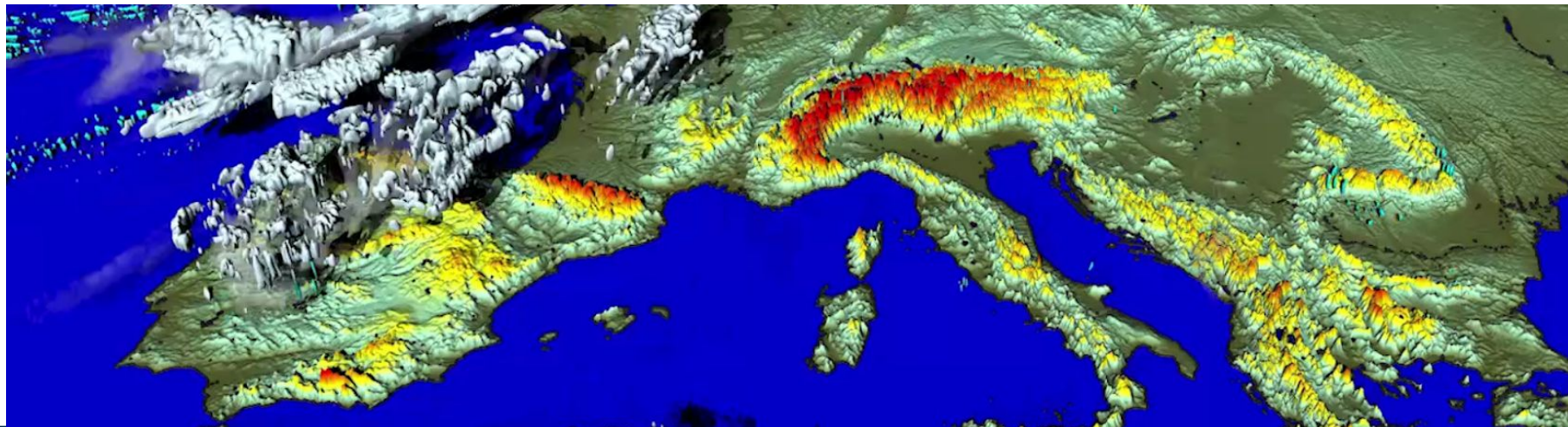




Coupled earth system modeling on heterogeneous HPC architectures with ParFlow in the Terrestrial Systems Modeling Platform

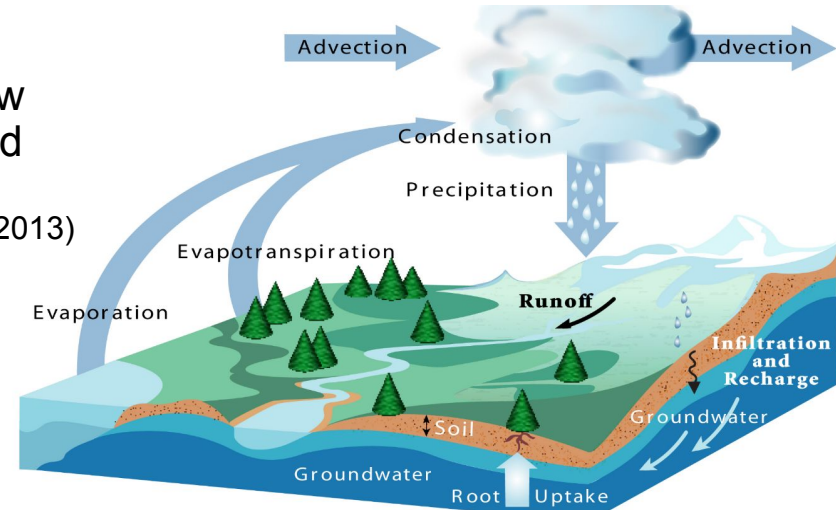
2021-4-28 | J. Hokkanen¹, S. Kollet¹, J. Kraus², A. Herten³, M. Hrywniak², D. Pleiter³ | ¹FZJ/IBG-3, ²NVIDIA GmbH, ³FZJ/JSC

Member of the Helmholtz Association



What is ParFlow?

