# Optimizing Data Offload in the IFS Using GPU-Aware Data Structures and Source-To-Source Translation

Johan Ericsson* (ECMWF), Ahmad Nawab (ECMWF), Balthasar Reuter (ECMWF), Philippe Marguinaud (Météo-France), Judicaël Grasset (Météo-France), and Michael Lange (ECMWF)

*contact: johan.ericsson@ecmwf.int

## Introduction

The adaptation of ECMWF's medium-range forecasting model, the Integrated Forecasting System (IFS), to heterogeneous computing architectures is an ongoing effort. The IFS consists of millions of lines of Fortran code that is highly optimized for modern CPUs. This poses significant challenges when porting the code to heterogeneous architectures, as data layouts and compute patterns need to be changed to efficiently utilise the hardware. In this poster we show how FIELD API [1], a GPU-aware data-structure library, co-developed between ECMWF and Météo-France, and Loki [2], a freely programmable source-to-source translation toolchain written in Python, can be used to generate efficient blocked data offload to GPUs.

## FIELD API

FIELD API is a Fortran library for handling data in heterogeneous environments, that has been specifically developed for the IFS. The main design idea behind FIELD API is to hide many of the complexities of CPU-GPU data transfers behind a *field* abstraction. At its core, a field is just an n-dimensional array, with type-bound methods to execute CPU-GPU data transfers. A field can either allocate its own data or wrap an existing data allocation. Data transfers are performed via an intuitive API, e.g. **`FIELD%SYNC_DEVICE_RDONLY()`**. FIELD API is written in object-oriented Fortran and uses fypp [3] for templating fields over different data types and to support multiple GPU programming models, e.g. OpenACC, OpenMP, and CUDA. This makes it possible to use the same code for data transfers between host and device when switching between different programming models.

### FIELD API Features

- A memory-pool based allocator for host memory that supports page-locked memory allocations.
- Support for partial offload of fields into smaller device memory buffers.
- Support for asynchronous data transfers and for splitting a field's offload over multiple streams to enable overlap of communication and computation

## Loki

Loki is a freely programmable source-to-source translation toolchain for Fortran developed at ECMWF. Loki is written in Python and parses Fortran into an abstract syntax tree (AST) that is enriched with metadata to form a symbolic intermediate representation (IR) of the source code. Loki comes with a set of predefined visitors that can be used to analyse and transform the IR. It is easy to extend and add new transformations that modify the IR and generate the corresponding Fortran source code.

## Blocking of Driver Loops

The CLOUDSC cloud microphysics mini-app is an IFS single-column physics performance benchmark that represents a computational proxy of various IFS physical parameterisation schemes. It comprises of an outer *driver loop* over the total number of **`NPROMA`** blocks (IFS fields are memory blocked to improve memory locality), and inner vector loops inside the actual compute kernel. This is a common pattern in the IFS and an example of a driver loop is shown in Listing 1 below.

```
    !$omp do schedule(runtime) firstprivate(PAUX, FLUX, TENDENCY_TMP, TENDENCY_LOC) &
    !$omp& default(shared) private(JKGLO,IBL,ICEND,TID)
    DO JKGLO=1,NGPTOT,NPROMA
        IBL=(JKGLO-1)/NPROMA+1

        CALL PAUX%UPDATE_VIEW(IBL)
        .
        .
        .
        CALL TENDENCY_TMP%UPDATE_VIEW(IBL)

        CALL CLOUDSC_SCC (1, . . ., PAUX%PT, PAUX%PQ, . . . )
      ENDDO
      !$omp end do nowait
```

