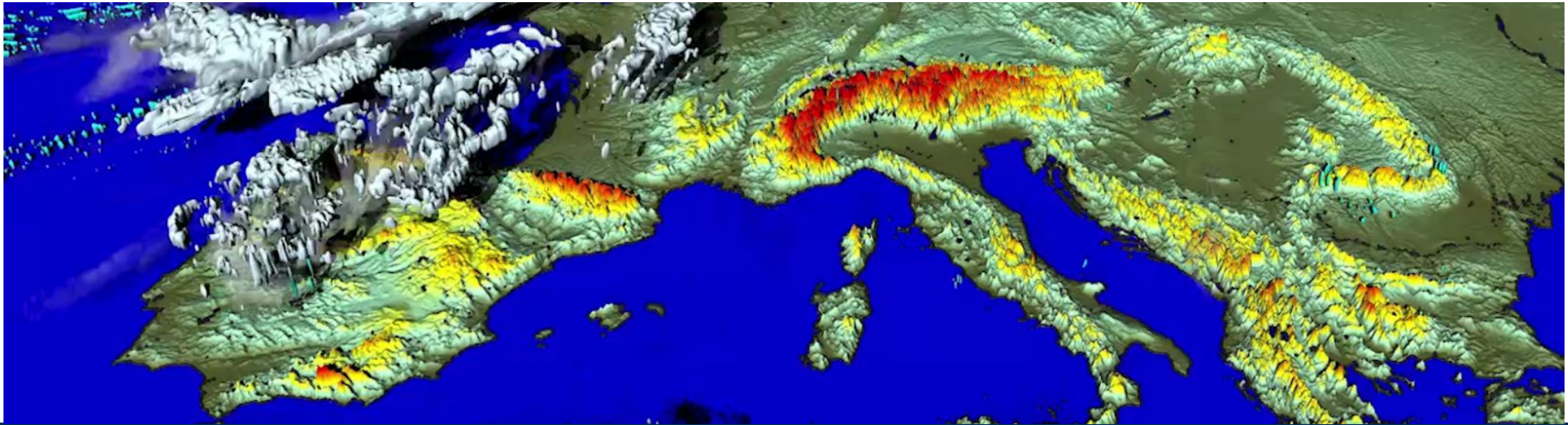
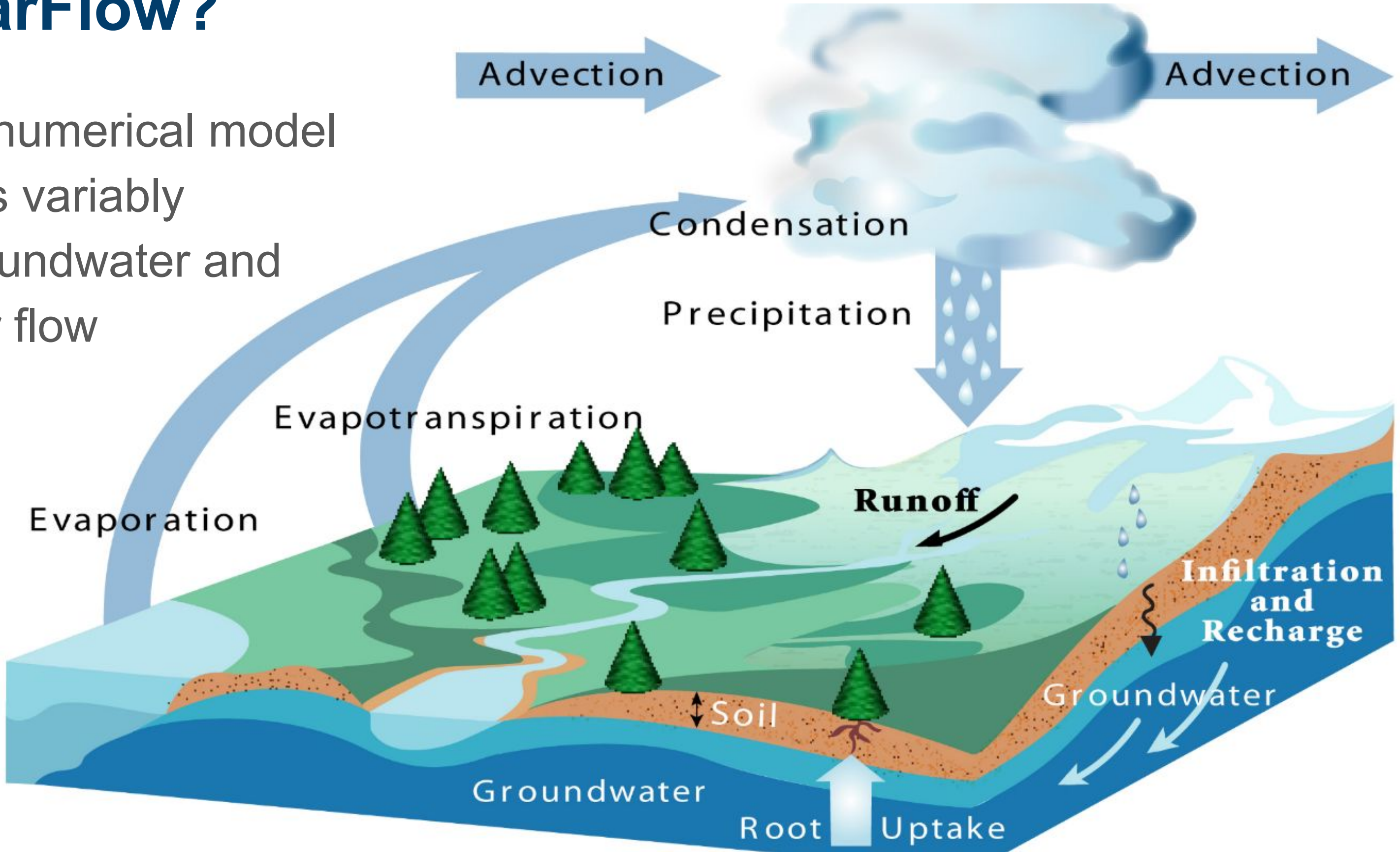


