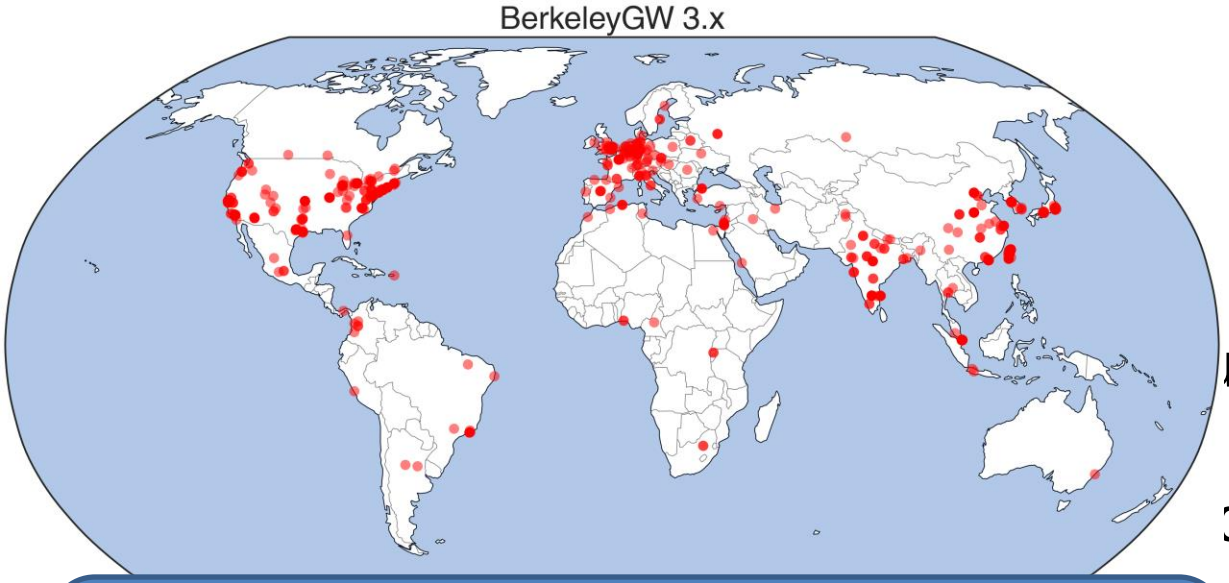
BerkeleyGW

A massively parallel software package to compute the electronic excited-state properties of materials via GW, Bethe-Salpeter equation (BSE) and beyond

Worldwide:

- Thousands of users
- Tens of active developers

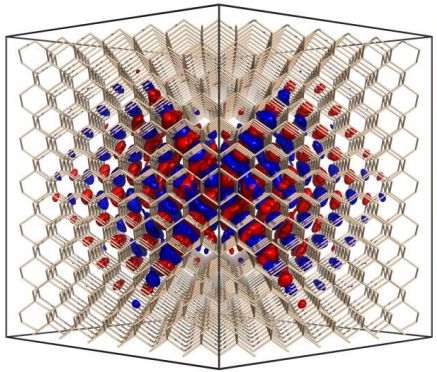
The BerkeleyGW Software Package

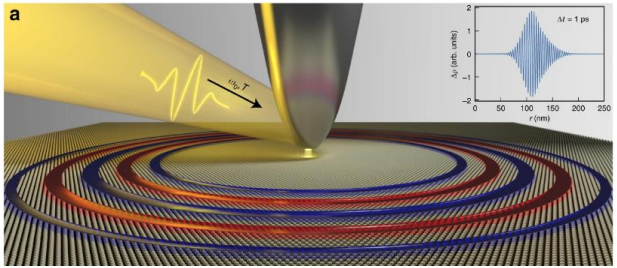


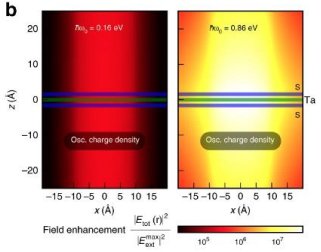
A ma ute the
electr a GW,
d

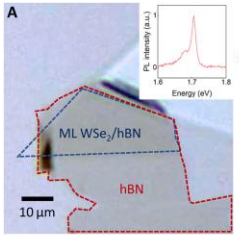
Worldwide download heatmap: more than 1,300 for 3.x since May 2021

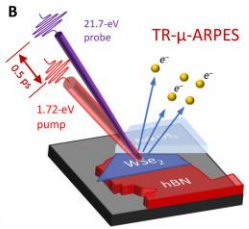
The BerkeleyGW Software Package: Highlights



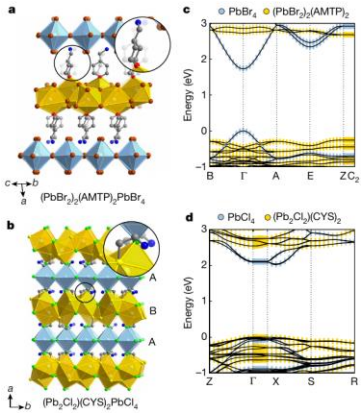






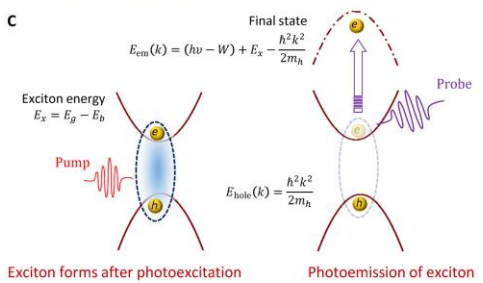


a Plasmon modes are excited in monolayer TaS_2 with an ultrafast laser pulse (energy $\hbar\omega_0 = 1.02$ eV, modulated with a Gaussian profile of width of T = 80



Recent BerkeleyGW highlights:

- Experimental measurement of the intrinsic excitonic wave function *Science Advances* **Vol. 7**, 17, (2021)
- Directed assembly of layered perovskite heterostructures as single crystals. *Nature* **597** 355-359 (2021)
- Universal slow plasmons and giant field enhancement in atomically thin quasi-two-dimensional metals *Nature Communications* **volume 11**, 1013 (2020)
- Large-Scale Excited-State GW Calculations on Leadership Class HPC Systems, In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, **art no. 4** SC '20. *Gordon Bell Finalist.*



The BerkeleyGW Software Package: General

Mostly written in Fortran 2003

Dependencies:

- Required: BLAS, LAPACK, FFTW2/3
- Recommended: ScaLAPACK, HDF5, ELPA, PRIMME

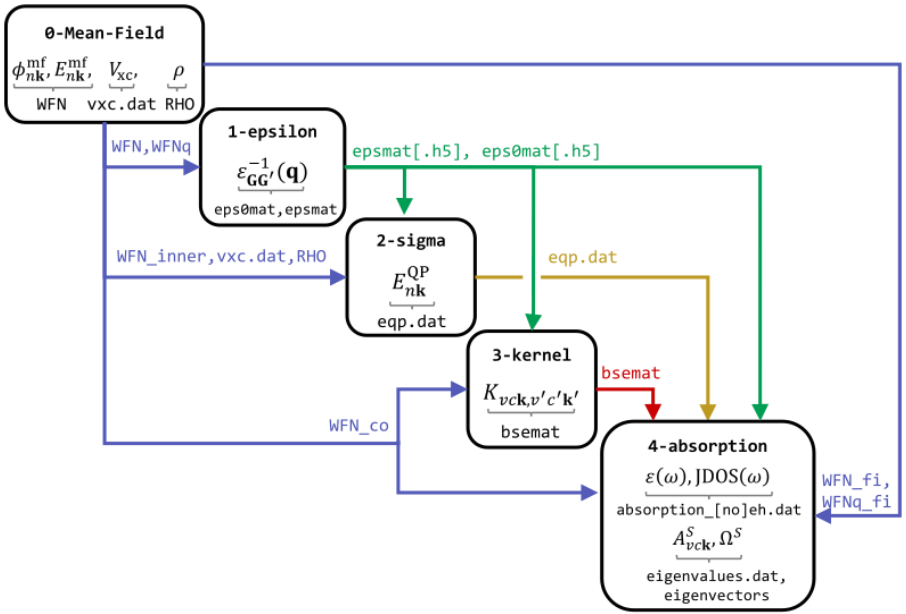
Parallel Programming Models:

- Multi-Node: MPI
- Muli-Core (CPU): OpenMP
- Accelerator: CUDA (separate branch), OpenACC/OpenMP-target (mainline)

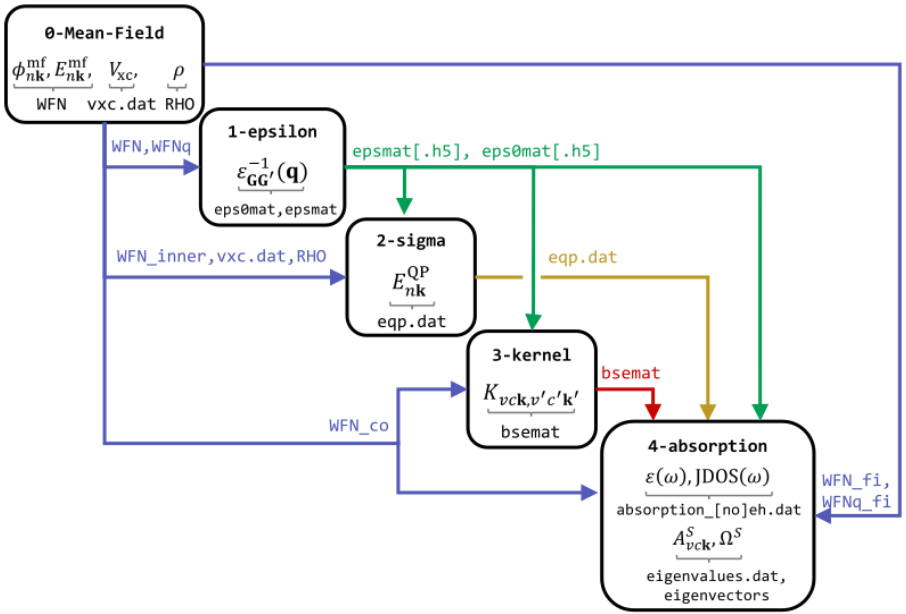
Common algorithmic motifs

- Large distributed matrix multiplications over short and fat matrices
- Distribute linear algebra: LU decomposition, triangular inversion, low rank approximations, eigendecomposition
- Node local FFTs and GEMMs
- large data reductions, tensor-like contractions

The BerkeleyGW Software Package: Workflow

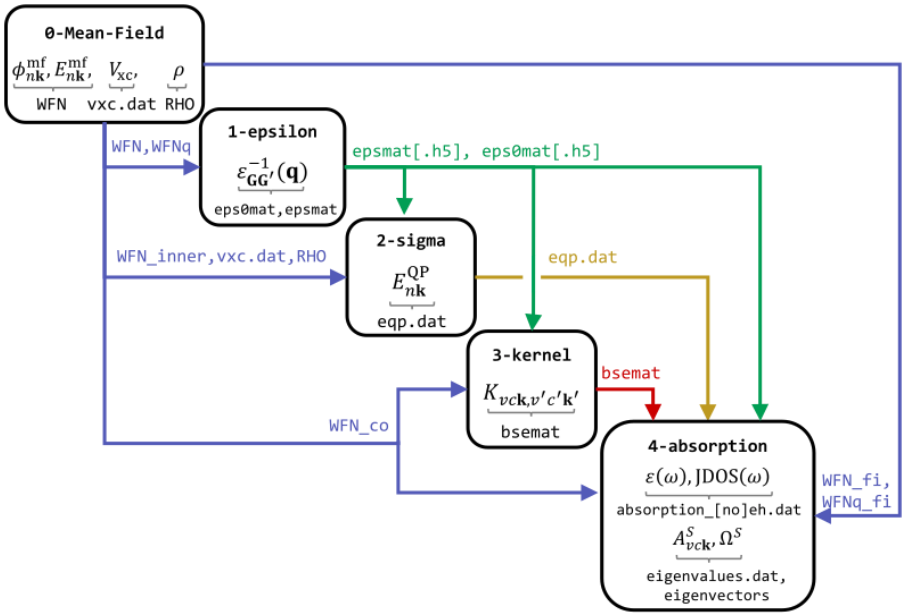


The BerkeleyGW Software Package: Workflow



Four major modules implementing the GW+BSE workflow

The BerkeleyGW Software Package: Workflow

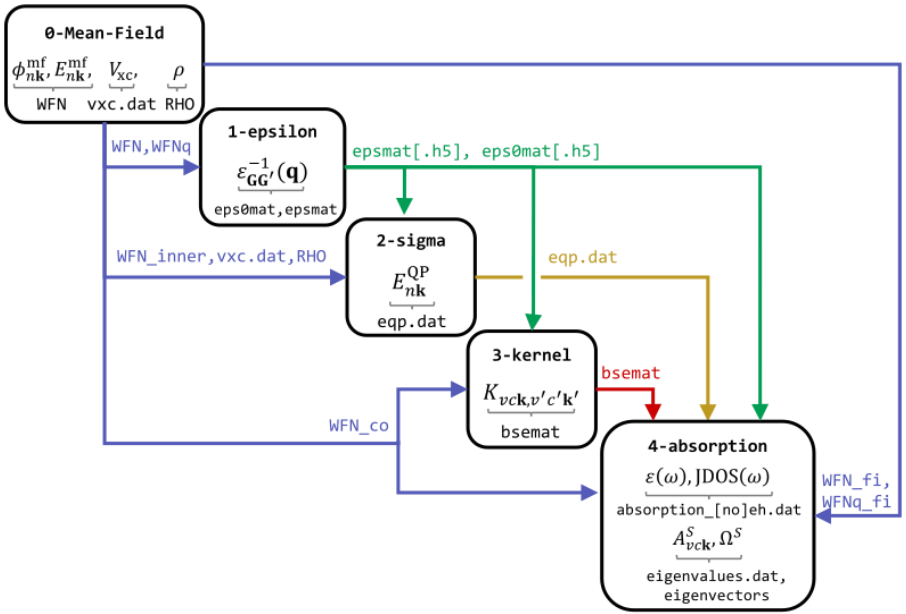


GW Workflow
epsilon + sigma

BSE Workflow
kernel + absorption

Four major modules implementing the GW+BSE workflow

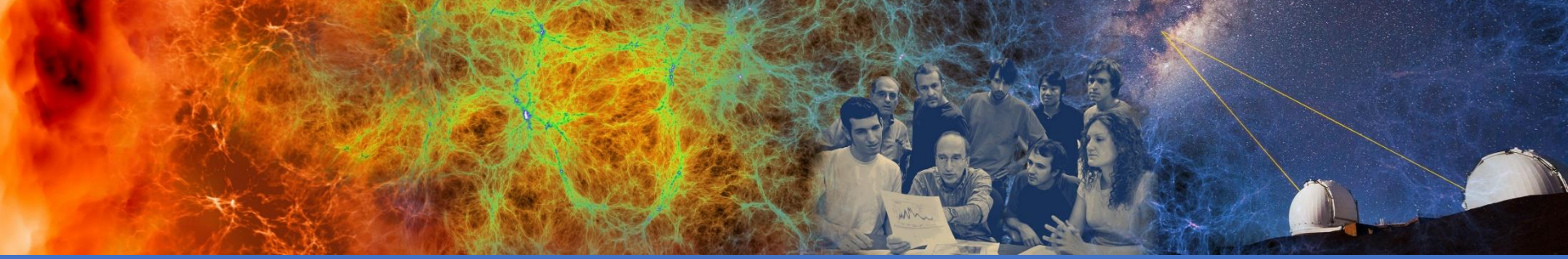
The BerkeleyGW Software Package: Workflow



