Code refactoring patterns targeting bandwidth optimized architectures and heterogeneous architectures

Jacob Poulsen

20th ECMWF workshop on high performance computing in meteorology

EVP overview

EVP is short for the solver for the *Elastic-Viscous-Plastic* equations from E. Hunke@LANL.

This algorithm constitutes a long acknowledged severe scaling obstacle for several forecast and climate systems

Previous efforts on dealing with this challenge have not focused on the implementation itself nor on the utilization of bandwidth-optimized hardware.

The parallelization refactoring will take place at the **core-level**, the **node-level** and the **cluster-level**.

The performance study associated with the code refactoring have focused on **Capacity Scaling** and **Strong Scaling** at the node level. The code refactoring is based on general patterns that can be re-used in other contexts.

Fingerprints

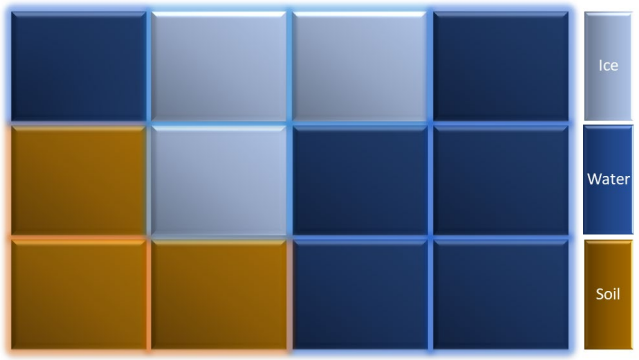
Memory:
- Irregular domain of active points
- Finite difference implying irregular access pattern
- Multiple definitions of active points (U&T cells, two masked grids)

Arithmetic:
- Short latency only (add, mult , div, sqrt)
- Computation intensity ~ 0.3 FLOP/Byte

Parallelization:
- Current hybrid approach based on general 2D blocking with thinning
- Halo swap after each outer iteration so any imbalance will be severely exposed by this component
- Two different inner iteration spaces

SLOCcount : ~4K (birds-eye view will do for now)

Testcases shared: Forecast and Climate, Winter and Summer



Birds-eye view on the EVP solver

```
do k = 1, niter      ! niter iteration per model timestep
  ! stage1: use variables on T cells and velocities on U cells to
  ! define stress* on T cells and stage-interface vectors
  do i=1,nt          ! nt is number of active T cells at given ts
    ! FD computations here
    ...
  enddo
  ! stage2, use variables on U cells and stage-interface
  ! to define new velocities* and new vars on U cells
  do j=1,nu          ! nu is number of active U cells at given timestep
    ! FD computations here
    ...
  enddo
  ! data dependencies: note that references in stage1 are set in stage2
  ! halo_swap with MPI neighbours
enddo
```

Refactoring target - Core

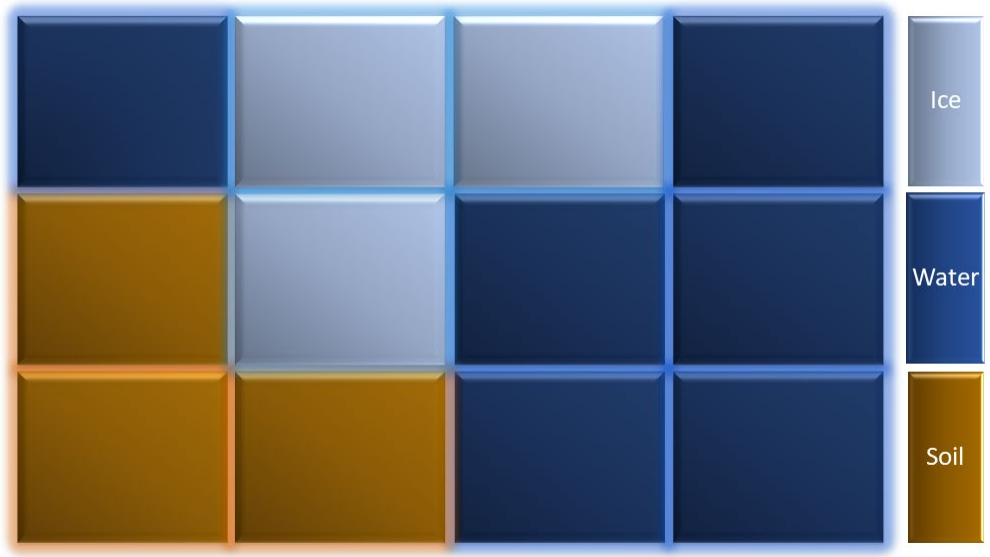
Core level:

- Focus on loops and ensure that loops of interest support SIMD code generation
  - EVP: 2D data structures -> 1D data-structures, 9 -point stencil via indirect addressing

Refactoring– core level (SIMD)

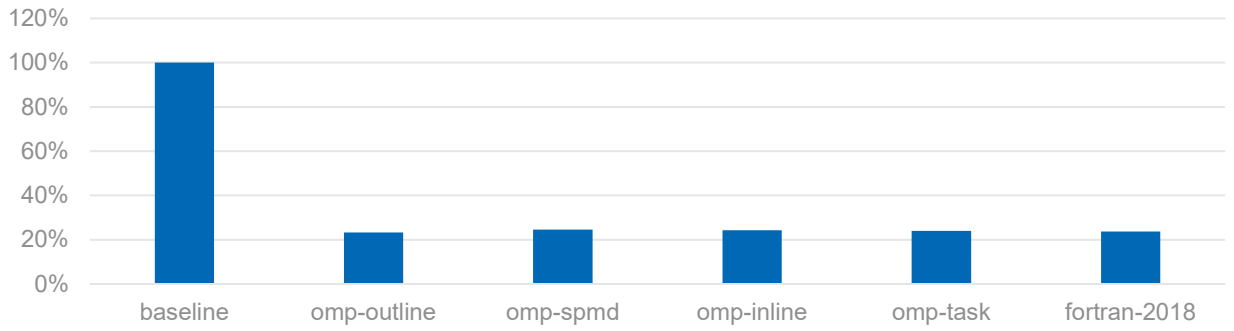
```
subroutine stress_v0(...,nx,ny,icellt,indxti,...)
  ! indirect addressing
  real (kind=dbl_kind), intent(in), dimension(nx,ny) :: cyp, ...
  ...
  do ij = 1, icellt ! loop over active points
     i = indxti(ij) ! lookup 2D orientation
     j = indxtj(ij)
     ! smaller FD-block with column dependencies (i+-1,j+-1)
     divune    = cyp(i,j)*uvel(i  ,j  ) - dyt(i,j)*uvel(i-1,j  )    &
               + cxp(i,j)*vvel(i  ,j  ) - dxt(i,j)*vvel(i  ,j-1)
     ...
     ! larger block with no column dependencies
     stressp_1(i,j) = (stressp_1(i,j) + c1ne*(divune - Deltane)) * denom1
     ...
  enddo
end subroutine stress_v0
```

```
subroutine stress_v1(...)
  ! direct addressing (but with gather operations to handle FD)
  real (kind=dbl_kind), intent(in), dimension(:), contiguous ::  cyp, ...
  ...
  do iw = 1, icellt ! loop over active points
     tmp_uvel_ee = uvel(ee(iw)) ! ee(iw) look up neighbour index
     ...
     ! smaller FD-block, column dependencies (i+-1,j+-1) via helper index ee, ...
     divune    = cyp(iw)*uvel(iw) - dyt(iw)*tmp_uvel_ee                    &
               + cxp(iw)*vvel(iw) - dxt(iw)*tmp_vvel_se
     ...
     ! larger block with no column dependencies
     stressp_1(iw) = (stressp_1(iw) + c1ne*(divune - Deltane)) * denom1
     ...
  enddo
end subroutine stress_v1
```



Single core (3rd gen 8360Y) for the same algorithm implemented via different approaches. Timings are averaged over 10 runs.

(lower is better)



Refactoring target- Node

Core level:

- Focus on loops and ensure that loops of interest support SIMD code generation

Node level:

- Review current parallelization pattern
- Focus on the parallelization unit used – improve granularity?
  - EVP: Block based granularity -> Point based granularity
- Focus on synchronization points – reduce them? And/Or make them more light weight ?
  - EVP: Explicit halo-swap between compute blocks -> OpenMP barrier
- Focus on temporary “stack” arrays (implementation overhead) used to convey results from one section to the next
  - EVP: Temporary arrays are already moved to the heap
- Focus on data, movement come at cost (Study the U -cells and the T-cells -> skip iterations in the 3%-4% range)
  - EVP: Merge iteration spaces to allow optimal runtime placement and binding, i.e. one decomposition (U ∪ T) instead of two decompositions (U and T)

We can now use OpenMP as a simple short-hand to code generation, cf. various options in the next couple of slides