Hydrologic modeling on GPUs with ParFlow — Leveraging accelerator architectures with modern techniques

2020-06-02 I J. Hokkanen[1], J. Kraus[2], A. Herten[3], D. Pleiter[3], S. Kollet[1] I [1]FZJ/IBG-3, [2]NVIDIA GmbH, [3]FZJ/JSC

What is ParFlow?

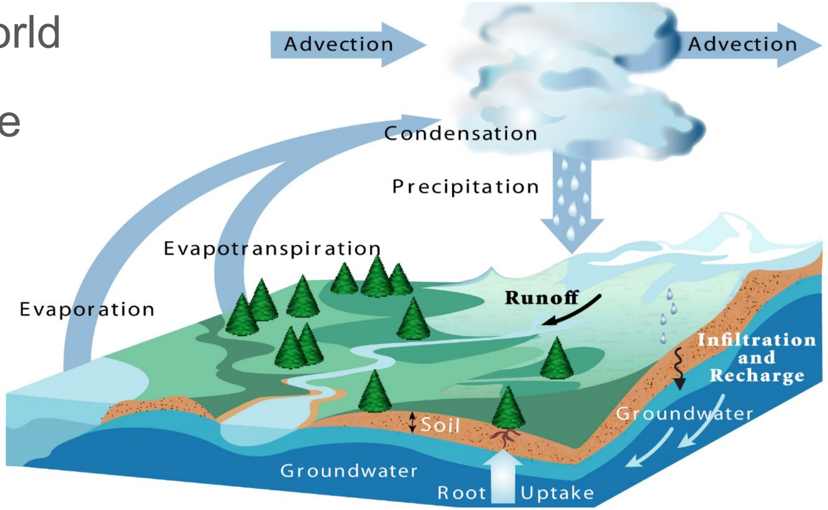
❖ ParFlow is a numerical model that simulates variably saturated groundwater and surface water flow



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

What is ParFlow?

