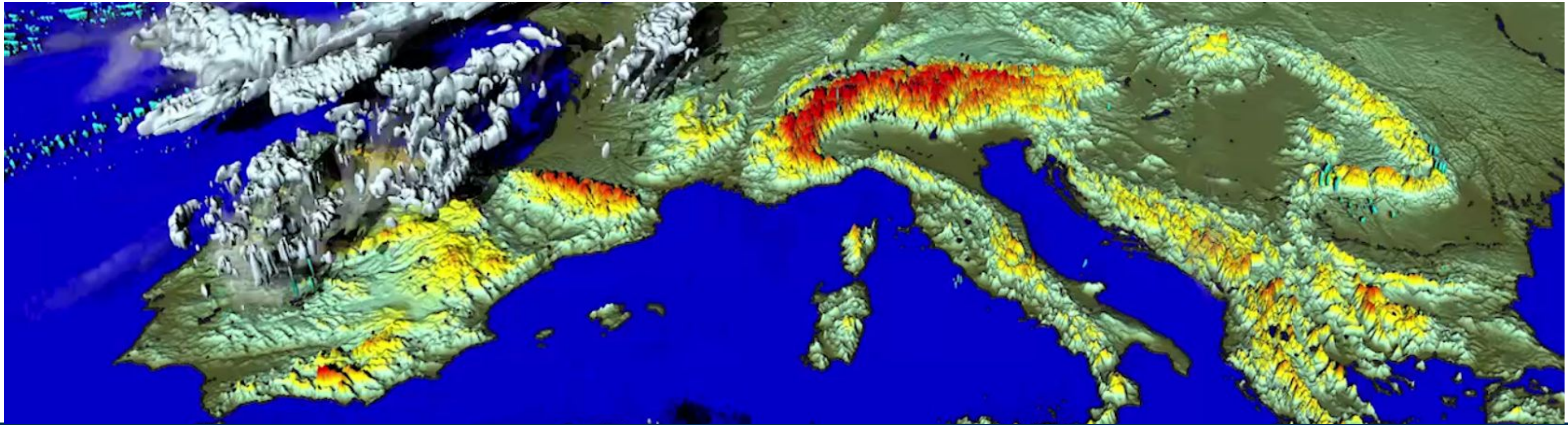


HPC application design: ParFlow GPU support

2022-07-05 | **Jaro Hokkanen**

CSC

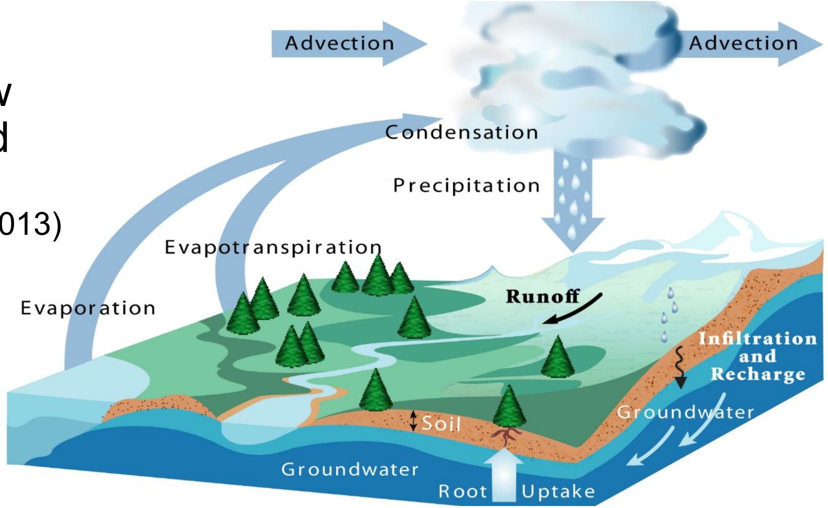
ESM
Advanced Earth System Modeling Capacity
Pilot ExaESM

Energy oriented Centre of Excellence
EoCoE

JÜLICH
Forschungszentrum

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 supercomputers environments
  (e.g. Gasper et al., 2014, Burstedde et al., 2018; Hokkanen et al., 2021)



