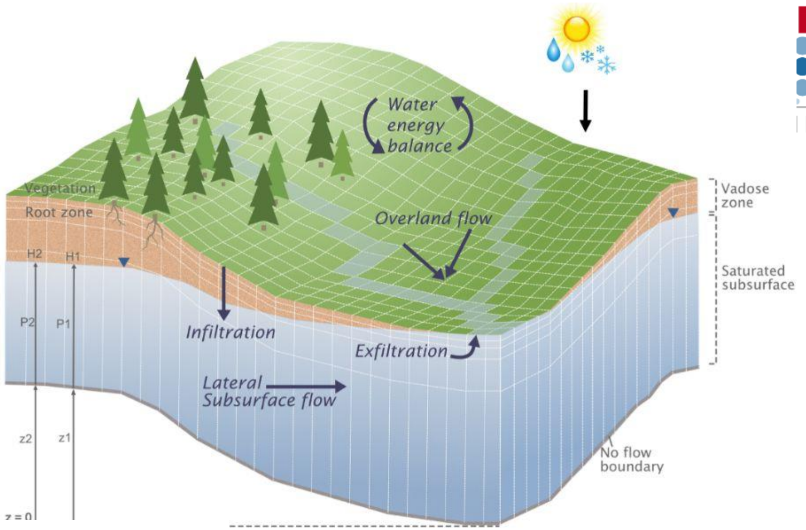
Sprint 4

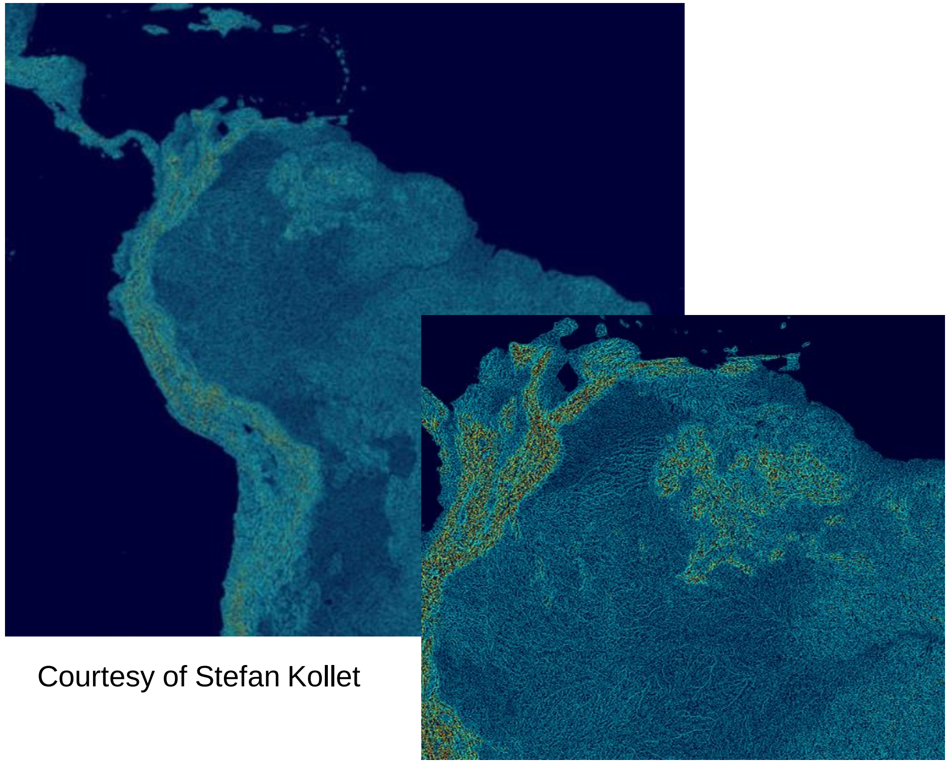
Challenges and results experienced during the first months of the ParFlow sprint

Daniel Caviedes-Voullième (FZ Jülich) & Jörg Benke (JSC)
Stefan Kollet, Andreas Herten (FZ Jülich)

ParFlow https://github.com/parflow

- Integrated hydrological model
- 3D Richards equation + 2D zero-inertia surface flow
- embedded Domain Specific Language (eDSL)
- CUDA and Kokkos-ready via eDSL