GW Workflow
epsilon + sigma

BSE Workflow
kernel + absorption

Four major modules implementing the GW+BSE workflow



BerkeleyGW on the Path to ExaScale

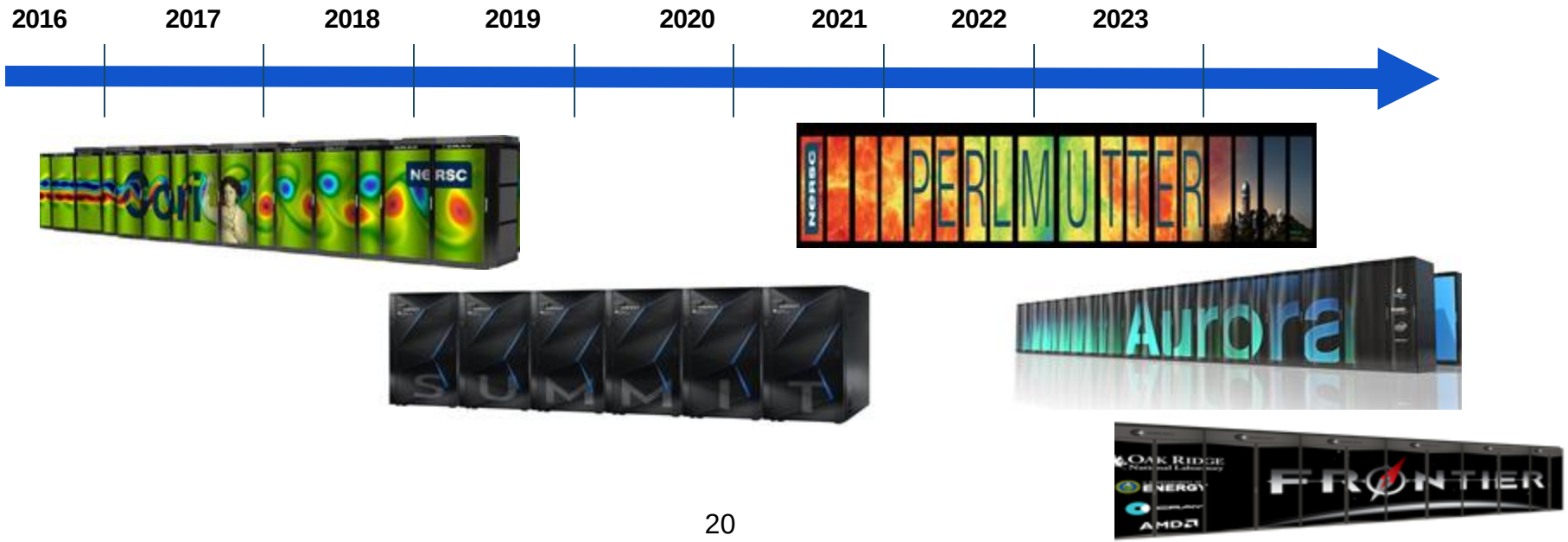
The Path to Exascale

Exascale for the DOE Office of Science Means: **GPU Accelerated Systems!**

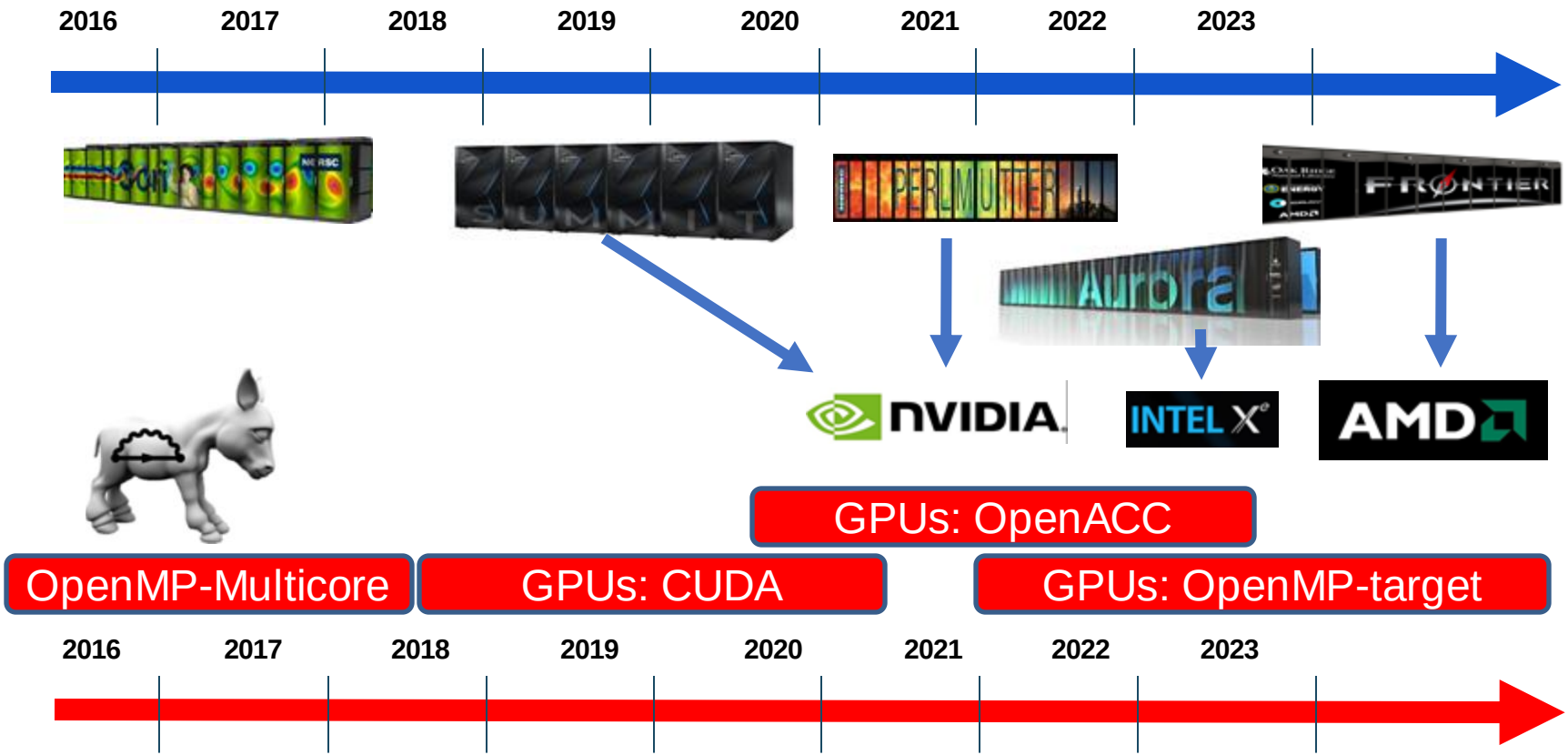
AMD

INTEL

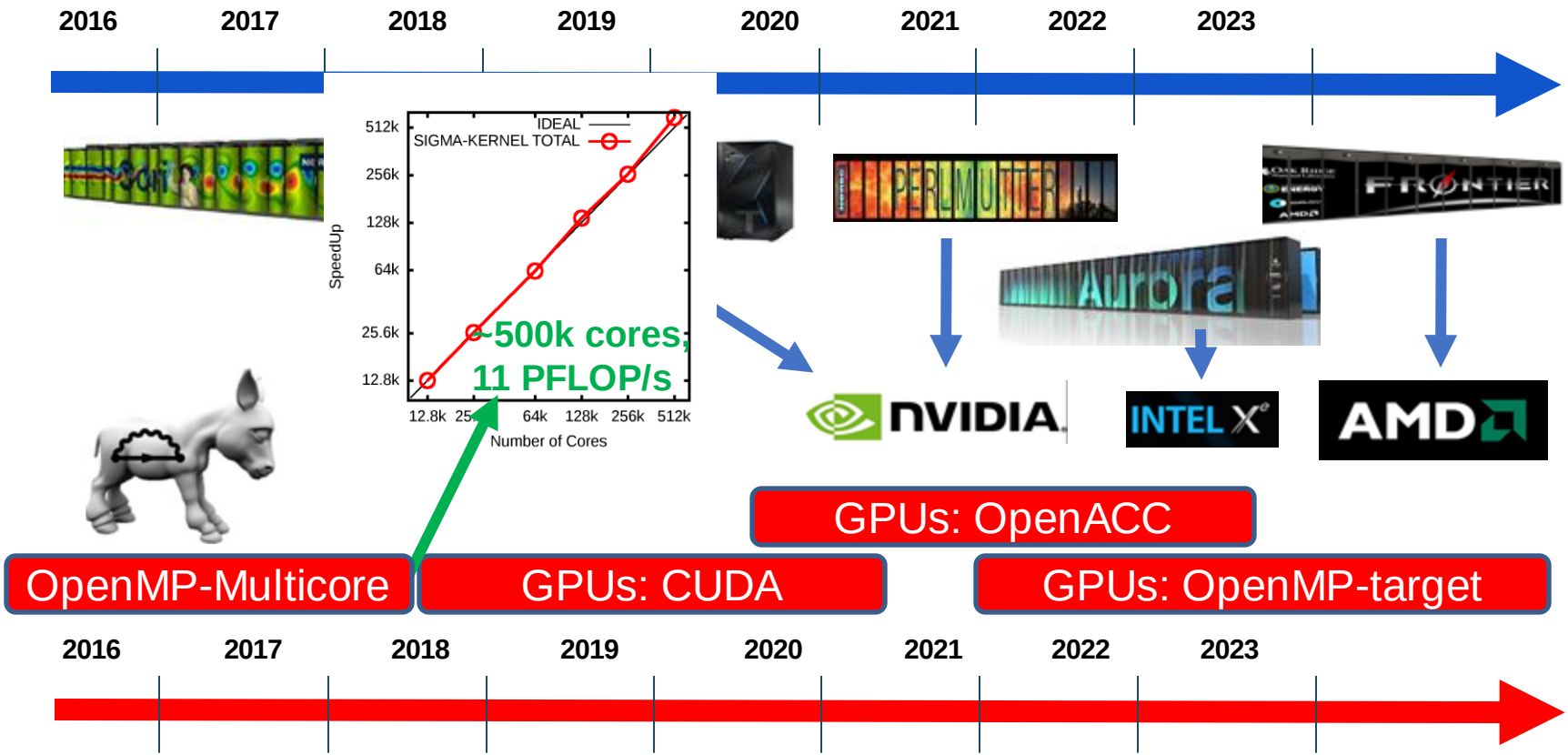
NVIDIA



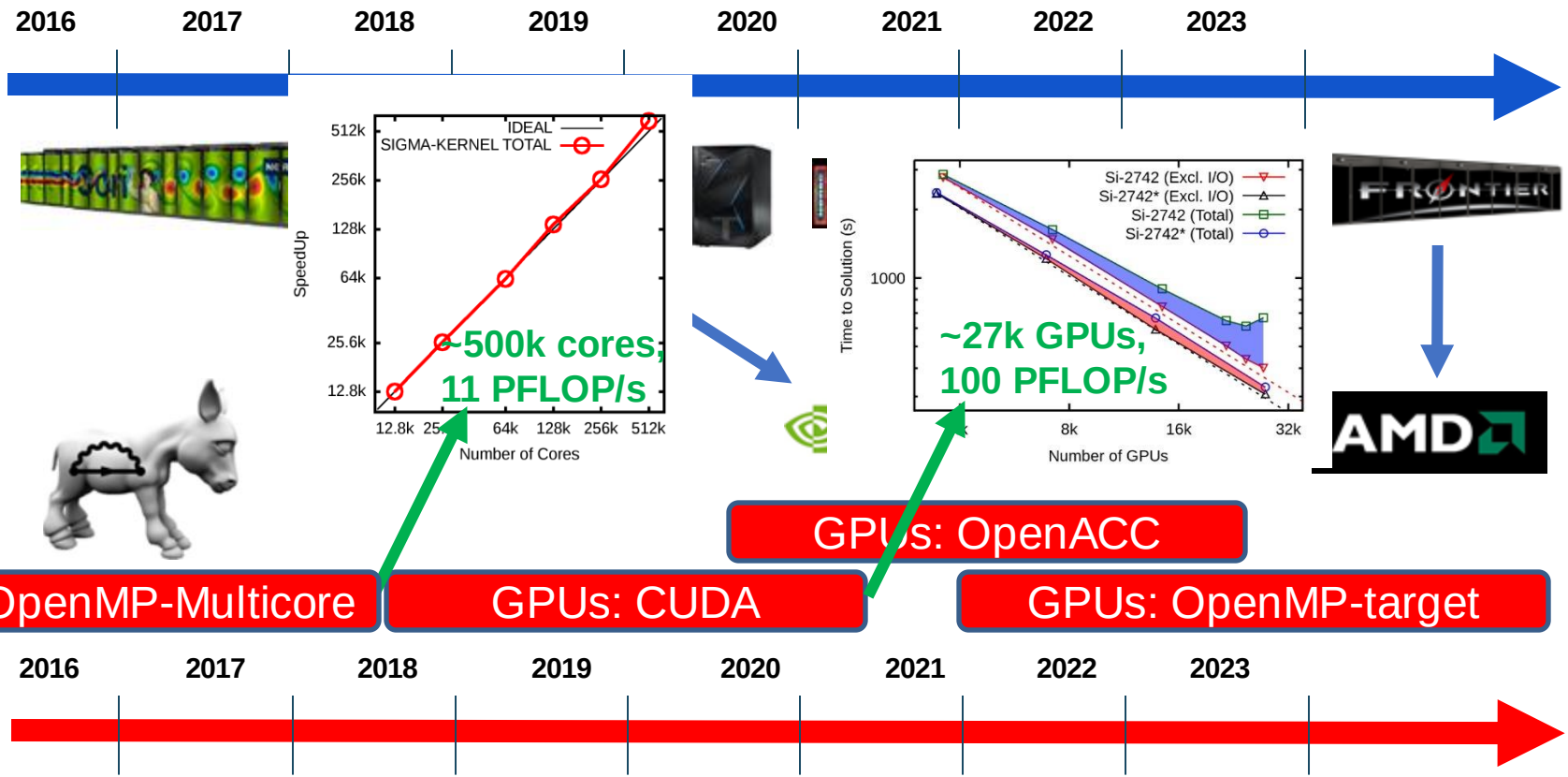
BerkeleyGW on the Path to Exascale



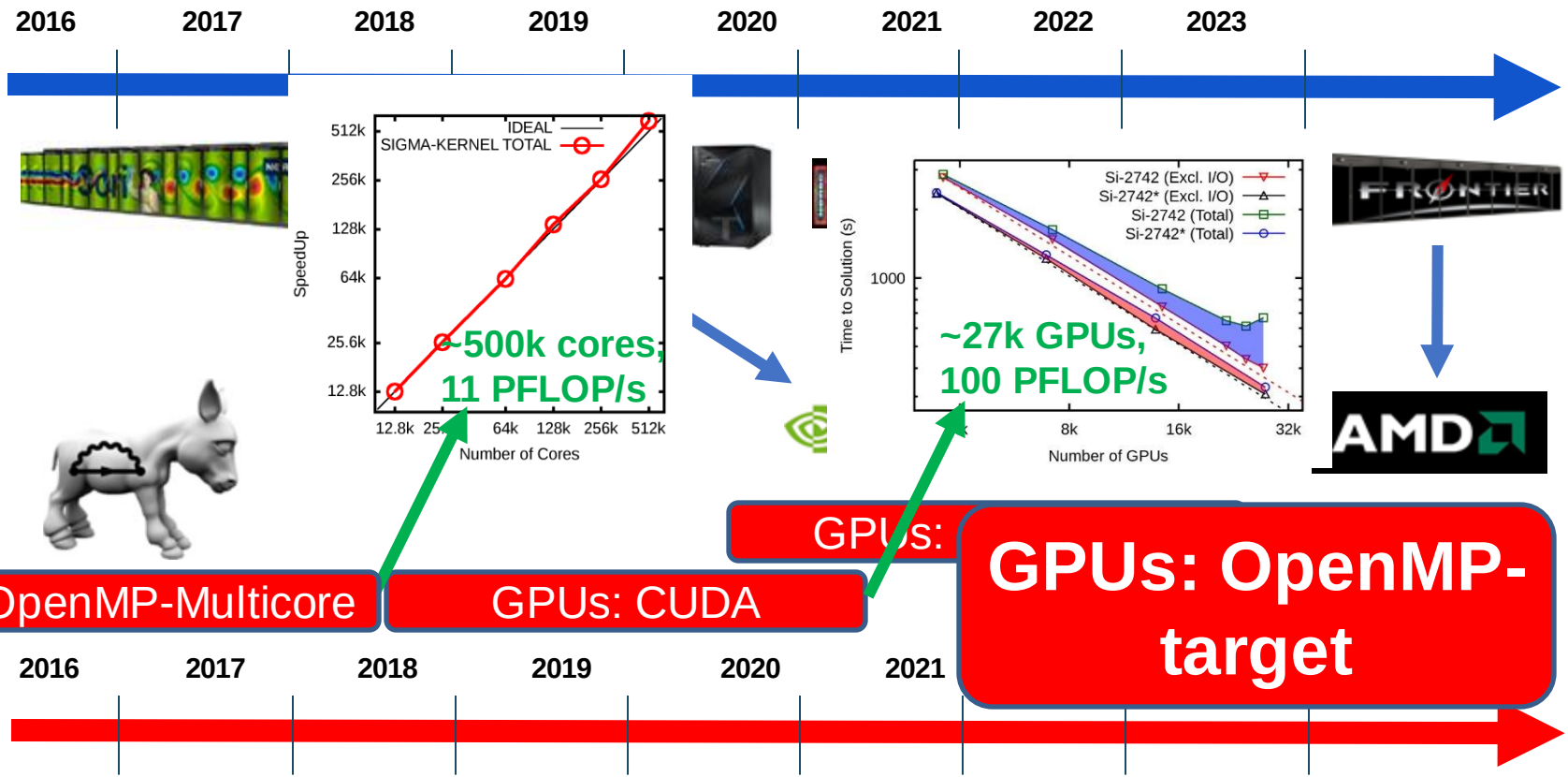
BerkeleyGW on the Path to Exascale



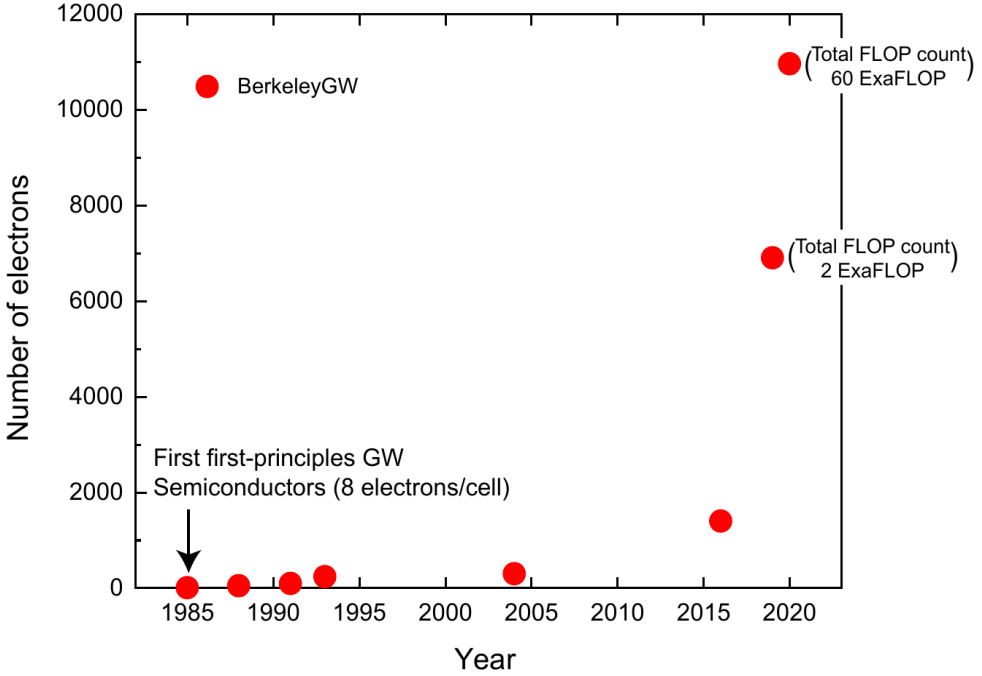
BerkeleyGW on the Path to Exascale

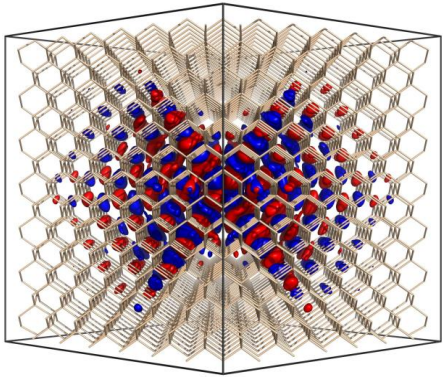


BerkeleyGW on the Path to Exascale



BerkeleyGW Overview Summary

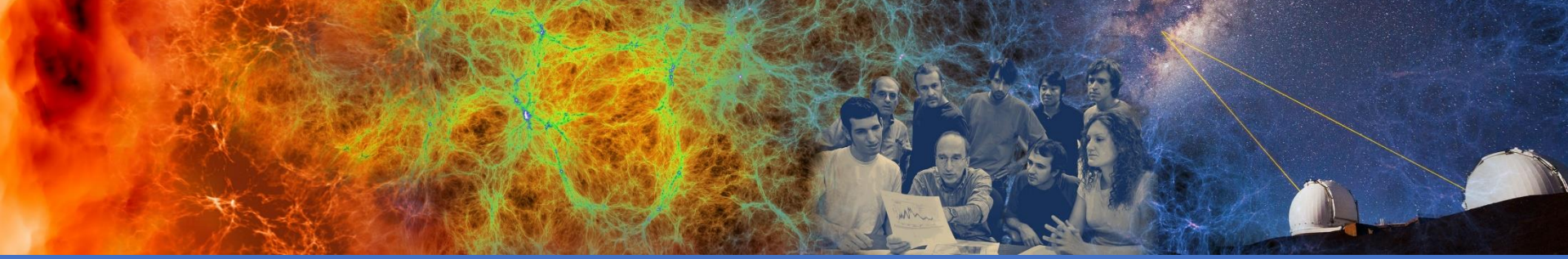




Divacancy defect in silicon
~11,000 electrons and over 2,700 atoms:
time to solution of the ~10s of mins.

