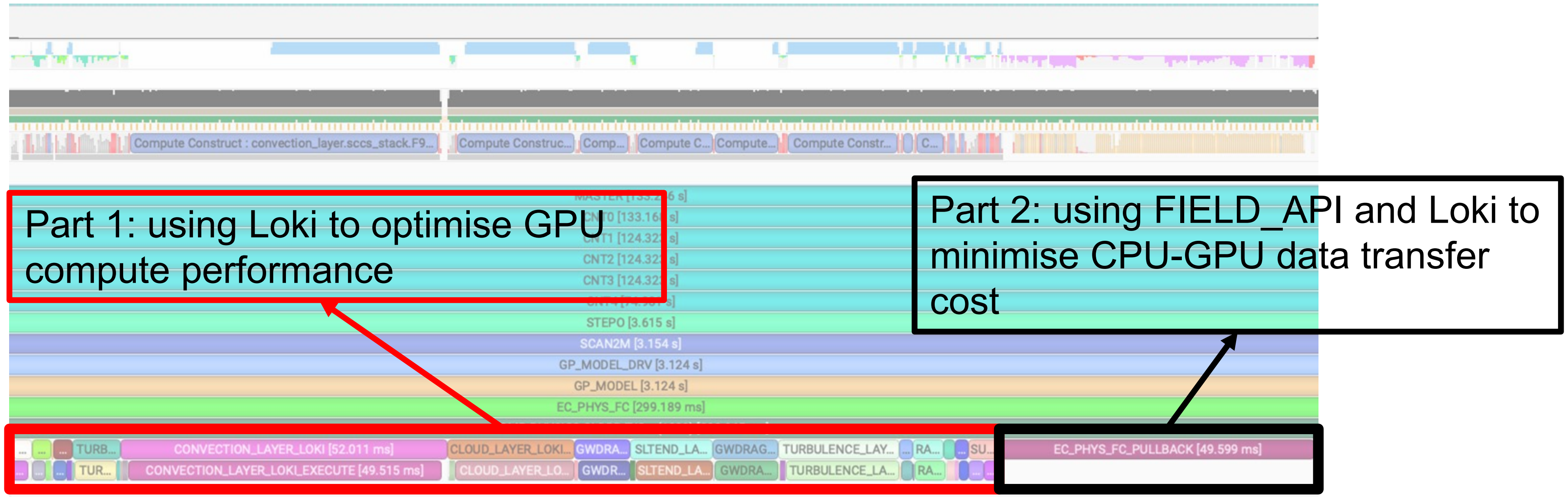
Cross-platform optimization for GPUs of various flavours

The low-level tech overview

A. Nawab, M. Staneker, J. Ericsson

Introduction

- We have seen the high-level strategy for the GPU porting of the IFS, and we have seen the performance it can deliver
- The current talk goes into deeper, code-level detail on how this portable performance is achieved



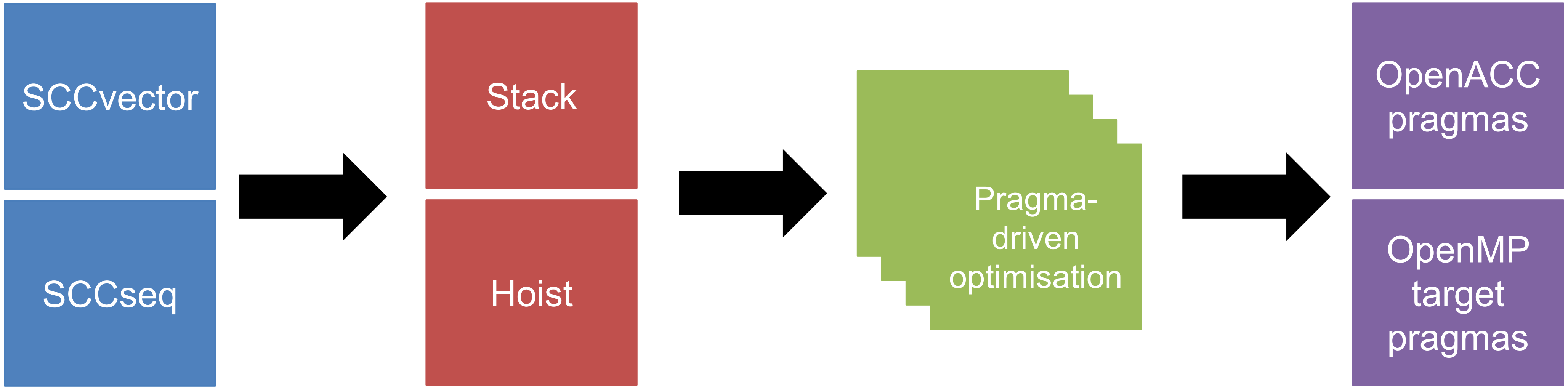
Part 1: using Loki to optimise GPU compute performance

Part 2: using FIELD_API and Loki to minimise CPU-GPU data transfer cost

- Conclude with a brief look at the holy grail of performance portability:
  - Fortan => CUDA/HIP transpilation

Transforming compute kernels with Loki

- Transforming compute kernels with Loki consists of 4 incremental steps:



1. Apply SCC - Single Column Coalesced

2. Pre-allocate on device temporaries

3. Apply kernel specific, developer guided optimisations

4. Annotate transformed code with target-specific pragmas

- The ecWAM source-term computation will be used as a case-study to illustrate the above pipeline

