EUVEL:
An EULER Vector Extension Library

Walter M. Lioen <walter@cwi.nl>
Margreet Louter-Nool <greta@cwi.nl>

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

In this report the vectorization of EULER, a multigrid code for first order accurate Euler flow computations on a self-adaptive grid, is described. To enhance the vectorization of the relaxation process we replaced Osher's flux-difference splitting scheme by the Van Leer flux-splitting scheme.

Primary: 65-04. Secondary: 65M50, 65M55, 65Y99. G.1.8. software, Euler equations, adaptive multigrid, vectorization. This work was supported by Cray Research, Inc., under grant CRG 91.01, via the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF).

1. Introduction

In this report we describe the vectorization on a Cray Y-MP of a Fortran multigrid code that performs 2D Euler flow computations with automatic mesh adaptation [6]. The implementation uses the data structure as described in [1]. The data handling subroutines and the subroutines for Euler flow computation are separate modules in the code. The former is called 'BASIS' and the module performing the adaptive multigrid Euler flow computation is called 'EULER'. We describe a so-called 'plug-compatible' EUler Vector Extension Library, which contains a set of vectorized subroutines with the same entry point names and calling sequences as the original BASIS and EULER subroutines. In the sequel, we will refer to this extension module as 'EUVEL'.

Since the implementation of the subroutines in BASIS and EULER is inherently scalar and since the greatest performance gain on the Cray Y-MP is expected from vectorization and not from parallelization, we focus our research on vectorization of the original EULER code using the same data structure. In Section 2, we describe an extension to BASIS to facilitate such vectorization. In Section 3, we describe the vectorization of some of the EULER subroutines. To enhance the vectorization of the relaxation process Osher's flux-difference splitting scheme [7] has been replaced by Van Leer's flux-splitting scheme [4].

2. Extensions to BASIS

A strict definition of the data structure can be found in [1]. The computational domain Ω is partitioned into a finite number of quadrilaterals, called cells. For the adaptive multigrid algorithm different levels of discretization are used. Each cell of the grid Ω^l on level $l > 0$ is a member of a division of a cell of Ω^{l-1}, into a set of 2×2 smaller cells. We call the cell on

Ω^{l-1} the parent cell and the four smaller cells of the subdivision on Ω^l the NE-kid cell, SE-kid cell, SW-kid cell, and NW-kid cell, respectively. (NE, SE, SW, and NW are abbreviations of North-East, South-East, South-West, and North-West, respectively.) The data structure is ordered by so-called patches: a point with possibly a horizontal wall, a vertical wall and a cell. The data structure consists of a tree of patches, with at most four new branches at each node: a so-called quad tree. With each patch a number of pointers, properties, coordinates and data contents are associated.

The workhorse of EULER is the BASIS subroutine called `Scan`, which accomplishes a depth-first quad tree traversal algorithm invoking `DoIt`, an external subroutine passed as argument to `Scan`, for every patch found. Another argument of `Scan` is an `INTEGER` array `Order(1:4)` which contains a permutation of the directions NE, SE, SW, NW: the order in which the four kid cells (branches in the tree) are visited. Since `DoIt` is called separately for every patch found we hardly find any performance gain by automatic vectorization. On the other hand, the data structure is well-suited for a self-adapting mesh. We have decided to leave the data structure unchanged and to add a pointer structure in order to make vectorization possible.

Each patch has an 'arbitrary' identification number. Therefore, two successive patch numbers can be related to patches that have nothing in common. All information concerning patches is originally stored in three large arrays:

`PNTR` – an `INTEGER` array,
`PPTY` – a `LOGICAL` array,
`DATA` – a `REAL` (or `DOUBLE PRECISION`) array.

The arrays are of dimension `MNOP` (`M`aximum `N`umber `O`f `P`atches). Instead of scanning the patches tree-wise, we operate on patches of which the pointers are listed in index arrays. Which specific index array is used depends on the kind of action (i.e. on the subroutine passed to `Scan`).

