ACH Meeting November 2024

Nicola Varini

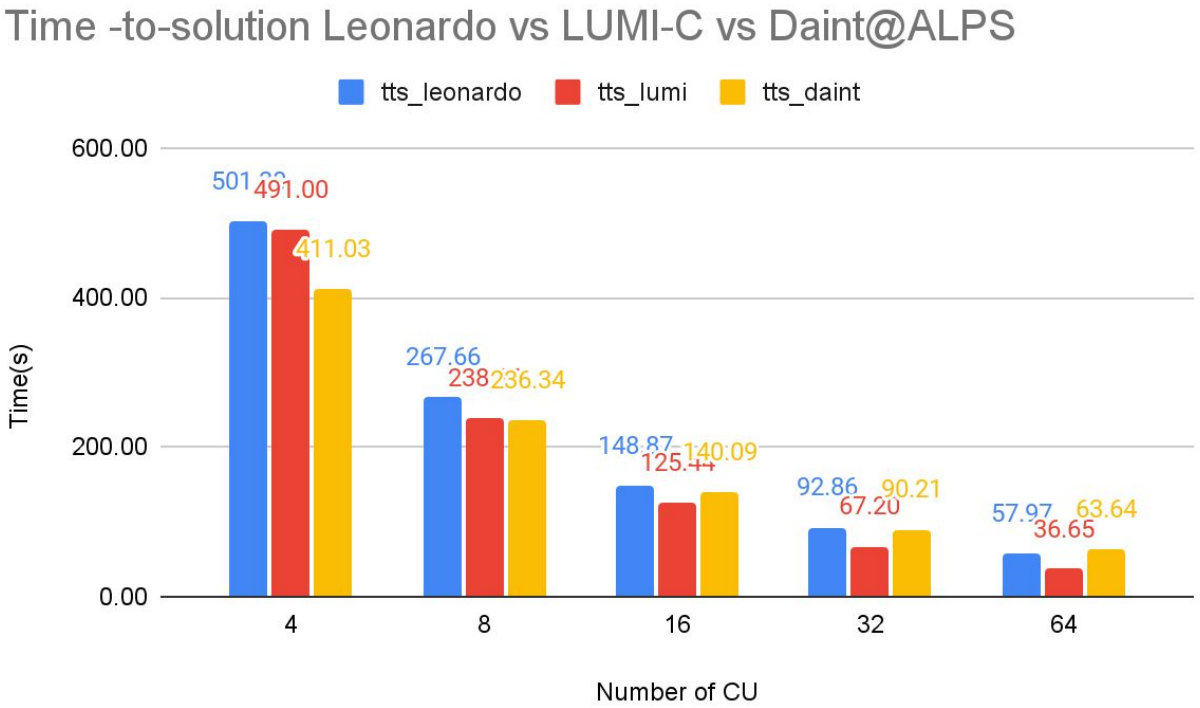
GBS

Nicola Varini

Overview

- Benchmark on Tier-0 systems
- LUMI vs Leonardo vs PizDaint@ALPS
- Take away from GPU porting

Leonardo vs Lumi vs Daint@ALPS - Time-to-solution



- TCV@0.9T
  - Nx = 300
  - Ny = 600
  - Nz = 128
- Benchmarks:
  - **Leonardo**: 4 A100/node
  - **LUMI**: 128 EPYC 7763/node
  - **Daint: 4 GH200/node**
- bc_model_yb='Tar'
  bc_model_yt='pAT'
  bc_model_xr='pAT'
  bc_model_xl='Mag'

