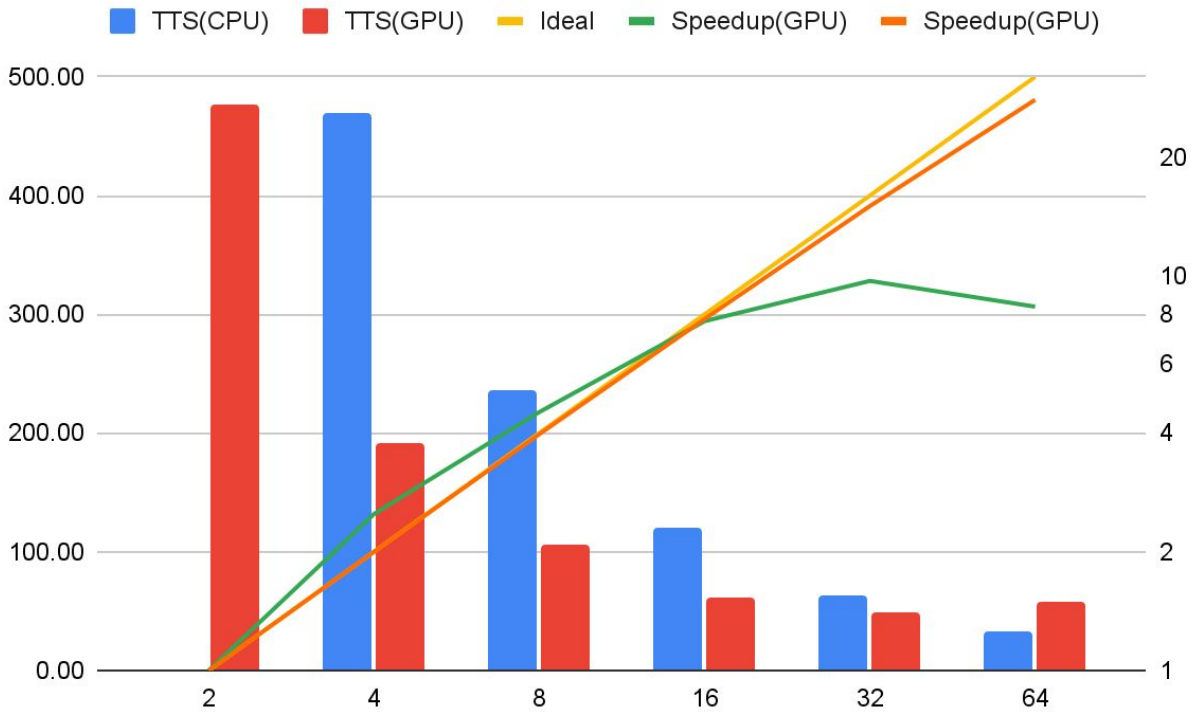
STATUS UPDATE FOR GBS

Nicola Varini

Overview

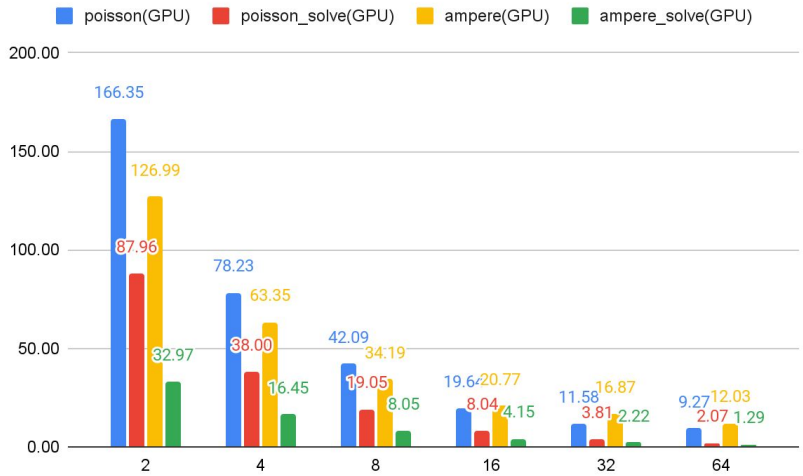
- To compare GPU benchmarks on different generations of NVIDIA GPU
- PizDaint GPU(P100) vs CPU(Haswell 12 cores)
- Leonardo GPU(A100) vs LUMI-C(AMD Epyc 128 cores)
- P100 vs A100 vs GH200
- Take away from GPU porting