First of all, pointers to patches that belong to the same level are gathered and stored into array `LevelW(0:MNOP)`. Array `LevelW` combined with `INTEGER` array `NLev(0:MNOL)`, where `MNOL` denotes the `M`aximum `N`umber `O`f (positive) `L`evels, tells us where we can find patches of level `k`:

`LevelW(1:NLev(0))` – contains all pointers to patches on level `0`,
`LevelW(NLev(k-1) + 1 : NLev(k))` – contains all pointers to patches on level `k`.

We have vectorized the collective symmetric point Gauß-Seidel relaxation by using a diagonal ordering. For this purpose, the elements of `LevelW`, referring to patches, are stored such that their diagonal number, being the sum of the ξ- and η-coordinates, is in monotonously non-descending order. The `INTEGER` array `NDiag(0:MNOD, 0:MNOL, 0:2)`, where `MNOD` denotes the `M`aximum `N`umber `O`f `D`iagonals, informs about the position of the diagonal elements within the array `LevelW`. The patches on diagonal `d` of level `k` can be found in the range

```
NDiag(d, k, 0) :  NDiag(d, k, 1)
```

of `LevelW`. These index arrays are all generated by subroutine `SortLv` and they must be updated only after the cell structure (the mesh) has been changed.

The changes the user must make to the EULER-based program are:

- `PARAMETER`-statements in `euvel.i` must be adapted,
- `euvel.i` must be included in the main program,
- calls to `NewDat` must be placed after the initial grid creation by `MkGeo` and after grid-adaptation with `Refn` and/or `UnRefn`.

Finally, the object module `euvel.o` must be loaded before the object modules `basis.o` and `euler.o`. As a result, the scalar BASIS and/or EULER routines are replaced by their vectorized EUVEL counterparts.

3. Vectorization of some of the EULER subroutines

We have restricted ourselves to the most time-consuming subroutines. All subroutines adapted were originally implemented using `Scan`.

3.1 The Flux, Transport and their Derivatives: `FluxTA`, `FTA`, `OsherA` *and* `VLeerA`

The P-variant of Osher's flux-difference splitting scheme has been replaced by Van Leer's flux-splitting scheme to enhance the vectorization of the relaxation. The relaxation process is responsible for approximately 80–90% of the total CPU-time. The vector length for the relaxation will be at most the number of cells in a diagonal of the finest grid. However, since we are dealing with an adaptively refined grid the vector length will be less in most cases. The reason for preferring Van Leer's scheme above Osher's scheme is its smaller number of branchings. In Osher's scheme there are sixteen different possibilities, whereas in Van Leer's scheme there are only four. The main problem in vectorizing is not the number of conditions, but the number of patches that satisfies a condition in the upwind scheme. Moreover, this small number of patches in a diagonal declines during the relaxation process, because the number of Newton-steps needed on the individual patches can differ. It is clear that the sixteen possibilities in Osher's scheme would result in even smaller vector lengths than the four possibilities in Van Leer's scheme.

The original subroutine `FTA` computes the `F`lux, the `T`ransport and the possibly left and/or right transport derivatives ('`A`ccent' from 'f'') on a single patch. We have created a generic source (to be preprocessed by cpp, the C language preprocessor) with the functionality of `FTA`. It operates on multiple patches (passed as an array of patch numbers), and it has Van Leer's flux-splitting scheme completely inlined. With cpp six different versions are generated all working on multiple patches, but with different variants for horizontal and vertical walls, and if necessary left or right derivatives are computed. It saves us from passing some original arguments, and even more important, it saves us from the resulting conditions in the generated subroutines. For example, one of the variants, `FTAHLS`, replaces `FTA` on `H`orizontal walls computing `L`eft derivatives and it operates on multiple patches (a `S`ubset). Inside a `FTA` variant, lists of patch numbers are pre-computed satisfying the subsonic and supersonic flow conditions considered in the Van Leer scheme; this is done completely vectorized. Next, the flux, the transport and the (possible) derivatives are computed using indirect addressing via the pre-computed lists (also completely vectorized).