**Listing. 1.** Example showing what the driver loop in one of the CPU variants of CLOUDSC looks like.

### Challenges When Adapting The Driver Loop Pattern to GPUs

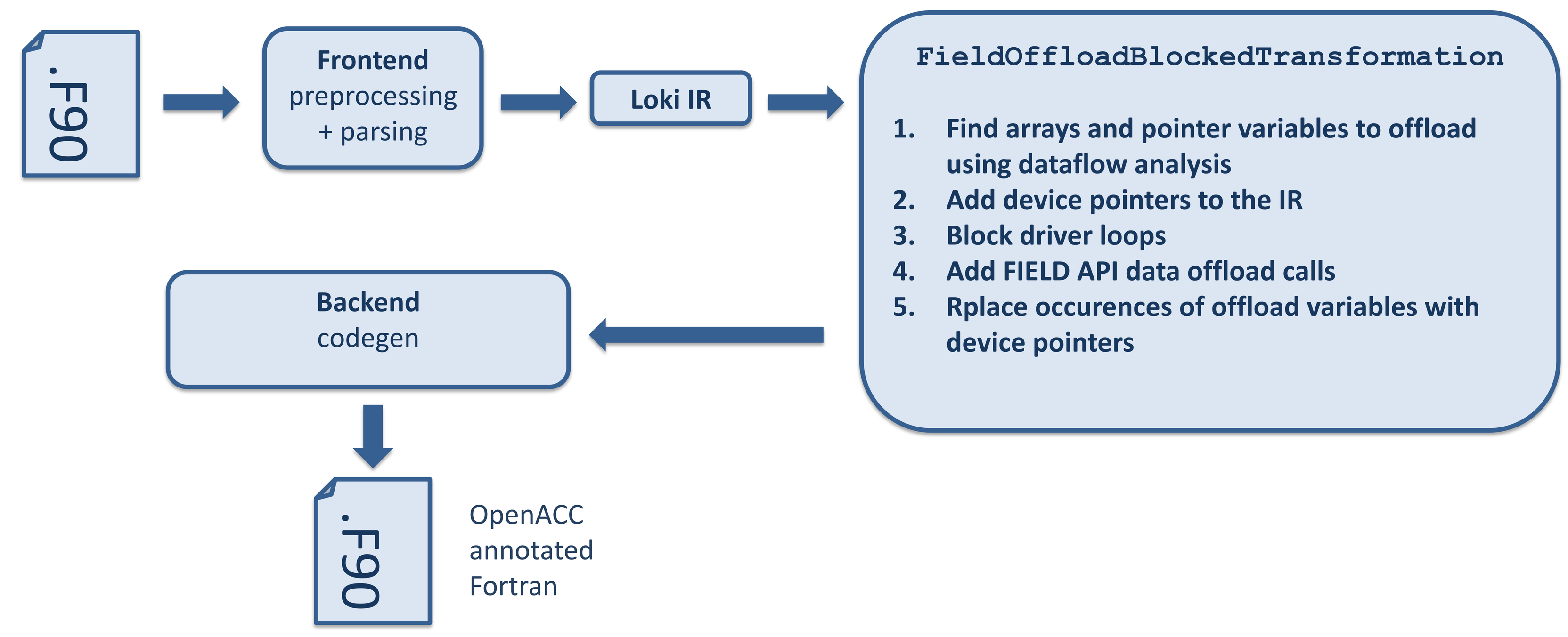
- For large problem sizes the fields may not fit on device.
- Performing data offload and pullback in each iteration is bad for performance.
- Data transfer times are often the main bottleneck.

### Solution - Loop Blocking

- Introduce an outer block loop that offloads fields in smaller chunks.
- Reduces the memory size required on the device.
- Chunks can be processed asynchronously if no loop-carried dependencies over the horizontal (**`NPROMA`**) dimension exists.

We contribute a Loki transformation, **`FieldOffloadBlockedTransformation`**, that can inject FIELD API data offload calls, and block data transfers, to add highly optimised asynchronous data offloads.