Refactoring– node level- outline (CPU)

```
! OpenMP outlined, classical, caller
do k = 1, niter      ! niter iteration per model timestep
  !$omp parallel do schedule(runtime) private(iw)
  do iw = 1,na       ! na is union of nt+nu with logical skipT, skipU
    call stress(iw,...)
  enddo
  !$omp end parallel do
  !implicit sync via classical openMP semantics
  !$omp parallel do schedule(runtime) private(iw)
  do iw = 1,na       ! na is union of nt+nu with logical skipT, skipU
    call stepu(iw,...)
  enddo
  !$omp end parallel do
  !implicit sync via classical openMP semantics
enddo
```

```
! OpenMP outlined, classical, callee
subroutine stress (iw,...)
  integer (kind=int_kind), intent(in) :: iw
  ...
  if (skipt(iw)) return
  ! work on active t cell
  ...
  call strain_rates(...)
  ...
  call visc_replpress(...)
  ...
end subroutine stress
```

```
! OpenMP outlined,SPMD, caller
!$omp parallel private(i)
  do k = 1, niter      ! niter iteration per model timestep
    call stress(...)
    !$omp barrier      ! explicit barrier as this is in parallel section
    call stepu(...)
    !$omp barrier      ! explicit barrier as this is in parallel section
  enddo
!$omp end parallel
```

```
! OpenMP outlined, SPMD, callee
subroutine stress (...)
  integer (kind=int_kind), intent(in) :: na
  ...
  call domp_get_domain(1,na,il,iu) ! domp_get_domain may contain balancing logic
                                   ! and return lower and upper for the thread
  do iw = il, iu                   ! il and iu are the individual thread bounds
     if (skipt(iw)) cycle
     ! work on active t cell
  enddo
end subroutine stress
```

Refactoring– node level– inline (CPU)

```
! OpenMP inlined, classical, caller
do k = 1, niter      ! niter iteration per model timestep
  call stress(na,...)! na is union of nt+nu with logical skipT, skipU
  call stepu(na,...) ! na is union of nt+nu with logical skipT, skipU
enddo
```

```
! OpenMP inlined, classical, callee
  !$omp parallel do schedule(runtime) &
  !$omp default(none) &
  !$omp private(…) &
  !$omp shared(…)
  do iw = 1, na
    if (skipme(iw)) cycle
    ...
    call strain_rates(...)
    ...
    call visc_replpress(...)
    ...
  enddo
  !$omp end parallel do
```

```
! OpenMP inlined, classical, caller
do k = 1, niter      ! niter iteration per model timestep
  call stress(na,...)! na is union of nt+nu with logical skipT, skipU
  call stepu(na,...) ! na is union of nt+nu with logical skipT, skipU
enddo
```

```
! OpenMP inlined, classical, callee
  !$omp parallel single schedule(runtime) &
  !$omp taskloop simd &
  !$omp default(none) &
  !$omp private(…) &
  !$omp shared(…)
  do iw = 1, na
    if (skipme(iw)) cycle
    ...
    call strain_rates(...)
    ...
    call visc_replpress(...)
    ...
  enddo
  !$omp end taskloop simd
  !$omp end single
  !$omp end parallel
```

Refactoring– node level (GPU)

```
! OpenMP target outlined, caller
!$omp target data  map(to:      ... )
!$omp              map(tofrom: ... )
!$omp target update to(         ... )
do k = 1, niter      ! niter iteration per model timestep
  !$omp target teams distribute parallel do
  do iw = 1,na       ! na is union of nt+nu
    call stress(iw,...)
  enddo
  !$omp end target teams distribute parallel do
  !$omp target teams distribute parallel do
  do iw = 1,na       ! na is union of nt+nu
    call stepu(iw,...)
  enddo
  !$omp end target teams distribute parallel do
enddo
!$omp end target data
```

```
! OpenMP target outlined, callee
subroutine stress (iw,...)
  integer (kind=int_kind), intent(in) :: iw
  ...
!$omp declare target
  if (skipme(iw)) return
  ! work on active t cell
  ...
  call strain_rates(...)
  ...
  call visc_replpress(...)
  ...
end subroutine stress
```

```
! OpenMP target inlined, caller
!$omp target data  map(to:      ... )
!$omp              map(tofrom: ... )
!$omp target update to(         ... )
do k = 1, niter
  call stress(1,na,...)
  ! serial so no explicit barrier needed here
  call stepu(1,na,...)
  ! serial so no explicit barrier needed here
enddo
!$omp end target data
```

```
! OpenMP target inlined, callee
subroutine stress (lb,ub,...)
  integer (kind=int_kind), intent(in) :: lb, ub
  ...
  !$omp target teams distribute parallel do
  do iw = lb, ub
    if (skipme(iw)) cycle
    ! work on active t cell
    ...
    call strain_rates(...)
    ...
    call visc_replpress(...)
    ...
  enddo
  !$omp end target teams distribute parallel do
end subroutine stress
```

```
! Fortran 2018 (both GPU and CPU), caller
do k = 1, niter
  call stress(1,na,...)
  ! serial so no explicit barrier needed
  call stepu(1,na,...)
  ! serial so no explicit barrier needed
enddo
```

```
! Fortran 2018 (both GPU and CPU), callee
subroutine stress (lb,ub,...)
  integer(kind=int_kind), intent(in) :: lb, ub
  ...
  do concurrent (iw=lb:ub) DEFAULT(NONE) &
                           SHARED(...)   &
                           LOCAL(...)
    if (skipme(iw)) cycle
    ! work on active t cell
    ...
    call strain_rates(...)
    ...
    call visc_replpress(...)
    ...
  enddo
end subroutine stress
```