3.2 Make right-hand-sides: `MkRhs`

Originally, the right hand sides were constructed using `Scan` over patches that built up the grid. Vectorization by looping over all patches on one level is not possible, because too many

actions must be performed for each patch: fluxes must be calculated and must be send to the memory locations in DATA on behalf of the right hand sides of a cell and/or its neighbors. Then, also the right hand sides of the kids are added, or, when there are no kids, some source term is evaluated. For the sake of vectorization it is better to split up the actions and to perform them on all patches residing on the same level. The new structure of the subroutine MkRhs looks as follows:

```
call ZRhsV( ... )

if (lev .eq. TopLev) then
   call MkRhsTV( ... )
else
   call MkRhsKV( ... )
   call MkRhsHV( ... )
   call MkRhsVV( ... )
end if
```

All subroutines operate on multiple patches listed in an index array and they perform one of the following subtasks:

- ZRhsV – initializes the right hand side locations of DATA on a given level;
- MkRhsTV – evaluates the source term on the highest level;
- MkRhsKV – adds the right hand sides of the kids to the patches, or, when there are no kids, a source term is evaluated, which is added to the patches;
- MkRhsHV – calculates the horizontal fluxes and sends them to corresponding DATA memory locations of the cells and/or their southern neighbors;
- MkRhsVV – calculates the vertical fluxes and sends them to corresponding DATA memory locations of the cells and/or their western neighbors.

The computation of the fluxes is bounded by many restrictions: distinction must be made between green walls (i.e. walls at fine-coarse grid interfaces), boundary walls, and ordinary walls. Moreover, it should be known whether the patches have kids or have not, and how the family situation next door looks like. Fortunately, the computation of the horizontal and vertical fluxes can be done independently, minimizing the number of conditions for a patch.

Again it turns out to be convenient to use index arrays in order to save tests on properties of patches and to make vectorization easier. On behalf of the subroutine MkRhs a 2-dimensional INTEGER array CP(0:MNOP, 1:3) is generated by the subroutine CmKids. CmKids, based on the index array LevelW, contains information about patches CP(:,1) and their horizontal (southern) CP(:,2) and vertical (western) CP(:,3) neighbors. The related index array nCP(1:3, 1:3, 0:MNOL) informs about start positions in array CP. Subroutine CmKids is called by NewDat in the main program after the cell structure has been adapted.

The flux transport computation across ordinary walls has been completely vectorized. For the green and boundary walls the first order flux with respect to left and right states required a slightly different approach. However, as soon as the left and right states have been computed for all green and boundary walls, the process can be accomplished analogously to the 'ordinary' case, which means completely vectorized. The fluxes are computed by the vectorized subroutines FTAHS and FTAVS described in Section 3.1. Finally, the sending of the

fluxes to the patches and/or their neighbors has been done in a straightforward way using the ordering defined by `CP`.

3.3 The residual: `Res`

The computation of the residual on the composite grid corresponds to the previously discussed computation of the right hand sides of the equations. First order fluxes in horizontal and vertical direction must be computed. Again, it is possible to separate the horizontal and vertical part. For both directions index arrays `ComHor` and `ComVer` of dimension `MNOP` are generated for the composite grid, including all levels ≥ 0. The computational complexity for the residual's computation and the right hand sides' computation is roughly the same.

Originally, after the residual was computed, another `Scan` through the data structure was made to construct the L_1-norm and L_∞-norm of the residual fields. The weighting is done by a multiplication of the residual with a factor 4^{-l}, where l is the level on which the cell resides. This implies, that for each patch its weighting factor must be computed, because it depends on its level. Operating along the index array `LevelW` has the advantage of an invariable weighting factor for a fixed level.

