Geosci. Model Dev. Discuss., 6, C1592–C1611, 2013
www.geosci-model-dev-discuss.net/6/C1592/2013/

Geoscientific Model Development
Discussions
Open Access

Interactive comment on "CUDA-C implementation of the ADER-DG method for linear hyperbolic PDEs" *by* C. E. Castro et al.

C. E. Castro et al.

ccastro@uta.cl

Received and published: 6 October 2013

The authors are grateful for the hints and constructive comments of both reviewers and the editor. We take these recommendations very seriously and will respond in the following general ways:

- A more concise description of the relevance of both methods for geoscientific applications will be added to the introduction;
- A clearer discussion of the performance improvements of the GPU implementation over the original CPU code will be given;
- A more detailed description of the GPU implementation will be given (see below);

- At the same time, we will better describe the research question behind our manuscript, which is concerned with the performance gain of GPU implementations provided maintaining the accuracy and minimizing the implementation effort.

We also apologize for not providing the two kernels used to code the method as it was the original plan and stated in the manuscript. Here we present a pseudo-code in order to compare the CPU-GPU main differences on the algorithm and the two kernels used to implement the ADER-DG GPU version.

Pseudo-Code and kernels

In order to demonstrate the changes to the code, necessary to include the GPU computation, we will integrate pseudo-code examples like the following into our manuscript. In the following example, we compare the main program parts of the original Fortran implementation with the GPU-enabled code:

Original Code

```
! set up initial conditions
   ...
! main time step loop
   DO WHILE(Time .LT. FinalTime &
     .AND. iTimeStep .LT. MaxNumTimeStep)

! time integration of DOF's
      CALL CauchyKowalewksi( parameters )

! compute volume integral
      CALL VolumeIntegral( parameters )

! Compute boundary integral (flux)
      CALL BoundaryIntegral( parameters )

! update DOFs
      CALL UpdateDof( parameters )

! some diagnostics and plotting
   ...
   END DO ! end of main while loop - time stepping loop
```

GPU enabled Code

```
! set up initial conditions
   ...
! call CUDA driver (contains time step loop)
   CALL AderDG_InCuda( parameters )
```

We sketch the Fortran subroutine `AderDG_InCuda` briefly and dive into the CUDA-C implementation further down:


```
SUBROUTINE AderDG_InCuda( parameters )

! initialization of static fields
   ...

! call CUDA-C kernel driver
   CALL ader_dg( parameters )

! deallocate static fields
   ...

END SUBROUTINE AderDG_InCuda
```

Now, we look into the CUDA-C implementation of the kernel driver and see essentially the same steps as in the Fortran time-stepping loop above. However, note that we have simplified the calling sequence slightly and that the two main kernels `TimeIntegrateDof` (representing the time integration and volume integrals steps above) and `FluxComputation` (representing the boundary integrals and update steps above) are now equipped with CUDA mapping primitives for the blocks and block dimensions.

```
extern "C" void ader_dg_( input_parameters ) {

/* allocating memory on GPU device               */
/* do this for all necessary arrays              */

   size = ( size_of_array )
   cudaMalloc( &array_cuda, size );
   cudaMemcpy( array_cuda, array_cpu, size, cudaMemcpyHostToDevice )

/* configure the block and grid sizes on GPU     */
/* Block configuration:                          */
/* number of components x number of DOFs x 1     */
```

```
    dim3 dimBlock(nComp,nDof,1);

/* Dimension configuration such that             */
/*   x_size x y_size < number of Elements        */
/* remember the unstructured grid!               */

   x_size = nElem/MAX_GRID_X_SIZE;
   y_size = MAX_GRID_X_SIZE;
   dim3 dimGrid(x_size,y_size,1);

/* time step loop                                */

   while (TIME < FinalTime & STEP < MaxNumTimeStep) {

/* time integration of DOFs                      */

      TimeIntegrateDof<<<dimGrid,dimBlock>>>( parameters );
      cudaThreadSynchronize();

/* flux computation                              */

      FluxComputation<<<dimGrid,dimBlock>>>( parameters );
      cudaThreadSynchronize();

/* advance time step                             */
      TIME += DT;
      STEP += 1;

/* output and diagnostics on CPU:                */
/* use cudaMemcpy to get data from GPU into CPU  */

   } /* end of while - time step loop */

/* deallocate GPU memory                         */

   cudaFree( array_cuda );
```



```
   return();
} /* end of ader_dg_ */
```