Performance study

Performance studies across CPU and GPU offerings:

1) Strong scaling, how fast can we run on the node? This work-load is dominated by bandwidth bound so the scaling will tail off once it become saturated. For DDR-based memory systems the saturation will happen earlier than for HBM -based memory systems.

2) Weak scaling, focus on the special case of weak scaling called capacity scaling. With **Capacity scaling** the code is not required to scale beyond the unit for each ensemble, so all w eak scaling challenges are on the HW and SW stack and not on a mix of the application itself and the HW and SW stack.

3) Simple roofline to quantify sustained performance using a CPU/GPU centric metric for easier interpretation of results and to ensure that our baselines for computing improvement factors are always running at “peak” performance.

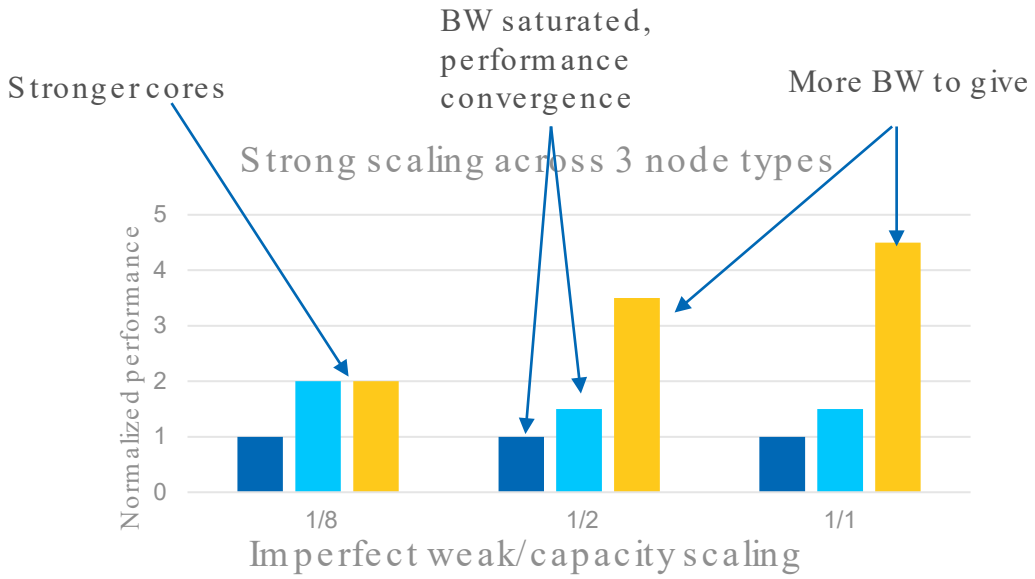
4) Performance/watt – we will also normalize results according to the watts required.

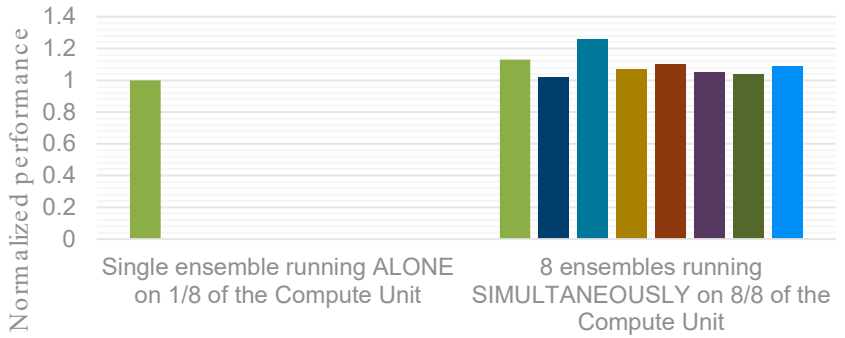
Method: All performance numbers reported are the **average time** obtained when repeating the test 10x times. All timings are obtained using omp_get_wtime ().