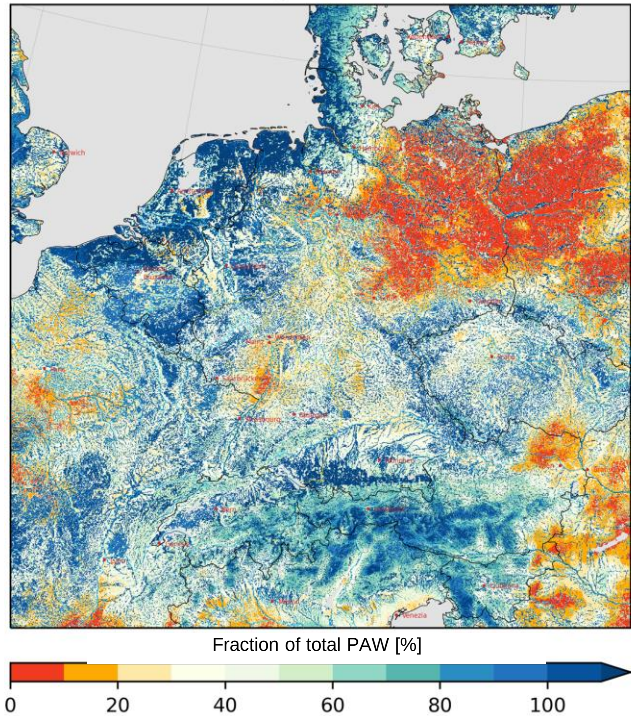




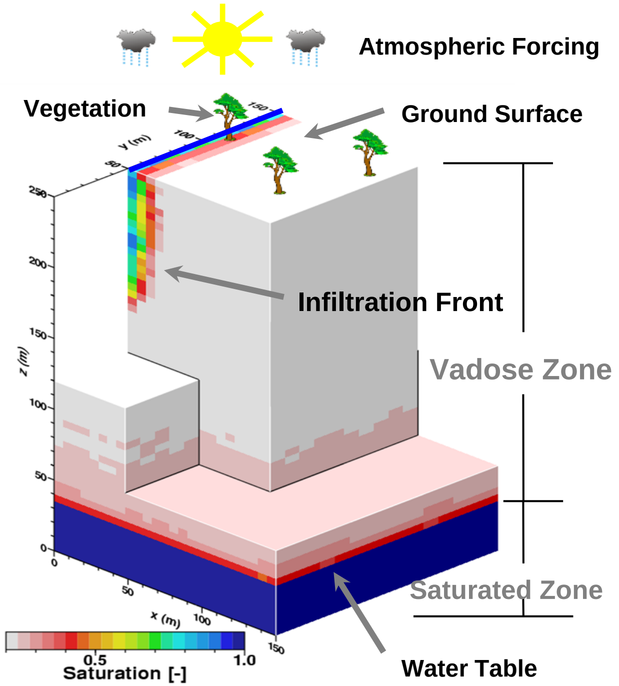
Courtesy of Stefan Kollet

Plant available water

2021-08-18 daily sum, 30cm depth



Courtesy of Klaus Görgen and Alexandre Belleflamme

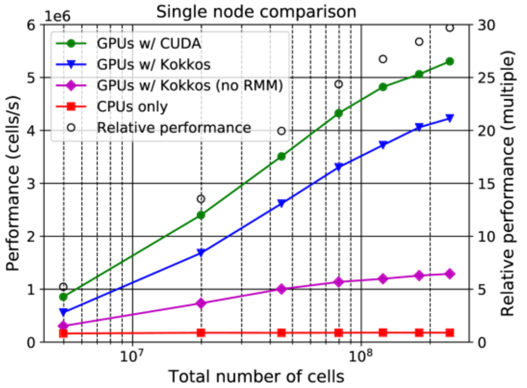


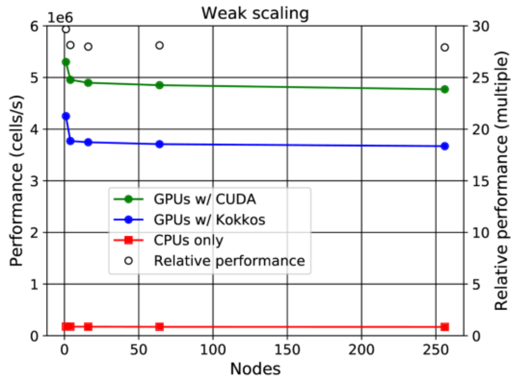
The big picture: ParFlow’s performance portability

```c
double *fp;
double *fp;
double value;
Subvector *f_sub;

/* some code missing here*/

for(k = iz; k < iz + nz; k++)
    for(j = iy; j < iy + ny; j++)
        for(i = ix; j < ix + nx; i++)
        {
            int ip = SubvectorEltIndex(f_sub, i, j, k);
            fp[ip] = pp[ip] - value;
        }
```