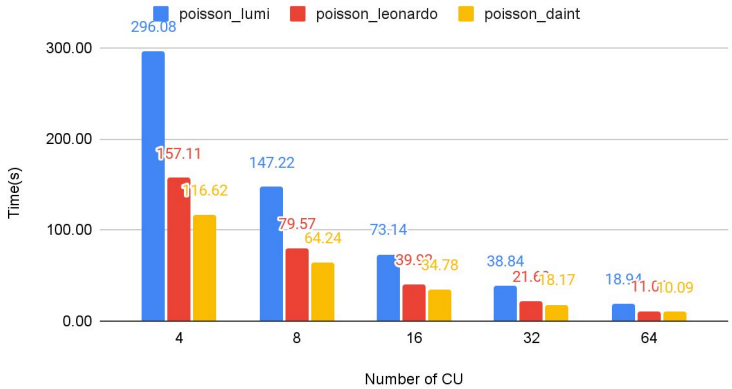
Scaling

- We used strong scaling along Z

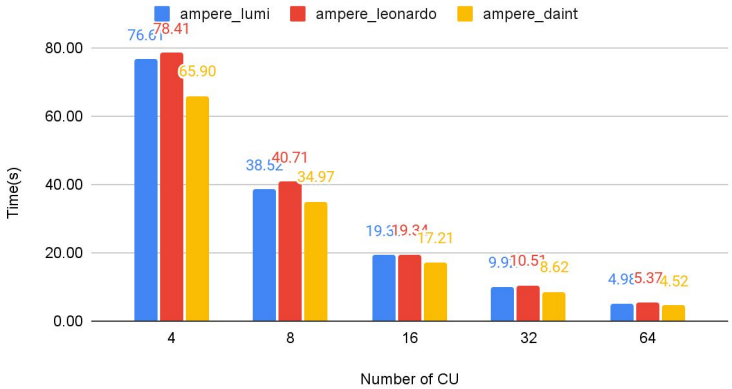
CU	NX(CPU-GPU)	NY(CPU-GPU)	NZ
4	128 1	128 1	4
8	128 1	128 1	8
16	128 1	128 1	16
32	128 1	128 1	32
64	128 1	128 1	64

Solvers

Poisson solver



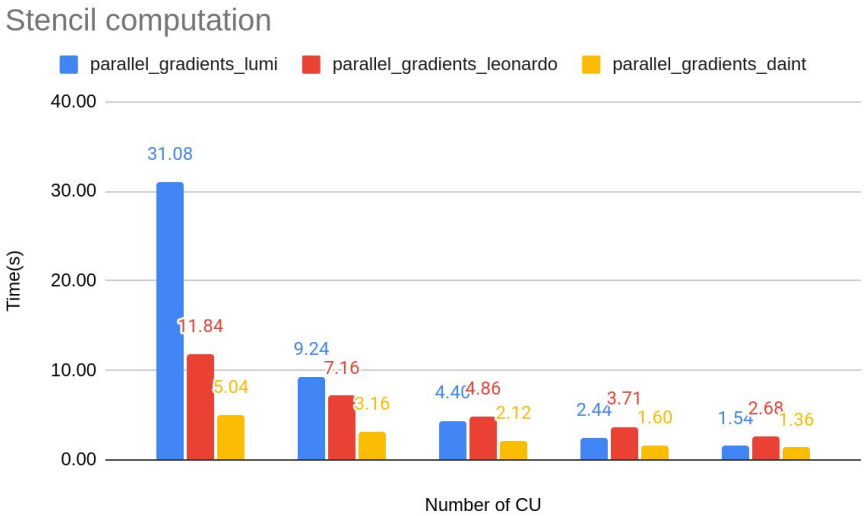
Ampere solver



- Performed by PETSc
  - GMRES solver
  - HYPRE BoomerAMG preconditioner
  - The number of unknown is fixed to the size of the poloidal plane

- The Poisson solver is **twice** faster on Leonardo compared to Lumi
- The Ampere solver performance are comparable