SCC - Single Column Coalesced

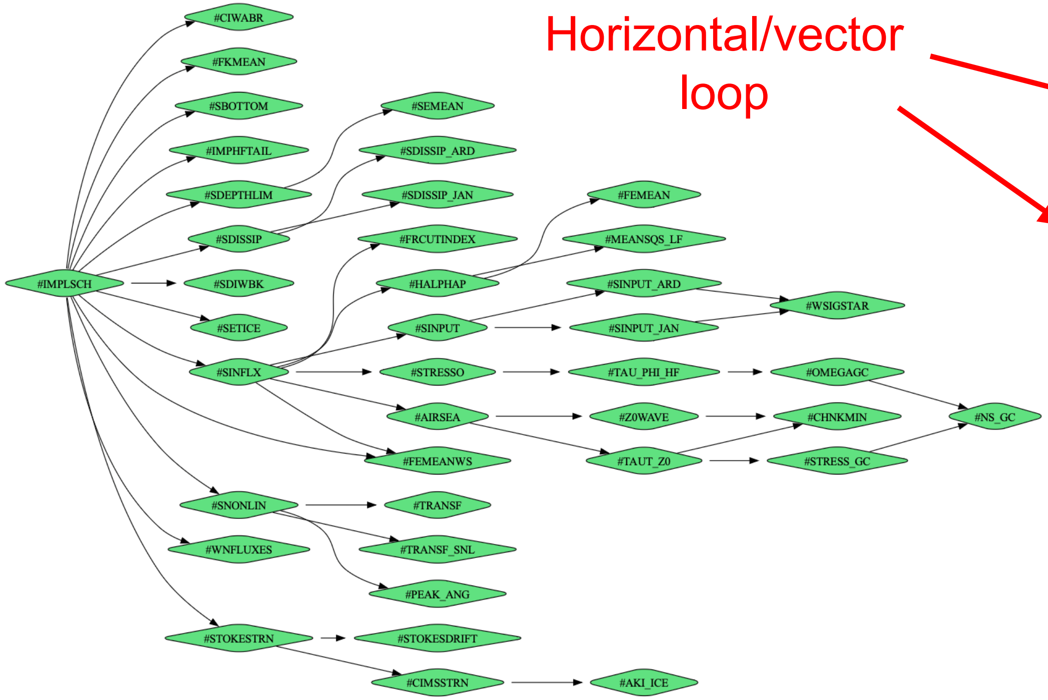
- SCC is the baseline Loki transformation for IFS single column code
- It reorders the control flow to target the SIMT execution model, as opposed to the SIMD execution model targeted in the baseline CPU code
- Essentially inverts loop order, so that ”horizontal” (i.e. NPROMA) loops are now the outermost
  - (Actually, it does a bit more than that but there’s only so much code I can discuss before putting everyone to sleep ;))

Baseline single column CPU control flow

Driver loop

```fortran
  !$OMP PARALLEL DO SCHEDULE(DYNAMIC,1) PRIVATE(ICHNK)
      DO ICHNK = 1, NCHNK

        CALL IMPLSCH (1, NPROMA_WAM, VARS_4D%FL1(:,:,:,ICHNK),    &
   &                     WVPRPT%WAVNUM(:,:,ICHNK), WVPRPT%CGROUP(:,:,ICHNK), WVPRPT%CIWA(:,:,ICHNK),
&  &                     WVPRPT%CINV(:,:,ICHNK), WVPRPT%XK2CG(:,:,ICHNK), WVPRPT%STOKFAC(:,:,ICHNK),
&  &                     WVENVI%EMAXDPT(:,ICHNK), &
```



Horizontal/vector loop

```fortran
DO IJ=KIJS,KIJL
  RAORW(IJ) = MAX(AIRD(IJ), 1.0_JWRB) * ROWATERM1
ENDDO

DO IJ=KIJS,KIJL·
  ALPFAC(IJ)  = ZALPFACX ! <1=some reduction, 1=no reduction to attenuation
ENDDO

DO K=1,NANG
  DO IJ=KIJS,KIJL
    COSWDIF(IJ,K) = COS(TH(K)-WDWAVE(IJ))
    SINWDIF2(IJ,K) = SIN(TH(K)-WDWAVE(IJ))**2
  ENDDO
ENDDO
```

SCCseq

- Supported by OpenACC + OpenMP target
- More complex, but (slightly) faster

```
DO ICHNK=1,NCHNK

  DO IJ=1,NPROMA_WAM
    CALL IMPLSCH_LOKI(1, NPROMA_WAM, VARS_4D%FL1(:, :, :, ICHNK), &
    & WVPRPT%WAVNUM(:, :, ICHNK), WVPRPT%CGROUP(:, :, ICHNK),  &
    & WVPRPT%CIWA(:, :, ICHNK), WVPRPT%CINV(:, :, ICHNK), &
    & WVPRPT%XK2CG(:, :, ICHNK), WVPRPT%STOKFAC(:, :, ICHNK),  &
    & WVENVI%EMAXDPT(:, ICHNK), WVENVI%DEPTH(:, ICHNK), &
    WVENVI%IOBND(:, ICHNK), WVENVI%IODP(:, ICHNK),  &
```

```
RAORW(IJ) = MAX(AIRD(IJ), 1.0_JWRB)*ROWATERM1

ALPFAC(IJ) = ZALPFACX      ! <1=some reduction, 1=no reduction to attenuation

DO K=1,NANG
  COSWDIF(IJ, K) = COS(TH(K) - WDWAVE(IJ))
  SINWDIF2(IJ, K) = SIN(TH(K) - WDWAVE(IJ))**2
END DO

IF (LWNEMOCOUWRS .and. .not.(LCIWA1 .or. LCIWA2 .or. LCIWA3)) THEN
  DO M=1,NFRE
    DO K=1,NANG
      SLICE(IJ, K, M) = 0.0_JWRB
    END DO
  END DO
END IF
```

No horizontal loop!

SCCvector

- Supported by OpenACC only
- Easier to encode, easier to debug
- Better support for reductions along horizontal dim (very rare in IFS grid-point code)