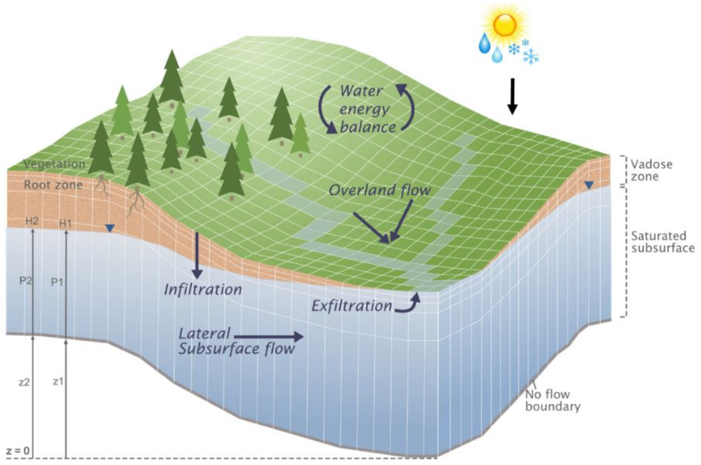
- ❖ Has been coupled to various land surface and atmospheric models
- ❖ Used for water cycle research, forecasting, data assimilation frameworks, hindcasting tools, and climate change projections
- ❖ More than 90 publications describing its development and application to diverse systems around the world
- ❖ A long development history dating back to the 1990s



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

Numerical model

- ❖ Cell centered finite difference (FD) scheme
- ❖ Implicit time integration
- ❖ Newton-Krylov methods for nonlinear problems
  - ➢ GMRES linear solver (Kinsol) with multigrid preconditioning (ParFlow, Hypre)
- ❖ Good parallel efficiency for up to 10^5 processes
- ❖ A large number of numerical kernels, none of which clearly dominate the run time
- ❖ Around 150k lines of C (https://github.com/parflow)



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

Considerations for accelerator utilization

- ❖ Architecture-specific ports are untenable
  - ➢ Single-source application is preferred
- ❖ Separation of concerns: Scientific development vs. accelerator utilization
  - ➢ Scientific code should ideally remain unchanged and be separated from the architecture-specific code
- ❖ No single programming model has emerged as a clear winner for accommodating new accelerator architectures (cf. message passing and MPI)
  - ➢ Betting big on a single programming model or library is risky!

ParFlow: Macro-based abstraction layer

- ❖ Macro-based abstraction layer forms a part of ParFlow embedded Domain-Specific Language (based on C host language)
- ❖ Hardly any changes to the scientific code and loop interface are required
  - ➢ Accelerator utilization is handled separately in ParFlow eDSL
- ❖ Adding support for one or more accelerator programming models or libraries is possible with a relatively small development effort
  - ➢ The whole application is <u>not</u> built "around" a single accelerator programming model or library!
  - ➢ High developer productivity and minimal invasivity!

ParFlow eDSL interface

Allocations & initializations

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

Message passing

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

Accessor macros

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

Loop macros

```
BCStructPatchLoop(i, j, k, fdir, ival, bc_struct,
  ipatch, is,
{
  int ip = SubvectorEltIndex(p_sub, i, j, k);
  double value = bc_patch_values[ival];
  pp[ip + fdir[0] * 1 + fdir[1] * sy_v + fdir[2] * sz_v]
    = value;
});
```

GPU-support for ParFlow eDSL

- The GPU-support is based on NVIDIA CUDA parallel computing platform
- Some of the most recent CUDA-related developments are leveraged such as
  - Unified Memory
  - Host-device lambdas
  - CUDA-based libraries such as CUB and RMM
- CUDA-aware MPI library and GPU-based application-side data packing routines are used for fast GPU-GPU communication

