Proposal of a new method for matrix element calculation in M3D-C1 with GPU optimization

Chang Liu

In this note, I will give a review of the current implementation of finite element sparse matrix element calculation in M3D-C1, discussing its pros and cons, and then introduce a new method which can be easily optimized for GPUs.

Note that we will only focus on the optimization of matrix element calculation for each element. The parallelization for different elements are currently done with MPI and OpenMP. It may not be possible to use the same parallelization strategy on GPU because it means each GPU thread will have different *79 arrays, and can lead out of memory errors. In addition, it also requires us to migrate the SCOREC library to GPU, which is a lot of work.

Review of current implementation of matrix element calculation in M3D-C1 (or J approach)

In the current version of code, the calculation of each matrix element is composed by three parts. The first part is the a j-loop over all the basis functions (nu79):

```
do j=1,dofs_per_element
     call vorticity_lin(mu79,nu79(j,:,:), &
     ss(:,j,:),dd(:,j,:),r_bf(:,j),q_bf(:,j),advfield,izone)
end do
```

The second part is inside these “lin” functions, where all the physics terms are calculated respectively,

```
subroutine vorticity_lin(trialx, lin, ssterm, ddterm, r_bf, q_bf,
advfield, & izone)
...
! Time Derivative
! ~~~~~~~~~~~~~~~
tempx = v1un(trialx,lin,rho79)*freq_fac
ssterm(:,u_g) = ssterm(:,u_g) + tempx
ddterm(:,u_g) = ddterm(:,u_g) + tempx*bdf
...
```

```
end subroutine vorticity_lin
```

And the details of each term is described in metricterms_new.f90

```
! V1un
! ====
function v1un(e,f,g)

     if(surface_int) then
          temp = 0.
     else
          temp = intx4(e(:,:,OP_DR),r2_79,f(:,OP_DR),g(:,OP_1)) &
               + intx4(e(:,:,OP_DZ),r2_79,f(:,OP_DZ),g(:,OP_1))
     end if

     v1un = temp
end function v1un
```

The third part is the numerical integral, which is done in functions like intx4, and is described in nintegrate_mode.f90,

```
pure function intx4(vari,varj,vark,varl)

  implicit none

  vectype, dimension(dofs_per_element) :: intx4
  vectype, dimension(dofs_per_element, npoints), intent(in) :: vari
  vectype, dimension(npoints), intent(in) :: varj, vark, varl

  integer :: k

  intx4 = 0.
  do k=1, npoints
     intx4 = intx4 + vari(:,k)*varj(k)*vark(k)*varl(k)*weight_79(k)
  enddo
end function intx4
```

Note that the output and the first input argument of this function has dimension of dofs_per_element, so the integral is done simultaneously for all the test functions (mu79). In fact, from the above line of multiplication calculation, the calculation was done first for all the other terms, including weight, before multiplying to mu79 (vari). So the estimate total number of multiplication operations is dofs_per_element*nterm*npoints*(3+dofs_per_element) (here nterm is the number of terms in the PDE).

The pros and cons of this method are as follows,

Pros:

The numerical integral function can be easily optimized by compilers, which has two layers of independent loops. This means that it can reach good speed on CPUs.

Cons:

The outside loop (j-loop) can run parallely, but the number of parallel workers is too small for GPU so it cannot reach significant speedup. In addition, for each j calculation, a lot of functions will be called and a large number of temporary arrays will be created, which can lead to OOM on GPUs.

Introduction to I-J approach

In order to reduce the complexity of calculation of each parallel worker, and increase the number of them, we can put both i-loop (loop over all the mu79 functions) and j-loop (loop over all the nu79 functions) outside. The new outside loops will look like

```
do j=1,dofs_per_element
      do j=1,dofs_per_element
            call vorticity_lin(mu79(i,:,:),nu79(j,:,:), &
            ss(:,j,:),dd(:,j,:),r_bf(:,j),q_bf(:,j),advfield,izone)
      end do
end do
```

The structure of other functions does not need to be changed much, except for the datatype (the dimension of some arrays needs to be reduced). The numerical integral function now becomes

```
pure function intx4(vari,varj,vark,varl)

  implicit none

  vectype :: intx4
  vectype, dimension(npoints), intent(in) :: vari
  vectype, dimension(npoints), intent(in) :: varj, vark, varl

  integer :: k

  intx4 = 0.
  do k=1, npoints
     intx4 = intx4 + vari(k)*varj(k)*vark(k)*varl(k)*weight_79(k)
  enddo
```

```
end function intx4
```

However, in this method, the total number of multiplication operations required is dofs_per_element*dofs_per_element*nterm*(4*npoints), which is about 4 times larger than the former approach. The reason is that, although the matrix element calculation can be done independently for all the test and basis functions, a lot of multiplication operations are repetitive for different i and j. I found that running the i-j approach on CPU always leads to worse performance than the current approach (2-3 times slower).