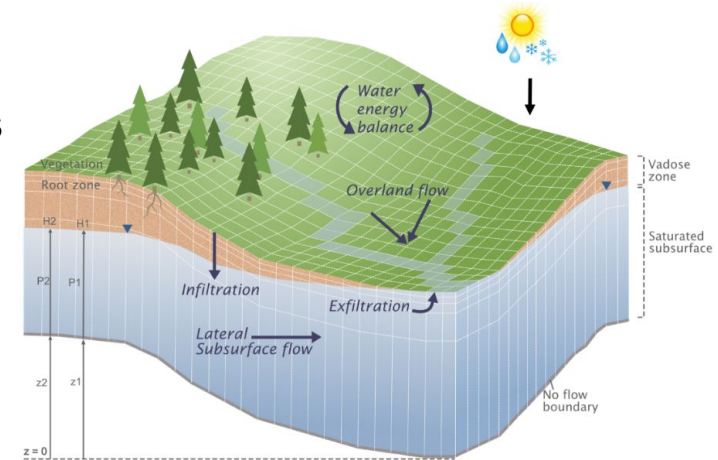
- 3D model for variably saturated subsurface flow including pumping and irrigation, and integrated overland flow
(Jones & Woodward, 2001; Kollet & Maxwell, 2006; Maxwell, 2013)
- Integrated with land surface and also regional climate model, TSMP (Shrestha et al., 2014)
- External coupling via OASIS3-MCT (Shrestha et al., 2014; Gasper et al., 2014)
- Parallel Data Assimilation Framework with PDAF (Kurtz et al., 2016)
- Optimized for modern heterogeneous supercomputer environments
(e.g. Gasper et al., 2014; Burstedde et al., 2018; Hokkanen et al., 2021)



https://github.com/hydroframe/ParFlow_Short_Course/blob/master/Slides/1.parflow_intro.pdf

ParFlow numerical model

- Cell centered finite difference / finite volume schemes
- Implicit time integration
- Newton-Krylov methods (Kinsol) with multigrid preconditioning (ParFlow, Hypre) for nonlinear problems
- A large number of numerical kernels, none of which clearly dominate the run time
- Around 150k lines of C



https://github.com/hydroframe/ParFlow_Advanced_ShortCourse/blob/master/slides/1.Intro_Overview.pdf



Desirable requirements for the accelerator support

- Good performance
- Single-source application
- Codebase long-term maintainability
- Minimum reliance on external dependencies
- Separation of concerns (scientific development vs. accelerator utilization)
- High developer productivity for adding support for new accelerators
- ParFlow solution: Support for accelerators is added into a macro-based abstraction layer (ParFlow embedded Domain-Specific Language, ParFlow eDSL)

ParFlow eDSL interface

Allocations & initializations

```
KW = NewVectorType(grid2d, 1, 1, cell_centered);  
InitVector(KW, 0.0);
```

Message passing

```
update_handle = InitVectorUpdate(  
    pressure, VectorUpdateAll);  
FinalizeVectorUpdate(update_handle);
```

Accessor macros

```
ix = SubgridIX(subgrid);  
iy = SubgridIY(subgrid);  
iz = SubgridIZ(subgrid);
```

Loop macros

```
GrGeomInLoop(i, j, k, gr, r, ix, iy, iz, nx, ny, nz,  
{  
    int ips = SubvectorEltIndex(ps_sub, i, j, k);  
    data[ips] = value;  
});
```