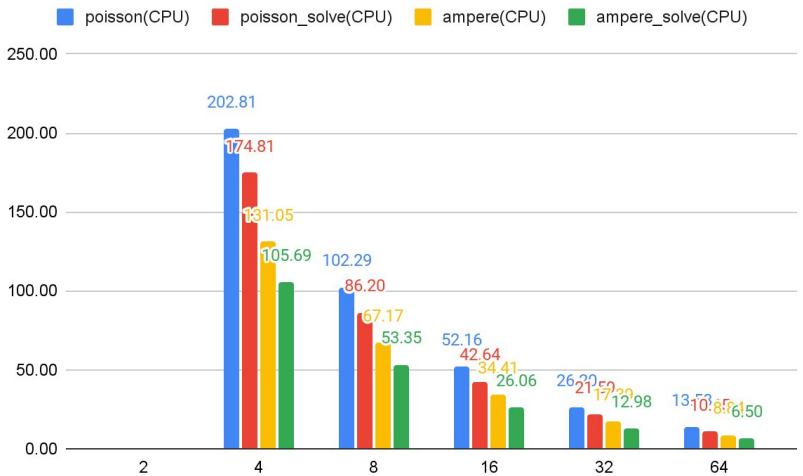
Daint CPU vs GPU – Time-to-solution



- TCV@0.9T
- Strong scaling on Z:
  - CPU 4x3 MPI/node
  - GPU 1 MPI/node
- bc_model_yb='Tar'
  bc_model_yt='pAT'
  bc_model_xr='pAT'
  bc_model_xl='Mag'
- The GPU speedup degrade after 32 nodes
- The GPU outperform the CPU until 32 nodes

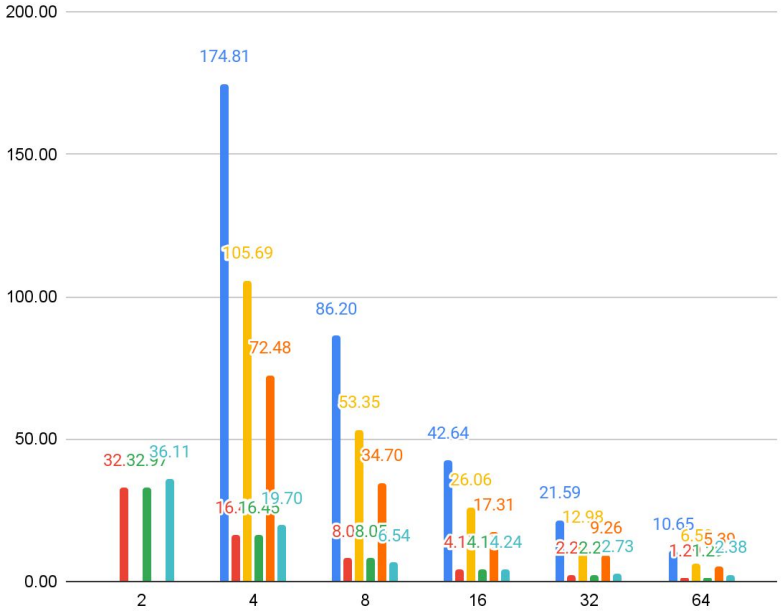
Solvers





- Poisson
  - pre process
  - Build the matrix
  - Solve(poisson solve)
  - Post process
- On GPU there's a much bigger gap between poisson_solve and poisson, same applies for ampere