Development steps

1. **CMAKE**: Configuring CMake for optional CUDA usage
2. **GPU affinity**: Selecting the correct GPU device for each process
3. **Memory management**: Recognizing which data is needed by the GPU(s) and modifying the memory allocations
4. **Loops**: Recognizing and parallelizing the correct loops for the GPU(s)
5. **MPI**: Solving MPI-CUDA compatibility problems
6. **Profiling and optimization**: Detecting page-faults, optimizing GPU-GPU data transfers, coalescing memory accesses, optimizing kernels, etc

CMake tasks for the GPU version

- Compiling ParFlow with GPU acceleration is optional and is activated by specifying -DPARFLOW_ACCELERATOR_BACKEND=cuda option to CMake
- CMake finds CUDA installation, sets CUDA paths, compiler arguments, host compiler, etc
- CMake finds CUB and RMM installations and sets library paths
- CMake determines which C source files use CUDA eDSL headers and must be compiled by the CUDA compiler

GPU affinity

- ❖ Each process uses only one GPU
- ❖ When more than one process per node is started, the employed GPU for each process is selected by

```
cudaSetDevice(amps_node_rank % num_devices)
```

  where *amps_node_rank* and *num_devices* are the node-local rank of the process and the node-local number of GPUs, respectively
- ❖ The optimum strategy when running simulations is to start the same number of processes per node as there are GPUs

Memory management — General

- In ParFlow, *malloc*s and *free*s are accessed through the eDSL API
- The allocation and deallocation for a memory region is (almost) always done in the same compilation unit
- Unified Memory allocations/deallocations (and loops) are controlled for each compilation unit separately allowing incremental development and flexibility

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

Memory management — eDSL (CPU memory only)

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

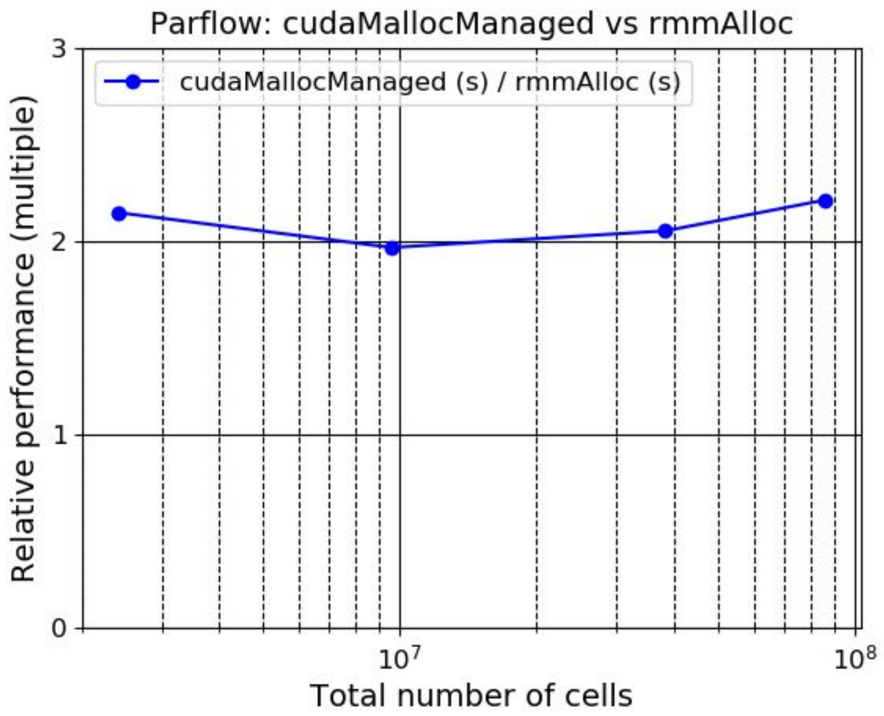
```
#define tfree(ptr) tfree_cuda(ptr)

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

Memory management — RMM pool allocator

❖ In CUDA C++, Unified Memory is allocated by calling *cudaMallocManaged()*

Then why bother adding more dependencies and use rmmAlloc to allocate Unified Memory instead of just directly calling cudaMallocManaged as RMM calls cudaMallocManaged internally anyway?

Memory management — RMM pool allocator



The RMM tradeoff:

- ❖ Slightly increases the average GPU memory usage in ParFlow (around 3-5%)
- ❖ Does not really increase the peak GPU memory usage in ParFlow

Loops — General

- ❖ In ParFlow, the loops over the grid cells are accessed through the eDSL API
- ❖ The eDSL defines general loops which take the loop body (which may be hundreds of lines of code) as a preprocessor macro argument
- ❖ Hundreds of loops with often very different loop body use the same looping macros
- ❖ The GPU implementation relies on modified loop macros which leverage host-device lambdas and general GPU kernels
- ❖ The loop body and required kernel arguments are passed as a lambda function for the general GPU kernels

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_CUDA
/* some code to find grid & block sizes */
PlusEqualsKernel<<<grid, block>>>
  (yp, alpha, ix, iy, iz, nx, ny, nz);
#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;
});
```