### Loki Transformation




```
    ! Allocate NQUEUES*BLOCK_SIZE on device for each field on the device
    CALL AUX%F_PT%CREATE_DEVICE_DATA(BLK_BOUNDS=[1,NQUEUES*BLOCK_SIZE])
    .
    .
    .
    CALL TENDENCY_LOC%F_CLD%CREATE_DEVICE_DATA(BLK_BOUNDS=[1,NQUEUES*BLOCK_SIZE])

    DO BLOCK_IDX=0, BLOCK_COUNT-1
      BLOCK_START=BLOCK_IDX*BLOCK_SIZE+1
      BLOCK_END=MIN((BLOCK_IDX+1)*BLOCK_SIZE, NGPBLKS)
      BLK_BOUNDS=[BLOCK_START, BLOCK_END]
      QUEUE = MODULO(BLOCK_IDX,NQUEUES)+1
      OFFSET = (QUEUE-1)*BLOCK_SIZE
      CHUNK_SIZE = BLK_BOUNDS(2) - BLK_BOUNDS(1) + 1

      ! partial async offload of fields to device
      CALL AUX%F_PT%GET_DEVICE_DATA_FORCE(PT, BLK_BOUNDS=BLK_BOUNDS, QUEUE=QUEUE, OFFSET=OFFSET)
      .
      .
      .
      CALL TENDENCY_LOC%F_CLD%GET_DEVICE_DATA_FORCE(TEND_LOC_CLD, BLK_BOUNDS=BLK_BOUNDS, QUEUE=QUEUE, OFFSET=OFFSET)

!$acc parallel loop gang vector_length(NPROMA) present(PT, . . ., ) async(QUEUE)
      DO IBLLOC=1, CHUNK_SIZE
        IBL= BLOCK_SIZE*BLOCK_IDX+IBLLOC
        JKGLO=(IBL-1)*NPROMA+1
        CALL CLOUDSC_SCC (1, . . ., PT(:,:,IBL), PQ(:,:,IBL), . . . )
      ENDDO
!$acc end parallel loop


      CALL AUX%F_PLUDE%SYNC_HOST_FORCE(BLK_BOUNDS=BLK_BOUNDS, QUEUE=QUEUE, OFFSET=OFFSET)
      .
      .
      .
      CALL TENDENCY_LOC%F_CLD%SYNC_HOST_FORCE(BLK_BOUNDS=BLK_BOUNDS, QUEUE=QUEUE, OFFSET=OFFSET)

    END DO ! End of block loop

! Wait for async work in all queues to finish
DO QUEUE=1,NQUEUES
  CALL WAIT_FOR_ASYNC_QUEUE(QUEUE)
END DO
```

**Listing. 2.** An example of a blocking with FIELD API based on the implementation of an asynchronous blocked driver loop in CLOUDSC.

## Performance Results

Performance of four different versions of CLOUDSC measured on a NVIDIA A100 40GB node on ECMWF's ATOS supercomputer:

1. **`SCC`** Uses simple OpenACC **`copy/copyin/copyout`** pragmas for data offload
2. **`SCC-FIELD`** Uses FIELD API for data offload
3. **`SCC-FIELD-BLOCKED`** Uses FIELD API with a blocked driver loop for offload
4. **`SCC-FIELD_BLOCKED-ASYNC`** Uses FIELD API with a blocked driver loop and splits the data offload and kernel computation into three asynchronous CUDA streams to overlap communication and computation.