Stencil computation: parallel gradients

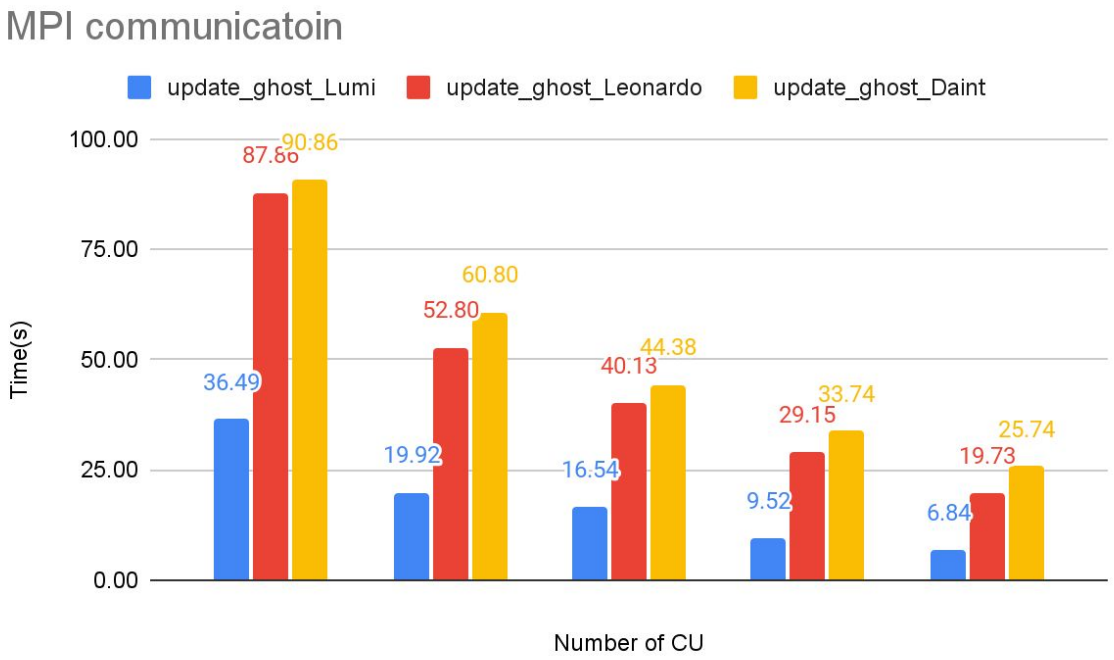


- Parallelization
  - CPU - 128 MPI tasks
  - GPU - 1 CUDA GPU
  - Strong scaling in z

CU	Stencil size per node
4	300x600x32
8	300x600x16
16	300x600x8
32	300x600x4
64	300x600x2

- GPU perform well with a lot of data

MPI Communication



- The MPI communication is far worse on GPU
- We used CUDA aware MPI and GPU Direct
- For CPU there's communication inside a poloidal plane

Shift of bottleneck

(Poisson+Ampere+Parallal_grad)/TTS			
nodes	Lumi	Leonardo	Daint
4	0.82	0.76	0.46
8	0.82	0.74	0.43
16	0.77	0.71	0.39
32	0.76	0.63	0.31
64	0.69	0.53	0.25

- The routines ported on GPU are performing well
  - 2X improvement in Poisson
  - Improvement in stencil - the more data the better
- On GPU, performance degrade more with scaling.
- For bigger system, we should expect better GPU performance