ParFlow eDSL for GPUs

- ParFlow can use a native CUDA implementation or the Kokkos library for GPUs
 - Both use the same interface for memory management and compute kernels
- Some of the most recent technology is used for best performance and developer productivity:
 - Unified Memory
 - Host-device lambdas
 - Pool allocator for Unified Memory (RMM)
- CUDA-aware MPI library and GPU-based application-side data packing routines can be used for fast GPU-GPU communication

ParFlow eDSL for GPUs: Development steps

1. **GPU affinity:** Select one GPU device for each MPI process
2. **Memory management:** Allocate Unified Memory for the GPU data
3. **Loops:** Use the same loop interface for CPU / GPU compute kernels
4. **MPI:** Use application-side GPU-based packing/unpacking kernels for MPI
5. **Profiling and optimization:** Detect page-faults, optimize GPU-GPU data transfers, coalesce GPU memory accesses, add pool allocator for Unified Memory, etc

ParFlow eDSL for GPUs: Incremental development

- In ParFlow, memory (de)allocations/compute kernels are accessed through the eDSL
- Unified Memory (de)allocations/compute kernels are controlled for each compilation unit separately allowing incremental development and flexibility
 - Some source files use Unified Memory and GPU-based loops, and others not

```
/* PFCUDA_COMP_UNIT_TYPE determines the compilation unit type:  
  1: Unified Memory allocation, Parallel loops (GPUs)  
  2: Unified Memory allocation, Sequential loops (CPUs)  
  default: Standard heap allocation, Sequential loops (CPUs) */  
#define PFCUDA_COMP_UNIT_TYPE 1 // Defined by CMake
```

Memory management: Host memory

Source file

```
vector = talloc(Vector, 1);
```

```
tfree(vector);
```

eDSL header file

```
#define talloc(type, count) \  
    (type*)malloc(sizeof(type) \  
        * (unsigned int)(count))
```

```
#define tfree(ptr) free(ptr)
```

Memory management: Unified Memory

CPU version eDSL header file

```
#define talloc(type, count) \  
    (type*)malloc(sizeof(type) \  
        * (unsigned int)(count))
```

GPU version eDSL header file

```
#define talloc(type, count) \  
    (type*)talloc_cuda(sizeof(type) \  
        * (unsigned int)(count))  
  
static inline void *talloc_cuda(size_t size)  
{  
    void *ptr = NULL;  
    // cudaMallocManaged(&ptr, size);  
    rmmAlloc(&ptr, size, 0, __FILE__, __LINE__);  
    return ptr;  
}
```

Memory management: Unified Memory

CPU version eDSL header file

```
#define tfree(ptr) free(ptr)
```

GPU version eDSL header file

```
#define tfree(ptr) tfree_cuda(ptr)

static inline void tfree_cuda(void *ptr)
{
    // cudaFree(&ptr);
    rmmFree(ptr, 0, __FILE__, __LINE__);
}
```

Loops: An example of a specialized GPU kernel

Original source file

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

What could be done, but...

```
#ifndef HAVE_CUDA
/* some code to find grid & block sizes */
CUDAKernel<<<grid, block>>>(ix, ...);
#else
BoxLoopI0(i, j, k, ix, iy, iz, nx, ny, nz,
{
    int ip;
    ip = SubvectorEltIndex(f_sub, i, j, k);
    fp[ip] = pp[ip] - value;
});
#endif
```

Loops: General GPU kernels

Original source file

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

New source file

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

Loops: General GPU kernels

CPU version eDSL header file

```
#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; \
                } \
}
```

GPU version eDSL header file (CUDA)

```
#define BoxLoopI0(i, j, k, \
    ix, iy, iz, nx, ny, nz, loop_body) \
{ \
    /* some code to find grid & block sizes */ \
    auto lambda_body = [=] __host__ __device__ \
        (int i, int j, int k) \
        { \
            i += ix; j += iy; k += iz; \
            loop_body; \
        } \
    BoxKernelI0<<<grid, block>>>(lambda_body, \
        nx, ny, nz); \
}
```