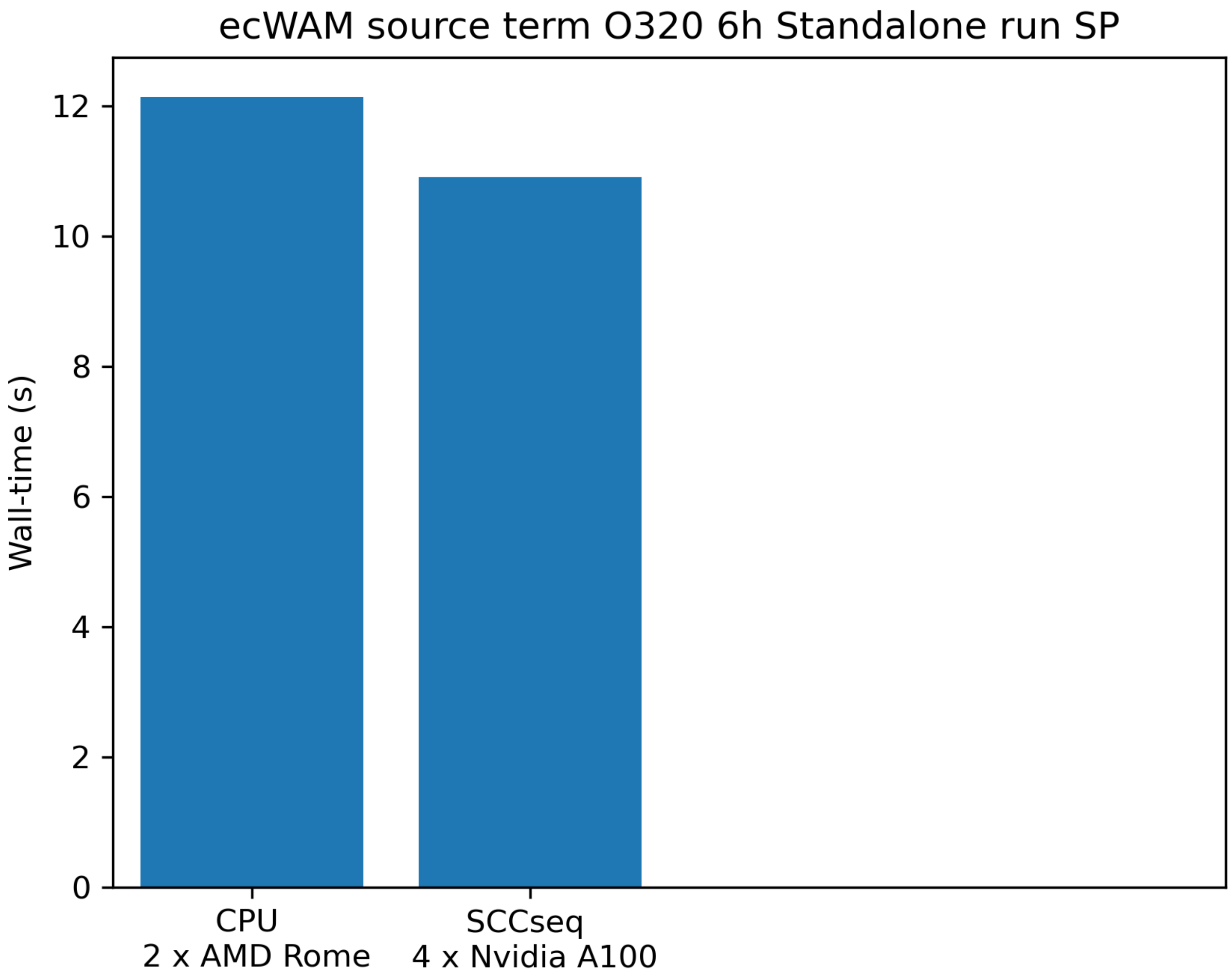
```
DO IJ=KIJS,KIJL
  RAORW(IJ) = MAX(AIRD(IJ), 1.0_JWRB)*ROWATERM1

  ALPFAC(IJ) = ZALPFACX      ! <1=some reduction, 1=no reduction to attenuation

  DO K=1,NANG
    COSWDIF(IJ, K) = COS(TH(K) - WDWAVE(IJ))
    SINWDIF2(IJ, K) = SIN(TH(K) - WDWAVE(IJ))**2
  END DO

  IF (LWNEMOCOUWRS .and. .not.(LCIWA1 .or. LCIWA2 .or. LCIWA3)) THEN
    DO M=1,NFRE
      DO K=1,NANG
        SLICE(IJ, K, M) = 0.0_JWRB
      END DO
    END DO
  END IF
END DO
```

Baseline SCC



Temporary arrays

- Allocations of temporary arrays in GPU kernels have a very(!) detrimental performance impact
- Solution: pre-allocate all temporaries before GPU kernel launch
- Comes in two flavours, ~equivalent performance:
  - Stack:
    - Allocate memory pool on device and convert temporary arrays to pointers extracted from this pool
    - Smaller device memory footprint than hoist, but needs architecture specific implementations
  - Hoist:
    - Hoist temporary arrays all the way up to the driver layer (i.e. from where the GPU kernel is launched)
    - Higher device memory footprint but universally applicable

Cray pointer stack - NVHPC

- Fortran pointer reshaping unsupported on GPU

```
ALLOCATE (ZSTACK(<size-expr>, NCHNK))
!$acc data create( ZSTACK )
...
!$acc parallel loop gang private( YLSTACK_L ) vector_length( NPROMA_WAM )
      DO ICHNK=1,NCHNK
        YLSTACK_L = LOC(ZSTACK(1, ICHNK))

!$acc loop vector
        DO IJ=1,NPROMA_WAM
          CALL IMPLSCH_LOKI(1, NPROMA_WAM, VARS_4D%FL1(:, :, :, ICHNK), &
          ...
          & VARS_4D%XLLWS(:, :, :, ICHNK), IJ=IJ, YDSTACK_L=YLSTACK_L)
        END DO
```

```
!$acc routine seq
INTEGER(KIND=8) :: YLSTACK_L
INTEGER(KIND=8), INTENT(INOUT) :: YDSTACK_L
POINTER(IP_RAORW, RAORW)
...
YLSTACK_L = YLSTACK_L + <size-expr>
IP_RAORW = YLSTACK_L
...
RAORW(IJ) = MAX(AIRD(IJ), 1.0_JWRB)*ROWATERM1
```

Stack array association

Stack array

Fortran pointer stack - ROCm AFAR

- Cray pointers unsupported on GPU

```
ALLOCATE (Z_JWRB_STACK(<size-expr>, NCHNK))
!$omp target enter data map( alloc: Z_JWRB_STACK )
...
!$omp target teams distribute thread_limit( NPROMA_WAM )
      DO ICHNK=1,NCHNK

!$omp parallel do
        DO IJ=1,NPROMA_WAM
          CALL IMPLSCH_LOKI(1, NPROMA_WAM, VARS_4D%FL1(:, :, :, ICHNK), &
          ...
          & VARS_4D%XLLWS(:, :, :, ICHNK), IJ, Z_JWRB_STACK(:, ICHNK))
        END DO
```