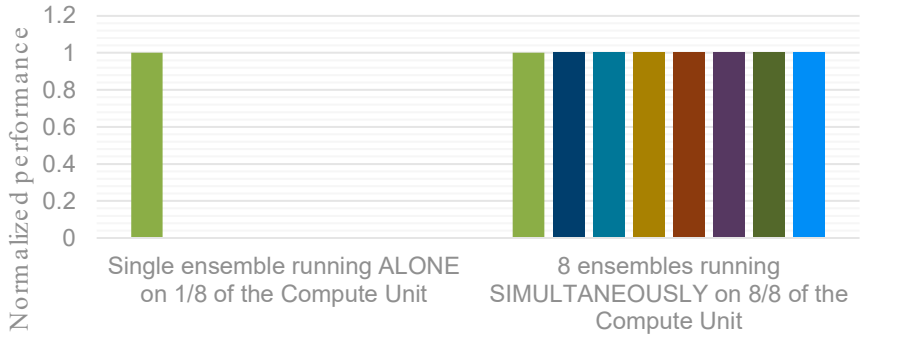
Definition for capacity measurements: An ensemble of 8 ensemble members is running the same workload simultaneously across NUMA domains or half tiles. The timing of an ensemble run is the time of the **slowest** ensemble member. We have repeated the ensemble runs 10 times and report **the average over these 10 times** .

For the GPU results, it is only the compute part that is reported (the kernel is a single model time - step so most traffic in the kernel is one -time initialization traffic). Moreover, the watts measurements on the GPU results are for the GPU device plus the host device.

Reproducible results: All results are obtained using open standards and freely available compilers (OneAPI). All the source code is made available via the github repository: https://github.com/dmidk/CICE -kernel