The L_1-norm for a residual component i is defined by

$$RMn(i) = \sum_p |x_p|$$

and the L_∞-norm by

$$RMx(i) = \max_p |x_p|.$$

Both norms can be calculated at vector speed using the BLAS subroutines `SASUM` and `ISAMAX`, respectively. Originally, the norms were computed simultaneously; in the modified source these norms are computed separately for each residual field i.

3.4 The relaxation: `Relax, Gauss`

As a relaxation procedure EULER uses collective symmetric point Gauß-Seidel relaxation. *Point* refers to the property that during the update of a state vector on a patch all other state vectors are kept fixed. *Collective* refers to the property that the update of the state vector on a patch is done for all of its four components simultaneously. *Symmetric* means that after a relaxation sweep a new sweep is made with the reverse ordering. At each cell visited during a relaxation sweep a system of four nonlinear equations is approximately solved with Newton iterations, the differential operator applied being $(\partial/\partial u, \partial/\partial v, \partial/\partial c, \partial/\partial z)^T$; see [2] for definitions/details.

We considered two possibilities for vectorizing the relaxation:

1. Replacing symmetric Gauß-Seidel by a red-black ordering leading to an essentially worse convergence factor (based on other experiments we expect to loose a factor 5).

2. Staying with symmetric Gauß-Seidel, but using a diagonal ordering (for this the nonlinear systems on a diagonal are completely decoupled) and replacing Osher's scheme by Van Leer's scheme. This results in longer vectors, because the number of possibilities in the upwind scheme reduces from sixteen to four.

As we already described in Section 3.1 we chose the latter approach. This means that, except for the replacement of Osher's flux-difference splitting scheme by Van Leer's flux-splitting scheme, we have precisely the same algorithm as in the original EULER code. It can easily be seen that the original relaxation using `Scan` with (SW, NW, SE, NE)-`Order` corresponds with the lexicographical ordering, which in turn is identical with the diagonal ordering with all non-linear systems on a diagonal completely decoupled. In Section 2, we already described how an additional pointer structure has been added to facilitate working on diagonals.

We must still describe how the decoupled non-linear systems on a diagonal are solved. First, the linearized (4×4) systems are constructed by using the previously described flux, transport and derivatives subroutines. Subsequently, a single Newton iteration step is performed. At this step multiple linear 4×4 systems must be solved. For this, a completely vectorized subroutine `GaussS` has been developed. This subroutine contains one completely vectorizable loop and the loop body solves a linear 4×4 system using Gaußian elimination with partial pivoting. In order to vectorize this loop and to enhance the vector and even the scalar performance the Gaußian elimination code for the solution of a single linear 4×4 system is completely unrolled. The remaining problem at this point is that one or more of the linear systems might be singular. This is taken care of by replacing the matrix-diagonal elements by '1' and separately marking the system singular.
Finally, we notice that we do not know beforehand, how many Newton iteration steps are to be performed. After a Newton step on all patches in a diagonal, we test whether the specified accuracy is reached, and then, the patch numbers are collected of the non-linear systems that did not reach the required accuracy. The Newton process is restarted on this probably much smaller subset. The iteration process is finished if either the subset becomes empty or after `MaxNwt` Newton iterations are performed.

3.5 Some other vectorized subroutines

Finally, we discuss some additional subroutines which have been vectorized, viz., `RstSol`, `BckUp` and `AddP1`. `RstSol` makes a restriction of the solution on that given level, which is sent to the memory locations of the solution on a given level. `BckUp` makes a copy from the solution memory locations on a given level. This copy is stored in the old solution memory locations. `AddP1` adds the correction from one level to another. The `Scan` approach originally used is too expensive, partly due to the subroutine call overhead. In the new implementation the actions are performed directly within the subroutines for multiple patches and can be vectorized, including the `IF`-tests on the patch properties. Additional index arrays are not used.

4. Vector Performance

We consider the same two test problems that were used during the development of BASIS and EULER. In this section, our main interest is the performance gain obtained by vectorization.