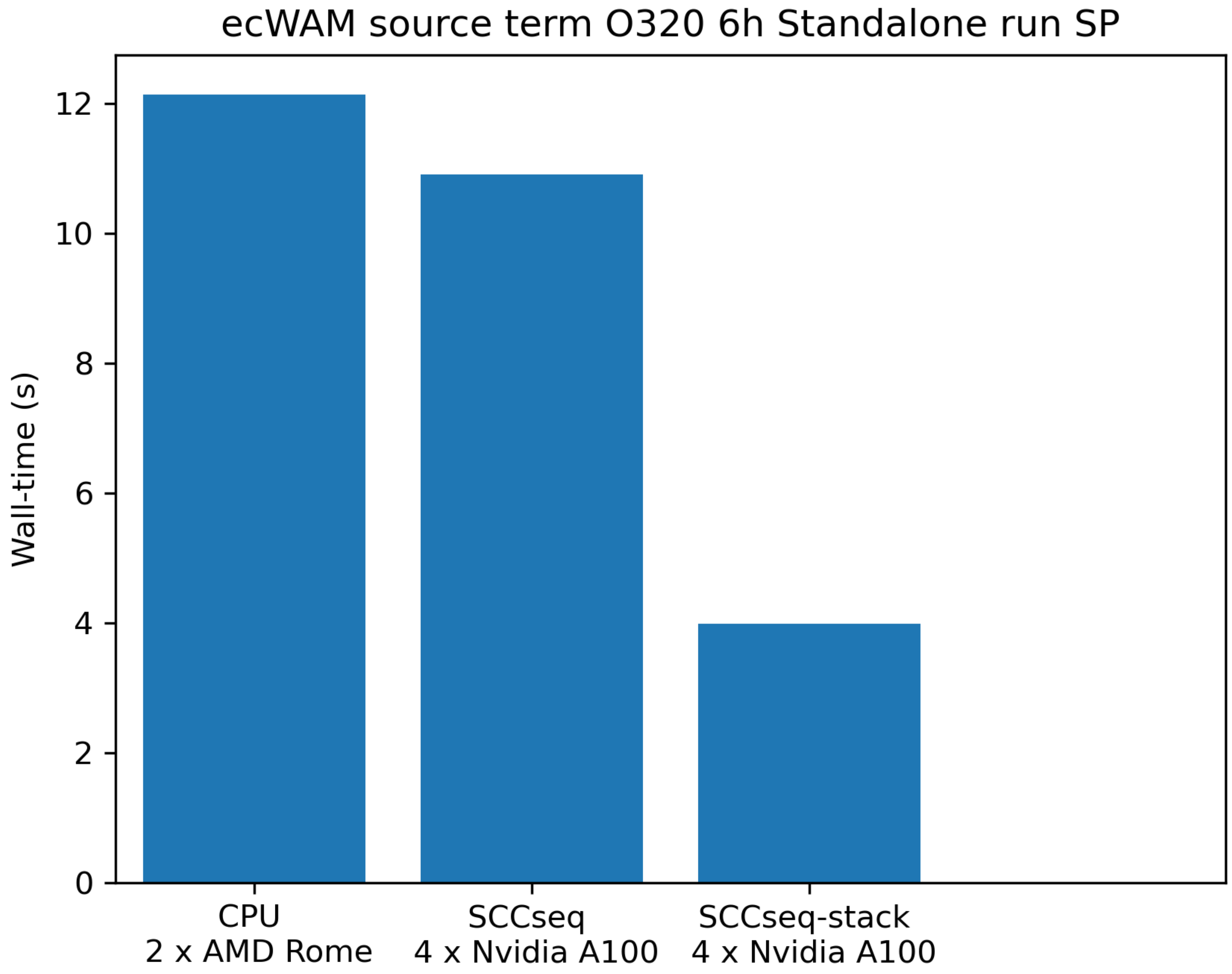
```
!$omp declare target
REAL(KIND=JWRB), TARGET, CONTIGUOUS, INTENT(INOUT) :: P_JWRB_STACK(<size-expr>)
RAORW(1:KIJL) => P_JWRB_STACK(<size-expr>)
...

RAORW(IJ) = MAX(AIRD(IJ), 1.0_JWRB)*ROWATERM1
```

Stack array association

Stack array

SCCseq-stack



Pragma-driven optimisations

- The Loki transformation steps seen until now are all general, and are applied to every Loki transformed IFS kernel
- The final step requires intervention from the developer, in the form of pragma hints to Loki to apply specific optimisations
- Examples include (but not limited to):
  - !$loki remove…!$loki end remove - mark region of code to be removed
  - !$loki inline - mark subroutine calls to be inlined
  - Loop transformations:
    - Loop interchange
    - Loop fusion/fission
    - Loop unroll

Optimisations typically performed by compilers

- Especially useful when applying such transformations by hand is not possible because:
  - It could hurt CPU performance
  - Might leave the code in a “messier” state, possibly inhibiting scientific development