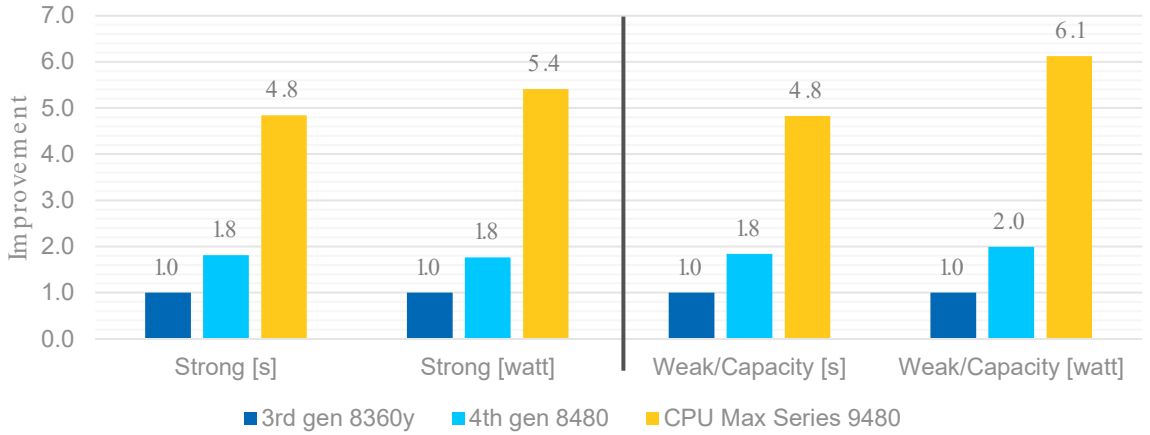




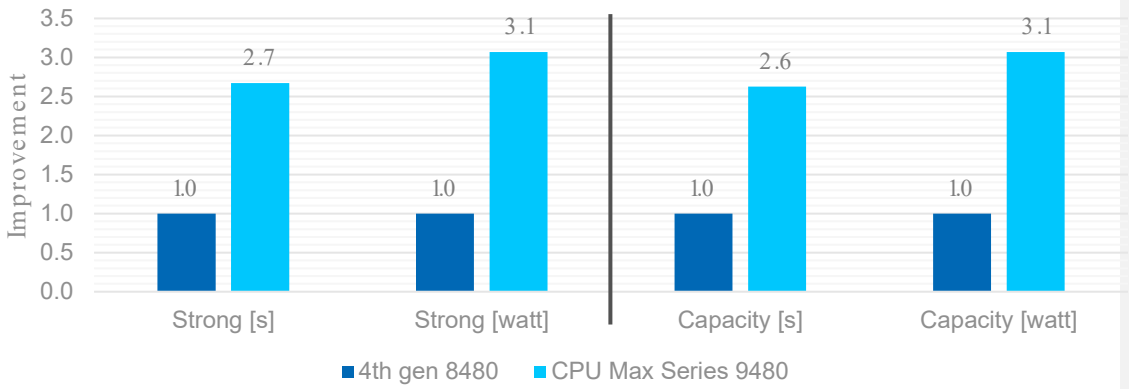


CPU performance

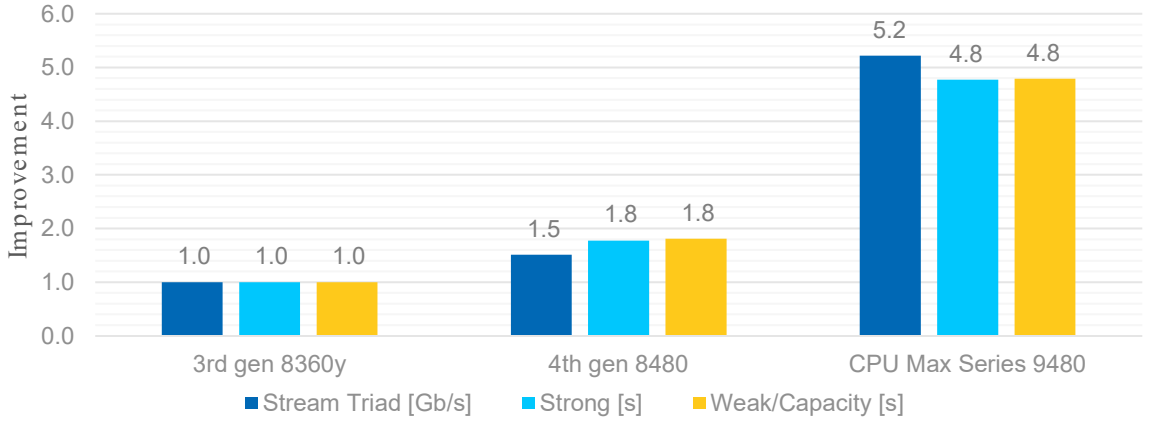
Normalized performance improvement of EVP
(higher is better)



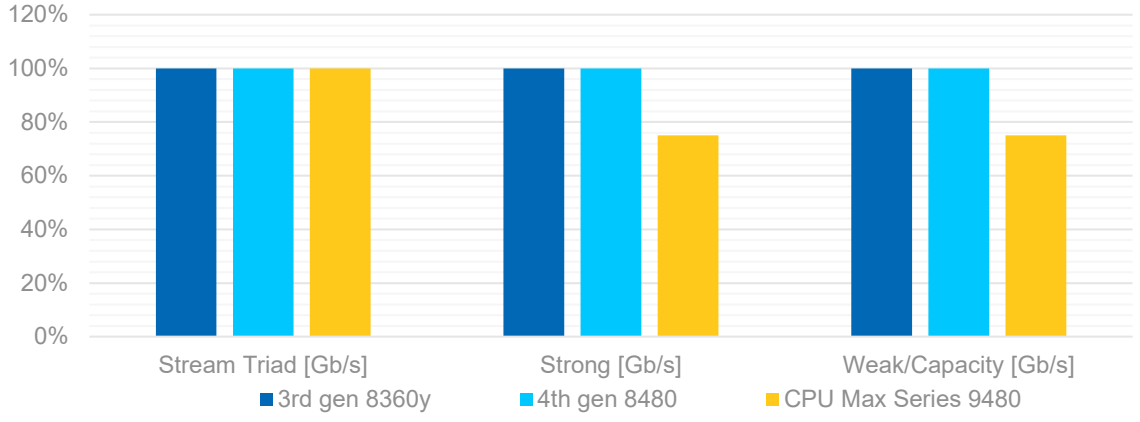
Normalized performance improvement of EVP, DDR5 vs. HBM
(higher is better)