JÜLICH Forschungszentrum

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

```
#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__\
    (const int i, const int j, const int k) \
      loop_body;                                       \
                                                       \
  BoxKernelI0<<<grid, block>>>(lambda_body, \
    ix, iy, iz, nx, ny, nz);                           \
}
```

Loops: General GPU kernels

```
template <typename LambdaBody>
__global__ static void BoxKernelI0(LambdaBody loop_body, const int ix, const int iy,
    const int iz, 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) return;

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

    loop_body(i, j, k);
}
```

MPI troubles

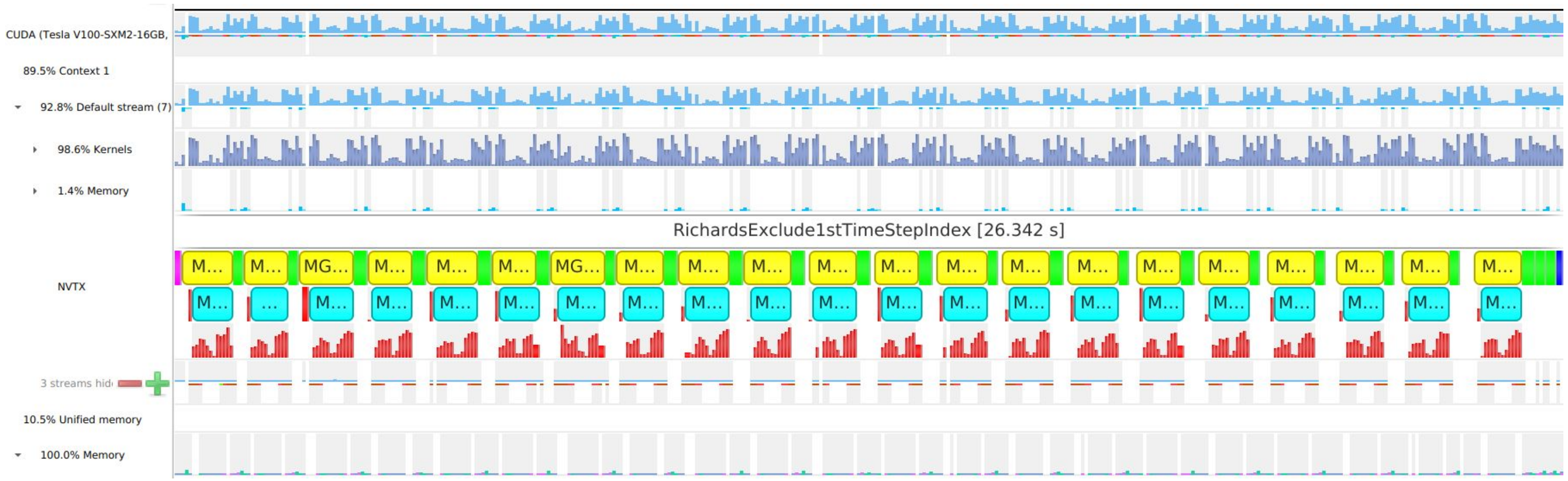
- ❖ Original ParFlow used derived MPI datatypes and relied on MPI library-side data packing and unpacking
- ❖ **Problem**: In the ParFlow GPU version, the pointers that were originally passed to the MPI library pointed to Unified Memory allocations
  - ➢ Frequent segfaults with MVAPICH2-GDR and ParastationMPI (not CUDA-aware)
  - ➢ ParastationMPI-CUDA released in 10/2019 worked, but GPU-GPU data transfers were extremely slow (GPUDirect P2P/RDMA not used)
- ❖ **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 MVAPICH2-GDR and ParastationMPI-CUDA)