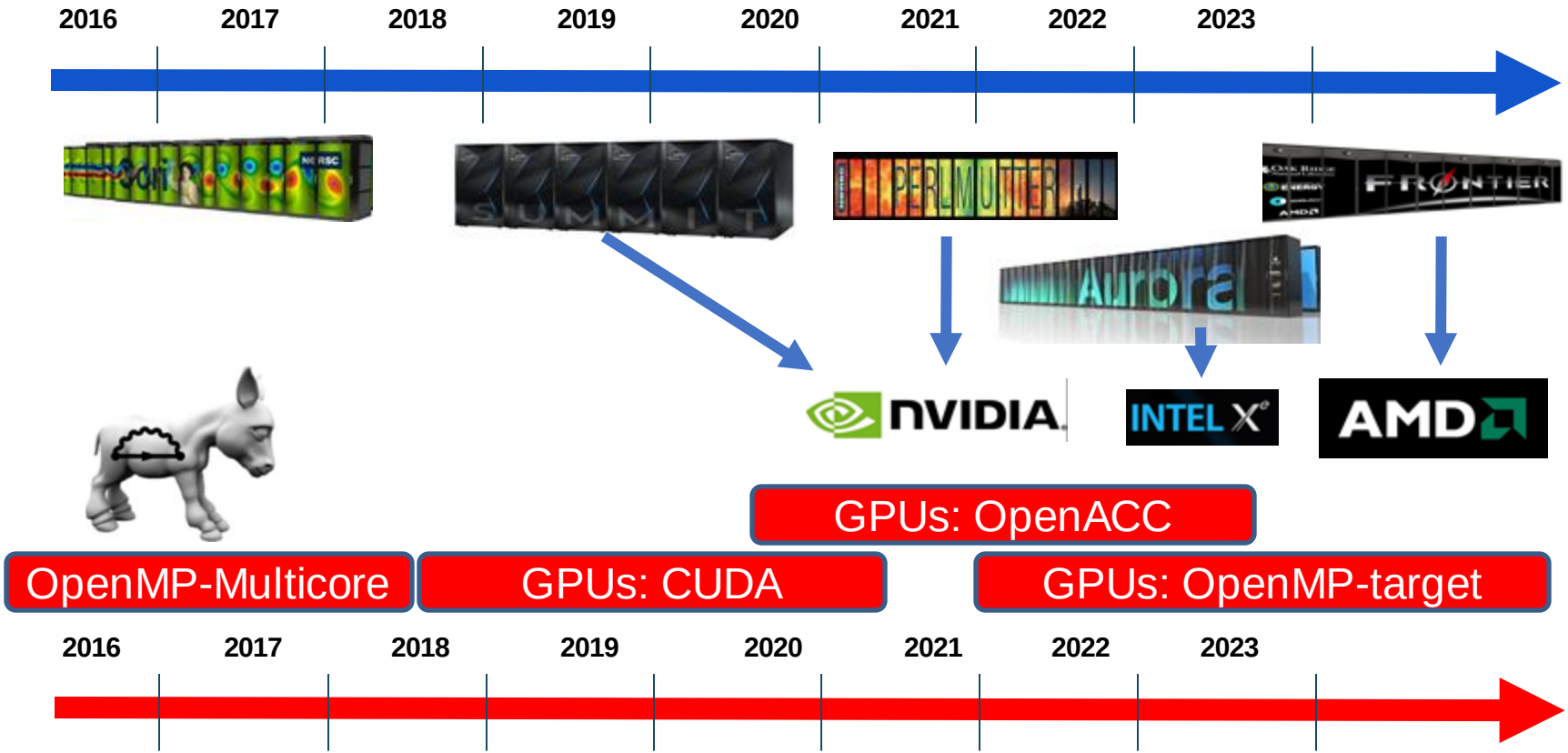
2020 ACM Gordon-Bell finalist:
https://dl.acm.org/doi/abs/10.5555/3433701.3433706

By software optimization of BerkeleyGW on leadership class HPC systems the application of GW to systems of increasing size closely follow the Moore's law

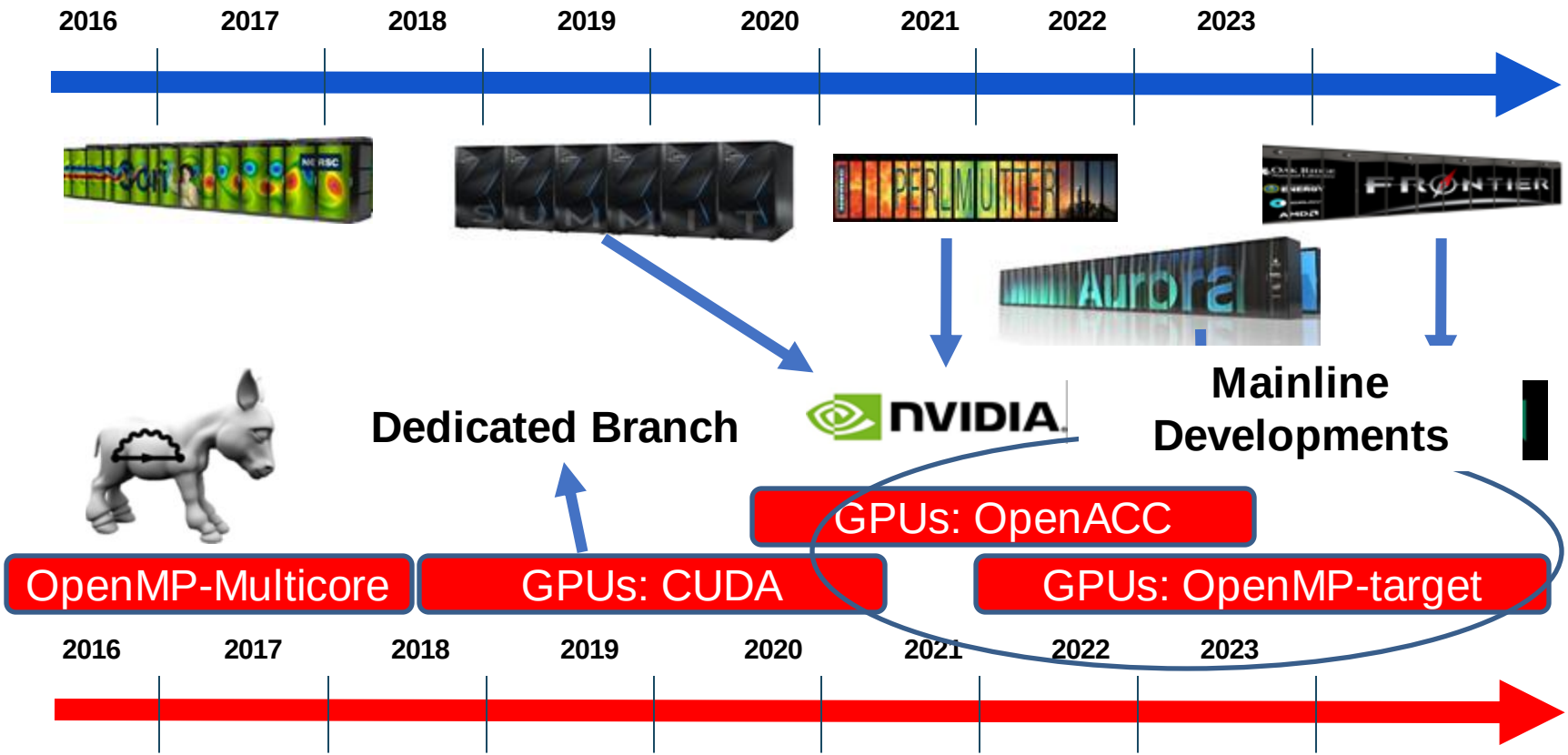


General Portability Strategy

BerkeleyGW: Transitioning to OpenMP-target



BerkeleyGW: Transitioning to OpenMP-target



BerkeleyGW: Transitioning to OpenMP-target

Port Pre-Existing OpenACC Kernels Into Mainline

```
      do iw=nstart,nend

        scht=0D0
        ssxt=0D0
        wxt = wx_array(iw)

        do ig = 1, ncouls

! Here we recompute Omega2 = wtilde2 * I_eps_array. This contains
! the factor of (1 - i tan phi) from Eqs. 21 & 22 of arXiv paper.

!FIXME: Here we use temporary variables wtilde, wtilde2, Omega2 while
! in the following sections we use wtilde_array and I_eps_array directly.
! JRD, please write a comment here explaining whether this is critical
! for performance or it doesn't matter.

          wtilde = wtilde_array(ig,my_igp)
          wtilde2 = wtilde**2
          Omega2 = wtilde2 * I_eps_array(ig,my_igp)

! Cycle bad for vectorization. Not needed wtilde is zero
!                 if (abs(Omega2) .lt. TOL_Zero) cycle
```

OpenACC

```
!$ACC DATA COPYIN(wtilde_array(:,wta_blks:wta_blke))

      do iw=nstart,nend

!$ACC PARALLEL PRESENT(I_eps_array, aqsntemp) vector_length(512)
!$ACC LOOP GANG VECTOR reduction(+:ssx_array_3,sch_array_3) collapse(3)
      do n1loc_blk = 1, n1loc_blksize
        do my_igp = wta_blks, wta_blke
          do ig_blk = 1, ig_blksize

!$ACC LOOP SEQ
        do ig = ig_blk, ncouls, ig_blksize
          wtilde = wtilde_array(ig,my_igp)
          wtilde2 = wtilde**2
          epsa    = I_eps_array(ig,my_igp)
          Omega2 = wtilde2 * epsa
          ssxcutoff = sexcut**2 * epsa * MYCONJG(epsa)
          vcoulx = vcoul_loc(my_igp)

!$ACC LOOP SEQ
        do n1_loc = n1loc_blk, ntband_dist, n1loc_blksize
          aqsmconj = MYCONJG(aqsmtemp_local(n1_loc,my_igp))
          matngmatmgp =  aqsmconj * aqsntemp(ig,n1_loc)
          vcoulxocc = vcoulx * occ_array(n1_loc)
```

BerkeleyGW: Transitioning to OpenMP-target

Add Performance Portability Layer

```
subroutine sigma_gpp()
  SCALAR, allocatable :: ssx_array(:), sch_array(:)
  real(DP) :: delwr, delw2, wdiffr, rden, wxt, ssxcutoff, limitone, limittwo
  complex(DPC) :: halfinvwtilde, delw, wdiff, cden
  integer :: igbeg, igend, igblk
  SCALAR :: ssx, sch, ssxt, scht, schtt

  PUSH_SUB(mtxel_cor.sigma_gpp)

  igblk = 512

  ! Some constants used in the loop below, computed here to save
  ! floating point operations
  limitone=1D0/(TOL_Small*4D0)
  limittwo=sig%gamma**2

  ! GSM: compute the static CH for the static remainder
  if (sig%exact_ch.eq.1) then
    call acc_static_ch(ngpown, ncouls, inv_igp_index, indinv, vcoul, &
      aqsntemp(:,n1), aqsmtemp(:,n1), achstemp, eps_scalar=I_eps_array)
  endif

!!--- Loop over three energy values which we compute Sigma ----------
  ...
```

Wrapper

```
subroutine sigma_gpp()
  PUSH_SUB(mxtel_cor.sigma_gpp)

  select case (sigma_gpp_algo)
  case (CPU_ALGO)
    call sigma_gpp_cpu()
  case (OPENACC_ALGO)
    call sigma_gpp_openacc()
  case (OMP_TARGET_ALGO)
    call sigma_gpp_omp_target()
  case default
    call die("Invald algorithm for sigma_gpp", only_root_writes = .true.)
  end select

  POP_SUB(mtxel_cor.sigma_gpp)

  return
end subroutine sigma_gpp
```

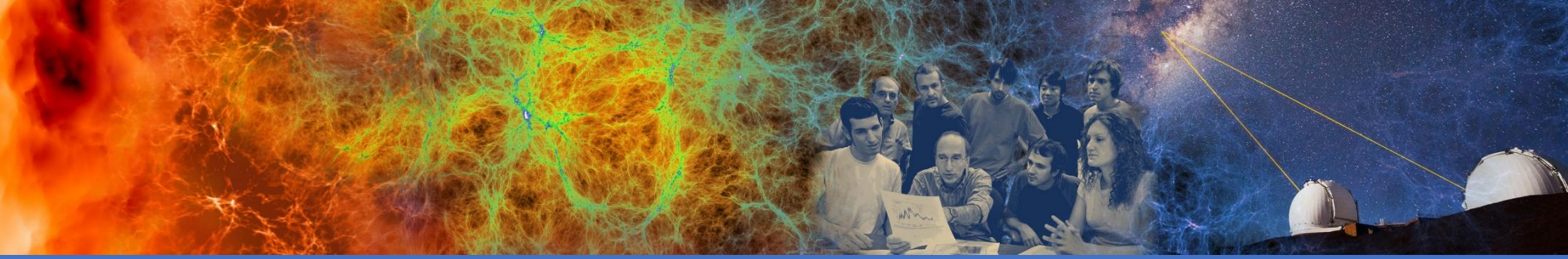
BerkeleyGW: Transitioning to OpenMP-target

OpenACC Kernels -> OMP-Target Kernels

```
n1 = 0
do n1_global = n_start, n_end
  n1 = n1 + 1

  inx_start = (n1_global-1)*wfnkq%nkpt+1
  inx_end   = inx_start + wfnkq%nkpt - 1
  acc_mtxel_sig%bands_vec(n1)%vec(:) = CMPLX(0.0D+00,0.0D+00)
  acc_mtxel_sig%bands_vec(n1)%vec(1:wfnkq%nkpt) = wfnkq%zkq(inx_start
  !$acc update device( acc_mtxel_sig%bands_vec(n1)%vec ) async(n1+1)
  acc_mtxel_sig%mtxel_vec(n1)%vec(:) = CMPLX(0.0D+00,0.0D+00)
  !$acc update device( acc_mtxel_sig%mtxel_vec(n1)%vec ) async(n1+1)

  ! zerofy FFT box
  call acc_zero_box(acc_mtxel_sig%mtxel_box(n1)%box, Nfft, queue=(n1+
  ! put
  call acc_put_into_fftbox(wfnkq%nkpt, acc_mtxel_sig%bands_vec(n1)%ve
                           acc_mtxel_sig%mtxel_box(n1)%box, Nfft, alp
  ! FFT
  call acc_run_fft(acc_mtxel_sig%mtxel_box(n1)%box, acc_mtxel_sig%mtx
  ! multiply
  call acc_box_multiply(acc_mtxel_sig%mtxel_box(n1)%box, acc_mtxel_si
  ! FFT
  call acc_run_fft( acc_mtxel_sig%mtxel_box(n1)%box, acc_mtxel_sig%mt
  ! get
  call acc_get_from_fftbox(ncoul, acc_mtxel_sig%mtxel_vec(n1)%vec, ac
                           acc_mtxel_sig%mtxel_box(n1)%box, Nfft, sca

  ! copy to host
  !$acc update self( acc_mtxel_sig%mtxel_vec(n1)%vec ) async(n1+1)

end do ! n1
```

OMP-target

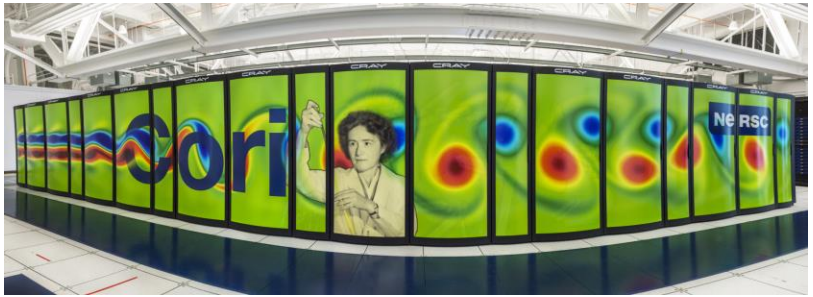
```
n1 = 0
do n1_global = n_start, n_end
  n1 = n1 + 1

  inx_start = (n1_global-1)*wfnkq%nkpt+1
  inx_end   = inx_start + wfnkq%nkpt - 1
  acc_mtxel_sig%bands_vec(n1)%vec(:) = CMPLX(0.0D+00,0.0D+00)
  acc_mtxel_sig%bands_vec(n1)%vec(1:wfnkq%nkpt) = wfnkq%zkq(inx_start:inx_end,jsp)
  !$omp target update to ( acc_mtxel_sig%bands_vec(n1)%vec )
  acc_mtxel_sig%mtxel_vec(n1)%vec(:) = CMPLX(0.0D+00,0.0D+00)
  !$omp target update to ( acc_mtxel_sig%mtxel_vec(n1)%vec )

  ! zerofy FFT box
  call acc_zero_box(acc_mtxel_sig%mtxel_box(n1)%box, Nfft, queue=(n1+1))
  ! put
  call acc_put_into_fftbox(wfnkq%nkpt, acc_mtxel_sig%bands_vec(n1)%vec, acc_mtxel_sig%gvec
                           acc_mtxel_sig%mtxel_box(n1)%box, Nfft, alpha=1.0D+00, queue=(n1
  ! FFT
  call acc_run_fft(acc_mtxel_sig%mtxel_box(n1)%box, acc_mtxel_sig%mtxel_plan(n1), 1 )
  ! multiply
  call acc_box_multiply(acc_mtxel_sig%mtxel_box(n1)%box, acc_mtxel_sig%eqp_box, Nfft, conj
  ! FFT
  call acc_run_fft( acc_mtxel_sig%mtxel_box(n1)%box, acc_mtxel_sig%mtxel_plan(n1), 1 )
  ! get
  call acc_get_from_fftbox(ncoul, acc_mtxel_sig%mtxel_vec(n1)%vec, acc_mtxel_sig%gvec_comp
                           acc_mtxel_sig%mtxel_box(n1)%box, Nfft, scale, queue=(n1+1) )

  ! copy to host
  !$omp target update from ( acc_mtxel_sig%mtxel_vec(n1)%vec )

end do ! n1
```



Benchmarks for Performance Assessment

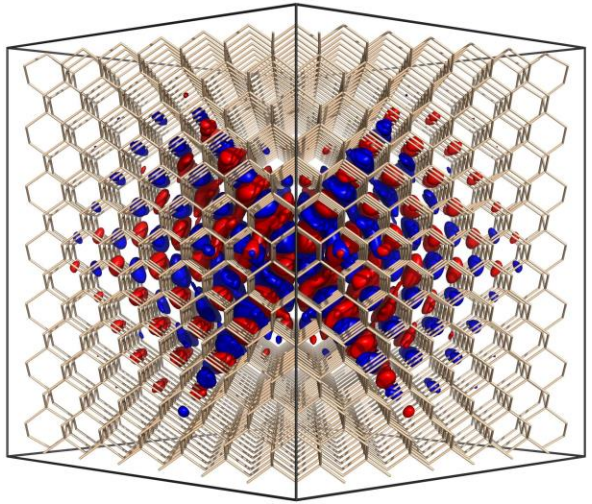
HPC Systems

- Summit (OLCF)
  4,608 nodes, each with 2 IBM POWER9 CPUs and 6 NVIDIA V100 GPUs
- Cori-GPU (NERSC)
  18 nodes, each with 2 Intel Xeon Skylake CPUs and 8 NVIDIA V100 GPUs
- Cori-Haswell (NERSC)
  2,688 Haswell nodes, each with 2 Intel Xeon E5-2698v3 CPUs
- Perlmutter (NERSC)
  ~1,500 nodes, each with 2 AMD Milan CPUs and 4 NVIDIA A100 GPUs



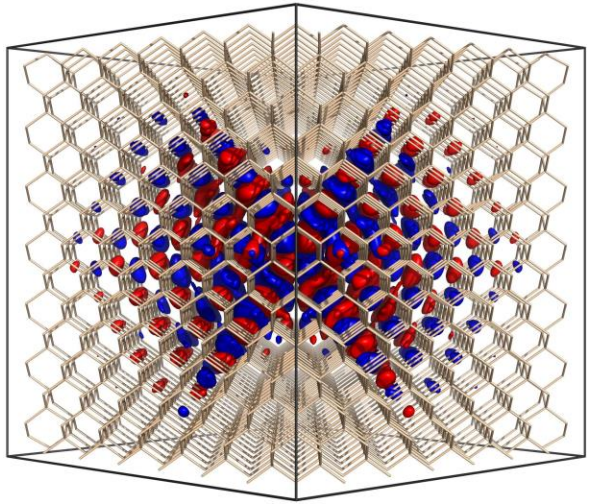
Benchmark for Performance Measurement

- Real applications: Defects in Semiconductors (silicon, silicon carbide) for QBit
- System size: up to **thousands of atoms**, **10000+ electrons**

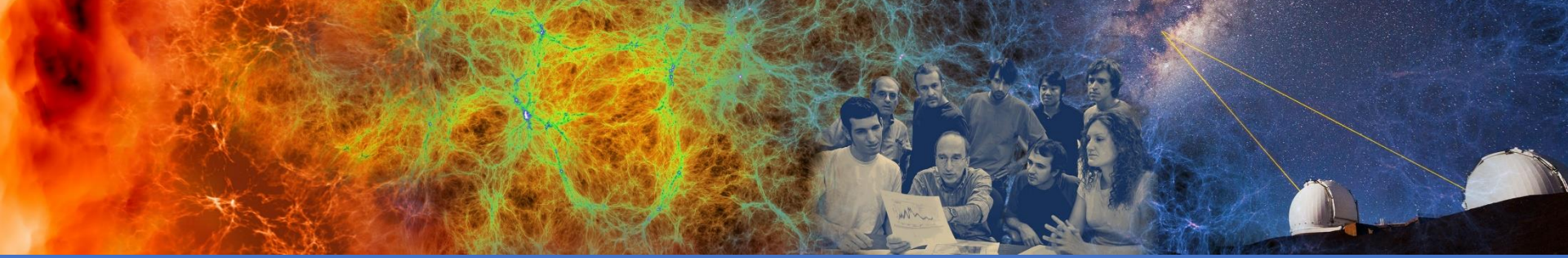


Benchmark for Performance Measurement

- Real applications: Defects in Semiconductors (silicon, silicon carbide) for QBit
- System size: up to **thousands of atoms**, **10000+ electrons**



Very Large-scale GW applications:
Requiring **100s of TB memory and 10s of total ExaFLOP count!**



System, Node and GPU Implementation

The GW Method in  BerkeleyGW

Dynamical properties of electrons as solution of Dyson's equation:

$$h_0(\mathbf{r})\phi_n(\mathbf{r}) + \int \Sigma(\mathbf{r},\mathbf{r}';E_n)\phi_n(\mathbf{r}')\mathrm{d}\mathbf{r}' = E_n\phi_n(\mathbf{r})$$