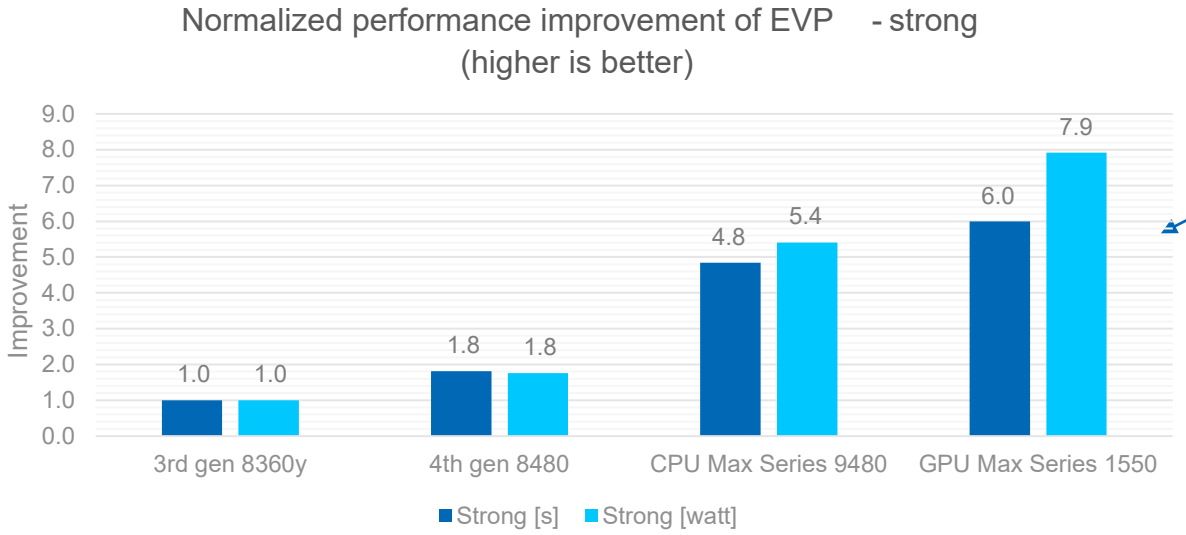
Normalized performance improvement of EVP vs. Stream Triad
(higher is better)



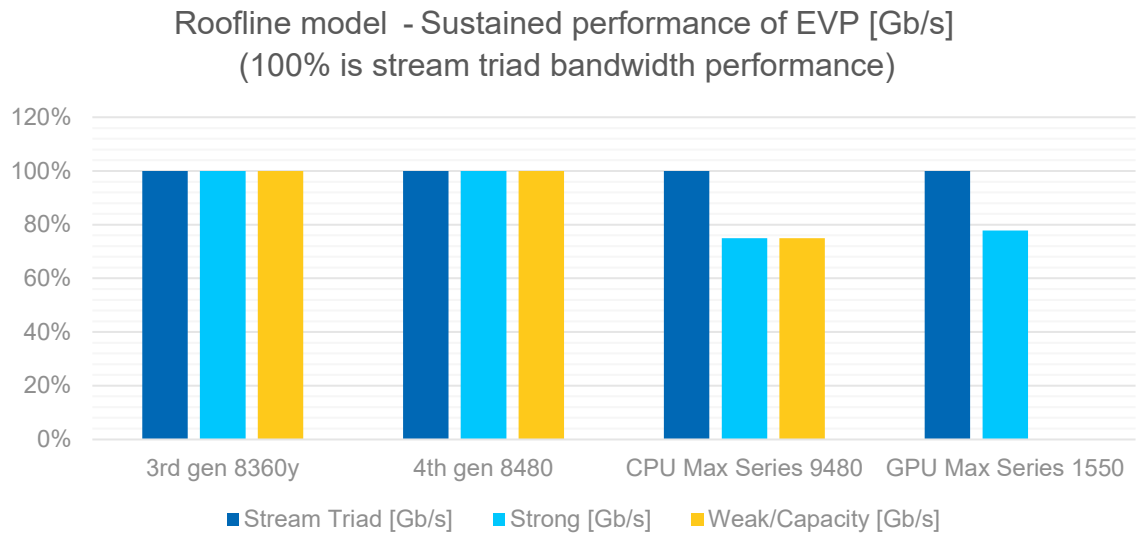
Roofline model - Sustained performance of EVP [Gb/s]
(100% is stream triad bandwidth performance)

