Update on the abstract accelerated infrastructure

Jacob King* (Tech-X)

Thanks to Eric Howell for code review

NIMROD Team Meeting
May 24th 2023

Work supported by US DOE

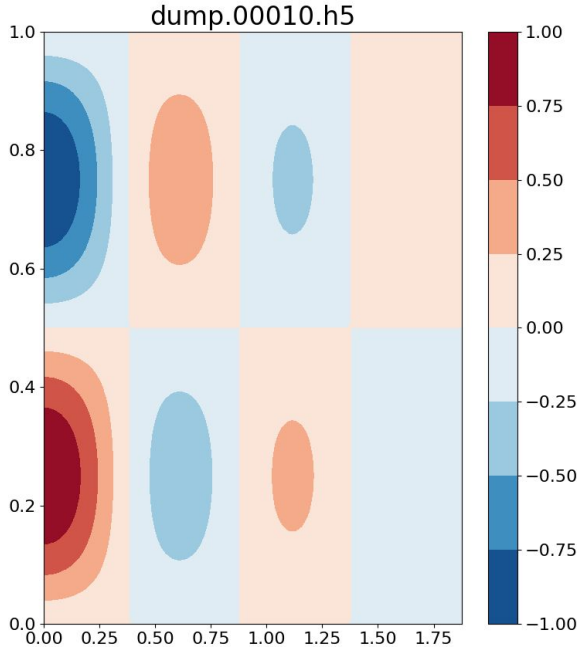
*Also affiliated with Fiat Lux

The full name is now *NIMROD abstract accelerated infrastructure* (NIMROD-AAI)

- Better reflects intention:
- Abstract types to enable implementation flexibility, envision varied:
  - Discretizations (e.g. structured, unstructured, dimensionality, shape)
  - Hardware optimizations (GPU vs CPU)
  - Function spaces (e.g. H1, HCurl)
- Accelerated – focus on GPU acceleration for new architectures
- Infrastructure – code does not include physical equation modules of interest, separate repository enforces a clean separation

Laplace example provides test case

- Known time-discretized solution with self-similar decay ensures correctness of implicit formulation
- Sine/Cosine waves and Bessel functions used in slab and cylindrical geometry, respectively
- Time-step loop is focus of GPU optimization



A plot of the cylindrical solution with a regularity condition (left), periodic boundary conditions (top/bottom) and an essential condition (right)

Test case parameters and performance expectations

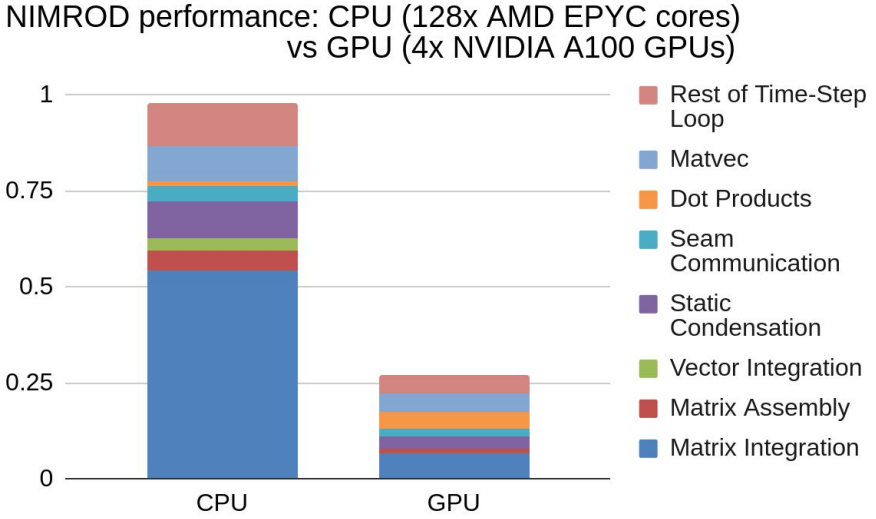
- nvhpc nvfortran compiler
- Rough calculations for Perlmutter:

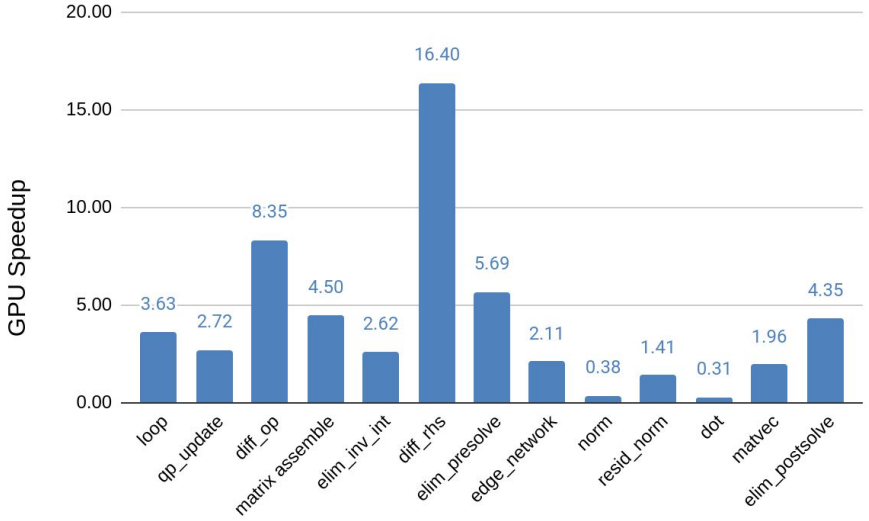
- Grid mx=my=256; pd=6
- Use nxbl=nybl=32 on CPU
- Use nxbl=nybl=2 on GPU
- nstep=4, dt=0.01, diff=1, theta=0.5

Nodes	Hardware	TFlops (DP)	Bandwidth (GB/s)
3072	2x EYPC Milan CPU	4	400
1536	4x Nvidia A100 GPU	40	8000
	1x EYPC Milan CPU	2	200

- 1 GPU node ≈ 10-20 CPU nodes
- All GPU nodes ≈ 5-10x all CPU nodes
- Compare performance on 1 CPU node (128 MPI procs) to 1 GPU node (4 MPI procs + 4 GPUs)