MPI + eDSL → MPI + eDSL (CUDA) → MPI + eDSL (CUDA, Kokkos[CUDA]) → MPI + eDSL (CUDA, Kokkos[CUDA,HIP])

2020 2021 2022 2023

```c
double *fp;
double *fp;
double value;
Subvector *f_sub;

/* some code missing here*/

BoxLoopI0(i, j, k, ix, iy, iz, nx, ny, nz,
        {
            int ip = SubvectorEltIndex(f_sub, i, j, k);
            fp[ip] = pp[ip] - value;
        });
```

eDSL

```c
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  auto lambda_body = [=] __host__ __device__
    (const int i, const int j, const int k)
      loop_body;

  /* some code missing for grid & block sizes */

  BoxKernelI0<<<grid, block>>>(lambda_body,
    ix, iy, iz, nx, ny, nz);
}
```

CUDA

Hokkanen et al. 2021. doi: 10.1007/s10596-021-10051-4

```c
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  auto lambda_body = KOKKOS_LAMBDA(int i, int j, int k)
  {
    i += ix; j += iy; k += iz;
    loop_body;
  }

  MDPolicyType_3D mdpolicy_3d({{0, 0, 0}},{{nx, ny, nz}});
  Kokkos::parallel_for(mdpolicy_3d, lambda_body);
}
```

Kokkos

Piotrowsk et al. IN PREPARATION

Sprint description and challenges

Scope of Request: duration: 5 months

Methods to be used:

- performance analyses
- Approach 1: eDSL + Kokkos backend
- Approach 2: HIP
- RAPIDS Memory Manager
- targeted systems: AMD MI250 (experimental hardware at JSC, maybe LUMI)

Criteria for fulfilment:

- performance portability demonstrated on AMD GPUs
- performance analyses
- proof-of-concept simulation at the global scale

⇒ required understanding the Kokkos workflow inside eDSL
⇒ interaction with Kokkos devs to figure out building Kokkos with ROCm
⇒ Kokkos v4.0.0 release warrants upgrades in eDSL+Kokkos implementation
⇒ we have managed to build, but we are still investigating why runtime crashes

⇒ lower priority, still early

⇒ overcoming several low(er)-level issues with ROCm

```
-DCMAKE_CXX_FLAGS='--include /p/…/ROCm/5.4.0-gobliflaf-11.2.0-3.2/include/hip/hip_runtime.h'
```

⇒ Build process aborted with system flang and hipfort (hipfc)

- solution: build ParFlow with hipfort and hipcc. Still not fully clear if ok.

⇒ only one experimental node is operational at JSC
⇒ participated in a workshop at JSC supported by AMD engineers (thanks to Andreas Herten and the Acc. Lab for this)

⇒ delayed response from LUMI for development access (March!)
⇒ need to build all dependencies in LUMI (from OpenMP and up)
⇒ LUMI and/or its filesystem often under maintenance

⇒ still debugging runtime, crashing at non-linear solver stage. Investigating…

Starting point: ParFlow eDSL (embedded Domain Specific Language)

Key idea: abstract code structures which repeat throughout the code into some macros

A typical loop in 3D space spanning indices i,j,k

```
double *fp;
double *fp;
double value;
Subvector *f_sub;

/* some code missing here*/

for(k = iz; k < iz + nz; k++)
    for(j = iy; j < iy + ny; j++)
        for(i = ix; j < ix + nx; i++)
        {
            int ip = SubvectorEltIndex(f_sub, i, j, k);
            fp[ip] = pp[ip] - value;
        }
```

eDSL

Same loop, abstracted into the BoxLoopIO macro

```
double *fp;
double *fp;
double value;
Subvector *f_sub;

/* some code missing here*/

BoxLoopIO(i, j, k, ix, iy, iz, nx, ny, nz,
         {
             int ip = SubvectorEltIndex(f_sub, i, j, k);
             fp[ip] = pp[ip] - value;
         });
```

eDSL macro definition for BoxLoopIO

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  for (k = iz; k < iz + nz; k++)
    for (j = iy; j < iy + ny; j++)
      for (i = ix; i < ix + nx; i++)
      {
        loop_body;
      }
}
```


Hokkanen et al. 2021. Leveraging HPC accelerator architectures with modern techniques — hydrologic modeling on GPUs with ParFlow. Computational Geosciences. doi: 10.1007/s10596-021-10051-4

Starting point: ParFlow eDSL (embedded Domain Specific Language)

Key idea: write all hardware dependent code inside the eDSL macros

Note that in this example, the macro is defined with the same name (**BoxLoopIO**) for all three backends

eDSL macro definition - sequential (host)

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  for (k = iz; k < iz + nz; k++)
    for (j = iy; j < iy + ny; j++)
      for (i = ix; i < ix + nx; i++)
      {
        loop_body;
      }
}
```

eDSL macro definition - Kokkos (host & device / sequential, parallel)