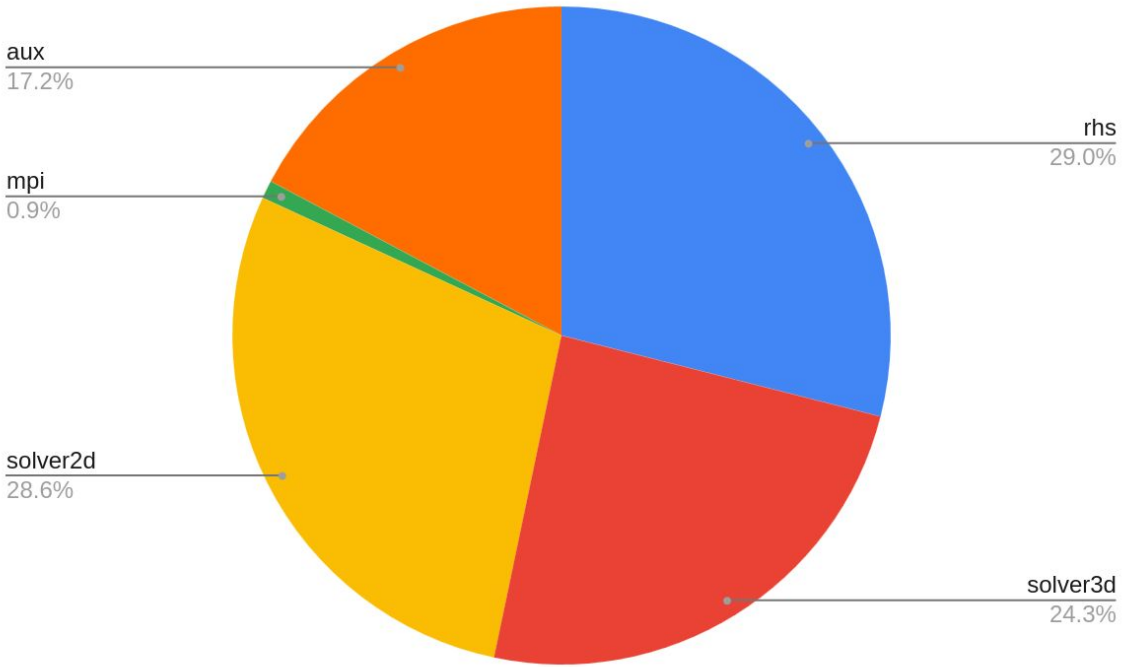
Future development

- The CUDA routines are performing as expected.
- CUDA force the developers to maintain two versions of the code.
- Shift of bottlenecks after the GPU porting
  - The most expensive routines go from 70% on CPU to 30% on GPU

GRILLIX

Nicola Varini and Andreas Stegmeir

GRILLIX



- The solver2d is performed by parallax:
  - Collaboration for GPU porting
- The **solver3d** will be ported on GPU by ACH
- The RHS will be ported to GPU last.

The 3D solver applied to heat flux solver

Reminder:

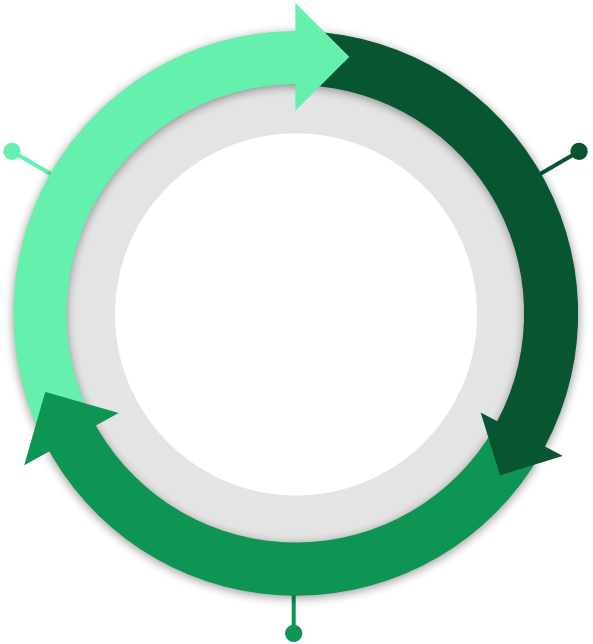
$$\lambda u - \xi \mathbf{P} c \mathbf{Q} u = b$$

In its most basic form, the provided matvec routine does the following:
(see GRILLIX: src/solver_aligned3d/solve_aligned3d_s.f90)

- Send/receive u to/from rank+1/rank-1
- Multiply Q*u blockwise →heat flux q
- Send/receive q to/from rank-1/rank+1
- Multiply P*q blockwise
- Note the different dimensionalities of quantities, since P and Q are generally non-square, representing canonical and staggered mesh)

Strategy for GPU porting

Miniapp injection back into the main codes



- Benchmarks definition and strategy.
- Profiling.

MINIAPPS

- Proof-of-concept for spmv
- Performance analysis

SPMV – Fortran version

```
!$omp parallel do
do i = 1, nrows
        do j = rows(i),rows(i+1)-1
           y_fortran(i) = y_fortran(i) + vals(j)*x(cols(j))
        end do
 end do
!$omp end parallel do
```

- Currently the sparse matrix vector product is performed on CPU with OpenMP
- Baseline version

SPMV – Kokkos version

C++

```
extern "C" void compute_spmv_(csrspmv *& p){
    csrspmv::ViewI& rows = *(p->rows);
    csrspmv::ViewI& cols = *(p->cols);
    csrspmv::ViewD& vals = *(p->vals);
   csrspmv::ViewD& x = *(p->x);
   csrspmv::ViewD& y = *(p->y);
   Kokkos::parallel_for(y.extent(0), KOKKOS_LAMBDA(const size_t idy){
          for(int idx=rows(idy);idx<rows(idy+1);idx++){
          y(idy) = y(idy)+ vals(idx)*x(cols(idx));
          }
  });
}
```

- The C++ routine is called from Fortran
- L2norm(y_fortran-y_kokkos) = 3E-14
- Succesfull offload observed with nsys