Preset status: GPU performance improved to ~4x





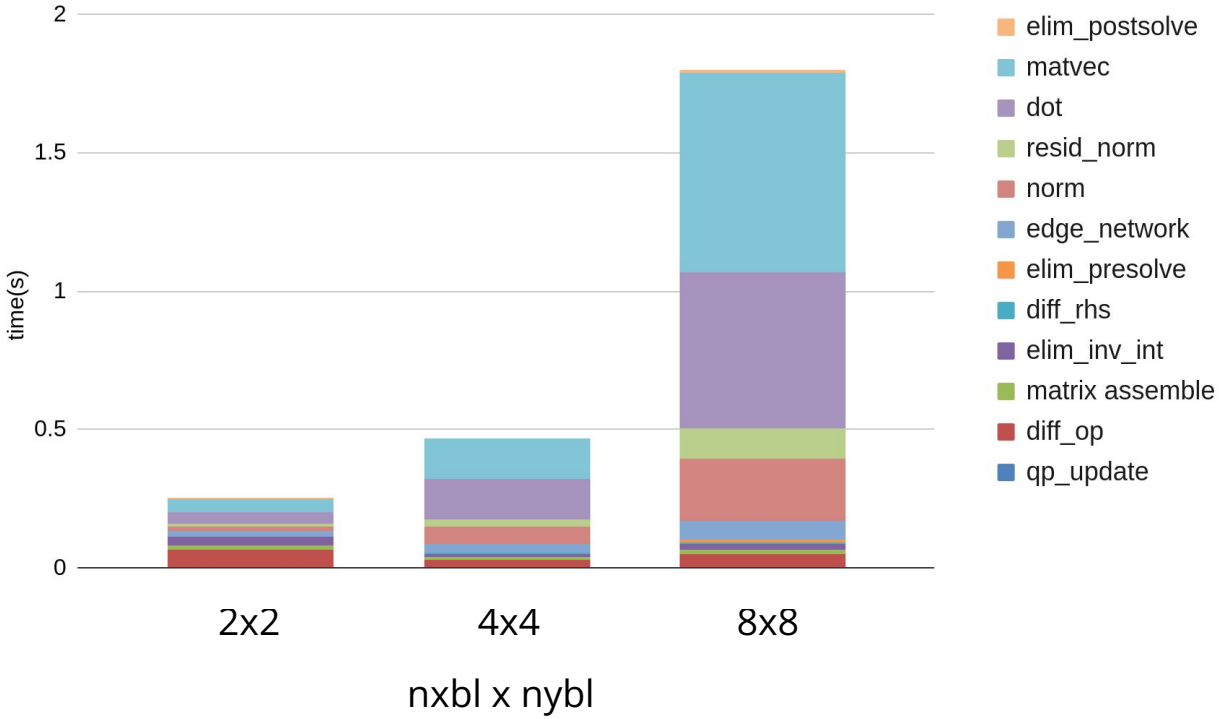
Optimization work not finished but ongoing expect 5-10x on application

Nsys profiler shows room for improvement

Profiling with nsys shows that `inf_norm`, `dot` and `matvec` routines are dominated by host-device memory transfers (red under OpenACC) and not compute (blue under CUDA HW). GPU kernel timings for the reduction operations are ~10x less than the reported times. Further optimization to remove the small host-device memory transfers is possible.

Good topic for a hackathon!

Using more, smaller blocks degrades performance



Items for improvement:

- Kernel fusion
- GPU RDMA
- CPU/GPU block decomp w/ small blocks on CPUs

Kernel fusion improves performance relative to May 2023

Caveats on the test case performance

Nonlinear NIMROD simulations:

- FE vector integration for matrix-free dot product which is well optimized (16x speed up)

Laplace example:

- Uses matvec to compute the action of the linear-operator which is not as well optimized (2x speed up)

Expect better performance on a nonlinear test case

Jacobi preconditioner implemented

Jacobi iteration for the problem $A\mathbf{x} = \mathbf{b}$ is defined as

$$\mathbf{x}^{(k+1)} = D^{-1} \cdot \left(\mathbf{b} - (L + U) \cdot \mathbf{x}^{k}\right) \tag{1}$$

where $\mathbf{x}^{k}$ is the solution vector at superscript iteration number, k, $\mathbf{b}$ the RHS vector, and the matrix $A = L + U + D$ is decomposed in lower triangular, upper triangular and diagonal components, respectively.

Rearranging terms, we find

$$\mathbf{x}^{(k+1)} = D^{-1} \cdot \mathbf{r}^{k} + \mathbf{x}^{k} \tag{2}$$

where the residual vector is defined as $\mathbf{r}^{k} = \mathbf{b} - A \cdot \mathbf{x}^{k}$.

This form is related to the Jacobi preconditioner, $D^{-1} \cdot \mathbf{r}^{k}$, used in `apply_precon`. The `update` step computes the updated D^{-1} and stores the result as a vector.