This CUDA-C kernel code performs time integration and volume integral of the presented ADER-DG GPU implementation.

```
__global__ void TimeIntegrateDof(Tensor2d Dof,
                                 Tensor3d A, Tensor3d B,
                                 Tensor3d Kxi, Tensor3d Ket,
                                 Tensor2d Mkl, Tensor3d J,
                                 Tensor2d TIDof,
                                 Tensor2d PickP,
                                 Tensor2d PickPointInfo,
                                 real_t Time, real_t FinalTime, real_t Dt,
                                 int_t Model, int_t nElem, int_t nComp, int_t Orde
                                 int_t nDof_k, int_t nDof_l, int_t nDof_m ) {

//    GPU implementation of ADER-DG method for linear hyperbolic pde. Time integra
//    See Geosci. Model Dev. Discuss., 6, 3743-3786, 2013
//    doi:10.5194/gmdd-6-3743-2013
//    Copyright (C) 2012  Crist\'obal E. Castro
//    Developed in KlimaCampus in the Numerical Methods in Geoscieces group
//    http://www.klimacampus.de/numericalmethods.html
//    contact email: ccastro@uc.cl

//    This program is free software: you can redistribute it and/or modify
//    it under the terms of the GNU General Public License as published by
//    the Free Software Foundation, either version 3 of the License, or
//    (at your option) any later version.

//    This program is distributed in the hope that it will be useful,
//    but WITHOUT ANY WARRANTY; without even the implied warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//    GNU General Public License for more details.
```

```
//    You should have received a copy of the GNU General Public License
//    along with this program.  If not, see <http://www.gnu.org/licenses/>.


    /* Each thread block compute the CK for one element
     * therefore we assign */
    int_t iElem   = blockIdx.y*gridDim.x + blockIdx.x;

    /* Use thread array to organize the work inside each element */
    int_t ip  = threadIdx.x;
    int_t il  = threadIdx.y;

    /* If iElem >= nElem this block does not map to an element */
    if (iElem >= nElem) {
        return;
    }

    /* Check PickPoint */
    /* If this element need to be stored, use index iPickP to identify
       where in the PickP data structure */
    int_t StorePickP = -1;
    for (int_t iPickP = 0; iPickP < PickPointInfo.n1; iPickP++ ) {
        int_t  PickPElem = float2int(PickPointInfo.entry[iPickP]);

        if (PickPElem == iElem+1) {
            StorePickP = iPickP;
        }
    }

    __shared__ real_t Astar[MAX_nCOMP][MAX_nCOMP][MAX_nDOF];
    __shared__ real_t Bstar[MAX_nCOMP][MAX_nCOMP][MAX_nDOF];

    /* Build star Jacobian matrices    */
    /* Parameter J is the inverse of J */
    int_t entryAIdx = nComp*nComp*nDof_m*iElem;
```



```
    int_t entryJIdx = 4*iElem;
    if (il < nDof_m) {
        for (int_t iq = 0; iq<nComp; iq++) {
            Astar[ip][iq][il] = A.entry[entryAIdx+nDof_m*nComp*iq+nDof_m*ip+il] *
                                J.entry[entryJIdx+0] +
                                B.entry[entryAIdx+nDof_m*nComp*iq+nDof_m*ip+il] *
                                J.entry[entryJIdx+2];
            Bstar[ip][iq][il] = A.entry[entryAIdx+nDof_m*nComp*iq+nDof_m*ip+il] *
                                J.entry[entryJIdx+1] +
                                B.entry[entryAIdx+nDof_m*nComp*iq+nDof_m*ip+il] *
                                J.entry[entryJIdx+3];
            /* This modification is for LeVeque test only*/
            if (Model == 1) {
                Astar[ip][iq][il] = Astar[ip][iq][il] * cos(PI*(Time+(Dt/2.0))/Fin
                Bstar[ip][iq][il] = Bstar[ip][iq][il] * cos(PI*(Time+(Dt/2.0))/Fin
            }

        }
    }

    __shared__ real_t TimeDerDof[MAX_nDOF][MAX_nCOMP];
    __shared__ real_t AuxTensor[MAX_nCOMP][MAX_nDOF][MAX_nDOF];
    __shared__ real_t AuxDof[MAX_nDOF][MAX_nCOMP];