GW Self-Energy Operator Σ : non-Hermitian, non-local, frequency dependent

(Note: In DFT, the role of self-energy is replaced by static and local $V_{\mathrm{xc}}(\mathbf{r})$)

The GW Method in  BerkeleyGW

Dynamical properties of electrons as solution of Dyson's equation:

$$h_0(\mathbf{r})\phi_n(\mathbf{r}) + \int \Sigma(\mathbf{r},\mathbf{r}';E_n)\phi_n(\mathbf{r}')\mathrm{d}\mathbf{r}' = E_n\phi_n(\mathbf{r})$$

GW Self-Energy Operator Σ : non-Hermitian, non-local, frequency dependent

(Note: In DFT, the role of self-energy is replaced by static and local $V_{\mathrm{xc}}(\mathbf{r})$)

The GW Method in  BerkeleyGW

Dynamical properties of electrons as solution of Dyson's equation:

$$h_0(\mathbf{r})\phi_n(\mathbf{r}) + \int \Sigma(\mathbf{r},\mathbf{r}';E_n)\phi_n(\mathbf{r}')\mathrm{d}\mathbf{r}' = E_n\phi_n(\mathbf{r})$$

GW Self-Energy Operator Σ : non-Hermitian, non-local, frequency dependent

(Note: In DFT, the role of self-energy is replaced by static and local $V_{xc}(\mathbf{r})$)

The GW Method in  BerkeleyGW

Dynamical properties of electrons as solution of Dyson's equation:

$$h_0(\mathbf{r})\phi_n(\mathbf{r}) + \int \Sigma(\mathbf{r},\mathbf{r}';E_n)\phi_n(\mathbf{r}')\mathrm{d}\mathbf{r}' = E_n\phi_n(\mathbf{r})$$

GW Self-Energy Operator Σ : non-Hermitian, non-local, frequency dependent

(Note: In DFT, the role of self-energy is replaced by static and local $V_{\mathrm{xc}}(\mathbf{r})$)

High GW Computational Cost in Two Major Bottlenecks:

The GW Method in  BerkeleyGW

Dynamical properties of electrons as solution of Dyson's equation:

$$h_0(\mathbf{r})\phi_n(\mathbf{r}) + \int \Sigma(\mathbf{r},\mathbf{r}';E_n)\phi_n(\mathbf{r}')\mathrm{d}\mathbf{r}' = E_n\phi_n(\mathbf{r})$$

GW Self-Energy Operator Σ : non-Hermitian, non-local, frequency dependent
(Note: In DFT, the role of self-energy is replaced by static and local $V_{xc}(\mathbf{r})$)

High GW Computational Cost in Two Major Bottlenecks:

- **Epsilon**: Inverse Dielectric Matrix **O(N^4)**

The GW Method in  BerkeleyGW

Dynamical properties of electrons as solution of Dyson's equation:

$$h_0(\mathbf{r})\phi_n(\mathbf{r}) + \int \Sigma(\mathbf{r},\mathbf{r}';E_n)\phi_n(\mathbf{r}')\mathrm{d}\mathbf{r}' = E_n\phi_n(\mathbf{r})$$

GW Self-Energy Operator Σ : non-Hermitian, non-local, frequency dependent

(Note: In DFT, the role of self-energy is replaced by static and local $V_{xc}(\mathbf{r})$)

High GW Computational Cost in Two Major Bottlenecks:

- **Epsilon**: Inverse Dielectric Matrix **O(N⁴)**
- **Sigma**: Self-Energy Matrix Elements **O(N⁴)**

ϵ^{-1} matrix

Breaking Down the Portability Strategy

Epsilon

Three major computational kernels:

- Matrix Elements (MTXEL kernel)
- Polarizability (**CHI-0 Kernel**)
- Inverse Dielectric Matrix (invert)

Breaking Down the Portability Strategy

Epsilon

Three major computational kernels:

- Matrix Elements (MTXEL kernel)
- Polarizability (**CHI-0 Kernel**)
- Inverse Dielectric Matrix (invert)

The **CHI-0 kernel**: large distributed matrix-multiplication over fat and short matrices

Breaking Down the Portability Strategy

Epsilon

Three major computational kernels:

- Matrix Elements (MTXEL kernel)
- Polarizability (**CHI-0 Kernel**)
- Inverse Dielectric Matrix (invert)

The **CHI-0 kernel**: large distributed matrix-multiplication over fat and short matrices

Sigma

Compute a set of (100-1000) Self-Energy matrix elements

Breaking Down the Portability Strategy

Epsilon

Three major computational kernels:

- Matrix Elements (MTXEL kernel)
- Polarizability (**CHI-0 Kernel**)
- Inverse Dielectric Matrix (invert)

The **CHI-0 kernel**: large distributed matrix-multiplication over fat and short matrices

Sigma

Compute a set of (100-1000) Self-Energy matrix elements

$$\Sigma_{lm} = \sum_{n}^{N_b} \sum_{GG'}^{N_G} M^{l*}_{Gn} [P(E_n)]_{GG'} M^{m}_{G'n}$$

Breaking Down the Portability Strategy

Epsilon

Three major computational kernels:

- Matrix Elements (MTXEL kernel)
- Polarizability (**CHI-0 Kernel**)
- Inverse Dielectric Matrix (invert)

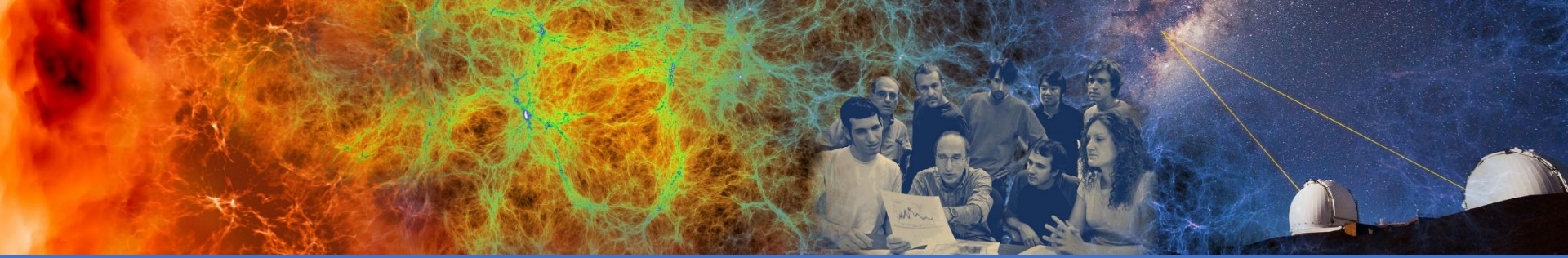
The **CHI-0 kernel**: large distributed matrix-multiplication over fat and short matrices

Sigma

Compute a set of (100-1000) Self-Energy matrix elements

$$\Sigma_{lm} = \sum_{n}^{N_b} \sum_{GG'}^{N_G} M^{l*}_{Gn} [P(E_n)]_{GG'} M^{m}_{G'n}$$

The **GPP kernel**: a big data reduction across different matrices with a complex matrix-vector interdependence



Epsilon Module

Breaking Down the Portability Strategy

Epsilon

Three major computational kernels:

- Matrix Elements (MTXEL kernel)
- Polarizability (**CHI-0 Kernel**)
- Inverse Dielectric Matrix (invert)

The **CHI-0 kernel**: large distributed matrix-multiplication over fat and short matrices

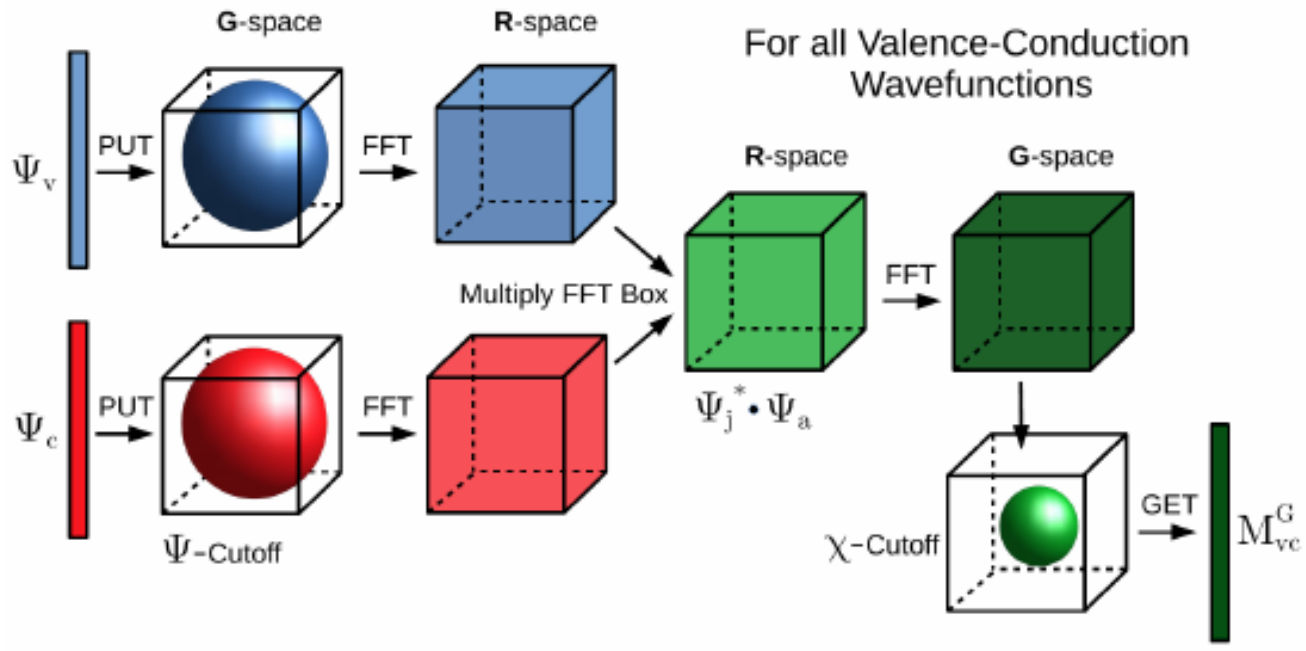
Sigma

Compute a set of (100-1000) Self-Energy matrix elements

$$\Sigma_{lm} = \sum_{n}^{N_b} \sum_{GG'}^{N_G} M^{l*}_{Gn} [P(E_n)]_{GG'} M^{m}_{G'n}$$

The **GPP kernel**: a big data reduction across different matrices with a complex matrix-vector interdependence

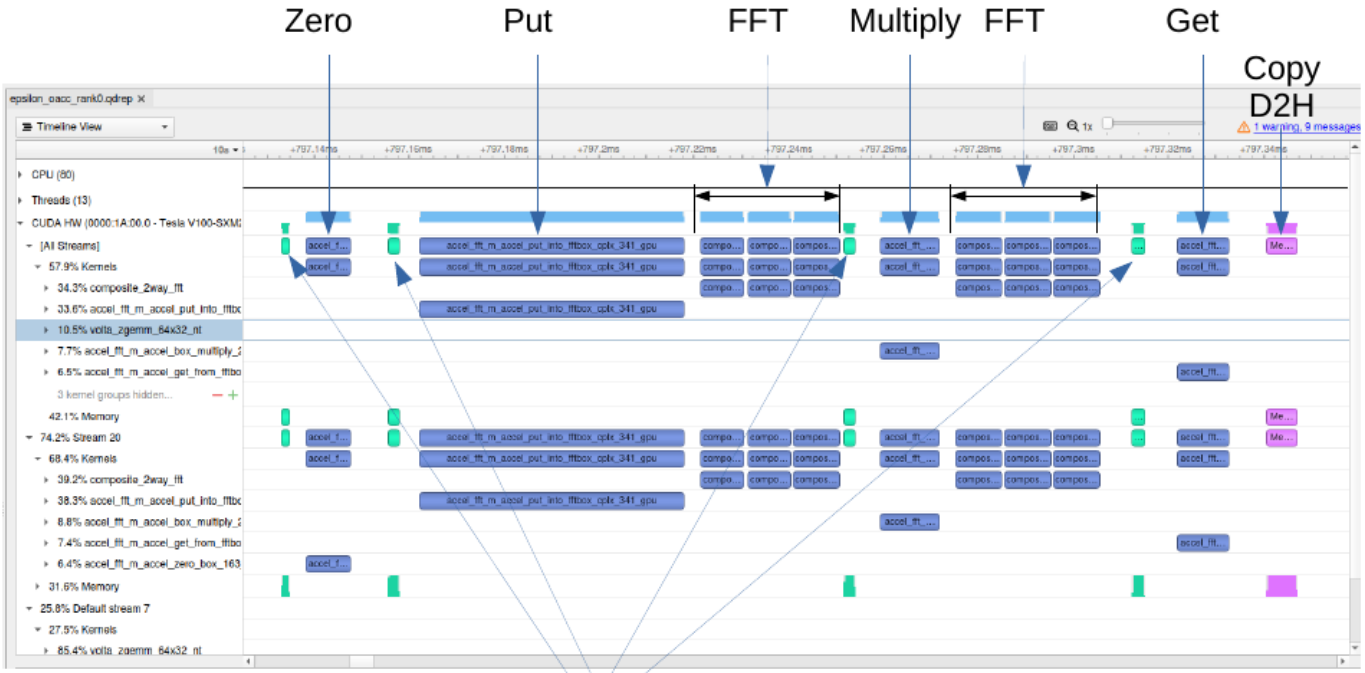
Epsilon: The MTXEL Kernel



Two nested loops executing a sequence of simple kernels

- Well Optimized CUDA Implementation
- Initial ACC/OMP port: *Single queue no loop blocking*

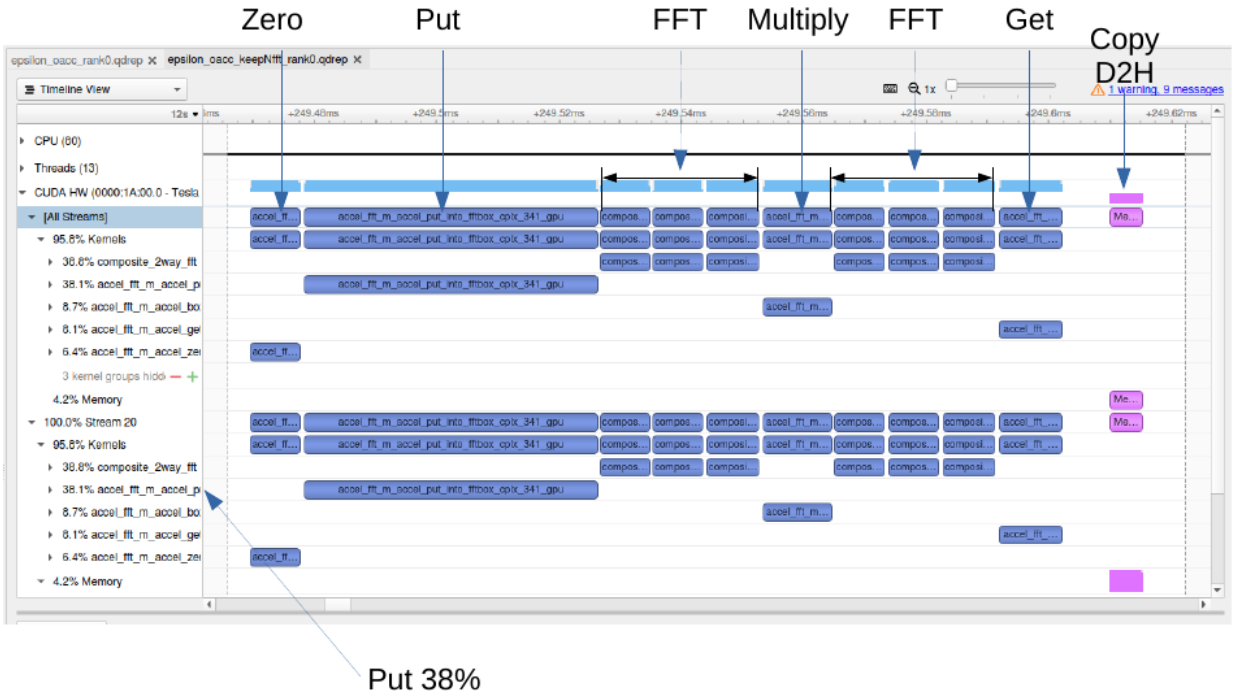
MTXEL Kernel ACC/OMP: Baseline (64s)



12 Bytes, I think this is coping in the Nfft array (integer array with 3 elements) before running the kernels

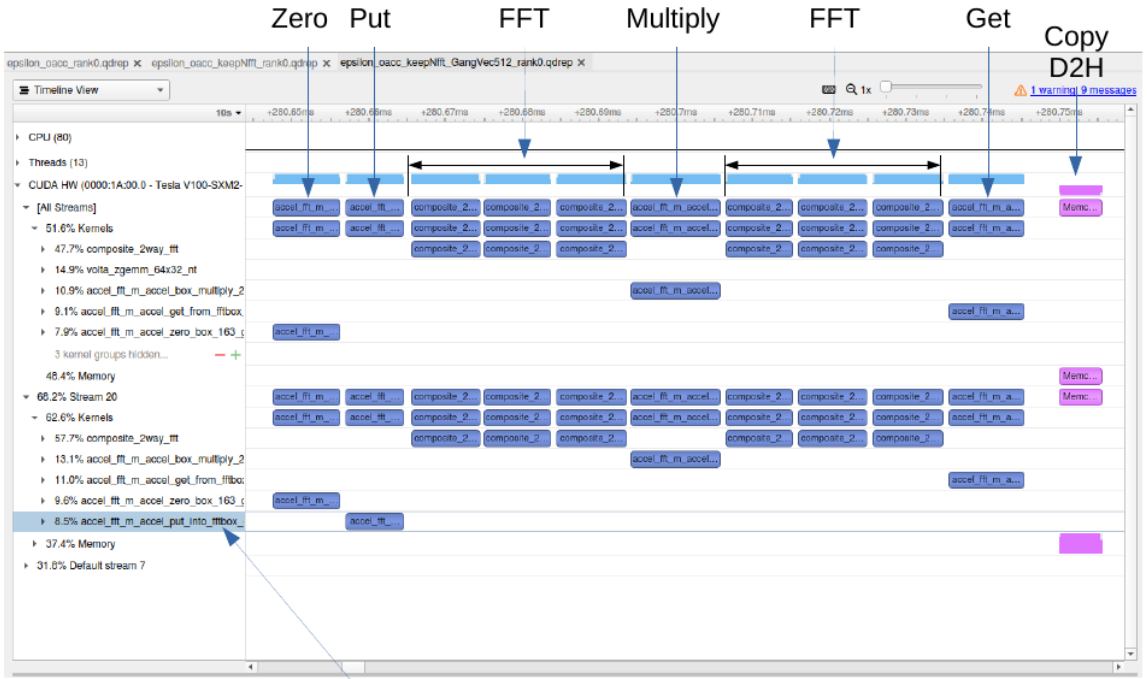
Move small array into GPU memory (58s)