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  auto lambda_body = KOKKOS_LAMBDA(int i, int j, int k)
  {
    i += ix; j += iy; k += iz;
    loop_body;
  }

  MDPolicyType_3D mdpolicy_3d({{0, 0, 0}},{{nx, ny, nz}});
  Kokkos::parallel_for(mdpolicy_3d, lambda_body);
}
```

Piotrowski et al. Lightweight embedded DSLs for geoscientific models. IN PREPARATION

eDSL macro definition - CUDA (device)

```
#define BoxLoopI0(i, j, k, ix, iy, iz,
  nx, ny, nz, loop_body)
{
  auto lambda_body = [=] __host__ __device__
    (const int i, const int j, const int k)
      loop_body;

  /* some code missing for grid & block sizes */

  BoxKernelI0<<<grid, block>>>(lambda_body,
    ix, iy, iz, nx, ny, nz);
}
```

Kernel launch

Hokkanen et al. 2021. Leveraging HPC accelerator architectures with modern techniques — hydrologic modeling on GPUs with ParFlow. Computational Geosciences. doi: 10.1007/s10596-021-10051-4

Starting point: resolving backends in eDSL

```
#define BoxLoopI1_cuda(i, j, k,
  ix, iy, iz, nx, ny, nz,
  i1, nx1, ny1, nz1, sx1, sy1, sz1,
  loop_body)
{
  if(nx > 0 && ny > 0 && nz > 0)
  {
    DeclareInc(PV_jinc_1, PV_kinc_1, nx, ny, nz, nx1, ny1, nz1, sx1, sy1, sz1);

    dim3 block, grid;
    FindDims(grid, block, nx, ny, nz, 1);

    const auto &ref_i1 = i1;

    auto lambda_body =
      GPU_LAMBDA(int i, int j, int k)
      {
        const int i1 = k * PV_kinc_1 + (k * ny + j) * PV_jinc_1
          + (k * ny * nx + j * nx + i) * sx1 + ref_i1;

        i += ix;
        j += iy;
        k += iz;
        loop_body;
      };

    BoxKernel<<<grid, block>>>(lambda_body, nx, ny, nz);
    CUDA_ERR(cudaPeekAtLastError());

    typedef function_traits<decltype(lambda_body)> traits;
    if(!std::is_same<traits::result_type, struct SkipParallelSync>::value)
      CUDA_ERR(cudaStreamSynchronize(0));
  }
  (void)i;(void)j;(void)k;
}
```

CUDA

Somewhere in Parflow we call **BoxLoopI1**

```
perm_x_elt = SubvectorElt(perm_x_sub, ix, iy, iz);
perm_y_elt = SubvectorElt(perm_y_sub, ix, iy, iz);
perm_z_elt = SubvectorElt(perm_z_sub, ix, iy, iz);

pi = 0;
BoxLoopI1(i, j, k,
          ix, iy, iz, nx, ny, nz,
          pi, nx_p, ny_p, nz_p, 1, 1, 1,
{
  perm_average_x += perm_x_elt[pi] * (cell_volume / well_volume);
  perm_average_y += perm_y_elt[pi] * (cell_volume / well_volume);
  perm_average_z += perm_z_elt[pi] * (cell_volume / well_volume);
});

FreeSubgrid(tmp_subgrid);       /* done with temporary subgrid */
```

HOST

```
#define BoxLoopI1_default(i, j, k,
                  ix, iy, iz, nx, ny, nz,
                  i1, nx1, ny1, nz1, sx1, sy1, sz1,
                  body)
  {
    DeclareInc(PV_jinc_1, PV_kinc_1, nx, ny, nz, nx1, ny1, nz1, sx1, sy1, sz1);
    for (k = iz; k < iz + nz; k++)
    {
      for (j = iy; j < iy + ny; j++)
      {
        for (i = ix; i < ix + nx; i++)
        {
          body;
          i1 += sx1;
        }
        i1 += PV_jinc_1;
      }
      i1 += PV_kinc_1;
    }
  }
```

BoxLoopI1_default

Starting point: memory management

From `parflow/pfsimulator/parflow_lib/mg_semi.c`