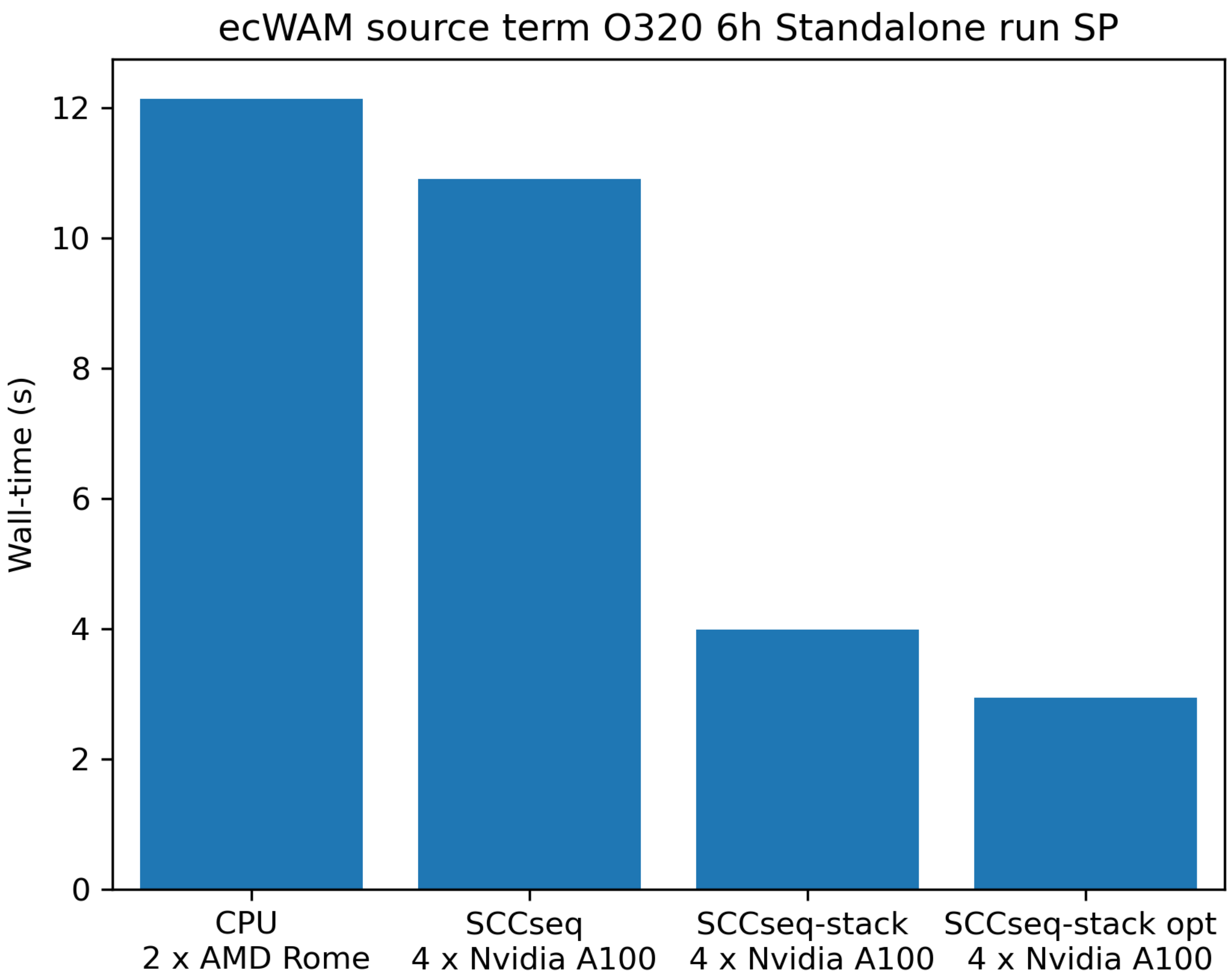
Split loads/stores for indirect array addresses

```
!$loki split-read-write
DO IJ=KIJS,KIJL
  SL(IJ,K2 ,MM ) = SL(IJ,K2 ,MM ) + AD(IJ)*FKLAMM1
ENDDO
DO IJ=KIJS,KIJL
  SL(IJ,K21,MM ) = SL(IJ,K21,MM ) + AD(IJ)*FKLAMM2
ENDDO
DO IJ=KIJS,KIJL
  SL(IJ,K2 ,MM1) = SL(IJ,K2 ,MM1) + AD(IJ)*FKLAMMA
ENDDO
DO IJ=KIJS,KIJL
  SL(IJ,K21,MM1) = SL(IJ,K21,MM1) + AD(IJ)*FKLAMMB
ENDDO
!$loki end split-read-write
```

```
!...loads
loki_temp_2 = SL(IJ, K2, MM) + AD*FKLAMM1
loki_temp_4 = SL(IJ, K21, MM) + AD*FKLAMM2
loki_temp_6 = SL(IJ, K2, MM1) + AD*FKLAMMA
loki_temp_8 = SL(IJ, K21, MM1) + AD*FKLAMMB

!...stores
SL(IJ, K2, MM) = loki_temp_2
SL(IJ, K21, MM) = loki_temp_4
SL(IJ, K2, MM1) = loki_temp_6
SL(IJ, K21, MM1) = loki_temp_8
```

SCCseq-stack optimised



FIELD_API

- A GPU-aware data-structure library co-developed with Météo-France for IFS/Arpege fields
- Once data-structures have been rewritten around FIELD_API, data transfers between CPU and GPU can be performed via intuitive and non-intrusive API calls:

```
CALL FIELD%GET_HOST/DEVICE_DATA_RDWR(PTR)
```

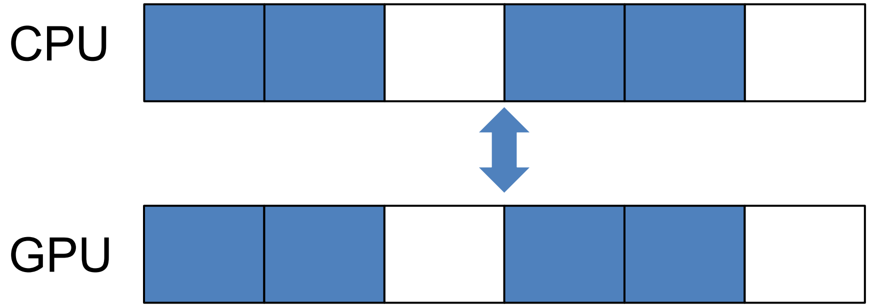
- FIELD_API now offers multiple offload backends:
  - NVHPC OpenACC +/- CUDA
  - NVHPC OpenMP target +/- CUDA
  - ROCm AFAR OpenMP target +/- HIPFORT

- **Data offload instructions in scientific code remain unchanged regardless of GPU architecture**

FIELD_API - offload optimisations

- FIELD_API offers three main features for reducing the data offload cost:

1. Fast strided data transfers via cuda/hipmemcpy2d:



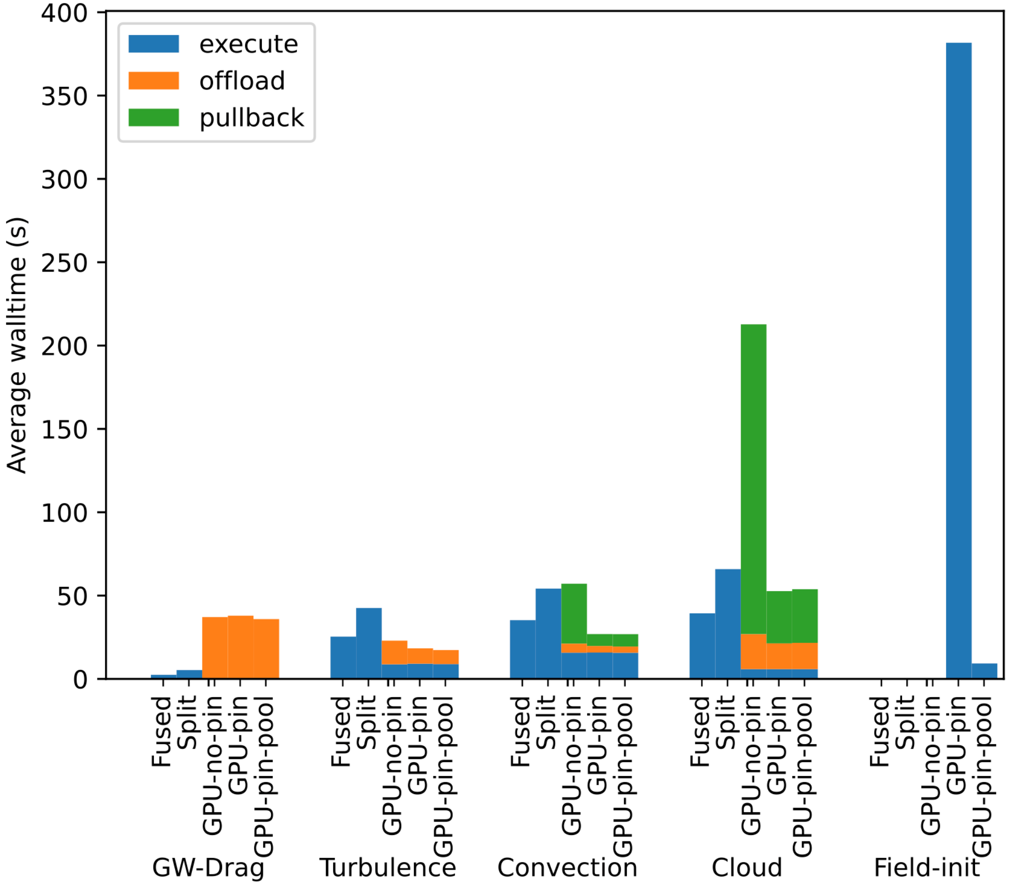
2. Asynchronous data transfers:

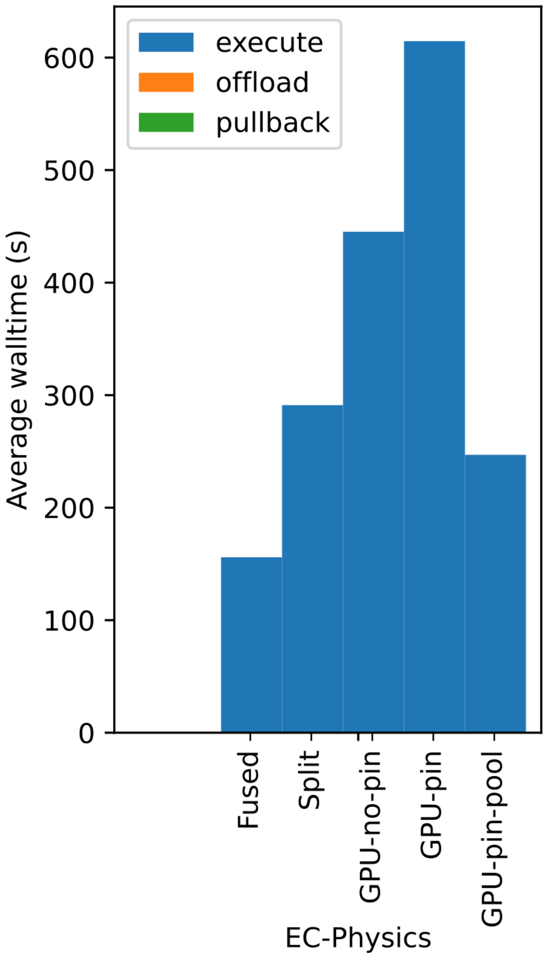
```
CALL FIELD%GET_DEVICE_DATA_RDWR(PTR,QUEUE=1)
...
CALL WAIT_FOR_ASYNC_QUEUE(QUEUE=1)
```

3. Pinning (i.e. page-locking) host allocations vida cuda/hiphostregister
   – Especially effective when combined with host memory pool

Pinning and pooling host allocations

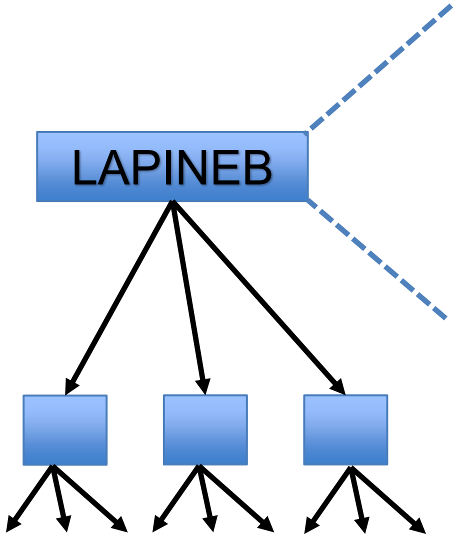
- Pinning host allocations can incur very large overheads
  - Can even lead to overall slowdowns, despite data transfer performance improvements
- FIELD_API has an optional memory pool for host allocations to mitigate pinning overheads





Minimising size of data offload

- Best case scenario is no data-transfer at all!
- GPU kernels that encompass large subroutine call-trees might access hundreds of fields stored in deeply nested data-structures
  - Very(!) difficult to manually compute minimal data offload
- Loki inter-procedural dataflow analysis + GPU deepcopy generation can be used