    __shared__ real_t AuxTIDof[MAX_nDOF][MAX_nCOMP];

    int_t iOrder = 0;
    int_t ik;
    real_t rOrder = 1.0;
    real_t TaylorCoeff = Dt/rOrder;


    TimeDerDof[il][ip] = Dof.entry[il+ip*nDof_l+iElem*(nDof_l*nComp)];
    AuxTIDof[il][ip]   = TimeDerDof[il][ip]*TaylorCoeff;

    /* Store Time derivative dof order 0 for PickPoints */
    if (StorePickP > -1) {
```

```
    PickP.entry[il+ip*nDof_l+iOrder*(nComp*nDof_l)+StorePickP*(Order*nDof_l*nC
}

__syncthreads();


/* Time derivative loop */
for (iOrder=1; iOrder<Order; iOrder++) {

    /* Compute next Taylor coeff */
    TaylorCoeff = TaylorCoeff*Dt/(rOrder+1.0);
    rOrder += 1.0;

    /* Astar_pqm * Dof_ql -> plm */
    for (int_t im = 0; im<nDof_m; im++) {
        AuxTensor[ip][il][im] = 0.0;
        for (int_t iq = 0; iq<nComp; iq++) {
            AuxTensor[ip][il][im] += Astar[ip][iq][im] * TimeDerDof[il][iq];
        }
    }
    __syncthreads();

    /* AuxTensor_plm * K_klm -> kp */
    ik = il;
    AuxDof[ik][ip] = 0.0;
    for (int_t im = 0; im<nDof_m; im++) {
        for (int_t il = 0; il<nDof_l; il++) {
            /* Here I am using only one stiff matrix K_klm derived in k
               Transposing the indices I can produce K_klm derived in l
                                                  and K_klm derived in m    */
            /* Model 1 is linear advection equation */
            if (Model == 1) {
                if (ik < nDof_m) {
                    AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                    ( Kxi.entry[il+ik*nDof_k+im*nDof_k*nDof_l]
                                     +Kxi.entry[im+il*nDof_k+ik*nDof_k*nDof_l]
```



```
                } else {  /* In this case the stiff matrix derived in m is zer
                    AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                    ( Kxi.entry[il+ik*nDof_k+im*nDof_k*nDof_l]
                }
            }
            /* Model 4 is elastic wave equation */
            if (Model == 4) {
                AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                ( Kxi.entry[il+ik*nDof_k+im*nDof_k*nDof_l] );
            }

        }
    }
    /* recover index */
    il = ik;

    /* Bstar_pqm * Dof_ql -> plm */
    for (int_t im = 0; im<nDof_m; im++) {
        AuxTensor[ip][il][im] = 0.0;
        for (int_t iq = 0; iq<nComp; iq++) {
            AuxTensor[ip][il][im] += Bstar[ip][iq][im] * TimeDerDof[il][iq];
        }
    }
    __syncthreads();

    /* Replace the time derivative dof storing the xi part previously computed
    TimeDerDof[il][ip] = AuxDof[il][ip];

    /* AuxTensor_plm * K_klm -> kp */
    ik = il;
    AuxDof[ik][ip] = 0.0;
    for (int_t im = 0; im<nDof_m; im++) {
        for (int_t il = 0; il<nDof_l; il++) {
            /* Here I am using only one stiff matrix K_klm derived in k
               Transposing the indices I can produce K_klm derived in l
                                                  and K_klm derived in m    */
```

```
            /* Model 1 is linear advection equation */
            if (Model == 1) {
                if (ik < nDof_m) {
                    AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                    ( Ket.entry[il+ik*nDof_k+im*nDof_k*nDof_l]
                                     +Ket.entry[im+il*nDof_k+ik*nDof_k*nDof_l]
                } else {  /* In this case the stiff matrix derived in m is zer
                    AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                    ( Ket.entry[il+ik*nDof_k+im*nDof_k*nDof_l]
                }
            }
            /* Model 4 is seismic wave equation */
            if (Model == 4) {
                AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                ( Ket.entry[il+ik*nDof_k+im*nDof_k*nDof_l] );
            }
        }
    }
    __syncthreads();

    /* recover index */
    il = ik;

    /* Add the eta part */