Copy in Nfft array before kernel execution



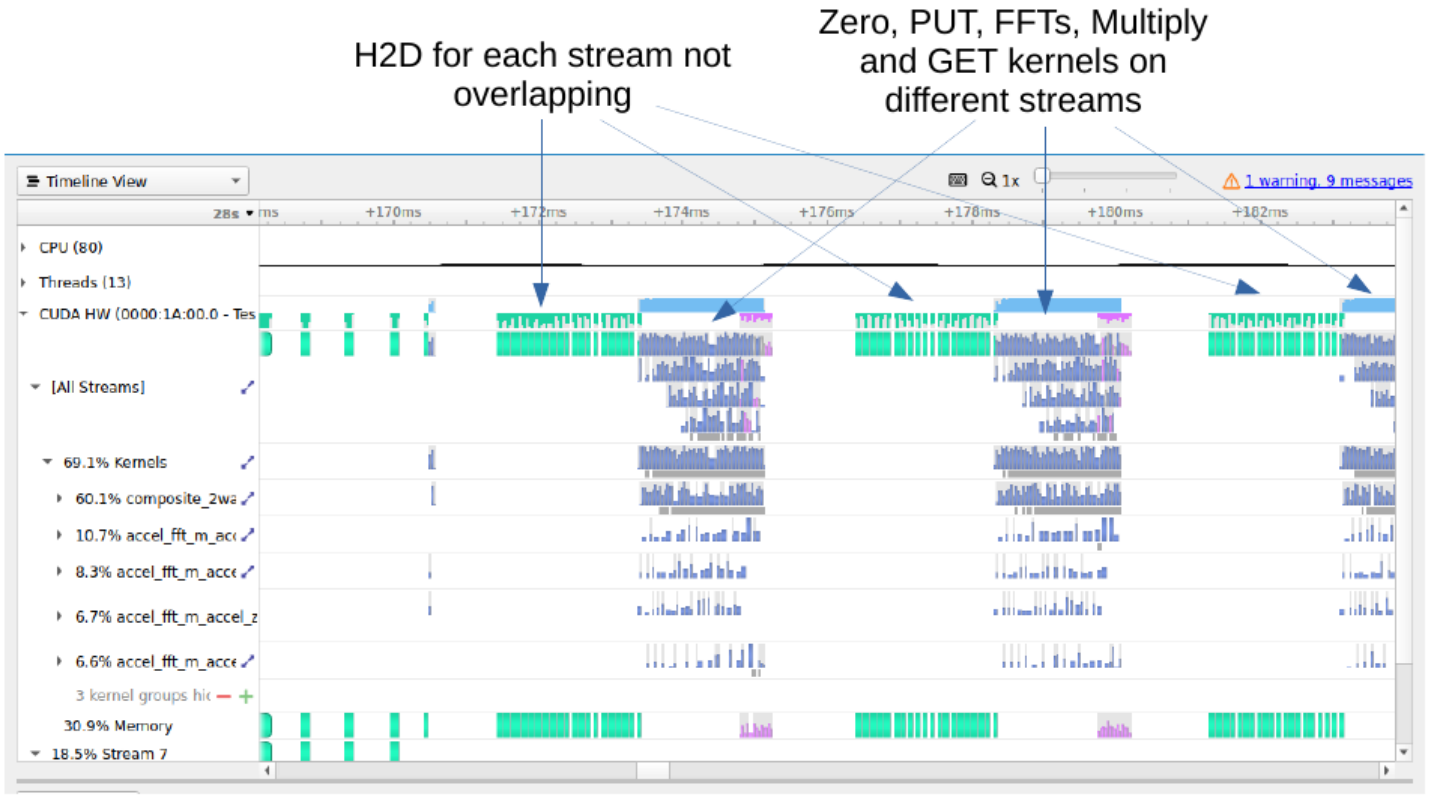
Test different loop constructs (53s)

Use different loop construct and experiment with vector length (512)

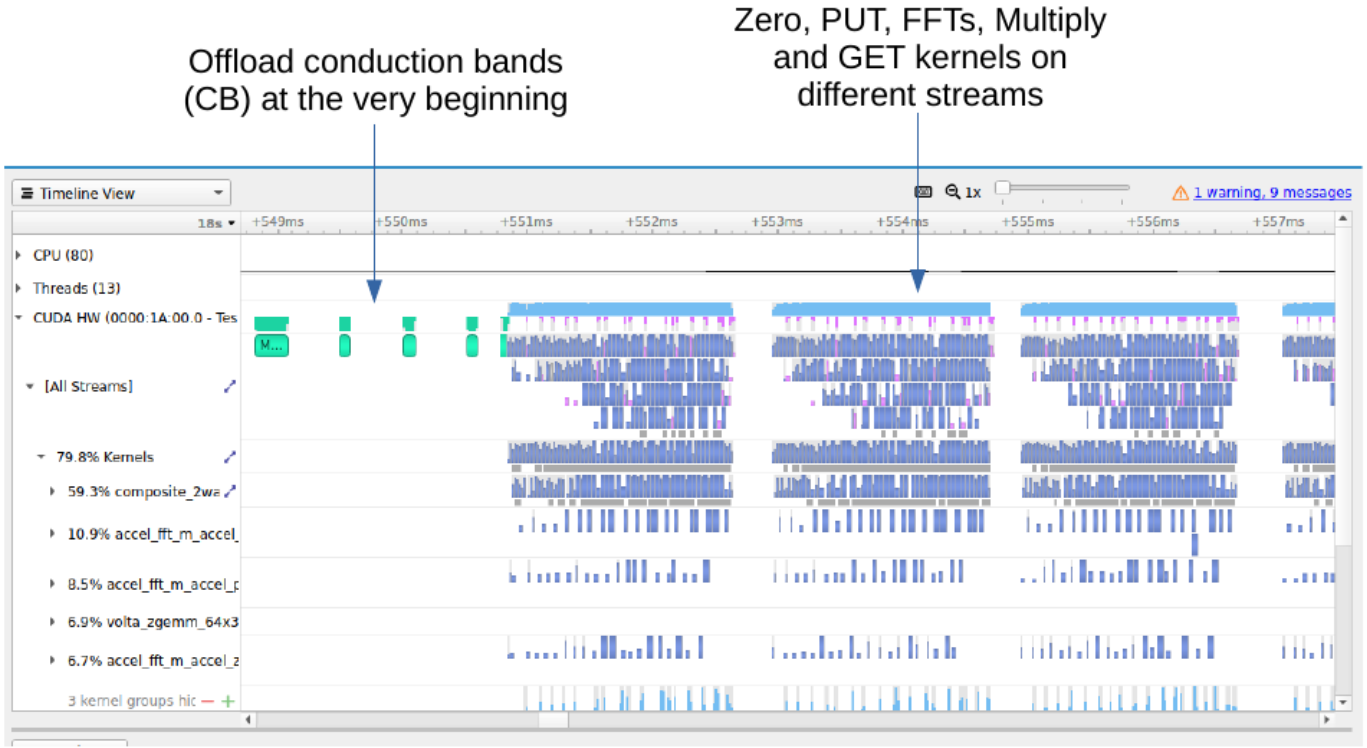


Using vector_length(512) loop gang vector construct for all kernel slightly improve performance 52.1 vs 52.6s

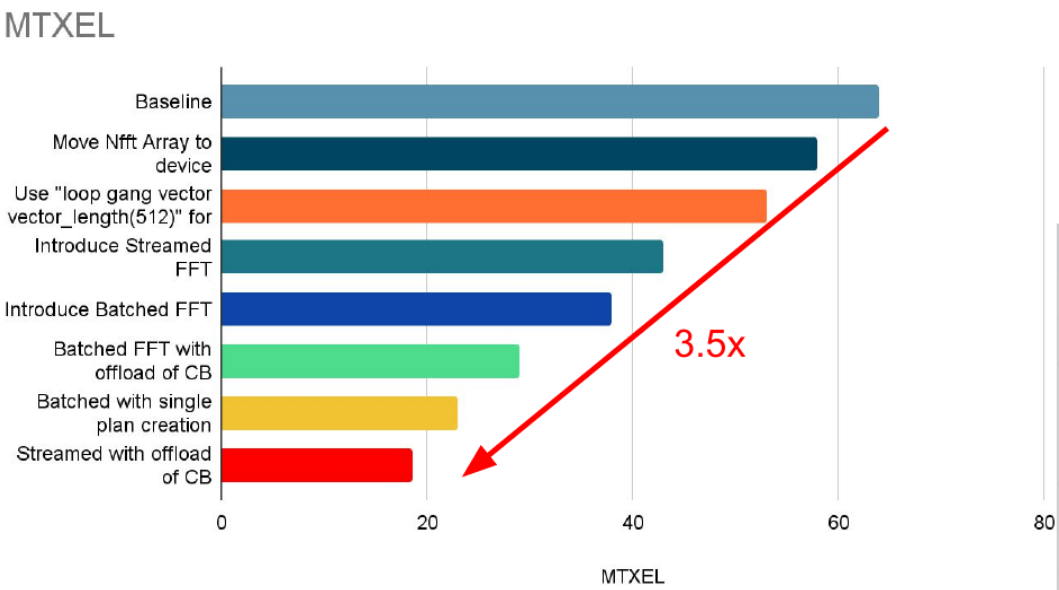
Introduce Queue/Streams (43s)

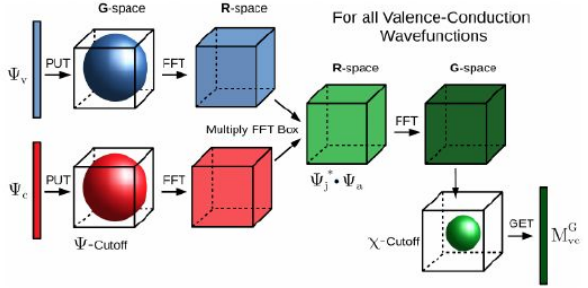


Queue/Streams + Offload Innermost Arrays (18.5s)



MTXEL Kernel ACC/OMP: Summary

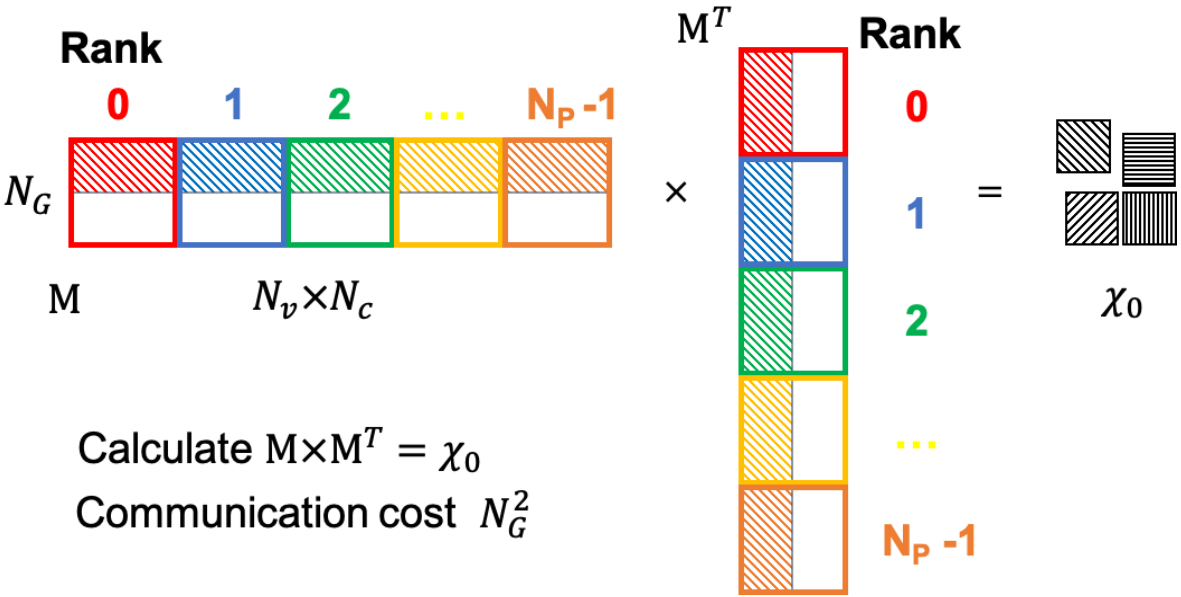




Time in seconds

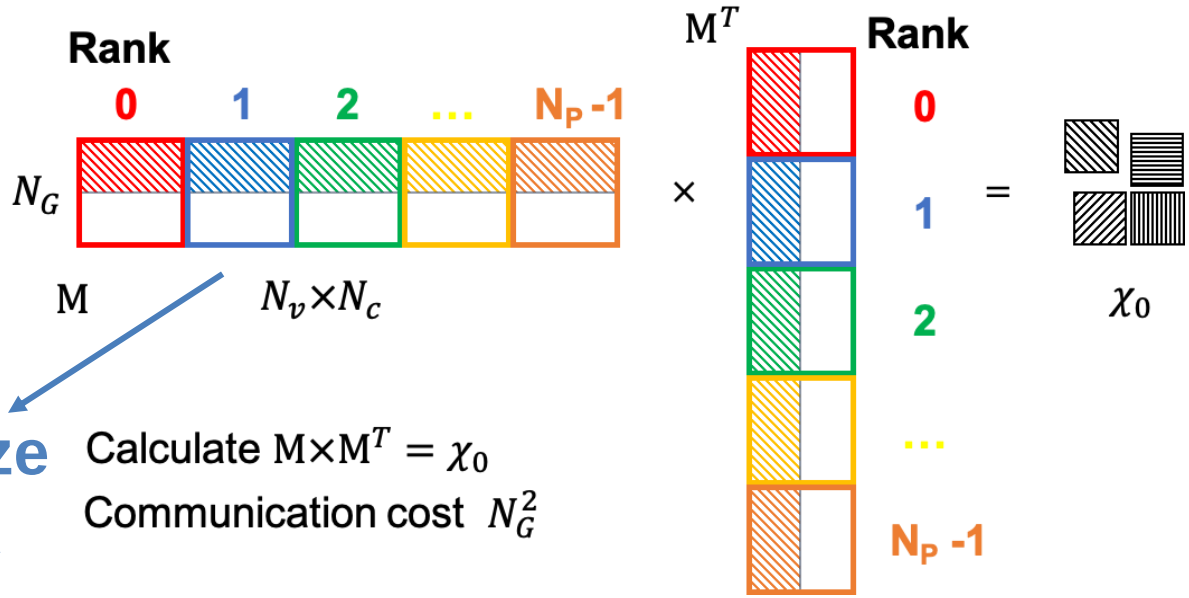
	MTXEL
Baseline	64
Move Nfft Array to device	58
Use "loop gang vector vector_length(512)" for PUT, Mult and Get kernels	53
Introduce Streamed FFT	43
Introduce Batched FFT	38
Batched FFT with offload of CB	29
Batched with single plan creation	23
Streamed with offload of CB	18.5

Epsilon: The CHI-0 Kernel



Data layout for **M** matrix in CHI-0 kernel

Epsilon: The CHI-0 Kernel



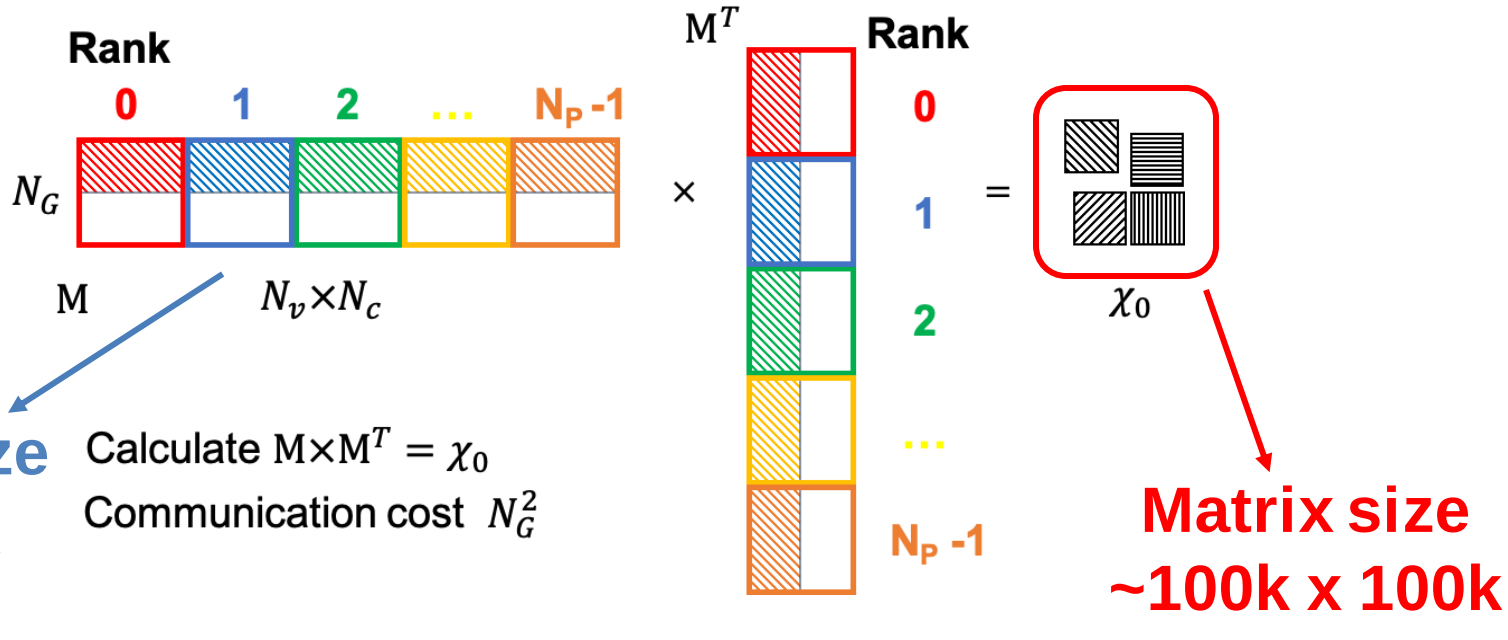
Matrix size ~100k x 100M

Calculate $M \times M^T = \chi_0$

Communication cost N_G^2

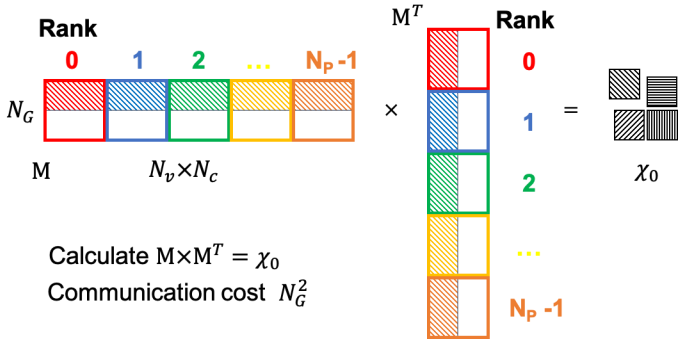
Data layout for **M** matrix in CHI-0 kernel

Epsilon: The CHI-0 Kernel



Data layout for **M** matrix in CHI-0 kernel

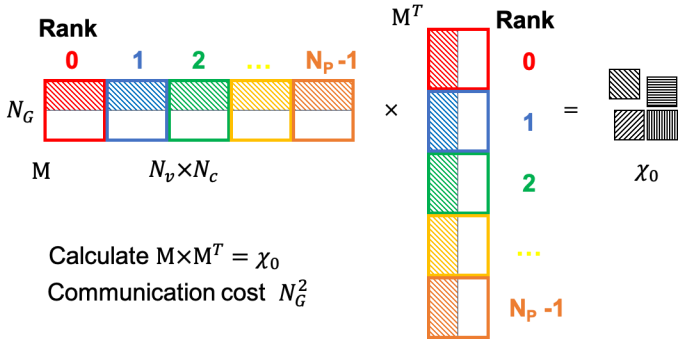
Epsilon: The CHI-0 Kernel



Data layout for **M** matrix in CHI-0 kernel

- Offload Data Preparation
  Accelerate data preparation (potential memory bottleneck)
- Batching Mechanism
  Avoid hitting OOM on GPU
- Non-blocking Cyclic Communication
  Overlap Computation (GPU) and MPI communication (CPU)

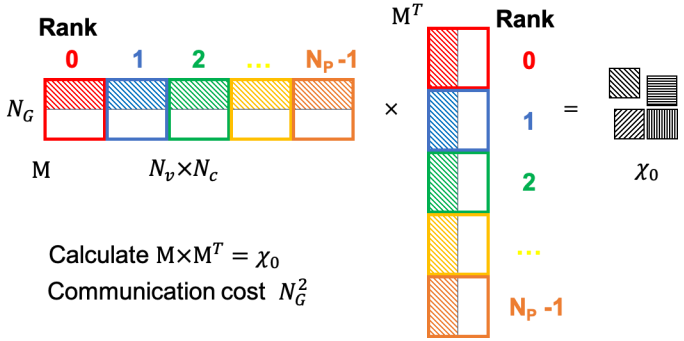
Epsilon: The CHI-0 Kernel



Data layout for **M** matrix in CHI-0 kernel

- **Offload Data Preparation**
  Accelerate data preparation (potential memory bottleneck)
- Batching Mechanism
  Avoid hitting OOM on GPU
- Non-blocking Cyclic Communication
  Overlap Computation (GPU) and MPI communication (CPU)