Abstract base requires preconditioner update/apply

```
  SUBROUTINE update_real(precon,mat,seam)
    USE linalg_utils_mod
    USE pardata_mod
    USE seam_mod
    IMPLICIT NONE

    !> preconditioner to update
    CLASS(precon_jacobi_real), INTENT(INOUT) :: precon
    !> matrix M to be used for preconditioning
    CLASS(rmat_storage), DIMENSION(:), INTENT(IN) :: mat
    !> associated seam
    TYPE(seam_type), INTENT(INOUT) :: seam

    INTEGER(i4) :: ibl
    INTEGER(i4) :: idepth
    INTEGER(i4), SAVE :: iftn=-1

    CALL timer%start_timer_l2(mod_name,'update_real',iftn,idepth)
!-------------------------------------------------------------------------------
!   matrices are stored without seaming, first extract the diagonal
!-------------------------------------------------------------------------------
    DO ibl=1,par%nbl
      CALL mat(ibl)%m%get_diag_as_vec(precon%inv_diag_mat(ibl)%v)
      CALL precon%inv_diag_mat(ibl)%v%edge_load_arr(seam%s(ibl),               &
                                                    do_avg=apply_ave_factor)
    ENDDO
!-------------------------------------------------------------------------------
!   next apply seaming
!-------------------------------------------------------------------------------
    CALL seam%edge_network
!-------------------------------------------------------------------------------
!   finally take the inverse that includes combined values along block borders
!-------------------------------------------------------------------------------
    DO ibl=1,SIZE(mat)
      CALL precon%inv_diag_mat(ibl)%v%edge_unload_arr(seam%s(ibl))
      CALL precon%inv_diag_mat(ibl)%v%invert
    ENDDO
    CALL timer%end_timer_l2(iftn,idepth)
  END SUBROUTINE update_real
```

```
SUBROUTINE apply_precon_real(precon,mat,res,zee,adr)
  USE pardata_mod
  IMPLICIT NONE

  !> preconditioner to apply
  CLASS(precon_jacobi_real), INTENT(INOUT) :: precon
  !> matrix M to be used for preconditioning
  CLASS(rmat_storage), DIMENSION(:), INTENT(IN) :: mat
  !> the residual res = rhs - A x_k
  TYPE(rvec_storage), DIMENSION(:), INTENT(IN) :: res
  !> the preconditioned residual, zee = M^-1 res
  TYPE(rvec_storage), DIMENSION(:), INTENT(INOUT) :: zee
  !> the full matrix vector product, A x_k
  TYPE(rvec_storage), DIMENSION(:), INTENT(IN) :: adr

  INTEGER(i4) :: ibl
  INTEGER(i4) :: idepth
  INTEGER(i4), SAVE :: iftn=-1

  CALL timer%start_timer_l2(mod_name,'apply_precon_real',iftn,idepth)
  DO ibl=1,par%nbl
    CALL precon%inv_diag_mat(ibl)%v%apply_diag_matvec(res(ibl)%v,zee(ibl)%v)
  ENDDO
  CALL timer%end_timer_l2(iftn,idepth)
END SUBROUTINE apply_precon_real
```