```
grid_l = talloc(Grid *, num_levels);
grid_l[0] = grid;

c_sra_l = talloc(SubregionArray *, (num_levels - 1));
f_sra_l = talloc(SubregionArray *, (num_levels - 1));

restrict_compute_pkg_l = talloc(ComputePkg *, (num_levels - 1));
prolong_compute_pkg_l = talloc(ComputePkg *, (num_levels - 1));


A_l = talloc(Matrix *, num_levels);
P_l = talloc(Matrix *, num_levels - 1);
```

these memory addresses are used in parallel regions, therefore they need to be allocated on the right memory space

From `parflow/pfsimulator/parflow_lib/backend_mapping.h`

```
#if defined(talloc_cuda) || defined(talloc_kokkos) ||
defined(talloc_omp)
  #define talloc CHOOSE_BACKEND(DEFER(talloc), ACC_ID)
#else
  #define talloc talloc_default
#endif
```

From `parflow/pfsimulator/parflow_lib/pf_cudamalloc.h`

```
#define talloc_cuda(type, count) \
  ((count) ? (type*)_talloc_device(sizeof(type) * (unsigned int)(count)) : NULL)
```

```
static inline void *_talloc_device(size_t size)
{
  void *ptr = NULL;

#ifdef PARFLOW_HAVE_RMM
  RMM_ERR(rmmAlloc(&ptr,size,0,__FILE__,__LINE__));
#elif defined(PARFLOW_HAVE_KOKKOS)
  ptr = kokkosAlloc(size);
#elif defined(PARFLOW_HAVE_CUDA)
  CUDA_ERR(cudaMallocManaged((void**)&ptr, size, cudaMemAttachGlobal));
  // CUDA_ERR(cudaHostAlloc((void**)&ptr, size, cudaHostAllocMapped));
#endif

  return ptr;
}
```

Starting point: kokkos

natESM

eDSL Kokkos wrappers

```
void* kokkosDeviceAlloc(size_t size){
#ifdef PARFLOW_HAVE_CUDA
  return Kokkos::kokkos_malloc<Kokkos::CudaSpace>(size);
#else
  return Kokkos::kokkos_malloc(size);
#endif
}
```

```
void kokkosDeviceFree(void *ptr){
#ifdef PARFLOW_HAVE_CUDA
  Kokkos::kokkos_free<Kokkos::CudaSpace>(ptr);
#else
  Kokkos::kokkos_free(ptr);
#endif
}
```

```
void* kokkosHostAlloc(size_t size){
#ifdef PARFLOW_HAVE_CUDA
  return Kokkos::kokkos_malloc<Kokkos::CudaHostPinnedSpace>(size);
#else
  return Kokkos::kokkos_malloc<Kokkos::HostSpace>(size);
#endif
}
```

eDSL loop macro definition - Kokkos

```
#define BoxLoopI1_kokkos(i, j, k,
  ix, iy, iz, nx, ny, nz,
  i1, nx1, ny1, nz1, sx1, sy1, sz1,
  loop_body)
{
  if(nx > 0 && ny > 0 && nz > 0)
  {
    DeclareInc(PV_jinc_1, PV_kinc_1, nx, ny, nz, nx1, ny1, nz1, sx1, sy1, sz1);

    const auto &ref_i1 = i1;

    auto lambda_body =
      KOKKOS_LAMBDA(int i, int j, int k)
      {
        const int i1 = k * PV_kinc_1 + (k * ny + j) * PV_jinc_1
          + (k * ny * nx + j * nx + i) * sx1 + ref_i1;

        i += ix;
        j += iy;
        k += iz;

        loop_body;
      };

    using MDPolicyType_3D = typename Kokkos::Experimental::MDRangePolicy<Kokkos::
    MDPolicyType_3D mdpolicy_3d({{0, 0, 0}}, {{nx, ny, nz}});
    Kokkos::parallel_for(mdpolicy_3d, lambda_body);

    typedef function_traits<decltype(lambda_body)> traits;
    if(!std::is_same<traits::result_type, struct SkipParallelSync>::value)
      Kokkos::fence();
  }
  (void)i;(void)j;(void)k;
}
```

Approach 1: HIPifying via kokkos

work in progress!!!

natESM