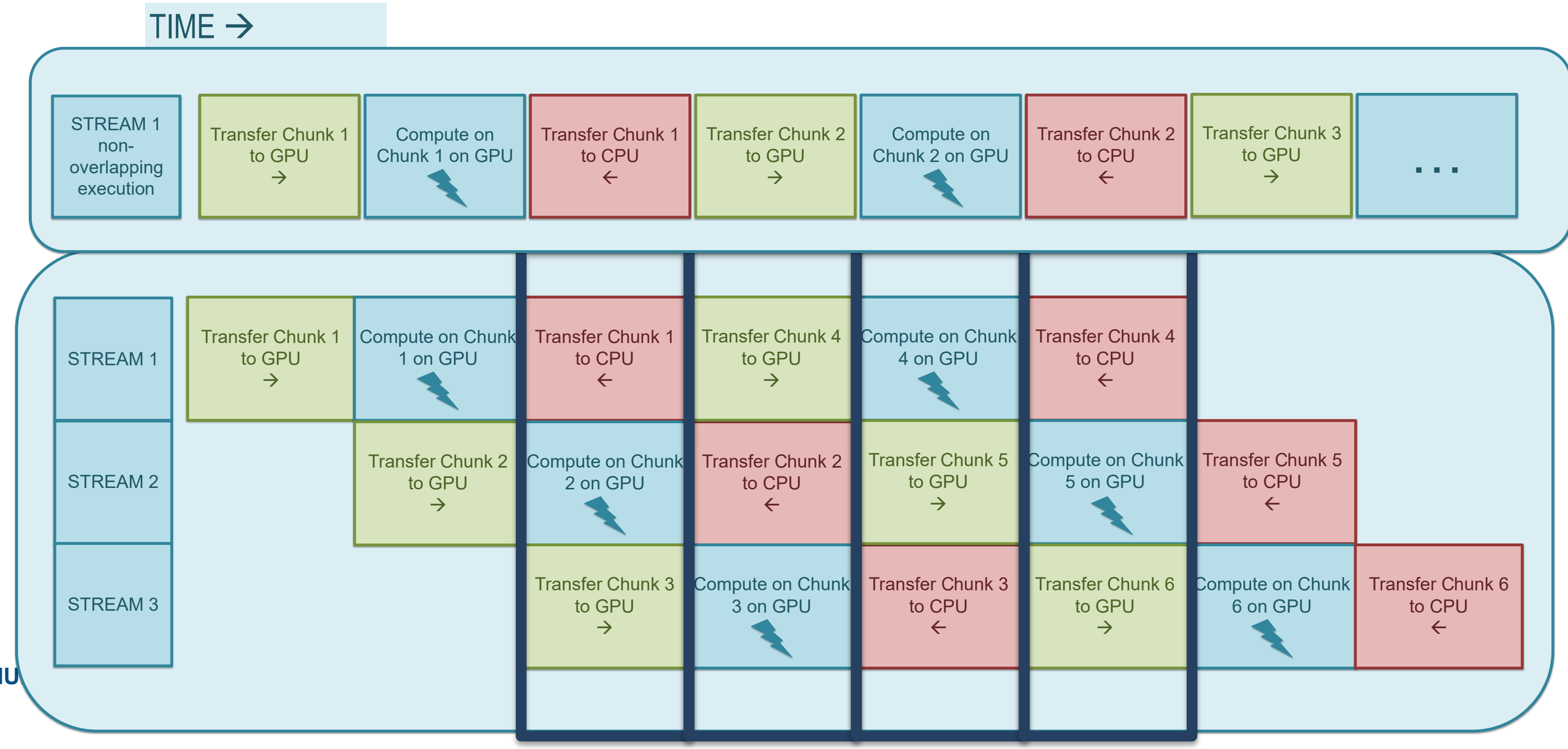
Loki generated

```
...
!$acc enter data copyin( YDVARS )
    IF (ALLOCATED(YDVARS%GFL_PTR)) THEN
!$acc enter data copyin( YDVARS%GFL_PTR )
      DO J1=LBOUND(YDVARS%GFL_PTR, 1),UBOUND(YDVARS%GFL_PTR, 1)
        IF (ASSOCIATED(YDVARS%GFL_PTR(J1)%FT1)) THEN
          CALL YDVARS%GFL_PTR(J1)%FT1%GET_DEVICE_DATA_RDWR(YDVARS%GFL_PTR(J1)%T1_FIELD)
!$acc enter data attach( YDVARS%GFL_PTR(J1)%T1_FIELD )
        END IF
      END DO
    END IF
...
```

```
ydvars:
  geometry:
    gnordl:
      p_field: read
    gnordm:
      p_field: read
  gfl_ptr:
    t1_field: readwrite
    ycomp:
      cname: read
      cslint: read
      ladv: read
      lgp: read
      lhorturb: read
      lmgrid: read
      lphy: read
      weno_alpha: read
  pcf_u:
    pc_ph_field: readwrite
  pcf_v:
    pc_ph_field: readwrite
  sp:
    t1_field: readwrite
  svd:
    t1_field: readwrite
  t:
    t1_field: readwrite
  u:
    t1_field: readwrite
  v:
    t1_field: readwrite
ylcpg_bnds:
  kbl: read
  kfdia: read
  kidia: read
```

Overlapping computation and data-offload

- Our most sophisticated data offload optimisation is to overlap computation and data offload
  - Leverages advanced Loki and FIELD_API functionality together
- Data offload and computation can be split into “chunks”, and distributed across multiple “streams”
- Stream execution can be overlapped to partially hide data-transfer cost
  - Overlap computation and communication
  - Overlap host-to-device and device-to-host data-transfers



Overlapping computation and data-offload

- Necessary boilerplate generated automatically via Loki starting from baseline CPU code