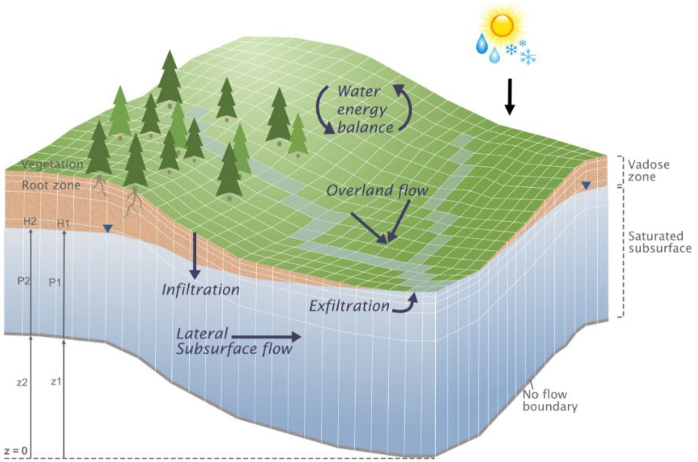
https://github.com/hydroframe/ParFlow_Short_Course/blob/master/Slides/1.parflow_intro.pdf

CSC

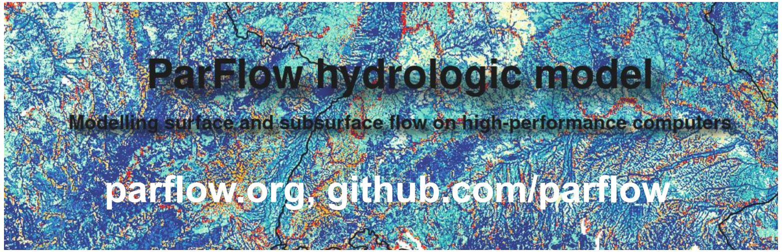
JÜLICH Forschungszentrum

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
- CUDA-aware MPI library and GPU-based application-side data packing routines can be used for fast GPU-GPU communication

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: HIP/NVCC compiler, Unified Memory allocation, Parallel loops (GPUs)
  2: HIP/NVCC compiler, Unified Memory allocation, Sequential loops (CPUs)
  default: HIP/NVCC compiler, 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)
```

CSC

JÜLICH Forschungszentrum

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_hip(sizeof(type) \
    * (unsigned int)(count))

static inline void *talloc_hip(size_t size)
{
   void *ptr = NULL;
// hipMallocManaged(&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_hip(ptr)

static inline void tfree_hip(void *ptr)
{
// hipFree(&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...

```
#ifdef HAVE_HIP
/* some code to find grid & block sizes */
HIPKernel<<<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/HIP)

```
#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

GPU version eDSL header file (CUDA/HIP)

```
#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);                                    \
}
```

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 HIP/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);
  }
}
```

A complete hello world example


```
#include <stdio.h>

/* Kernel macro selection (CPU/GPU) */
// #define BoxLoop BoxLoopCPU
#define BoxLoop BoxLoopGPU

/* Compute kernel macro for CPU (API) */
#define BoxLoopCPU(i, nx, loop_body)              \
{                                                 \
  for (i = 0; i < nx; i++)                        \
  {                                               \
    loop_body;                                    \
  }                                               \
}

/* Compute kernel macro for GPU (API) */
#define BoxLoopGPU(i, nx, loop_body)              \
{                                                 \
  auto lambda = [=] __host__ __device__ (int i)   \
  {                                               \
    loop_body;                                    \
  };                                              \
                                                  \
  int blocksize = 1024;                           \
  int gridsize = (nx - 1 + blocksize) / blocksize; \
                                                  \
  _BoxKernel<<<gridsize, blocksize>>>(lambda, nx); \
  cudaStreamSynchronize(0);                       \
  (void)i;                                        \
}
```

```
/* General GPU kernel */
template <typename LambdaBody> __global__ static
void _BoxKernel(LambdaBody lambda, const int nx)
{
  const int i =
    blockIdx.x * blockDim.x + threadIdx.x;
  if(i < nx)
  {
    lambda(i);
  }
}

/* Memory allocation macro (API) */
#define alloc_managed(type, count)                  \
  (type*)_alloc_managed(count * sizeof(type));

/* Function to allocate Unified Memory */
static inline void *_alloc_managed(size_t size)
{
  void *ptr = NULL;
  hipMallocManaged((void**)&ptr, size);
  return ptr;
}

/* Memory deallocation macro (API) */
#define free_managed(ptr) _free_managed(ptr);

/* Function to deallocate Unified Memory */
static inline void _free_managed(void *ptr)
{
  hipFree(ptr);
}
```