GPU vs CPU routines – poisson solver, ampere solver, parallel gradients



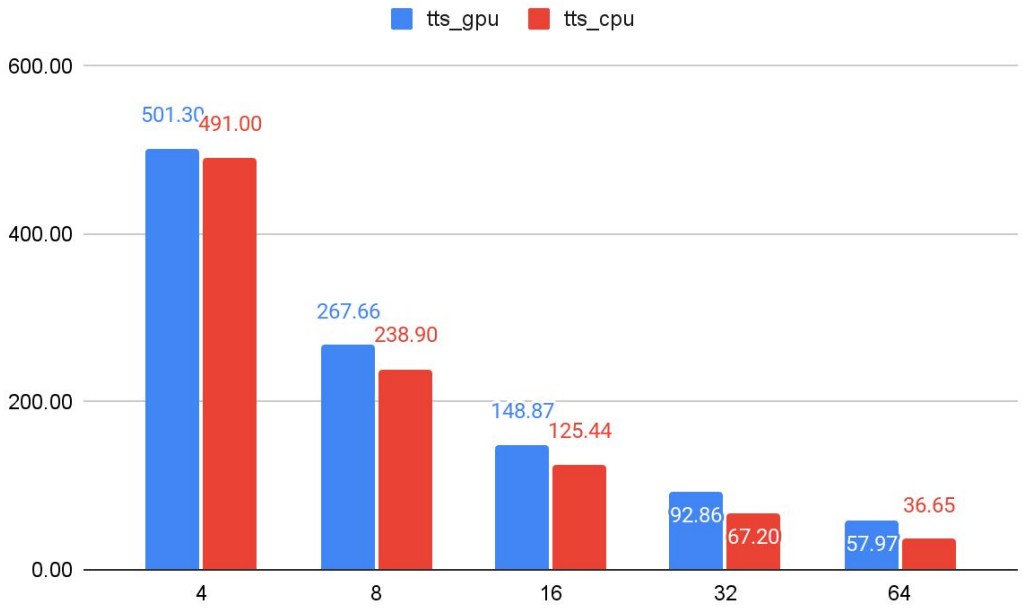
- The routines ported in CUDA are faster but the MPI is slower

CPU(s)/GPU(s) Speedup				
nodes	poisson solve speedup	ampere solve speedup	paralllel gradient speedup	update_ghost speedup
4	4.6	6.4	3.7	0.2
8	4.5	6.6	5.3	0.3
16	5.3	6.3	4.1	0.3
32	5.7	5.8	3.4	0.2
64	5.1	5.0	2.3	0.1

- The CPU most expensive routines perform well on GPU.
- There is shift of bottleneck at scale.

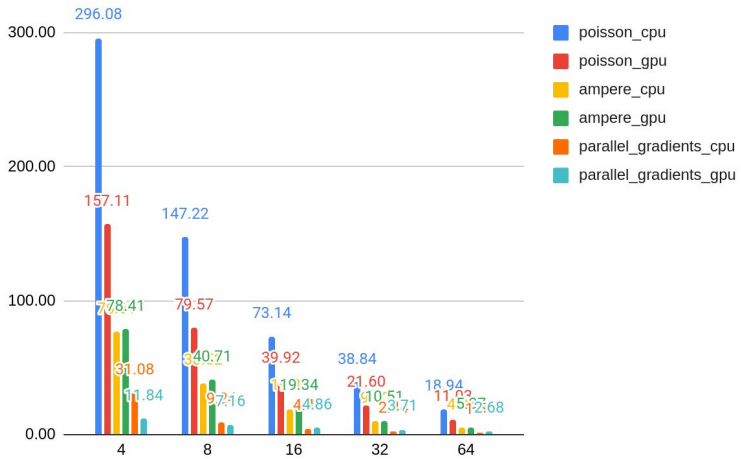
(Poisson_solve+Ampere_solve+Parallel_grad)/TTS		
nodes	GPU	CPU
4	0.39	0.75
8	0.32	0.74
16	0.26	0.71
32	0.18	0.70
64	0.10	0.67

Leonardo vs LUMI-C



- Computational unit:
  - 1 A100 Leonardo
  - vs 1 LUMI-C node(128 cores)
- Good scaling until 64 cu
- LUMI-C is faster at scale, but performance are comparable

GPU vs CPU routines - poisson solver, ampere solver, parallel gradients



Speedup T_cpu/T_gpu				
Nodes	Poisson	Ampere	Parallel gradients	update_ghost
4	2.07	0.98	2.63	0.42
8	2.03	0.95	1.29	0.38
16	2.12	1.00	0.91	0.41
32	2.05	0.94	0.66	0.33
64	2.04	0.93	0.57	0.35

(Poisson+Ampere+Parallel_gradients)/TTS		
Nodes	GPU	CPU
4	0.49	0.82
8	0.48	0.82
16	0.43	0.77
32	0.39	0.76
64	0.33	0.69

Looking at the future – 4GPU comparison

Machine	TTS	Poisson_solve	Ampere_solve	Parallel grad
Daint	191.14	38.0	16.45	19.70
Leonardo	61.89	21.28	10.87	1.18
GH200	45.38	16.95	7.62	0.50