Function pointer used to enable flexibility for BCs, etc.

```
  TYPE, ABSTRACT :: precon_real
!-------------------------------------------------------------------------------
!   Input
!-------------------------------------------------------------------------------
    !* custom function to call before preconditioning (e.g. to save state)
    PROCEDURE(precon_func), POINTER, NOPASS :: f_precon_init
    !* custom function to call after preconditioning (e.g. for BC/regularity)
    PROCEDURE(precon_func), POINTER, NOPASS :: f_precon_finalize
!-------------------------------------------------------------------------------
!   Diagnostic output
!-------------------------------------------------------------------------------
    !* return code (for external packages)
    INTEGER(i4) :: return_code
    !* error message
    CHARACTER(128) :: error_msg
  CONTAINS

    ! Deferred routines
    PROCEDURE(dealloc_real), PASS(precon), DEFERRED :: dealloc
    PROCEDURE(update_real), PASS(precon), DEFERRED :: update
    PROCEDURE(apply_precon_real), PASS(precon), DEFERRED :: apply_precon
    ! Defined routines
    PROCEDURE, PASS(precon) :: apply => apply_real
  END TYPE precon_real
```

```
  SUBROUTINE apply_real(precon,mat,res,zee,adr)
    USE matrix_mod
    USE vector_mod
    IMPLICIT NONE

    !> preconditioner to apply
    CLASS(precon_real), INTENT(INOUT) :: precon
    !> matrix M to be used for preconditioning
    CLASS(rmat_storage), DIMENSION(:), INTENT(IN) :: mat
    !> the residual res = rhs - A x_k
    TYPE(rvec_storage), DIMENSION(:), INTENT(IN) :: res
    !> the preconditioned residual, zee = M^-1 res
    TYPE(rvec_storage), DIMENSION(:), INTENT(INOUT) :: zee
    !> the full matrix vector product, A x_k
    TYPE(rvec_storage), DIMENSION(:), INTENT(IN) :: adr

!-------------------------------------------------------------------------------
!   call init
!-------------------------------------------------------------------------------
    IF (ASSOCIATED(precon%f_precon_init))                                      &
      CALL precon%f_precon_init(mat,res,zee,adr)
!-------------------------------------------------------------------------------
!   call apply
!-------------------------------------------------------------------------------
    CALL precon%apply_precon(mat,res,zee,adr)
!-------------------------------------------------------------------------------
!   call finalize
!-------------------------------------------------------------------------------
    IF (ASSOCIATED(precon%f_precon_finalize))                                  &
      CALL precon%f_precon_finalize(mat,res,zee,adr)
  END SUBROUTINE apply_real
```

Mirrors functionality of threed_dot_mgt for preconditioning, plan to extend abstract design to dot product

Recent: extension to complex types

Development plan followed:

- Implement and optimize real types first
- Extend to complex when ready to avoid code refactoring overhead
- Nodal function space representation and linear algebra complex types added
- Remaining: preconditioning, iteration and dump file inclusion

Complex types required for upcoming fluid implementation

Future development directions

Near term priorities

- FFTs #33 – Plan to use FFTW and cuFFT
- Export to CSR/CSC format #114 and interface with external solvers
- Rework seam norm/tang #121 for abstraction

- GPU RDMA in seams #111
- Generalized infrastructure for surface and boundary integrals #115
- Heterogeneous CPU/GPU block decomposition #116
- Batched FEM evaluation at unstructured points #117
- Abstract interface to solve the inverse problem #118
- Use CI@NERSC to test for performance regression on Perlmutter #119

Aside: How should bndry Norm/Tang be defined?

In the base version of NIMROD norm/tang in the seams are set at the seam nodes at run time using the formulas

$$\hat{t} = \frac{\left(\frac{\partial R}{\partial x}, \frac{\partial Z}{\partial y}\right)}{\left|\left(\frac{\partial R}{\partial x}, \frac{\partial Z}{\partial y}\right)\right|}$$

and $\hat{n} = \hat{t} \times \hat{\Phi}$. There are a couple issues with this to unpack.

1. This formula makes an assumption about the block orientation and is not general.
2. With C0 elements, there is a discontinuity across elements. The values at the vertex nodes are averaged.
3. For edge integration at quadrature points with nimdevel's Lagrange edge types, evaluation of the FEM basis is used. This can produce odd behavior at corners with the above averaging procedure.

Could this be an issue with the n=0 resistive wall?

NIMROD python code organized into formal package

With apologies to Brian Nelson, we've called it nimpy

https://gitlab.com/NIMRODteam/nimpy

Routines for basis function and mathematical evaluation, FSA, flux-coordination transformations, plotting, etc.

Installed package is mature, reviewed code but scripts directory provides a place for storing/exchanging developing capabilities.

Thanks to Eric Howell for many contributions

Final remarks

- NIMROD abstract accelerated infrastructure is running on the GPUs
  - A Perlmutter GPU node is faster than a Perlmutter CPU node
- GPU optimization and improving code features is ongoing
- Building generalized multispecies code on top of abstract infrastructure