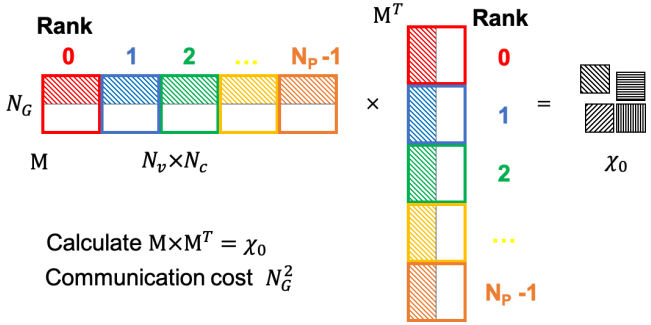
Epsilon: The CHI-0 Kernel



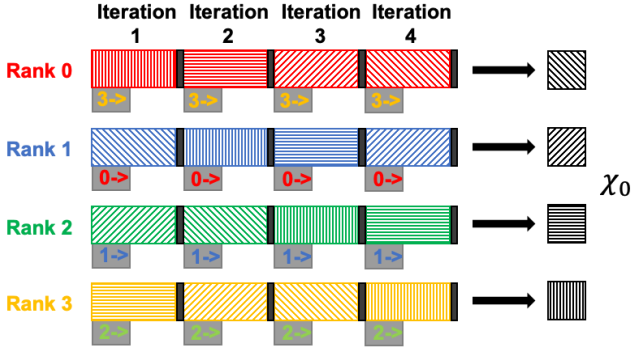
Data layout for **M** matrix in CHI-0 kernel

- Offload Data Preparation
  Accelerate data preparation (potential memory bottleneck)
- **Batching Mechanism**
  Avoid hitting OOM on GPU
- Non-blocking Cyclic Communication
  Overlap Computation (GPU) and MPI communication (CPU)

Epsilon: The CHI-0 Kernel



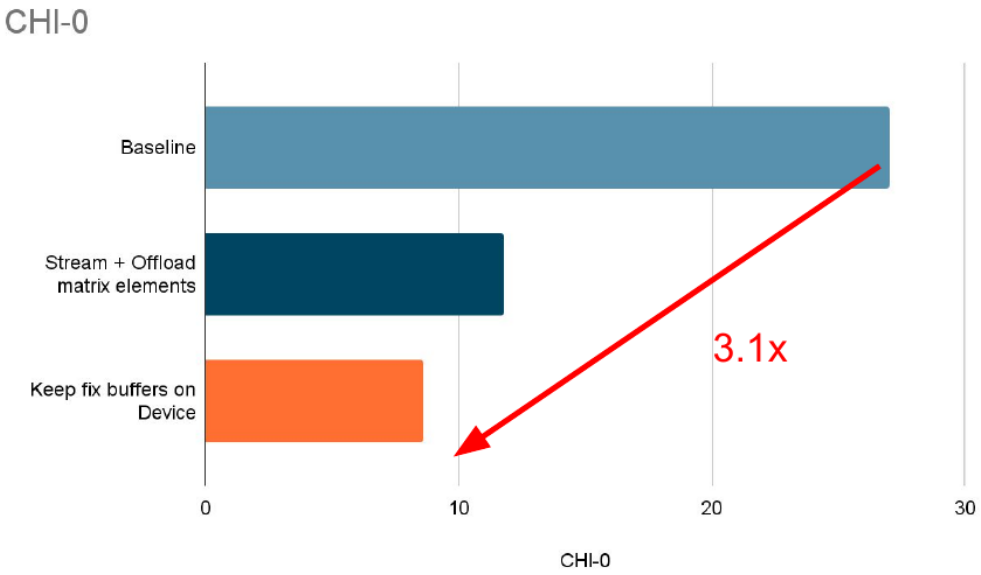
Data layout for **M** matrix in CHI-0 kernel

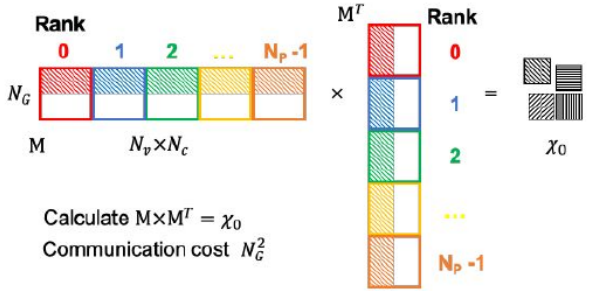


Comput./Commun. pattern for non-blocking cyclic scheme

- Offload Data Preparation
  Accelerate data preparation (potential memory bottleneck)
- Batching Mechanism
  Avoid hitting OOM on GPU
- **Non-blocking Cyclic Communication**
  Overlap Computation (GPU) and MPI communication (CPU)

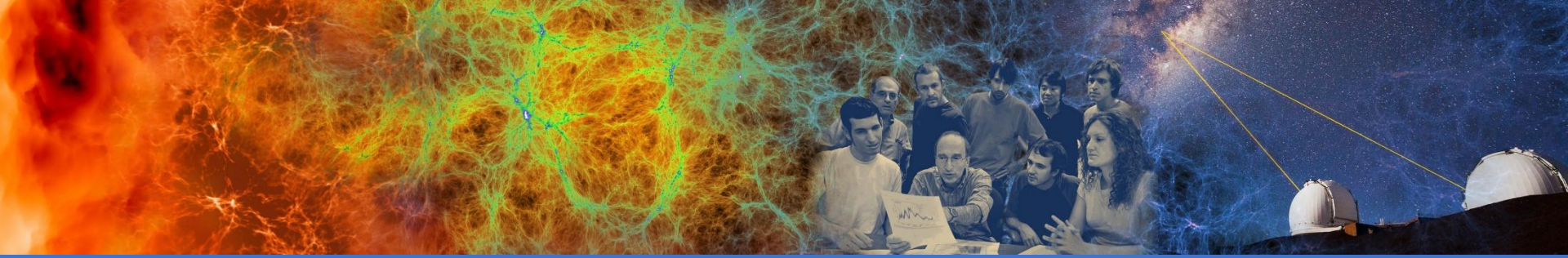
Epsilon CHI-0 Kernel: ACC/OMP Summary





Time in seconds

	CHI-0
Baseline	27
Stream + Offload matrix elements	11.8
Keep fix buffers on Device	8.6



Sigma Module

Breaking Down the Portability Strategy

Epsilon

Three major computational kernels:

- Matrix Elements (MTXEL kernel)
- Polarizability (**CHI-0 Kernel**)
- Inverse Dielectric Matrix (invert)

The **CHI-0 kernel**: large distributed matrix-multiplication over fat and short matrices

Sigma

Compute a set of (100-1000) Self-Energy matrix elements

$$\Sigma_{lm} = \sum_{n}^{N_b} \sum_{GG'}^{N_G} M^{l*}_{Gn} [P(E_n)]_{GG'} M^{m}_{G'n}$$

The **GPP kernel**: a big data reduction across different matrices with a complex matrix-vector interdependence

Sigma: The GPP Kernel

Compute a set of (100-1000) Self-Energy matrix elements

Two Level Parallelization Strategy:

- Inter-Pool parallelization (*independent self-energy matrix elements*)
- Intra-Pool parallelization (*same self-energy matrix elements*)

Sigma: The GPP Kernel

Inter-Pool Parallelization

16 MPI tasks, 2 pools, 4 Self-Energies: $\{\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4\}$

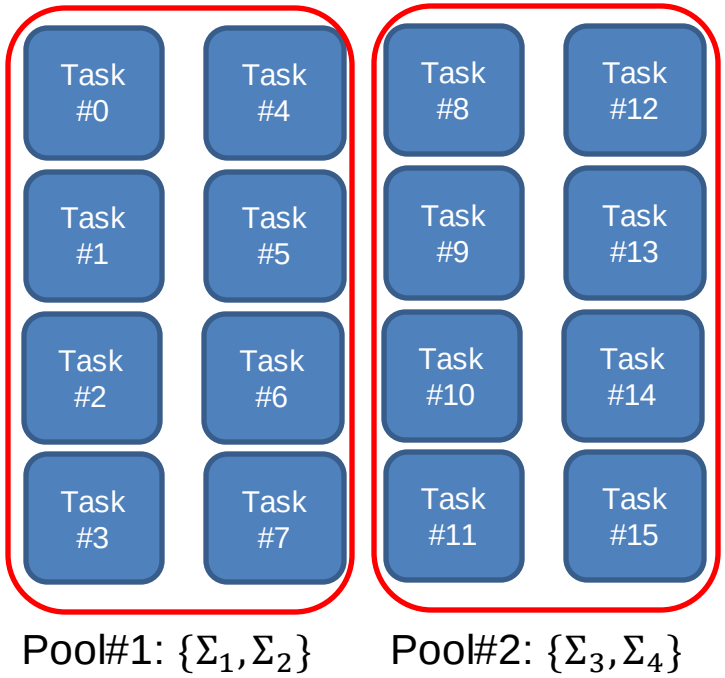
Two Level Parallelization Strategy:

- **Inter-Pool parallelization** (*independent self-energy matrix elements*)
- Intra-Pool parallelization (*same self-energy matrix elements*)

Sigma: The GPP Kernel

Inter-Pool Parallelization

16 MPI tasks, 2 pools, 4 Self-Energies: $\{\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4\}$

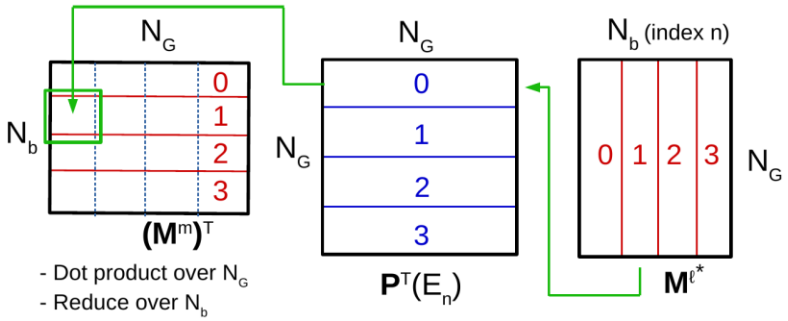


Two Level Parallelization Strategy:

- **Inter-Pool parallelization** (*independent self-energy matrix elements*)
- Intra-Pool parallelization (*same self-energy matrix elements*)

Sigma: The GPP Kernel

Intra-Pool data layout

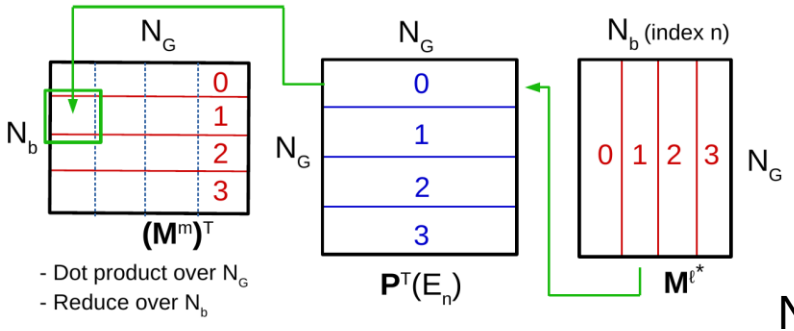


Two Level Parallelization Strategy:

- Inter-Pool parallelization (*independent self-energy matrix elements*)
- **Intra-Pool parallelization** (*same self-energy matrix elements*)

Sigma: The GPP Kernel

Intra-Pool data layout



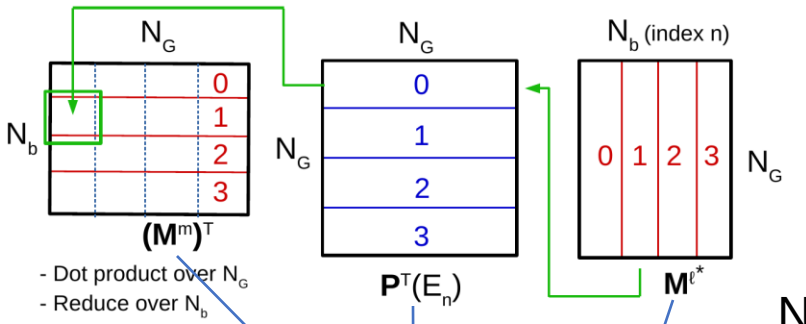
Two Level Parallelization Strategy:

- Inter-Pool parallelization (*independent self-energy matrix elements*)
- **Intra-Pool parallelization** (*same self-energy matrix elements*)

Non-Blocking Cyclic communication: **Overlap MPI communication (host) with computation (GPU)**

Sigma: The GPP Kernel

Intra-Pool data layout



$$\Sigma_{lm} = \sum_{n}^{N_b} \sum_{GG'}^{N_G} M^{m}_{G'n} [P^{\mathsf{T}}(E_n)]_{G'G} M^{l*}_{Gn}$$

Two Level Parallelization Strategy:

- Inter-Pool parallelization (*independent self-energy matrix elements*)
- Intra-Pool parallelization (*same self-energy matrix elements*)

Non-Blocking Cyclic communication: Overlap MPI communication (host) with computation (GPU)

GPP-Kernel: Compute Intensive part entirely on GPU

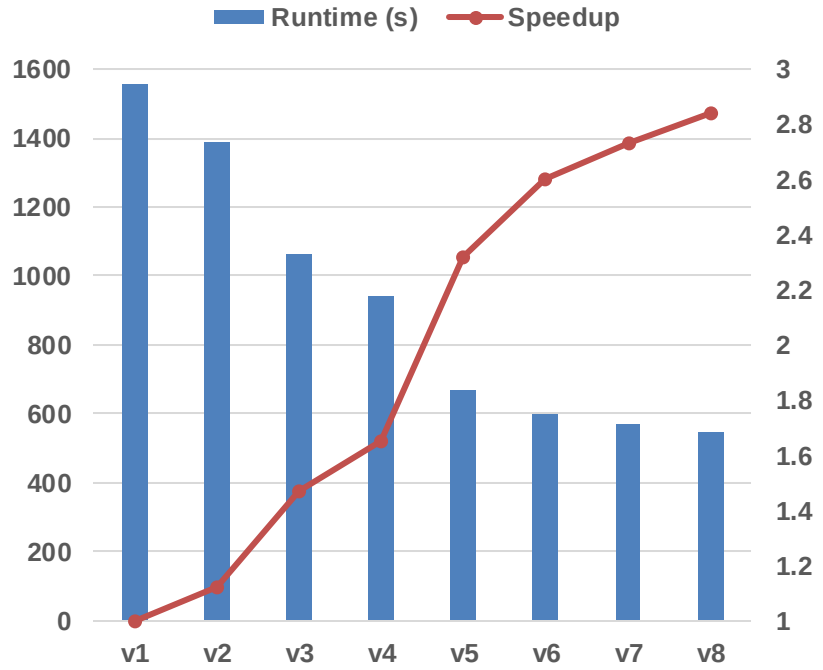
GPP Kernel: Computational Characteristics

```
# pseudo code per invocation
for band = 1, nbands # O(1k)
 for igp = 1, ngpown # O(10k)
  for ig = 1, ncouls # O(100k)
   for iw = 1, nw    # small
    Complex number arithmetic
Reduce to arrays[iw]
```

- Tensor contraction
  - Bandwidth bound
- Reduction of 10^{12} numbers
  - Shared mem for partial sums
- Double complex numbers
  - High register usage
- Multiple multi-dim arrays
  - Memory access pattern
- Long-latency operations
  - Divisions, square roots

Optimization Path CUDA

1. Baseline*
2. Replace divides with reciprocals
3. Replace square roots with power of 2
4. Replace divides and square roots
5. Loop re-ordering
6. Further increase occupancy
7. Cache blocking
8. Add more arrays to shared memory



*Collapse 3 of the other loops

Optimization ACC/OMP: Staring Kernel

Sigma GPP Kernel - CPU

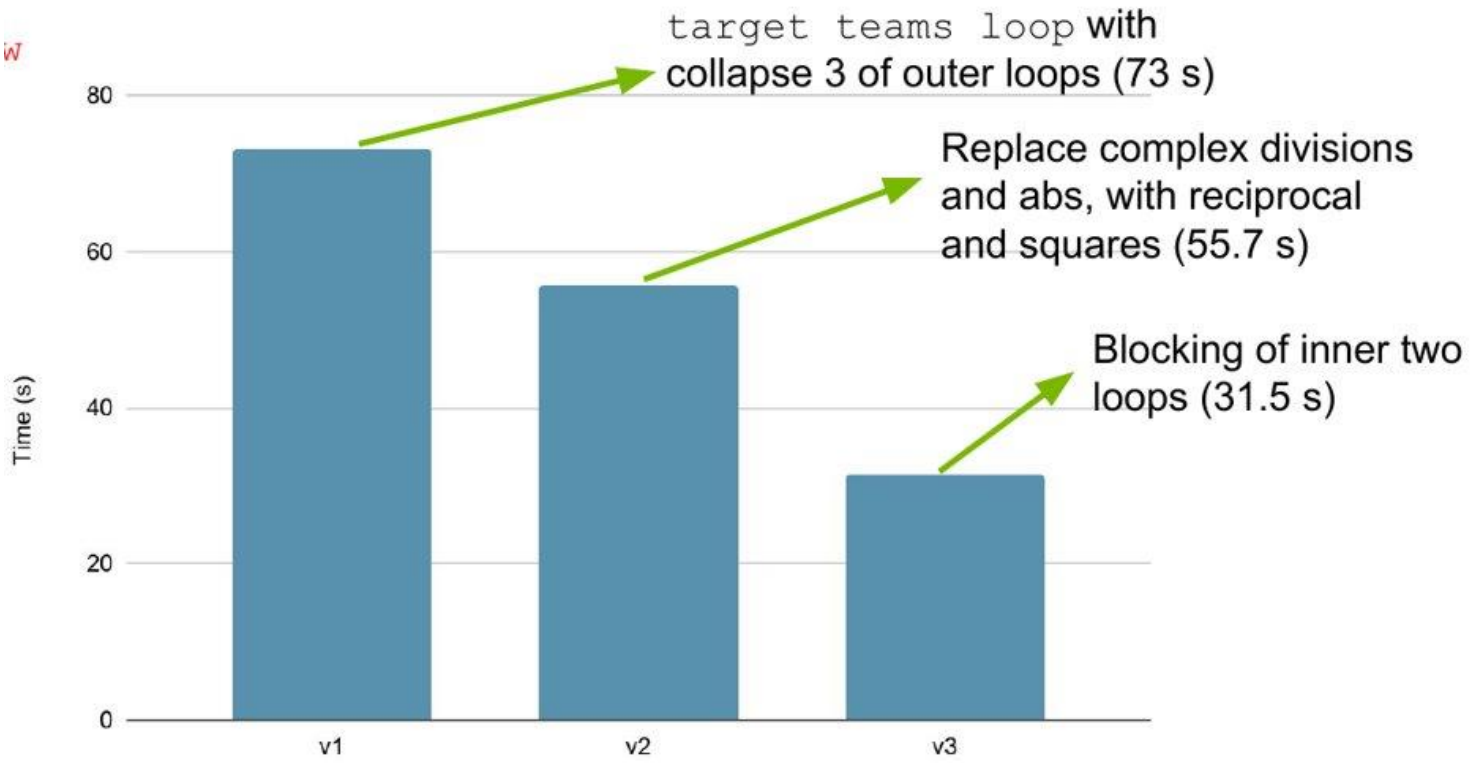
```
!CPU version
!$OMP PARALLEL DO default(private) &
!$OMP          shared(ntband_dist, ngpown, ncouls, nstart, nend, &
!$OMP                 wtilde_array, I_eps_array, wx_array_t, &
!$OMP                 limittwo, limitone, n1true_array, &
!$OMP                 aqsmtemp_local, aqsntemp, vcoul_loc, occ_array ) &
!$OMP collapse(3) reduction(+:acht_n1_loc,ssx_array,sch_array)
    do n1_loc = 1, ntband_dist
      do my_igp = 1,  ngpown
        do ig = 1, ncouls
          do iw=nstart,nend
            wtilde = wtilde_array(ig,my_igp)
            wtilde2 = wtilde**2
            Omega2 = wtilde2 * I_eps_array(ig,my_igp)

            wdiff = wx_array_t(iw,n1_loc) - wtilde

            delw = wtilde / wdiff
            delwr = delw*CONJG(delw)
            wdiffr = wdiff*CONJG(wdiff)

            if (wdiffr.gt.limittwo .and. delwr.lt.limitone) then
              sch = delw * I_eps_array(ig,my_igp)
              cden = wx_array_t(iw,n1_loc)**2 - wtilde2
              ssx = Omega2 / cden
            else if ( delwr .gt. TOL_Zero) then
              sch = 0.0d0
```