```
#define PF_KOKKOS_MEMSPACE_CONCAT(a,b) a::b
#if PARFLOW_HAVE_CUDA
    #define PF_KOKKOS_MEMSPACE    PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,CudaSpace)
    #define PF_KOKKOS_MEMPINSPACE PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,CudaHostPinnedSpace)
#elif PARFLOW_HAVE_HIP
    #define PF_KOKKOS_MEMSPACE    PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,Experimental::HIPSpace)
    #define PF_KOKKOS_MEMPINSPACE PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,Experimental::HIPHostPinnedSpace)
#elif
    #define PF_KOKKOS_MEMSPACE    PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,HostSpace)
    #define PF_KOKKOS_MEMPINSPACE PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,HostSpace)
#endif
```

```
void* kokkosDeviceAlloc(size_t size){
#ifdef PARFLOW_HAVE_CUDA
  return Kokkos::kokkos_malloc<Kokkos::CudaSpace>(size);
#else
  return Kokkos::kokkos_malloc(size);
#endif
}
```

```
void* kokkosDeviceAlloc(size_t size){
  return Kokkos::kokkos_malloc<PF_KOKKOS_MEMSPACE>(size);
}
```

Approach 1: HIPifying via kokkos

work in progress!!!

natESM

HIPified eDSL Kokkos macros

```
#define PF_KOKKOS_MEMSPACE_CONCAT(a,b) a::b
#if PARFLOW_HAVE_CUDA
    #define PF_KOKKOS_MEMSPACE    PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,CudaSpace)
    #define PF_KOKKOS_MEMPINSPACE PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,CudaHostPinnedSpace)
#elif PARFLOW_HAVE_HIP
    #define PF_KOKKOS_MEMSPACE    PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,Experimental::HIPSpace)
    #define PF_KOKKOS_MEMPINSPACE PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,Experimental::HIPHostPinnedSpace)
#elif
    #define PF_KOKKOS_MEMSPACE    PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,HostSpace)
    #define PF_KOKKOS_MEMPINSPACE PF_KOKKOS_MEMSPACE_CONCAT(Kokkos,HostSpace)
#endif
```

HIPified eDSL Kokkos wrappers

```
void* kokkosDeviceAlloc(size_t size){
  return Kokkos::kokkos_malloc<PF_KOKKOS_MEMSPACE>(size);
}
```

```
void kokkosDeviceFree(void *ptr){
  Kokkos::kokkos_free<PF_KOKKOS_MEMSPACE>(ptr);
}
```

```
void* kokkosHostAlloc(size_t size){
  return Kokkos::kokkos_malloc<PF_KOKKOS_MEMPINSPACE>(size);
}
```

eDSL loop macro definition - Kokkos

```
#define BoxLoopI1_kokkos(i, j, k,
  ix, iy, iz, nx, ny, nz,
  i1, nx1, ny1, nz1, sx1, sy1, sz1,
  loop_body)
{
  if(nx > 0 && ny > 0 && nz > 0)
  {
    DeclareInc(PV_jinc_1, PV_kinc_1, nx, ny, nz, nx1, ny1, nz1, sx1, sy1, sz1);

    const auto &ref_i1 = i1;

    auto lambda_body =
      KOKKOS_LAMBDA(int i, int j, int k)
      {
        const int i1 = k * PV_kinc_1 + (k * ny + j) * PV_jinc_1
          + (k * ny * nx + j * nx + i) * sx1 + ref_i1;

        i += ix;
        j += iy;
        k += iz;

        loop_body;
      };

    using MDPolicyType_3D = typename Kokkos::Experimental::MDRangePolicy<Kokkos::
    MDPolicyType_3D mdpolicy_3d({{0, 0, 0}}, {{nx, ny, nz}});
    Kokkos::parallel_for(mdpolicy_3d, lambda_body);

    typedef function_traits<decltype(lambda_body)> traits;
    if(!std::is_same<traits::result_type, struct SkipParallelSync>::value)
      Kokkos::fence();
  }
  (void)i;(void)j;(void)k;
}
```

Outlook & open questions

- Performance evaluation of the Kokkos(HIP) solution
- Scaling up in LUMI
- Hard - HIP backend
- Addressing the memory pooling problem
  - best solution: redesign memory allocation paradigm in ParFlow
  - hacky solution: evaluate the Kokkos pool allocator, or the Umpire memory manager

Sprint 4

Challenges and results experienced during the first months of the ParFlow sprint

Daniel Caviedes-Voullième (FZ Jülich) & Jörg Benke (JSC)
Stefan Kollet, Andreas Herten (FZ Jülich)