Fortran

```
call allocate_kokkos_view(kokkos_view,
      nrows, nnz, rows, cols, vals, x, y_kokkos)
call compute_spmv(kokkos_view)
```

SPMV - CUDA version

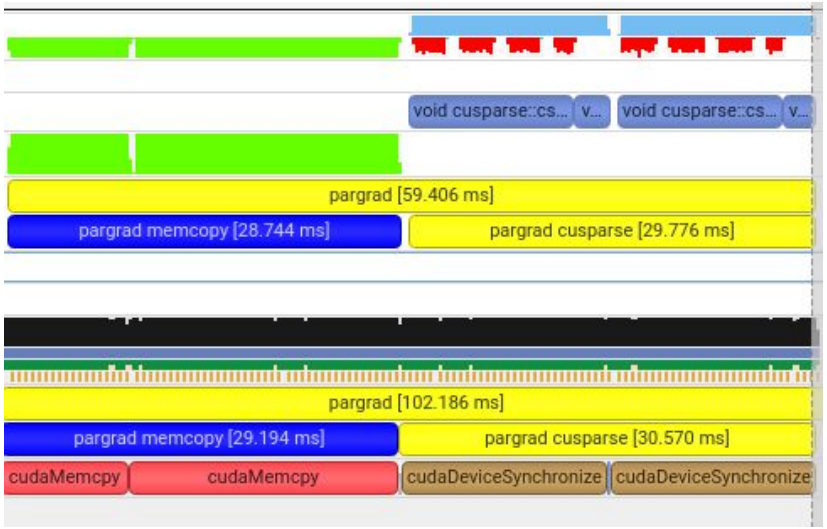
```
extern "C" void allocate_cusparse_struct_(matvec_struct *& mv, const int *numRows,
    const int *numCols, const int *nnz, int *csrRowPtr, int *colInd, double *vals,
    double *x, double *y) {

  size_t bufferSize;

  mv = new matvec_struct();
  mv->alpha = 1.0;
  mv->beta = 0.0;
  CHECK_CUSPARSE(cusparseCreate(&(mv->handle)));
  // Create sparse matrix and dense vector descriptors
  CHECK_CUSPARSE(cusparseCreateCsr(&(mv->matA), *numRows, *numCols, *nnz,
                                   csrRowPtr, colInd, vals,
                                   CUSPARSE_INDEX_32I, CUSPARSE_INDEX_32I,
                                   CUSPARSE_INDEX_BASE_ZERO, CUDA_R_64F));
  CHECK_CUSPARSE(cusparseCreateDnVec(&(mv->vecX), *numCols, x, CUDA_R_64F));
  CHECK_CUSPARSE(cusparseCreateDnVec(&(mv->vecY), *numRows, y, CUDA_R_64F));

  cusparseSpMV_bufferSize(
      mv->handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
      &mv->alpha, mv->matA, mv->vecX, &mv->beta, mv->vecY, CUDA_R_64F,
      CUSPARSE_SPMV_CSR_ALG1, &bufferSize);
  CHECK_CUDA(cudaMalloc(&(mv->dBuffer), bufferSize));
}

extern "C" void spmv_cusparse_(matvec_struct *& mv) {
    cusparseSpMV(
        mv->handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
        &(mv->alpha),mv->matA,mv->vecX,&(mv->beta),mv->vecY, CUDA_R_64F,
        CUSPARSE_SPMV_CSR_ALG1, mv->dBuffer);

    cudaDeviceSynchronize();
}
```

- We can also perform the sparse matrix-vector operation
- L2norm(y_cuda-y_fortran) =2.5E-14
- Successful kernel generation observed with nsys

```
call allocate_cusparse_struct(cuda_ptr, nrows, nrows, nnz, rows, cols, vals, x, y_cuda)
call spmv_cusparse(cuda_ptr)
call spmv_clean(cuda_ptr)
```

Benchmark and profiling

- Nvtx events in nsys
- We can see the cusparse library in the timeline



Next steps

- We would like to leave the door open to different paradigms.
- The SPMV operation in grillix is performed in different routines
  - We need to choose the right level of abstraction
- Benchmarks
  - In-depth comparison between OpenMP threads and CUDA