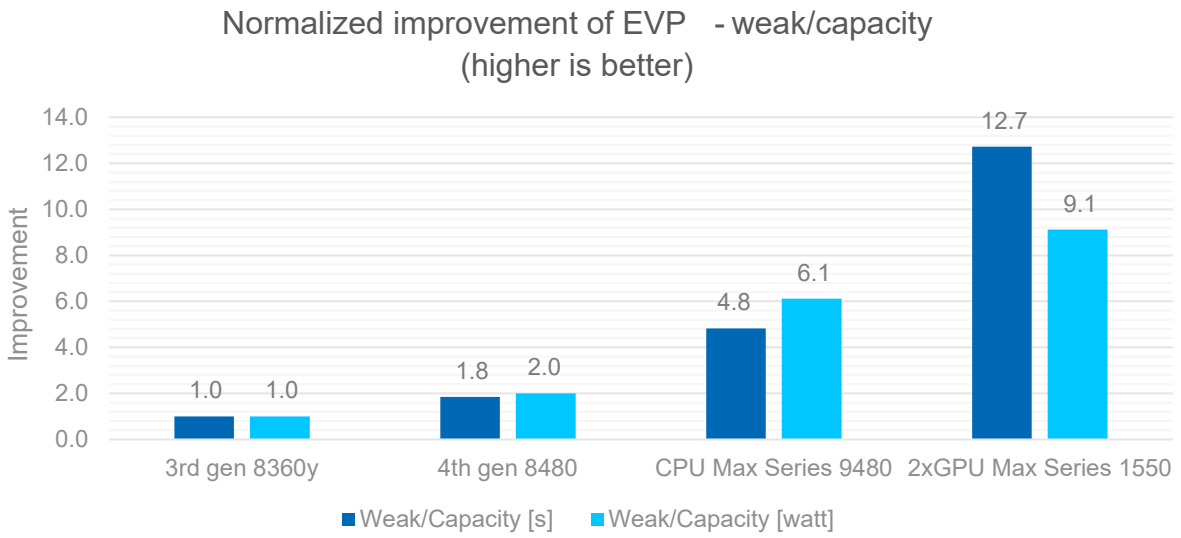


CPU vs. GPU performance



Uses implicit scaling across the 2 tiles





8 tasks on 2 devices (two MPI -tasks per tile).

Refactoring target- Cluster

Core level:
- Focus on loops and ensure that loops of interest support SIMD code generation

Node level:
- Review current parallelization pattern
- Focus on the parallelization unit used – can we make it more fine grained ?
- Focus on synchronization points – can we reduce them, can we make them more light weight ?
- Focus on temporary “stack” arrays (implementation overhead) used to convey results from one section to the next

Cluster level:
- See first three items for the Node level

Refactoring- cluster level

The core and node refactoring have given us decent node performance but is there also something to refactor at the cluster level? Two opportunities to consider:

1. MPI neighborhood collectives
2. An MPMD hook for the EVP allowing instant support of heterogeneous architectures

The EVP algorithm is a posterchild for the MPMD pattern in the sense that it both have **unmet performance** requirements in current parallelization and it **poses a severe bottleneck to the overall scaling** of the whole model system.

This is WIP (analysis) as the current refactoring is being integrated upstream. Once the node part has been fully integrated upstream then we plan to continue with the MPMD hook for EVP.

Conclusions

- OpenMP can be used to easily express the parallelism in your code and can easily target both CPUs and GPUs. Good performance is achievable with OpenMP on all architectures assuming the existence of parallelism and that we carefully adjust our runtime environment.

- Code refactoring can give huge performance gains and does not have to be extremely invasive.

- Roofline modelling based on stream triad seem to be too simplistic for some of the emerging memory technologies
  - How many flops can we stuff into a stream -triad (with AI=0.08) before BW utilization drops below say 1.7Tb/s
  - READ/WRITE ratio is in the 4:1 range for EVP.
  - READ/SIMD GATHER is in the 5:1 range for EVP
  - How many percent of the READs can be SIMD GATHER before BW utilization drops below say 1.7Tb/s
    - SIMD GATHER amounts to 12% vs. 19% of the READ operations in stage1 and stage2, respectively

- Architectural design using sub -units pose performance challenges to the SW developers that one need to take into account to achieve good performance on these architectures. Proper mapping (placement+binding) and overall proper runtime environment is vital to achieve good performance for strong scaling across multiple sub -units.

- The freely available oneAPI compiler from Intel allow you to build and run all the refactoring samples across multi architectures. Go play and please reach out if you have any questions.

Thanks to

- CICE community: Elizabeth C. Hunke, Till S. Rasmussen, Mads H. Ribergaard, Anthony P. Craig
- Intel: Ruchira Sasanka, Chris Dahnken, Mikko Byckling, Carsten Uphoff, Ben Hoertz , Tim Mefford, Mike Greenfield, Kristina Kermanshahche


intel