    TimeDerDof[il][ip] += AuxDof[il][ip];

    /* Divide by the mass matrix */
    TimeDerDof[il][ip] = - TimeDerDof[il][ip]/Mkl.entry[il*nDof_k+il];

    /* Store Time derivative dof order iOrder for PickPoints */
    if (StorePickP > -1) {
        PickP.entry[il+ip*nDof_l+iOrder*(nComp*nDof_l)+StorePickP*(Order*nDof_
    }

    /* Store the contribution to the time integrated dof */
    AuxTIDof[il][ip] += TimeDerDof[il][ip]*TaylorCoeff;
```



```
}

__syncthreads();

/* Save the time integrate dof in global memory */
TIDof.entry[il+ip*nDof_l+iElem*(nDof_l*nComp)] = AuxTIDof[il][ip];

/* Start the volume integral */

    /* Astar_pqm * TIDof_ql -> plm */
    for (int_t im = 0; im<nDof_m; im++) {
        AuxTensor[ip][il][im] = 0.0;
        for (int_t iq = 0; iq<nComp; iq++) {
            AuxTensor[ip][il][im] += Astar[ip][iq][im] * AuxTIDof[il][iq];
        }
    }

    __syncthreads();

    /* AuxTensor_plm * K_klm -> kp */
    ik = il;
    AuxDof[ik][ip] = 0.0;
    for (int_t im = 0; im<nDof_m; im++) {
        for (int_t il = 0; il<nDof_l; il++) {
            /* Here I am using the stored stiff matrix K_klm derived in k */
            /* Model 1 is linear advection equation */
            if (Model == 1) {
                AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                  Kxi.entry[ik+il*nDof_k+im*nDof_k*nDof_l];
            }
            /* Model 4 is seismic wave equation */
            if (Model == 4) {
                if (ik < nDof_m) {
                    AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                    ( Kxi.entry[ik+il*nDof_k+im*nDof_k*nDof_l]
```

```
                                        +Kxi.entry[im+il*nDof_k+ik*nDof_k*nDof_l]
                    } else {  /* In this case the stiff matrix derived in m is zer
                        AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                        ( Kxi.entry[ik+il*nDof_k+im*nDof_k*nDof_l]
                    }
                }
            }
        }

        /* recover index */
        il = ik;

        /* Add xi coordinate volume integral */
        Dof.entry[il+ip*nDof_l+iElem*(nDof_l*nComp)] += AuxDof[il][ip]/Mkl.entry[i

        /* Bstar_pqm * TIDof_ql -> plm */
        for (int_t im = 0; im<nDof_m; im++) {
            AuxTensor[ip][il][im] = 0.0;
            for (int_t iq = 0; iq<nComp; iq++) {
                AuxTensor[ip][il][im] += Bstar[ip][iq][im] * AuxTIDof[il][iq];
            }
        }

        __syncthreads();

        /* AuxTensor_plm * K_klm -> kp */
        ik = il;
        AuxDof[ik][ip] = 0.0;
        for (int_t im = 0; im<nDof_m; im++) {
            for (int_t il = 0; il<nDof_l; il++) {
                /* Here I am using the stored stiff matrix K_klm derived in k */
                /* Model 1 is linear advection equation */
                if (Model == 1) {
                    AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                      Ket.entry[ik+il*nDof_k+im*nDof_k*nDof_l];
                }
```



```
                /* Model 4 is seismic wave equation */
                if (Model == 4) {
                    if (ik < nDof_m) {
                        AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                        ( Ket.entry[ik+il*nDof_k+im*nDof_k*nDof_l]
                                         +Ket.entry[im+il*nDof_k+ik*nDof_k*nDof_l]
                    } else {  /* In this case the stiff matrix derived in m is zer
                        AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                        ( Ket.entry[ik+il*nDof_k+im*nDof_k*nDof_l]
                    }
                }
            }
        }
        __syncthreads();

        /* recover index */
        il = ik;

        /* Add eta coordinate volume integral */
        Dof.entry[il+ip*nDof_l+iElem*(nDof_l*nComp)] += AuxDof[il][ip]/Mkl.entry[i

    /* End the volume integral */

    return;
}
```