Profiling & Optimization

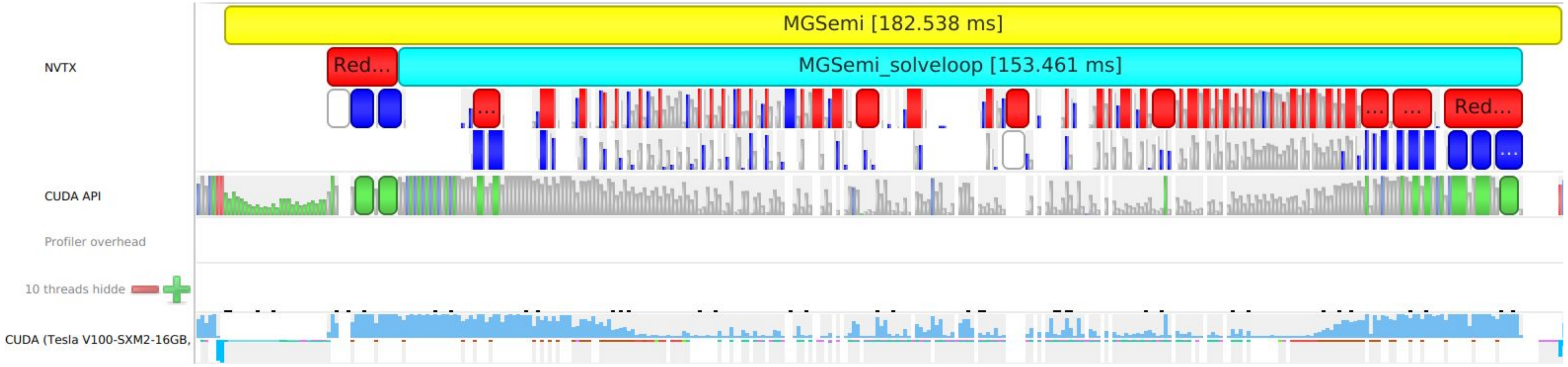
- **Used profiling tools**:
  - Nvidia Visual Profiler
  - Nvidia Nsight Systems + Compute

- **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)