← Headers

Driver function

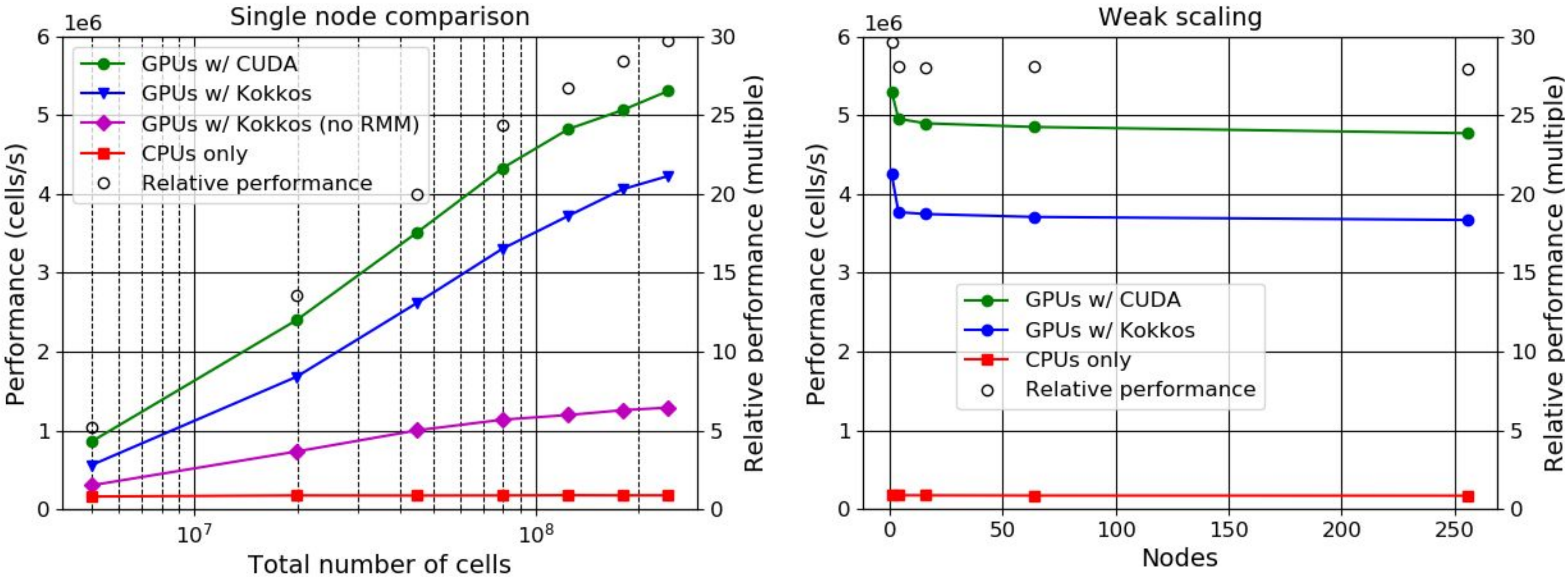
```
/* Driver function */
int main(int argc, char *argv [])
{
  int i, nx = 10;
  int* array = alloc_managed(int, nx);

  BoxLoop(i, nx,
  {
    array[i] = i;
  });

  BoxLoop(i, nx,
  {
     int thread = array[i];
     printf("Hello from GPU thread %d\n", thread);
  });

  free_managed(array);
}
```

Results from JUWELS Booster supercomputer



CSC

JÜLICH
Forschungszentrum

Summary

- The ParFlow results from JUWELS Booster supercomputer demonstrate a very good weak scaling with up to 28X speedup from A100 GPUs over hundreds of nodes
- Hiding GPU support into a macro-based DSL 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
- Does the approach work for Fortran? -Yes, but only with CUDA: The approach has been tested in COSMO atmospheric model using CUF kernels instead of lambda functions

Background: ParFlow-GPU, JUWELS Booster, global, 1km, 60GB / output timestep;
showing S. America, surface drainage (Hokkanen, Kollet, and Naz)