This CUDA-C kernel code performs boundary integrals (fluxes) of the presented ADER-DG GPU implementation.

```
__global__ void FluxComputation(Tensor2d Dof,
                                Tensor2d TIDof,
                                Tensor3d A, Tensor3d B,
                                Tensor2d EdgeInfo,
                                Tensor2d FluxInt_i,
                                Tensor3d FluxInt_j,
```

```
                                Tensor2d Mkl, Tensor3d J,
                                real_t Time, real_t FinalTime, real_t Dt,
                                int_t Model, int_t nElem, int_t nComp, int_t Order
                                int_t nDof_k, int_t nDof_l, int_t nDof_m ) {


//    GPU implementation of ADER-DG method for linear hyperbolic pde. Flux computa
//    See Geosci. Model Dev. Discuss., 6, 3743-3786, 2013
//    doi:10.5194/gmdd-6-3743-2013
//    Copyright (C) 2012  Crist\'obal E. Castro
//    Developed in KlimaCampus in the Numerical Methods in Geoscieces group
//    http://www.klimacampus.de/numericalmethods.html
//    contact email: ccastro@uc.cl

//    This program is free software: you can redistribute it and/or modify
//    it under the terms of the GNU General Public License as published by
//    the Free Software Foundation, either version 3 of the License, or
//    (at your option) any later version.

//    This program is distributed in the hope that it will be useful,
//    but WITHOUT ANY WARRANTY; without even the implied warranty of
//    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//    GNU General Public License for more details.

//    You should have received a copy of the GNU General Public License
//    along with this program.  If not, see <http://www.gnu.org/licenses/>.


    /* For debugging */
    int_t PlotArrays = nElem;


    /* Each thread block compute the numerical flux for one element
     * therefore we assign */
    int_t iElem   = blockIdx.y*gridDim.x + blockIdx.x;
```



```
    /* Use thread array to organize the work inside each element */
    int_t ip  = threadIdx.x;
    int_t il  = threadIdx.y;

    int_t nSide = FluxInt_i.n2;

    /* If iElem >= nElem this block does not map to an element */
    if (iElem >= nElem) {
        return;
    }

    __shared__ real_t AJac[MAX_nCOMP][MAX_nCOMP][MAX_nDOF];
    __shared__ real_t BJac[MAX_nCOMP][MAX_nCOMP][MAX_nDOF];
    __shared__ real_t AuxJac[MAX_nCOMP][MAX_nCOMP][MAX_nDOF];
    __shared__ real_t AuxDof[MAX_nDOF][MAX_nCOMP];
    __shared__ real_t AuxTensor[MAX_nCOMP][MAX_nDOF][MAX_nDOF];

    /* Store Jacobian matrices */
    int_t entryAIdx = nComp*nComp*nDof_m*iElem;
    if (il < nDof_m) {
        for (int_t iq = 0; iq<nComp; iq++) {
            AJac[ip][iq][il] = A.entry[entryAIdx+nDof_m*nComp*iq+nDof_m*ip+il];
            BJac[ip][iq][il] = B.entry[entryAIdx+nDof_m*nComp*iq+nDof_m*ip+il];
        }
    }

    /* Variable to store max wave speed */
    real_t Smax;

    /* Compute module of Jacobian mapping */
    int_t entryJIdx = 4*iElem;
    /* Parameter J is the inverse of J */
    real_t Jmod = J.entry[entryJIdx+0]*J.entry[entryJIdx+3]
                -J.entry[entryJIdx+1]*J.entry[entryJIdx+2];
    Jmod = 1.0/Jmod;
```

```
    /* Auxiliary counter */
    int_t ik;

    /* Flux for each side */
    for (int_t iSide = 0; iSide<nSide; iSide++) {

        int_t entryEdgeInfo   = EdgeInfo.n1*iSide+iElem*nSide*EdgeInfo.n1;

        real_t S        = EdgeInfo.entry[0+entryEdgeInfo];
        real_t nx       = EdgeInfo.entry[1+entryEdgeInfo];
        real_t ny       = EdgeInfo.entry[2+entryEdgeInfo];
        int_t  NeigElem = float2int(EdgeInfo.entry[3+entryEdgeInfo]);