Pros:

There are two layers of independent loops (i and j) outside which can be easily parallelized on GPUs. There are also no OOM problems.

Cons:

The performance gets worse on the CPU. A large number of multiplication operations are unnecessary.

Separation of physics term and numerical integral

In this Hackathon I tried a different approach to combine the benefits of the above two methods. The goal is to reach better or equal performance on CPUs, and also make the GPU optimization easy.

As shown above, the calculation of multiplication of all the intermediate terms (varj, vark, varl, weight79) are repetitive for all the test functions and basis functions. In fact, this work can be done separately and the results can be stored in a temporary array. Then this array is multiplied with all the mu79 and nu79 functions.

In this approach, we need to separate the calculation of all the physical terms in PDE and the numerical integral calculation. The PDE terms will be calculated first and stored in a temporarily sparse matrix, and the numerical integral will be done later. The PDE calculation is done in functions like

```
subroutine vorticity_lin(nterm, term, op1, op2, ssarray, ddarray,
advfield, izone)

integer, dimension(MAX_TERMS), intent(out) :: term, op1, op2
vectype, dimension(MAX_PTS,MAX_TERMS), intent(out) :: ssarray,
ddarray

  ! Time Derivative
  ! ~~~~~~~~~~~~~~~
```

```
  temp=r2_79*rho79(:,OP_1)
  if (surface_int) temp=0.
  ADDTERM(u_g,OP_DR,OP_DR,temp,temp)
  ADDTERM(u_g,OP_DZ,OP_DZ,temp,temp)

end subroutine
```

Where ADDTERM is a macro and is defined as

```
#define ADDTERM(xterm,xop1,xop2,ssterm,ddterm) \
     iterm=iterm+1; \
     term(iterm)=xterm; \
     op1(iterm)=xop1; \
     op2(iterm)=xop2; \
     ssarray(:,iterm)=ssterm*weight_79; \
     ddarray(:,iterm)=ddterm*weight_79
```

Here ssarray and ddarray are the temporary arrays that store the results of different terms for all the quadrature points.

The outside calculation can be done as

```
call vorticity_lin(nterm, term, op1, op2, ssarray, ddarray,
advfield,izone)

do iterm=1,nterm
     do j=1,dofs_per_element
          tempss=ssarray(:,iterm)*nu79(j,:,op2(iterm))
          tempdd=ddarray(:,iterm)*nu79(j,:,op2(iterm))
          do i=1,dofs_per_element
               ssterm(i,j,iterm)=sum(mu79(i,:,op1(iterm))*tempss)
               ddterm(i,j,iterm)=sum(mu79(i,:,op1(iterm))*tempdd)
          end do
     end do
end do
do i=1,dofs_per_element
     do j=1,dofs_per_element
          do iterm=1,nterm

ss(i,j,term(iterm))=ss(i,j,term(iterm))+ssterm(i,j,iterm)

dd(i,j,term(iterm))=dd(i,j,term(iterm))+ddterm(i,j,iterm)
          end do
     end do
end do
```

After calculating ssarray and ddarray, the results are used to multiply with nu79 and mu79. Note that here we first do the multiplication with nu79 and store the results in tempss and tempdd, and then do the multiplication with mu79. This is followed by a summation reduction to obtain the final results of ss and dd. The total number of multiplication operation is nterm*dofs_per_element*(npoints+dofs_per_element*npoints), which is even smaller than the current implementation (j approach).

Pros:

The total number of multiplication operations required is small. The numerical integration is done separately from the physics part. According to our measurement, most of the cpu time (>90%) is spent on the numerical integration (two loops in the above code), thus one only needs to do GPU optimization for these two loops. This can be easily done as the loops are mostly independent, and the speedup is significant.

Cons:

As shown above, the calculation of all the physical terms in PDE needs to be rewritten. Basically we need to combine ludef_t.f90 and metricterms_new.f90 and make it a big file, which is rather complicated.

Summary

According to the tests done and what we learned from Hackathon, we can either use the I-J approach or the new approach which separates the physics part and the numerical integration part for the GPU optimization. The I-J approach can be easily implemented, but it can lead to worse performance on CPUs, and mediocre performance on GPUs. The new approach is very attractive regarding performance on both CPUs and GPUs, but it requires significant work of code rewrite.

I think if we decide to go with the new approach, we should use python or some other languages to write an interpreter or lexer to interpret what we have now in ludef_t.f90 and metricterms_new.f90, and translate it into the new form. Given the difficulties in understanding the new form by humans, we can then make this interpreter as part of our make file and do this translation every time we make the project.