Loops: General GPU kernels

CPU version eDSL header file

```
#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;\
            }\
}
```

GPU version eDSL header file (Kokkos)

```
#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);\
}
```

Loops: A general CUDA kernel example

```
template <typename LambdaBody>
__global__ static void BoxKernelI0(LambdaBody loop_body,
  const int nx, const int ny, const int nz)
{
  int i = ((blockIdx.x * blockDim.x) + threadIdx.x);
  int j = ((blockIdx.y * blockDim.y) + threadIdx.y);
  int k = ((blockIdx.z * blockDim.z) + threadIdx.z);

  if(i < nx && j < ny && k < nz)
  {
    loop_body(i, j, k);
  }
}
```

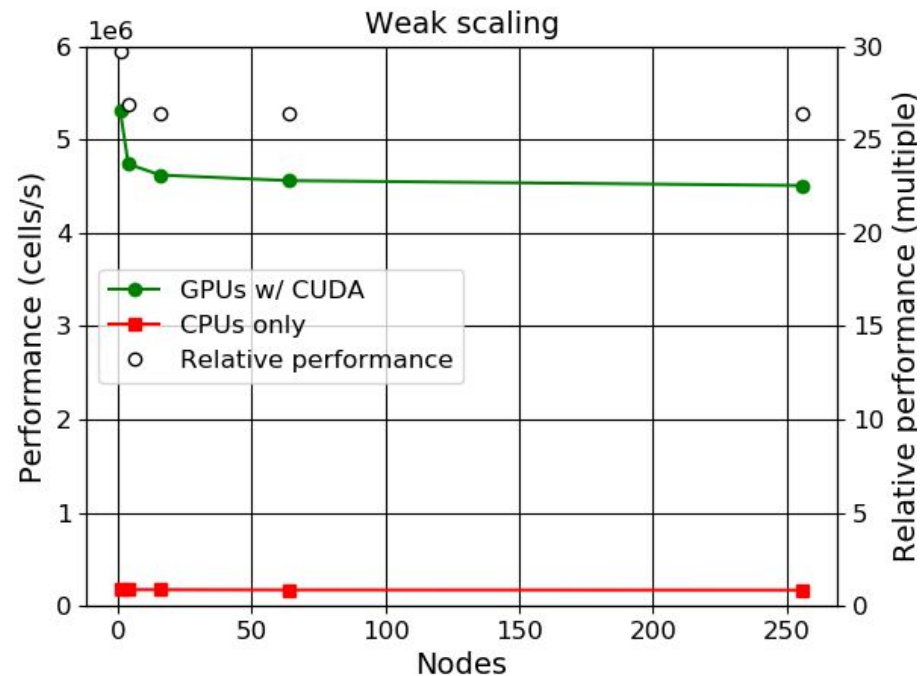
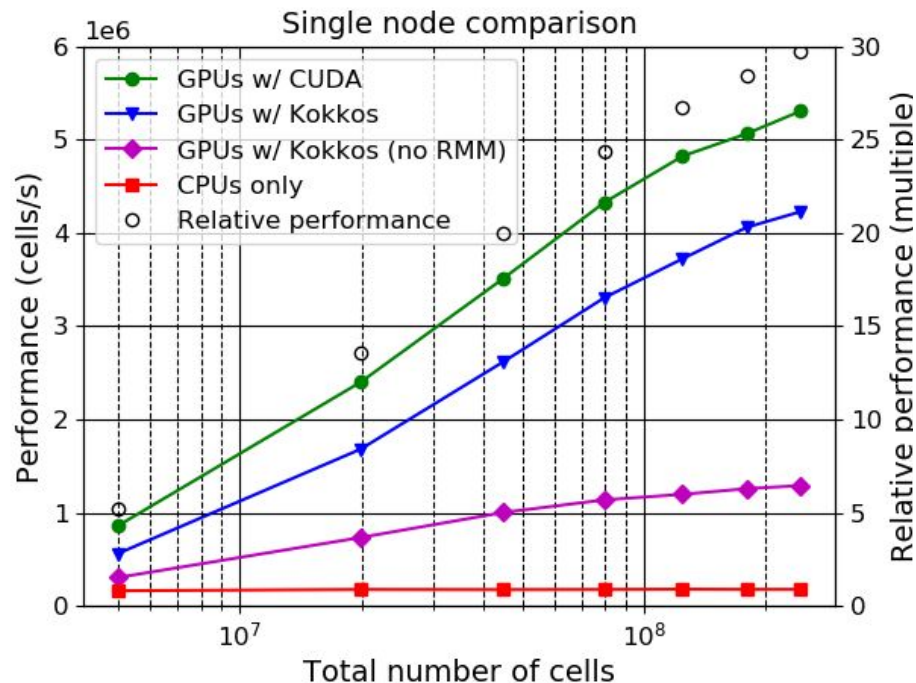
MPI issues and GPU-GPU communication