Double complex data reduction of 2 arrays

3 nested loops collapsed(3)

Array size: number of frequencies (nend-nstart+1)

Optimization Path ACC/OMP



Optimization Path ACC/OMP

	CPU-Only (16 MPI)	CUDA (4 MPI + 4 GPU)	OpenACC (4 MPI + 4 GPU)	OpenMP Offload (4 MPI + 4 GPU)
MTXEL	62 s	1.7 s	3.2 s	3.5 s
GPP	1245 s	27.3 s	30.2 s	31.5 s
Total Runtime	1311 s	34.5 s	43.4 s	45.9 s

Run Settings:

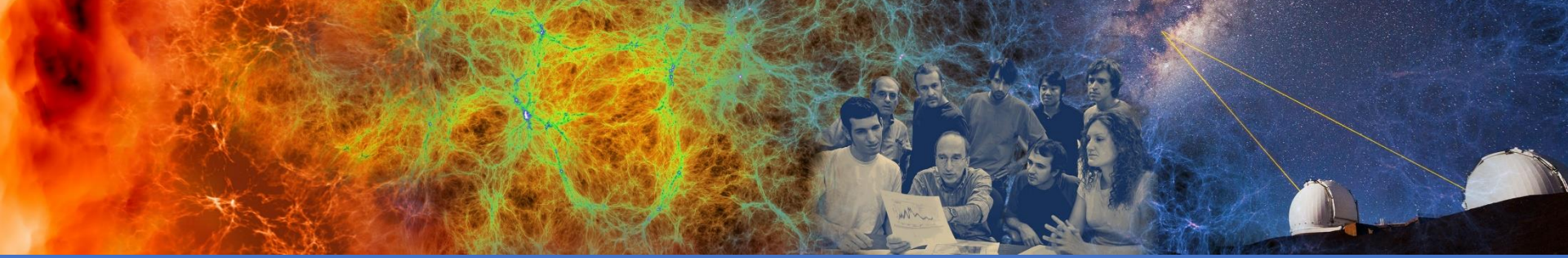
- BerkeleyGW Si214 test case
- Cori GPU (single node: 2 20-core Intel Xeon Gold 6148 @ 2.40 GHz + 8 Nvidia V100s)
- Nvidia HPC SDK 20.11
- CUDA 11.1.1

Optimization Path ACC/OMP

	CPU-Only (16 MPI)	CUDA (4 MPI + 4 GPU)	OpenACC (4 MPI + 4 GPU)	OpenMP Offload (4 MPI + 4 GPU)
MTXEL	62 s	1.7 s	3.2 s	3.5 s
GPP	1245 s	27.3 s	30.2 s	31.5 s
Total Runtime	1311 s	34.5 s	43.4 s	45.9 s

Run Settings:

- BerkeleyGW Si214 test case
- Cori GPU (single node: 2 20-core Intel Xeon Gold 6148 @ 2.40 GHz + 8 Nvidia V100s)
- Nvidia HPC SDK 20.11
- CUDA 11.1.1



Performance Results

CPU vs GPU

Epsilon

Runtime comparison in seconds on Cori-GPU with 2 nodes, with GPU and CPU only for Si-214

	MTXEL	CHI-0	Invert	Total
CPU Only	616	1120	10.3	1794
GPU Full-Offload	24.4	47.7	9.6	96.3

CPU vs GPU

Epsilon

Runtime comparison in seconds on Cori-GPU with 2 nodes, with GPU and CPU only for Si-214

	MTXEL	CHI-0	Invert	Total
CPU Only	616	1120	10.3	1794
GPU Full-Offload	24.4	47.7	9.6	96.3

25x **23.5x**

CPU vs GPU

Epsilon

Runtime comparison in seconds on Cori-GPU with 2 nodes, with GPU and CPU only for Si-214

18.6x speedup (overall)

	MTXEL	CHI-0	Invert	Total
CPU Only	616	1120	10.3	1794
GPU Full-Offload	24.4	47.7	9.6	96.3

25x **23.5x** **18.6x**

CPU vs GPU

Epsilon

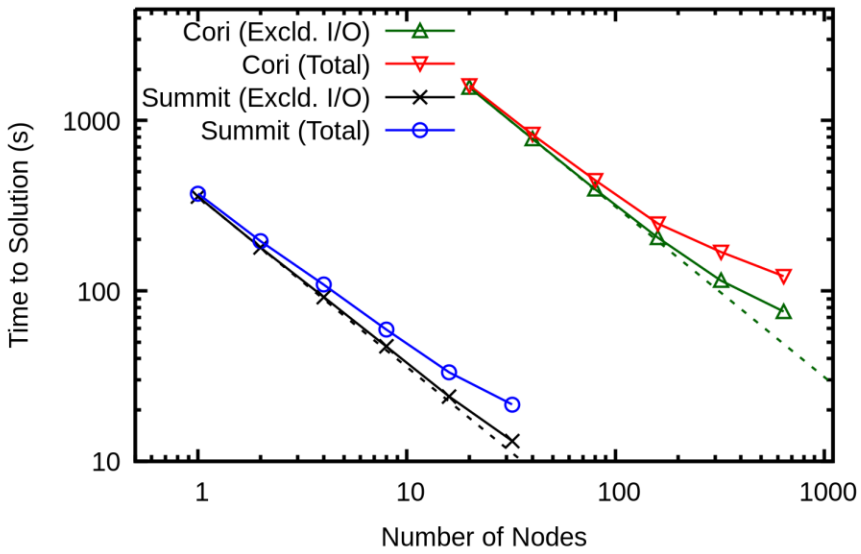
Runtime comparison in seconds on Cori-GPU with 2 nodes, with GPU and CPU only for Si-214

18.6x speedup (overall)

	MTXEL	CHI-0	Invert	Total
CPU Only	616	1120	10.3	1794
GPU Full-Offload	24.4	47.7	9.6	96.3
	25x	**23.5x**		**18.6x**

Sigma

Runtime comparison between Cori-Haswell and Summit for Si-510



CPU vs GPU

Epsilon

Runtime comparison in seconds on Cori-GPU with 2 nodes, with GPU and CPU only for Si-214

18.6x speedup (overall)

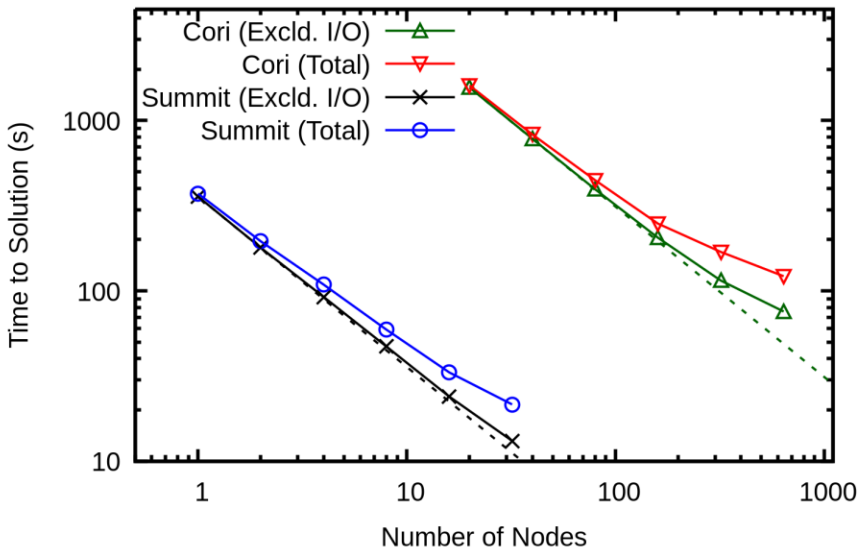
	MTXEL	CHI-0	Invert	Total
CPU Only	616	1120	10.3	1794
GPU Full-Offload	24.4	47.7	9.6	96.3

25x **23.5x** **18.6x**

Sigma

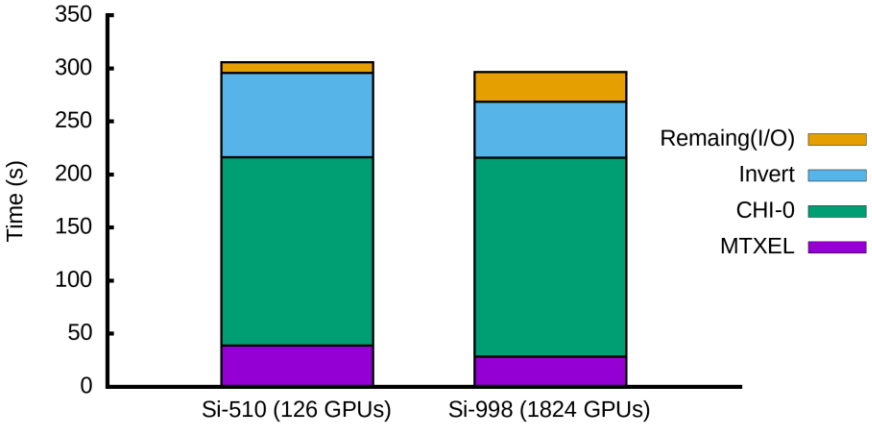
Runtime comparison between Cori-Haswell and Summit for Si-510

86x speedup (1:1 node)

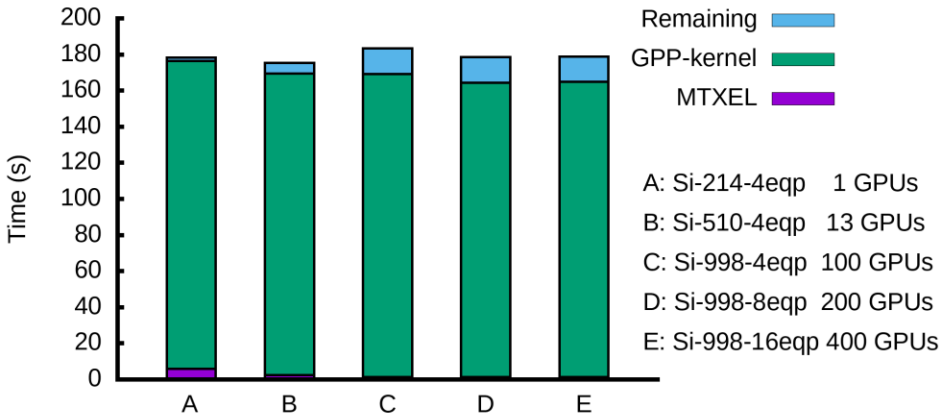


Weak Scaling

Epsilon

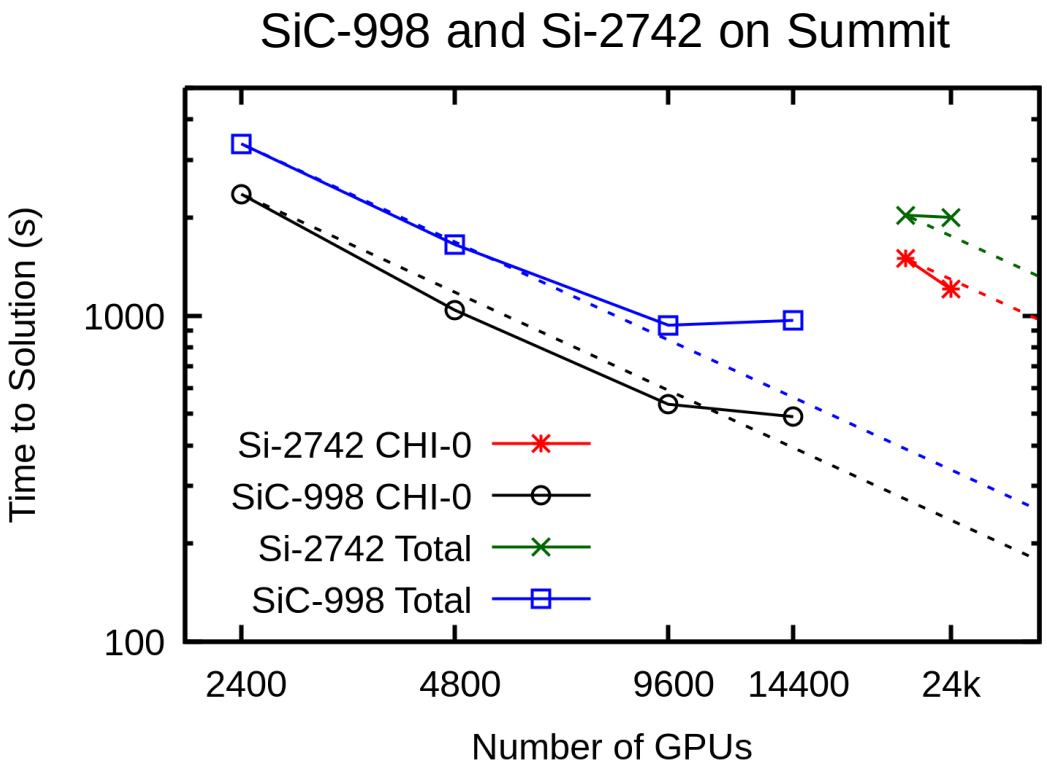


Sigma



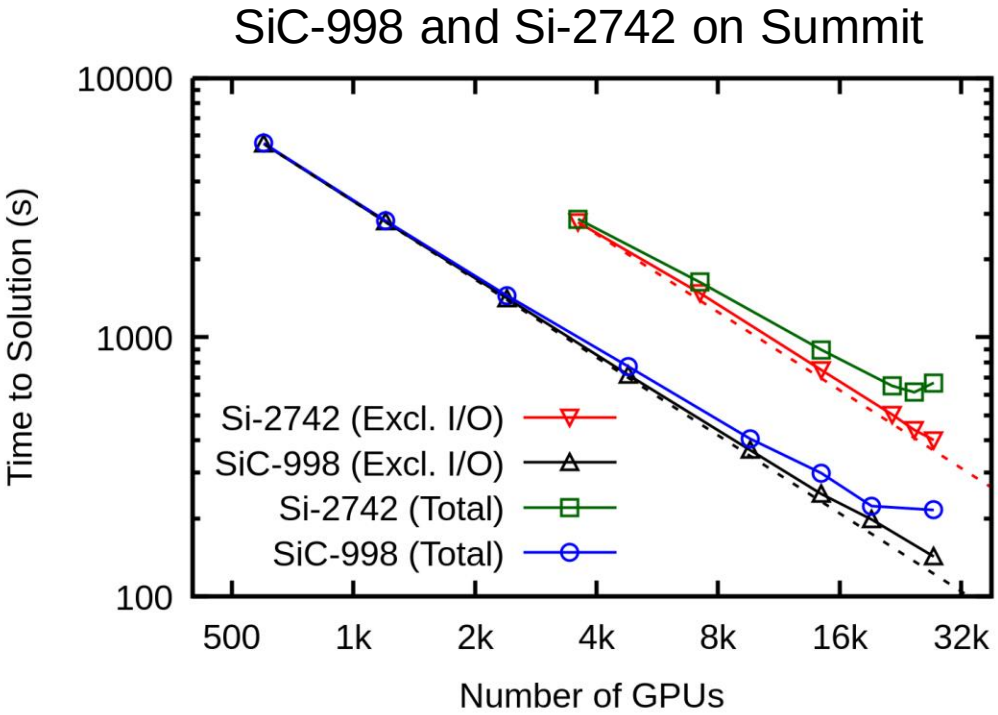
- Only the $O(N^4)$ part considered (CHI-0 and GPP-kernel)
- Good weak scaling within 5%

Epsilon: Strong Scaling



CHI-0 kernel scales linearly well to thousands of GPUs

Sigma: Strong Scaling (Preliminary)



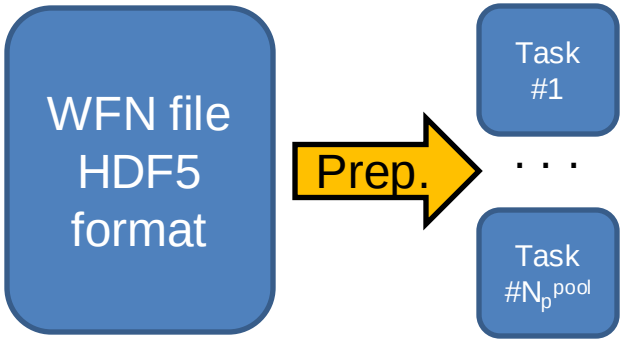
Scaling up to 4560 Summit nodes (27,360 GPU's) 99% of Summit.

I/O Optimization

- Exploit node local solid-state memory (SSD)
  - Prepare data → distributed form, at no cost
  - Prestage data → node local SSD
  - Runtime → Each rank read directly from local SSD

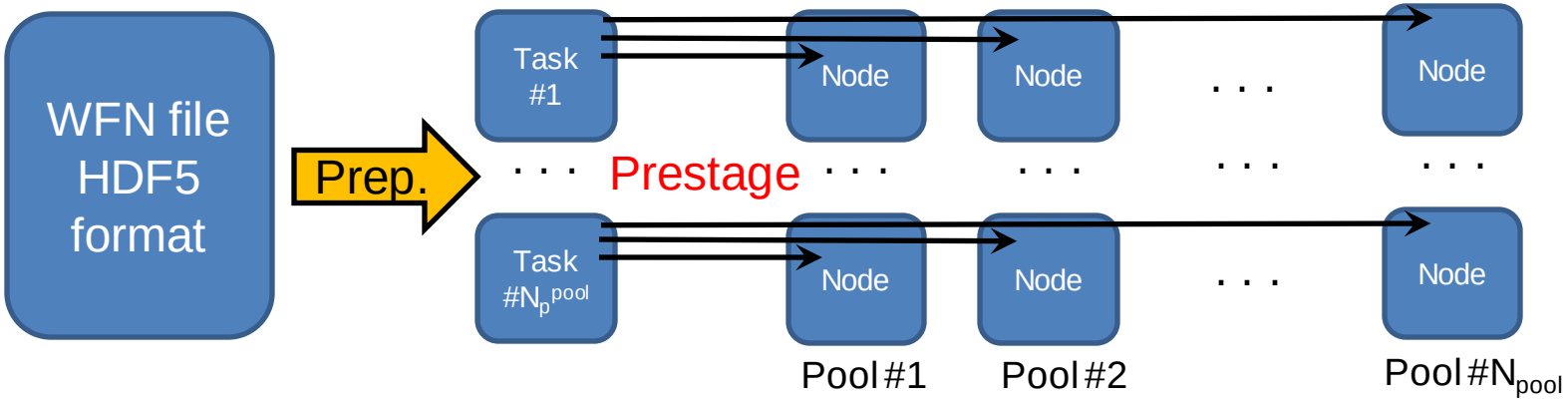
I/O Optimization

- Exploit node local solid-state memory (SSD)
  - **Prepare data** → distributed form, at no cost
  - Prestage data → node local SSD
  - Runtime → Each rank read directly from local SSD



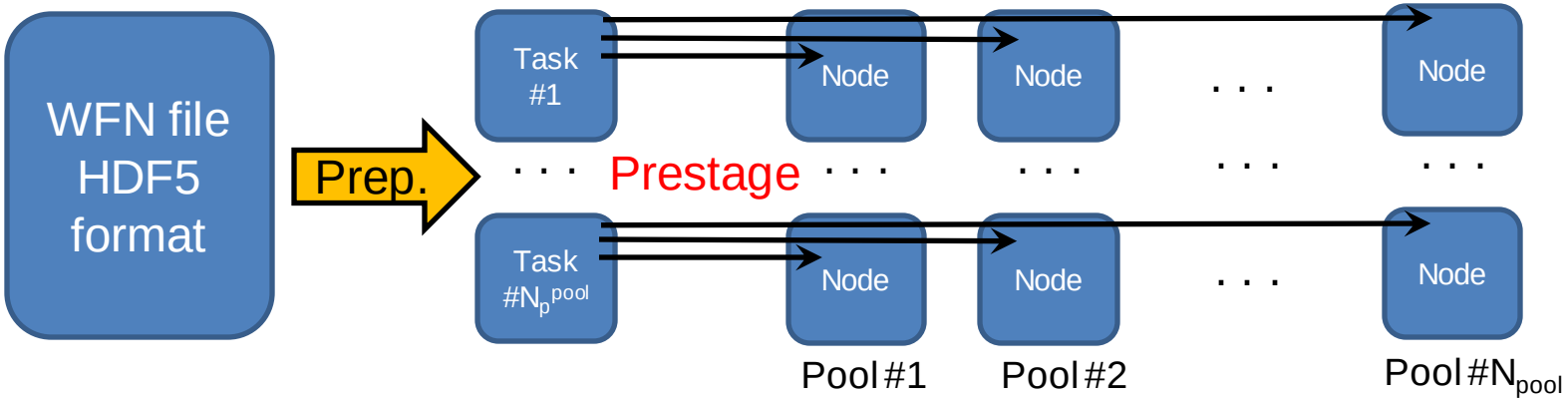
I/O Optimization

- Exploit node local solid-state memory (SSD)
  - Prepare data → distributed form, at no cost
  - **Prestage data** → node local SSD
  - Runtime → Each rank read directly from local SSD