        int_t  NeigSide = float2int(EdgeInfo.entry[4+entryEdgeInfo]);

        /* This is PDE dependent */
        switch (Model) {
            case 1:
                Smax = abs(nx*AJac[0][0][0] + ny*BJac[0][0][0]); /* For linear adv
                break;
            case 4:
                Smax = sqrt(AJac[3][0][0]*AJac[0][3][0]); /* Compute cp from cell
                break;
            default:
//                printf(" Flux computation of model %d not implemented in cuda\n"
            return;
        }


        /* Store time integrated dof */
        AuxDof[il][ip] = TIDof.entry[il+ip*nDof_l+iElem*(nDof_l*nComp)];
        __syncthreads();

        /* Compute flux function for local contribution */
        if (il < nDof_m) {
            for (int_t iq = 0; iq<nComp; iq++) {
                AuxJac[ip][iq][il] = nx*AJac[ip][iq][il]+ny*BJac[ip][iq][il];
```



```
                if (il == 0 && ip == iq) {
                    AuxJac[ip][iq][il] += Smax;
                }
                AuxJac[ip][iq][il] *= 0.5*S;
            }
        }
        __syncthreads();

        /* A_pqm * TIDof_ql -> plm */
        for (int_t im = 0; im<nDof_m; im++) {
            AuxTensor[ip][il][im] = 0.0;
            for (int_t iq = 0; iq<nComp; iq++) {
                AuxTensor[ip][il][im] += AuxJac[ip][iq][im] * AuxDof[il][iq];
            }
        }
        __syncthreads();

        /* AuxTensor_plm * F_klm -> kp */
        int_t entryFluxInt_i  = iSide*nDof_k*nDof_l*nDof_m;
        ik = il;
        AuxDof[ik][ip] = 0.0;
        for (int_t im = 0; im<nDof_m; im++) {
            for (int_t il = 0; il<nDof_l; il++) {
                AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                  FluxInt_i.entry[ik+il*nDof_k+im*nDof_k*nDof_l+en
            }
        }
        /* recover index */
        il = ik;

        /* Add flux contribution */
        Dof.entry[il+ip*nDof_l+iElem*(nDof_l*nComp)] -= AuxDof[il][ip]/(Mkl.entry[
        __syncthreads();

        /* Store time integrated dof from neighbor element*/
```

```
        AuxDof[il][ip] = TIDof.entry[il+ip*nDof_l+(NeigElem-1)*(nDof_l*nComp)];
        __syncthreads();

        /* Compute flux function for neighbour contribution */
        if (il < nDof_m) {
            for (int_t iq = 0; iq<nComp; iq++) {
                AuxJac[ip][iq][il] = nx*AJac[ip][iq][il]+ny*BJac[ip][iq][il];
                if (il == 0 && ip == iq) {
                    AuxJac[ip][iq][il] -= Smax;
                }
                AuxJac[ip][iq][il] *= 0.5*S;
            }
        }
        __syncthreads();

        /* A_pqm * TIDof_ql -> plm */
        for (int_t im = 0; im<nDof_m; im++) {
            AuxTensor[ip][il][im] = 0.0;
            for (int_t iq = 0; iq<nComp; iq++) {
                AuxTensor[ip][il][im] += AuxJac[ip][iq][im] * AuxDof[il][iq];
            }
        }
        __syncthreads();

        /* AuxTensor_plm * F_klm -> kp */
        int_t entryFluxInt_j  = (NeigSide-1)*nSide*nDof_k*nDof_l*nDof_m;
        ik = il;
        AuxDof[ik][ip] = 0.0;
        for (int_t im = 0; im<nDof_m; im++) {
            for (int_t il = 0; il<nDof_l; il++) {
                AuxDof[ik][ip] += AuxTensor[ip][il][im] *
                                  FluxInt_j.entry[ik+il*nDof_k+im*nDof_k*nDof_l+en
            }
        }
        /* recover index */
        il = ik;
```



```
        __syncthreads();


        /* Add flux contribution */
        Dof.entry[il+ip*nDof_l+iElem*(nDof_l*nComp)] -= AuxDof[il][ip]/(Mkl.entry[
        __syncthreads();

    }
    return;

}
```

Interactive comment on Geosci. Model Dev. Discuss., 6, 3743, 2013.