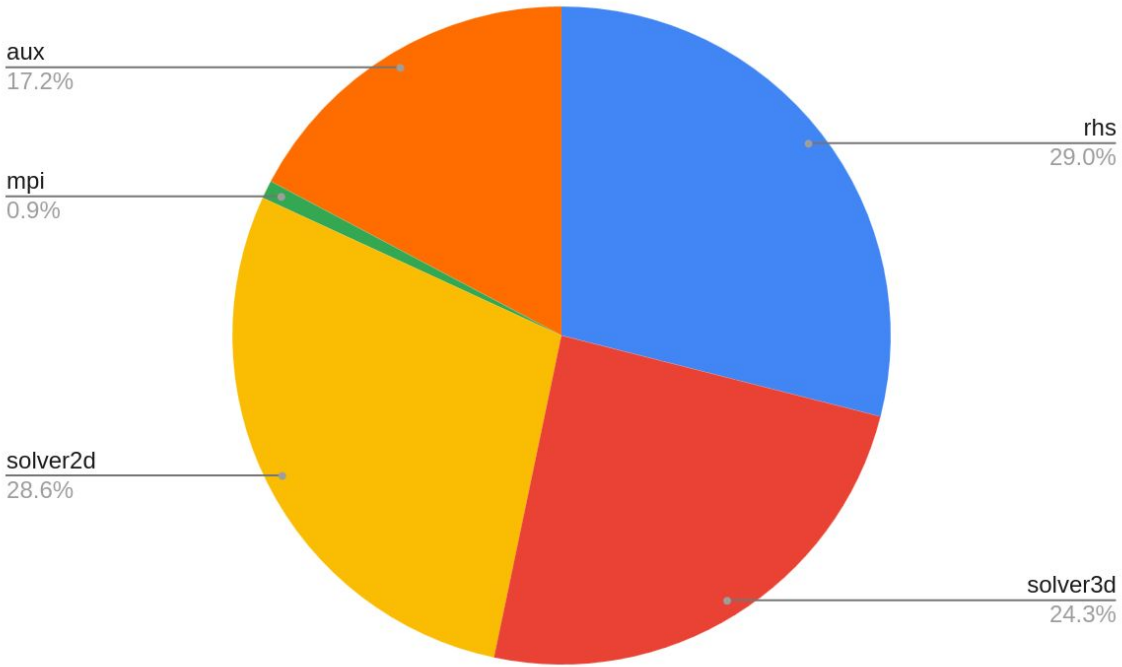
Future development

- 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
- How to improve the remaining part?
  - It will require to expose more asynchronicity.
  - We are thinking about kokkos for performance portability.

STATUS UPDATE FOR GRILLIX

Nicola Varini and Andreas Stegmeir

ITER - Dirichlet BT timers breakdown



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

Mathematical background

Parallel heat flux problem

$$\partial_t u = \nabla \cdot \left(\mathbf{b} \chi \nabla_{\parallel} u \right)$$

→ Following problem has to be solved for u

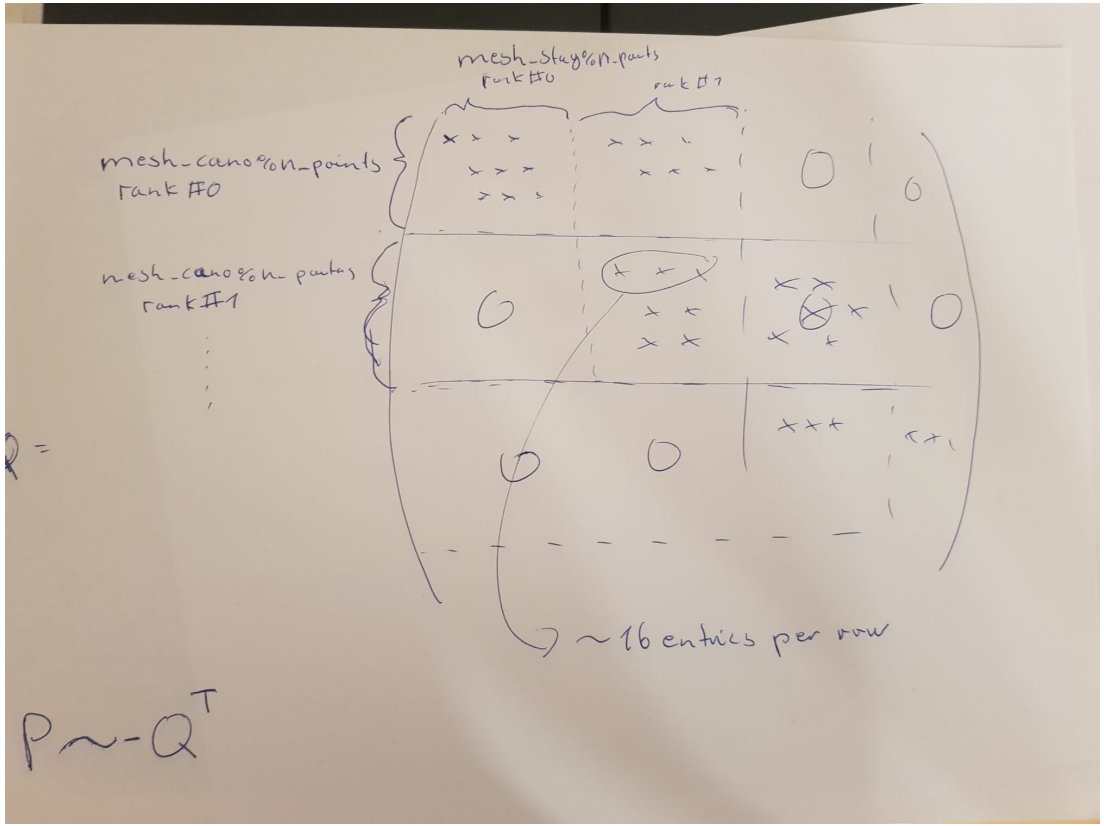
$$\lambda u - \xi \nabla \cdot \left(\mathbf{b} c \nabla_{\parallel} u \right) = b$$