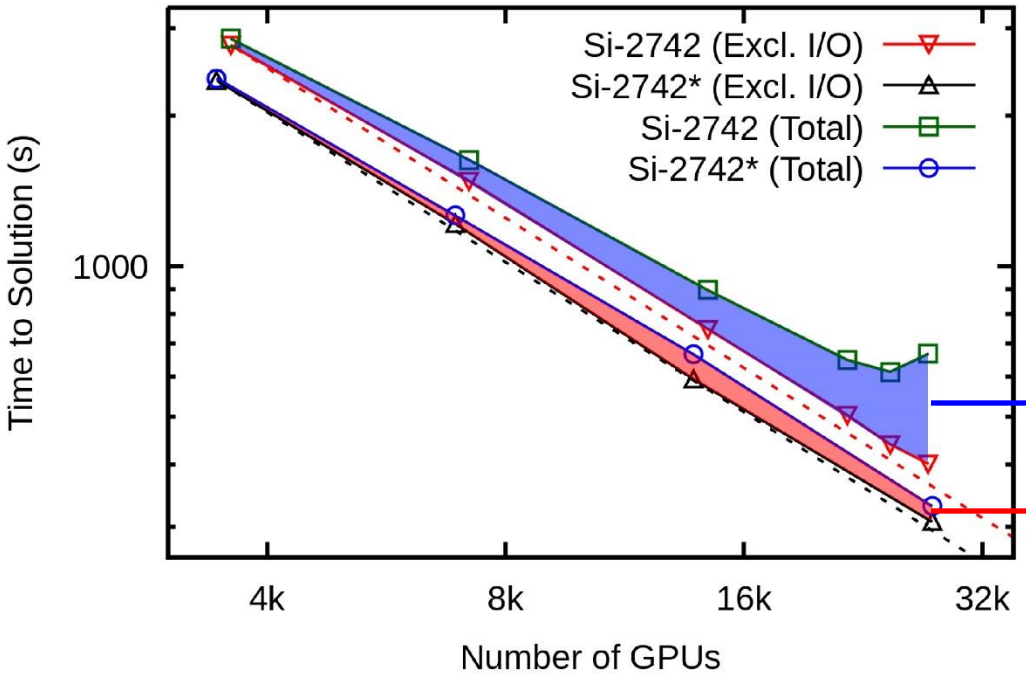


I/O Optimization

- Exploit node local solid-state memory (SSD)
  - Prepare data → distributed form, at no cost
  - Prestage data → node local SSD
  - **Runtime** → Each rank read directly from local SSD



BerkeleyGW: Full Application at Scale



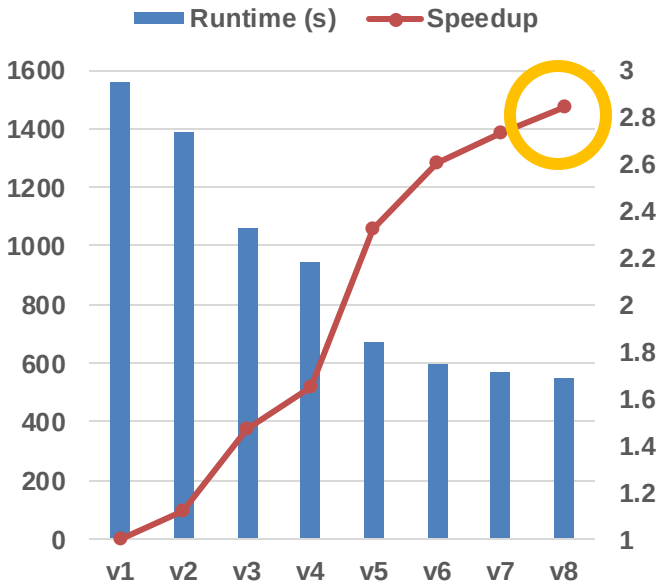
I/O Optimization (SSD):
266s to 23s

(Blue shade) **no** Prestage to SSD

(Red shade) **with** Prestage to SSD

*Results with optimized I/O include some further optimization in GPP-kernel (v8)

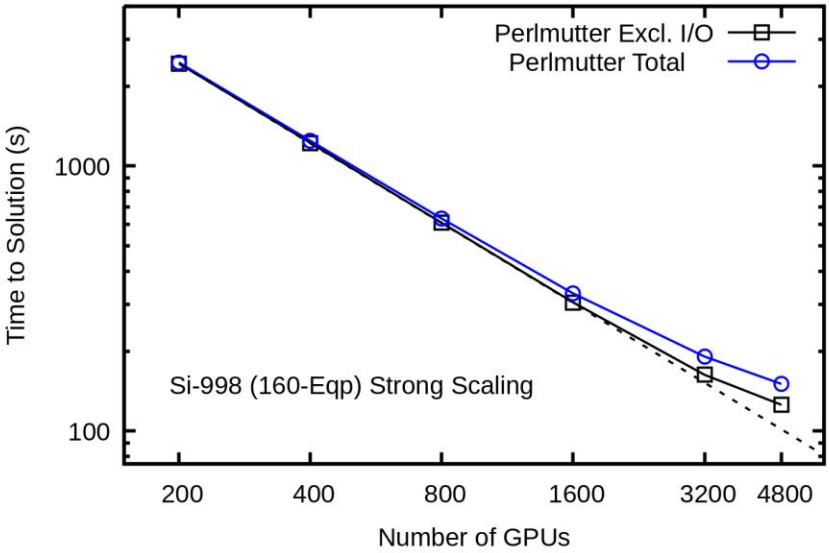
BerkeleyGW: Full System Run

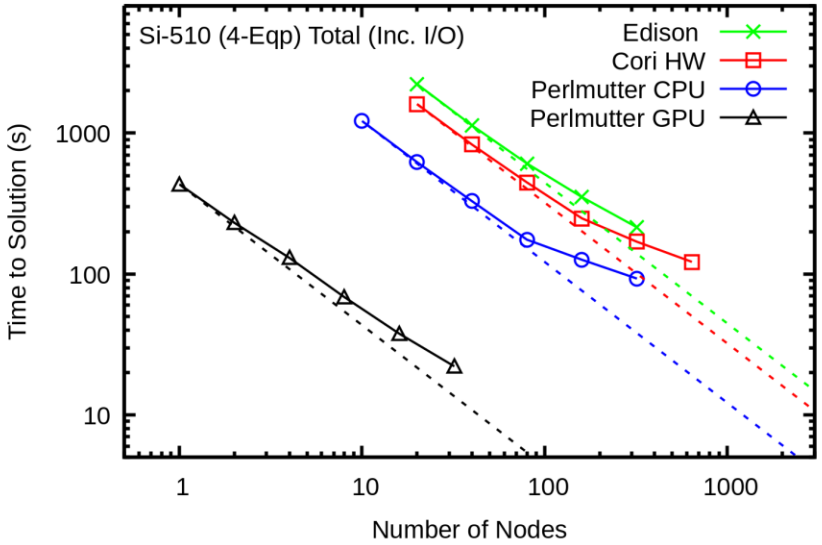


Application: 105.9 PFLOP/s

Application	BerkeleyGW
Benchmark	Si-2742
# Eqp	256
# of GPUs	27,648 (full Summit)
Pool Size	108 GPUs
Compute Time	592 s
I/O Time	39 s
Throughput	105.9 PFLOP/s
% of R_{peak}	52.7% of 200.79 PFLOP/s

Sigma: Performance on Perlmutter (GPU)

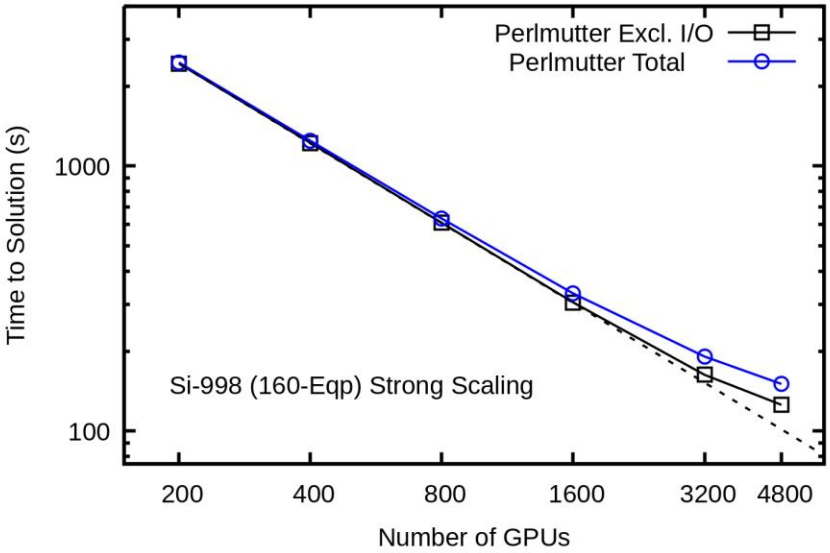


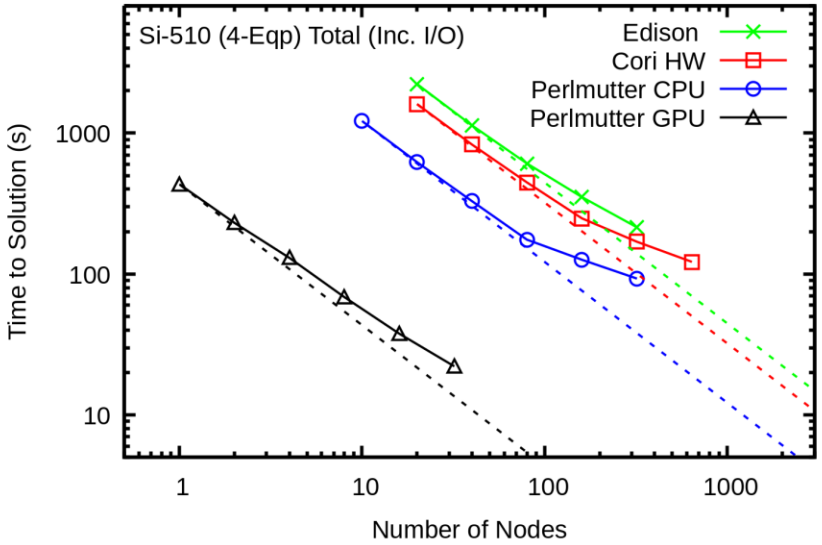


Strong Scaling for Sigma measured on Perlmutter@NERSC (Cray Shasta, Node: 2 AMD Milan + 4-A100 GPUs)

- Left: Strong scaling to (almost) entire Perlmutter
- Right: Comparison between CPU (Cori-Haswell) and GPU (Perlmutter)

Sigma: Performance on Perlmutter (GPU)

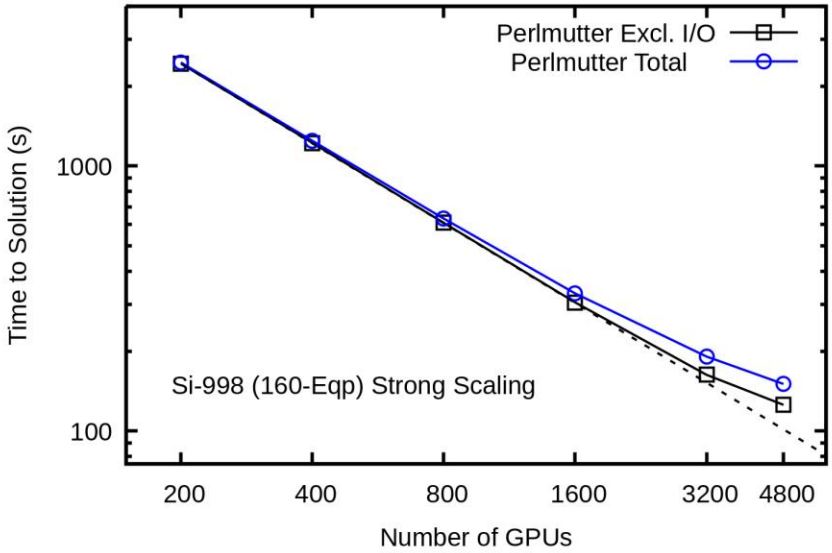


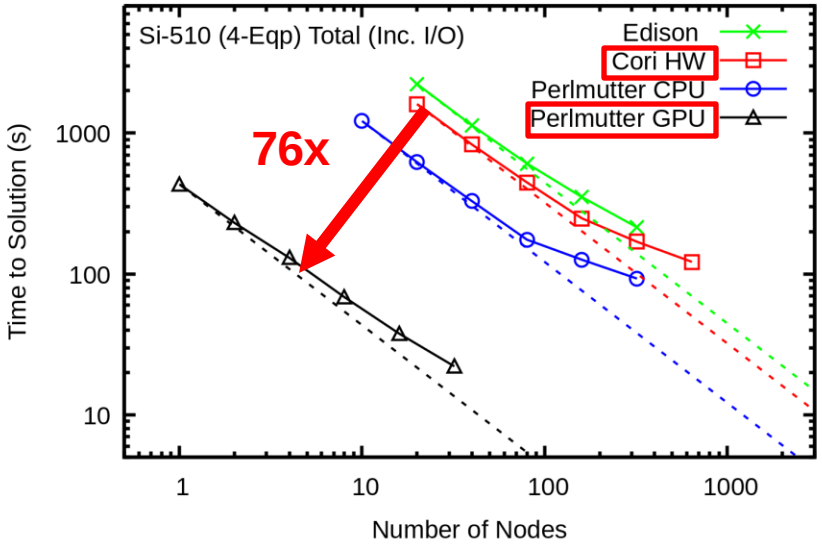


Strong Scaling for Sigma measured on Perlmutter@NERSC (Cray Shasta, Node: 2 AMD Milan + 4-A100 GPUs)

- **Left: Strong scaling to (almost) entire Perlmutter**
- Right: Comparison between CPU (Cori-Haswell) and GPU (Perlmutter)

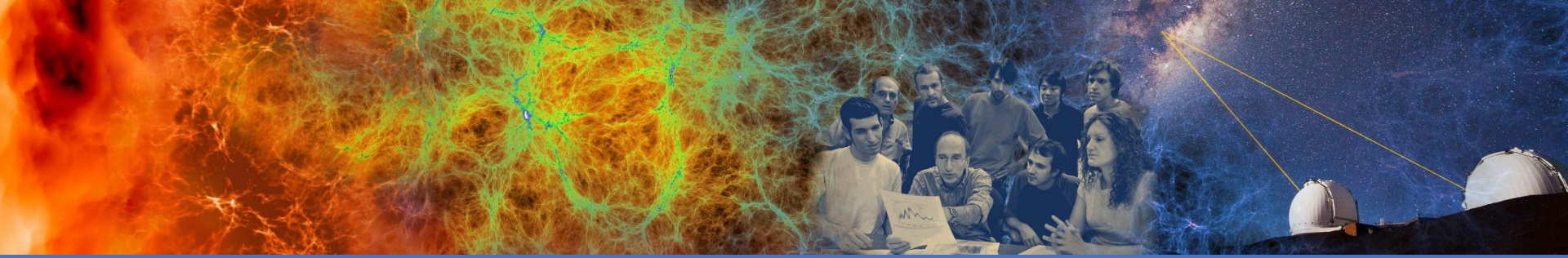
Sigma: Performance on Perlmutter (GPU)





Strong Scaling for Sigma measured on Perlmutter@NERSC (Cray Shasta, Node: 2 AMD Milan + 4-A100 GPUs)

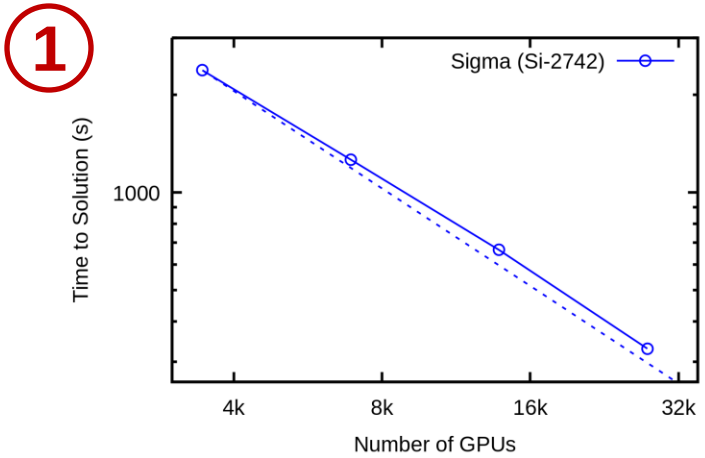
- Left: Strong scaling to (almost) entire Perlmutter
- **Right: Comparison between CPU (Cori-Haswell) and GPU (Perlmutter)**

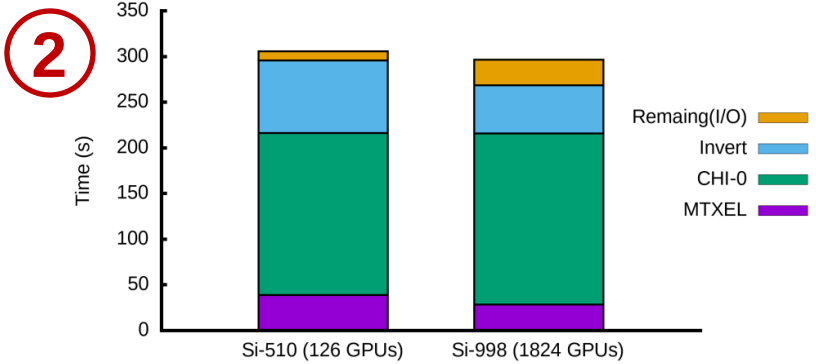


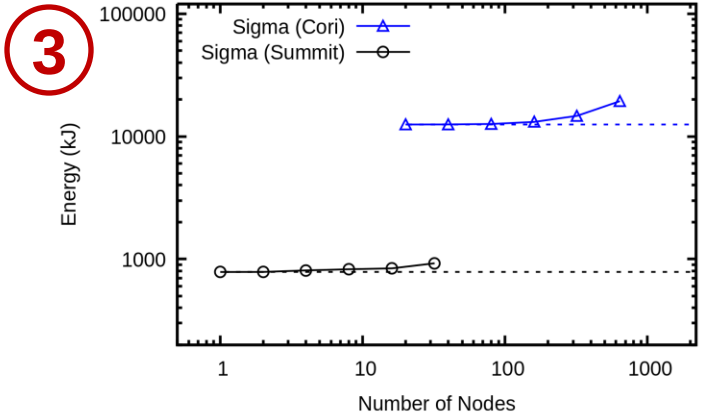
Summary

Summary

1. Linear Strong Scaling
2. Linear Weak Scaling
3. Up to 16x Energy Savings
4. Performance portability across programming models







Acknowledgement

- This research used resources at the National Energy Research Scientific Computing Center (NERSC), which is supported by the U.S. Department of Energy Office of Science under contract DE-AC02-05CH11231.

- This research used resources at the Oak Ridge Leadership Computing Facility (OLCF) through the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program, which is supported by the U.S. Department of Energy Office of Science under Contract No. DE-AC05-00OR22725.

- This work was supported by the Center for Computational Study of Excited-State Phenomena in Energy Materials (C2SEPEM), funded by the U.S. Department of Energy Office of Science under Contract No. DEAC02-05CH11231.

OpenMP®

Large-Scale Materials Science Codes Porting Strategies on GPU Architectures, the BerkeleyGW Case Study

www.openmp.org

A recording of this webinar, as well as the slides, will be available later today on our website under “News & Events”