**CLOUDSC Performance of Original and Blocked Versions**

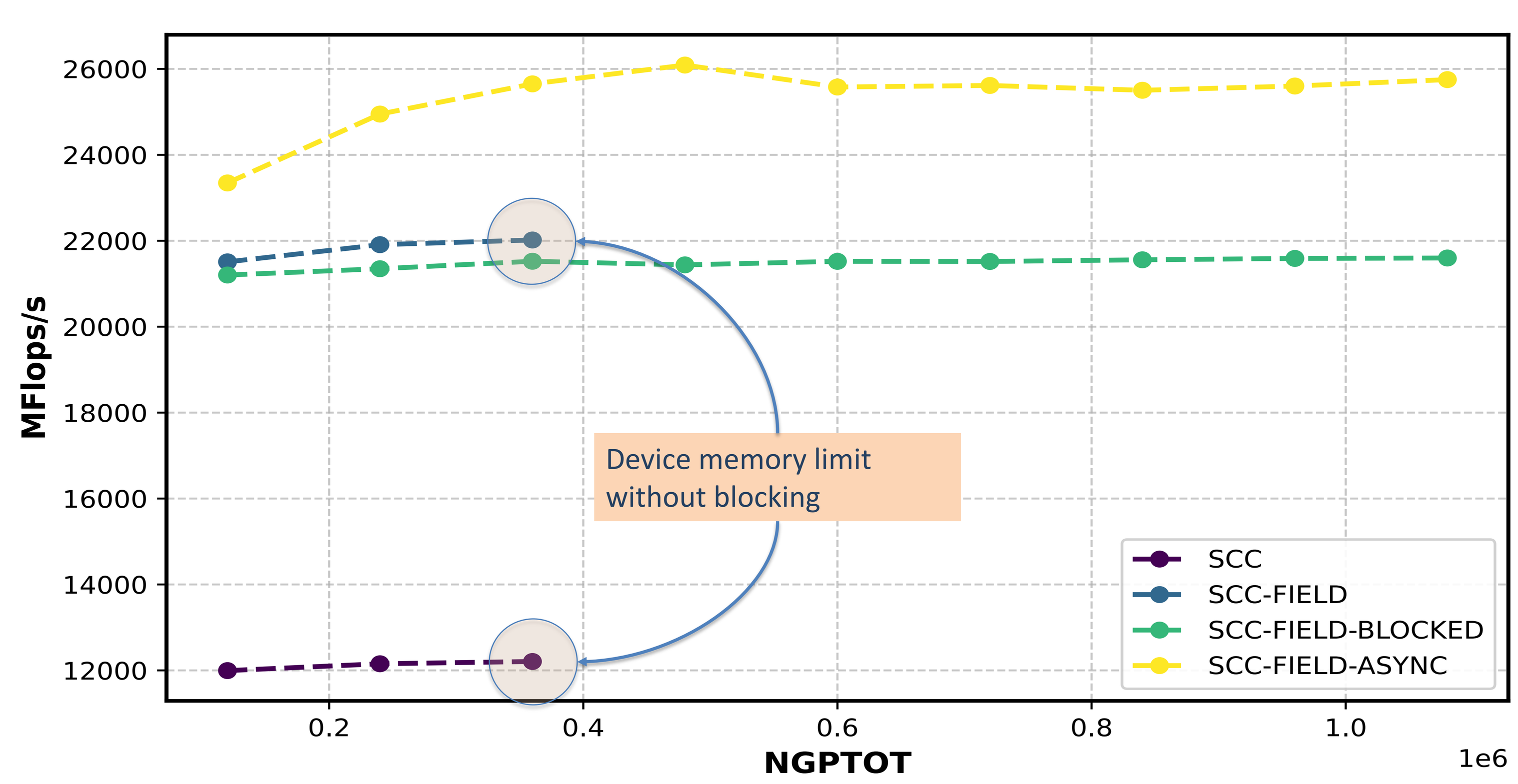



**Figure 1.** Performance of four different versions of CLOUDSC measured on a NVIDIA A100 40GB node on ECMWF's ATOS supercomputer:

## References

[1] FIELD API https://github.com/ecmwf-ifs/field_api
[2] Loki: Freely programmable source-to-source translation https://github.com/ecmwf-ifs/loki
[3] Fypp – Python Powered Fortran metaprogramming https://github.com/aradi/fypp
[4] CLOUDSC https://github.com/ecmwf-ifs/dwarf-p-cloudsc

Funded by the European Union

Destination Earth

implemented by ECMWF esa EUMETSAT