with coefficients lambda, xi, c and known right hand side b

→ On discrete level expressed via parallel gradient matrix **Q** and parallel divergence Matrix **P**

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

Matrix structure



Parallel gardient matrix Q:

- Contains interpolation coefficients obtained from field line tracing
- generally non-square
- Connects only rank and rank+1
- Sparse roughly 32 entries per row

Parallel divergenc matrix P

- Rougly negative transpose of Q (Structurally exact)

Parallel diffusion matrix P*Q

- square matrix
- never explicitly constructed

The 3D solver

Parallel Iterative methods library (PIM), see https://gitlab.mpcdf.mpg.de/phoenix/pim

- Krylow based methods (CG, GMRES, BCGSTAB, …)
- MPI aware
- Matrix vector multiplication and preconditioner provided as external functions with prescribed interface
  →Full matrix does not explicitly need to be constructed.
- Generic wrapper module available in PARALLAX:

```
module subroutine create_PIM(self, comm, ndim_loc, resmax,  matvec, precondl, precondr, maxiter, nrestart,  dbgout)
      class(solver3d_PIM_t), intent(inout) :: self
      …
      procedure(matvec_interface) :: matvec
      procedure(precond_interface), optional :: precondl
      procedure(precond_interface), optional :: precondr
       …
```

→ Most work requires providing the matvec routine with the following interface

```
abstract interface
      subroutine matvec_interface(u, v, ipar)
         !! Interface for matrix vector multiplication v = A*u
         import FP
         real(FP), dimension(*) :: u
         real(FP), dimension(*) :: v
         integer :: ipar(*)
      end subroutine
   end interface
```

The 3D solver applied to heat flux solver

Reminder:

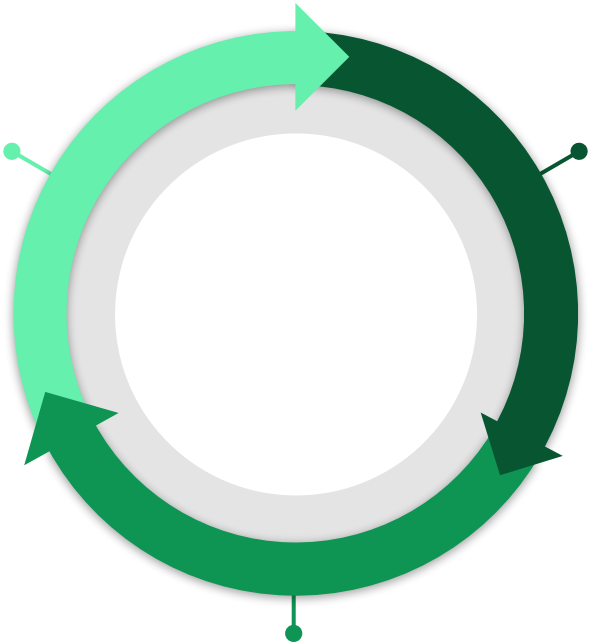
$$\lambda u - \xi \mathbf{P} c \mathbf{Q} u = b$$

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)
```

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
  - This summer