- Before GPU support, derived MPI datatypes and MPI library-side data packing/unpacking were commonly used in ParFlow
- **Problem:** In the ParFlow GPU version, the pointers that were originally passed to the MPI library pointed to Unified Memory allocations
 - No GPU-based packing/unpacking support from MPI libraries
 - Frequent segfaults with CUDA-aware MVAPICH2-GDR
 - CUDA-aware ParastationMPI worked, but GPU-GPU data transfers were very slow (no GPUDirect P2P/RDMA)
- **Solution:** New application-side GPU-based data packing routines with pinned GPU staging buffers and a simple MPI_BYTE data type for MPI communication (GPUDirect P2P/RDMA now works with CUDA-aware MPI libraries)

Profiling & Optimization

- **Used profiling tools:**
 - Nvidia Visual Profiler
 - Nvidia Nsight Systems + Compute
- **Observed importance (ordered):**
 1. Minimizing recurring page faults and host-device data transfer (large impact)
 2. Adding efficient parallel reduction kernels (large impact)
 3. Using pool allocator for Unified Memory (large impact)
 4. Coalescing device memory accesses (large impact)
 5. Avoiding unnecessary synchronizations (small impact)
 6. Tweaking with grid & block sizes (small impact)

Results from JUWELS Booster supercomputer



From continental to global simulations with GPUs

- ParFlow-GPU, JUWELS Booster, global, 1km, 60GB / output timestep;
S. America, surface drainage

Simulation from JUWELS Booster
Early Access Program
by Hokkanen, Kollet, and Naz

Heterogeneous jobs with TerrSysMP

- ParFlow can be coupled with other independently developed earth system compartment models such as land surface and atmospheric models
- Leveraging heterogeneous jobs can lead to better performance and more efficient usage of the HPC resources

```
#SBATCH -N 4 --ntasks-per-node=48 -p batch
#SBATCH hetjob
#SBATCH -N 1 --ntasks-per-node=48 -p batch
#SBATCH hetjob
#SBATCH -N 1 --ntasks-per-node=4 --gres=gpu:4 -p develgpu

srun --het-group=0 ./lmparbin_pur : \
    --het-group=1 ./clm : \
    --het-group=2 ./parflow cordex
```

Slurm job script for running ParFlow on a GPU node and COSMO and CLM on CPU-only nodes

Summary

- The ParFlow results from JUWELS Booster supercomputer demonstrate a very good weak scaling with up to **26X** speedup from A100 GPUs over hundreds of nodes
- Hiding GPU support into ParFlow eDSL resulted in good developer productivity, codebase long-term maintainability, and performance gain
- An existing DSL is not a prerequisite: Only a minimal abstraction layer for accessing the most relevant loops and memory (de)allocations is required
- Unified Memory support played an important role in good developer productivity
- Pool allocator for GPU memory can play a very important role for high performance gain