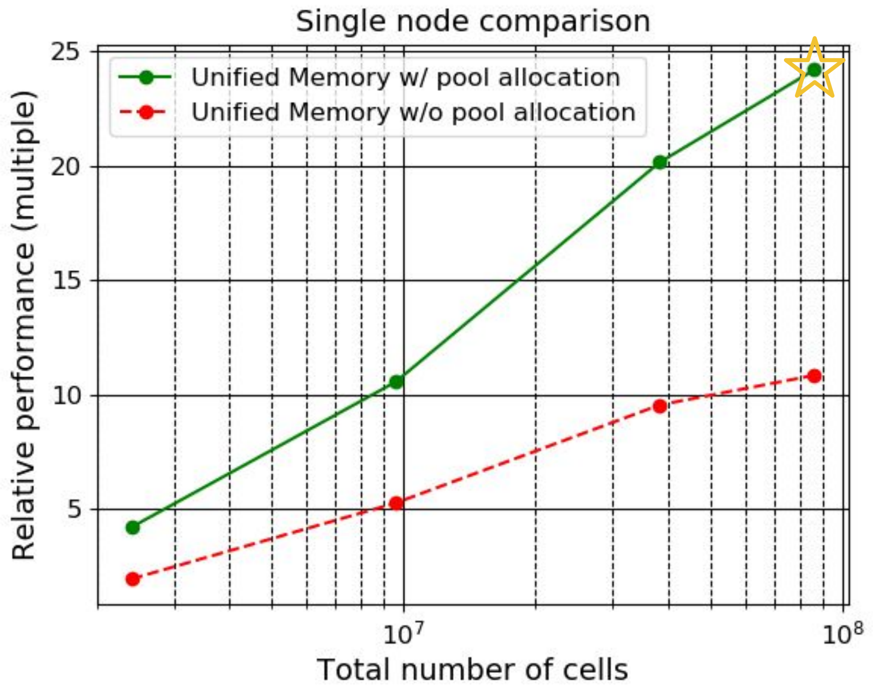
Profiling & Optimization: Nsight Systems

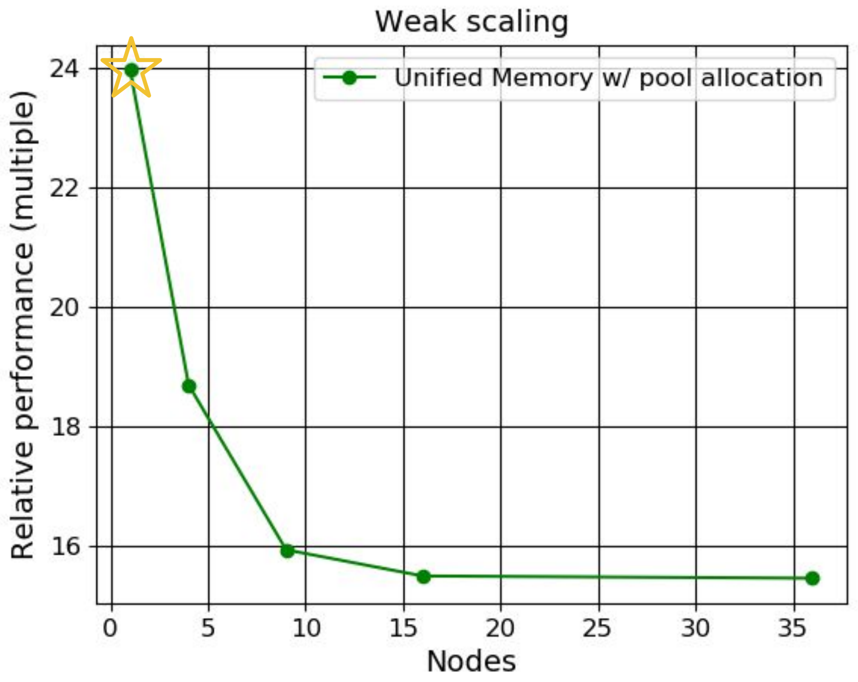


Profiling & Optimization: Nsight Systems



Results (JUWELS supercomputer)





Summary of the utilized approach

- ❖ Requires at least a minimal abstraction layer for accessing the most relevant loops and memory allocations/deallocations
- ❖ The same interface for loops, memory allocations, and data structure access patterns is used regardless of the used architecture
- ❖ All compute kernels and device functions are naturally defined in the same compilation unit such that the compiler can fully optimize the machine code
- ❖ The codebase maintainability is well preserved
- ❖ Significant savings with future developments to support new architectures are possible

Questions & Discussion