Baseline

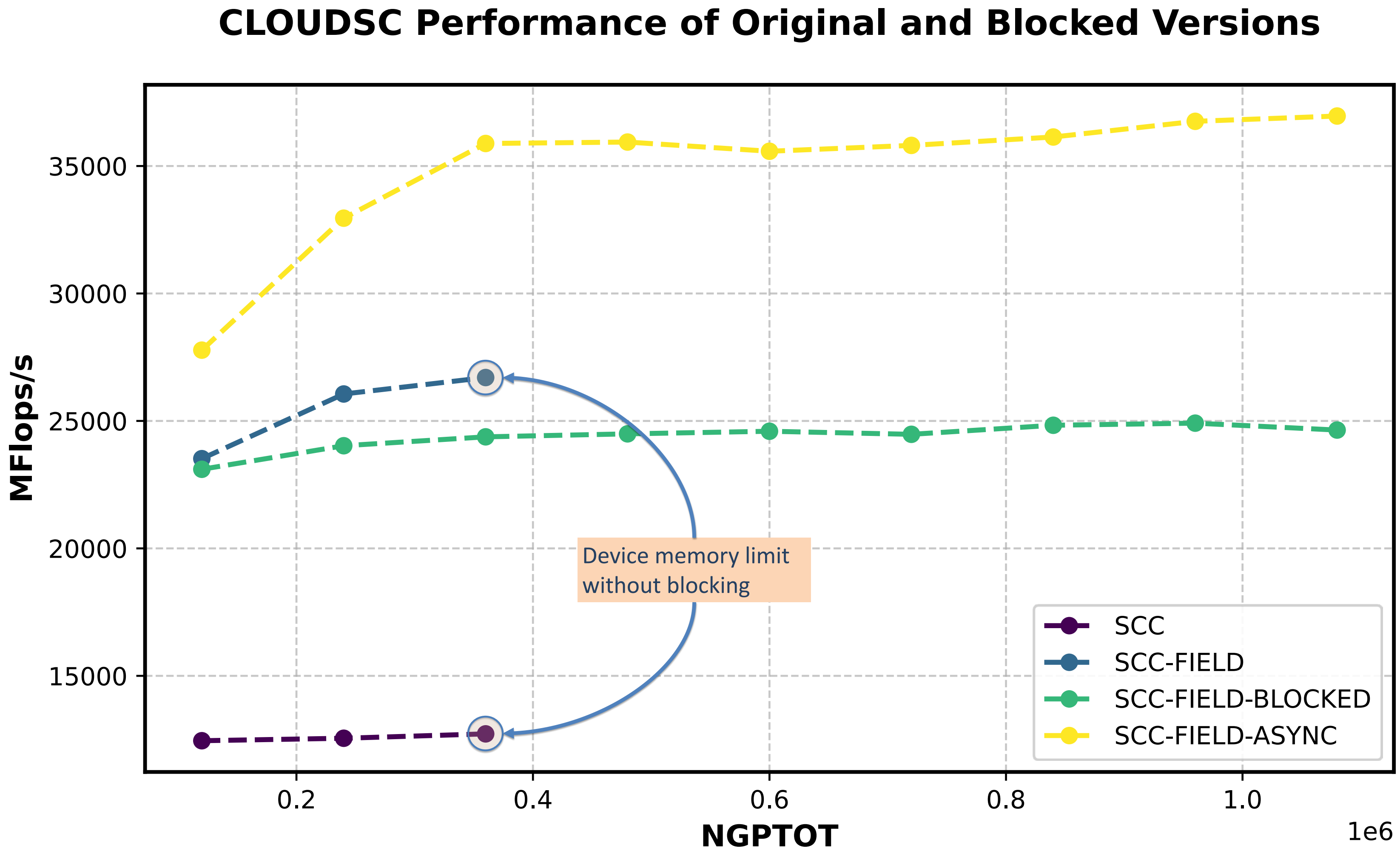
```
!$loki data
!$loki driver-loop
DO I=1,NLEV
  CALL STATE%UPDATE_VIEW(I)
  CALL KERNEL_ROUTINE(NLON, NLEV, STATE%A)
END DO
!$loki end data
```

Loki generated

```
DO I_LOOP_BLOCK_IDX=1,I_LOOP_NUM_BLOCKS
  LOKI_BLOCK_QUEUE = MODULO(I_LOOP_BLOCK_IDX, LOKI_BLOCK_NQUEUES) + 1
  LOKI_BLOCK_OFFSET = (LOKI_BLOCK_QUEUE - 1)*I_LOOP_BLOCK_SIZE
  I_LOOP_BLOCK_START = (I_LOOP_BLOCK_IDX - 1)*I_LOOP_BLOCK_SIZE + 1
  I_LOOP_BLOCK_END = MIN(I_LOOP_BLOCK_IDX*I_LOOP_BLOCK_SIZE, NLEV)
  CALL STATE%F_A%GET_DEVICE_DATA_FORCE(LOKI_DEVPTR_PREFIX_STATE_A, QUEUE=LOKI_BLOCK_QUEUE,  &
  & BLK_BOUNDS=(/ I_LOOP_BLOCK_START, I_LOOP_BLOCK_END /), OFFSET=LOKI_BLOCK_OFFSET)
  !$acc parallel loop present(LOKI_DEVPTR_PREFIX_STATE_A) async(LOKI_BLOCK_QUEUE)
  DO I_LOOP_LOCAL=1,I_LOOP_BLOCK_END - I_LOOP_BLOCK_START + 1
    I_LOOP_ITER_NUM = I_LOOP_BLOCK_START + I_LOOP_LOCAL - 1
    I = I_LOOP_ITER_NUM
    CALL KERNEL_ROUTINE(NLON, NLEV, LOKI_DEVPTR_PREFIX_STATE_A(:, :, I))
  END DO
  !$acc end parallel loop
  CALL STATE%F_A%SYNC_HOST_FORCE(QUEUE=LOKI_BLOCK_QUEUE,BLK_BOUNDS=(/I_LOOP_BLOCK_START,I_LOOP_BLOCK_END/),&
  & OFFSET=LOKI_BLOCK_OFFSET)
END DO
```

Overlapping computation and data-offload

- ~40% reduction in CLOUDSC dwarf walltime using 3 streams
- Especially impressive considering limited stream overlap potential in CLOUDSC dwarf
  - Data offload time is ~10x larger than computation
  - Device-to-host copy 2x larger than host-to-device copy



ECMWF

Loki moonshot: Fortran => CUDA/HIP/SYCL transpilation

- The best GPU performance can only be achieved by using a low-level kernel language e.g. CUDA
- M. Staneker has developed Loki functionality to transpile single column Fortran to CUDA/HIP/SYCL
- Very(!!) complex, not yet ready for production
  - Currently at proof-of-concept stage
  - Works fully automated for CLOUDSC dwarf, partially automated for ecWAM source-term

global var and derived type handling	InlineTrafo	GlobalVarHoistTransformation	GlobalVariableAnalysis	DerivedTypeArgumentsTransformation
single column	SCCBaseTransformation	SCCDevectorTransformation	SCCDemoteTransformation	SCCRevectorTransformation
blocking to kernel(s)	BlockIndexLowerTransformation	BlockIndexInjectTransformation	BlockLoopLowerTransformation	
device vars, launch configuration, …	SCCLowLevelGPU			
handling of temporaries	HoistTemporaryArraysAnalysis	HoistTemporaryArraysCstyleTransformation		
de/reference, wrapper, bindings, …	FortranCTransformation			
housekeeping	DependencyTransformation			

heavily relying on Loki infrastructure/utilities

CLOUDSC - Fortran OpenACC vs CUDA