4.1 Shock reflection

The first test problem is the shock reflection problem used in [6]. Here the finest grid has dimensions 128×64, so the maximal diagonal length (being the maximal vector length) of the relaxation process is 64. The vector speed-ups measured for one FAS cycle can be found in Table 1. In order to build the additional index-arrays to allow vectorization the EUVEL

Table 1: Speed-up achieved with EUVEL on shock reflection problem

subroutine	CPU time original	CPU time using EUVEL	Speed-Up
`AddP1`	0.0107	0.0009	12
`BckUp`	0.0079	0.0002	40
`MkRhs`	0.2070	0.0279	7.4
`Relax`	2.1587	0.5294	4.1
`RstSol`	0.0106	0.0006	18
`FAS` Total	2.3949	0.5590	4.3
`Res`	0.1707	0.0208	8.2
`NormRP`	0.0275	0.0008	34

Table 2: Timing of additional EUVEL subroutines for shock reflection problem

subroutine	CPU time
`SortLv`	0.0091
`CmKids`	0.0020
`CmPar`	0.0031

subroutine `NewDat` has been added. `NewDat` calls the three subroutines `SortLv`, `CmKids`, and `CmPar`, whose timings can be found in Table 2. As can be seen the total cost of maintaining the additional data structure is 2.5% of the total CPU-time.

4.2 Transonic airfoil flow

Here we consider the same transonic flow problem around the NACA0012-airfoil as in [5]. Besides the replacement of Osher's flux-difference splitting scheme by Van Leer's, the solution method differs from this problem due to the special geometry. For the solution of the airfoil problem a cylindrical grid (a rectangular grid with coinciding lower and upper boundaries) is used. The `d`-th diagonal, having `d` as sum of the ξ- and η-coordinates, extends over the lower boundary to the diagonal in the rectangular grid having `d` + `NY` as sum of the ξ- and η-coordinates. `NY` is the number of points in the η-direction. In fact, the diagonals on the rectangular grid render into spirals on the cylindrical grid. In Table 3, we can find the vector speed-up for this problem.

5. Conclusions

We were able to vectorize the original EULER code and obtained a vector speed-up of 4–5. To judge this speed-up we must bear in mind, first, the indirect addressing, necessary for the adaptive grid, and second, the relatively small average vector length due to the adaptive grid, too. Furthermore, the use of adaptive grids instead of uniformly refined grids has already decreased the amount of computational work with a factor 5–10 for realistic problems. We

Table 3: Speed-up achieved with EUVEL on airfoil problem

subroutine	CPU time original	CPU time using EUVEL	Speed-Up
`AddP1`	0.0229	0.0019	12
`BckUp`	0.0167	0.0004	40
`MkRhs`	0.4193	0.0641	6.5
`Relax`	4.4489	0.9914	4.5
`RstSol`	0.0427	0.0025	17
`FAS` Total	4.9510	1.0605	4.7
`Res`	0.3396	0.0347	9.8
`NormRP`	0.0500	0.0015	33

could obtain slightly higher vector speeds by changing the refinement criterion. However, in that case, the increase of computational work is not compensated by the increased vector speed.

The price to be paid for vectorizing the original Euler code is the replacement of Osher's flux-difference splitting scheme by the somewhat less accurate Van Leer flux-splitting scheme. As indicated by Koren [3] this is a relatively low price, because it is possible to compensate for this by applying defect correction with Osher's flux-difference splitting scheme.

As indicated in the introduction, vectorization instead of parallelization of the code was a well-considered approach. Apart from the fact that the greatest performance gain on a Cray Y-MP was expected from vectorization, it was a real challenge to vectorize the inherently scalar code. Parallelization is still possible using domain decomposition or, chopping up the quad tree. Compared to our vectorization efforts, parallelization should be relatively easy.

Acknowledgements

Financial support for this work was provided by Cray Research, Inc. under a 1991 University Research & Development Grant for the project 'Numerical Multigrid Software for the Cray Y-MP4'. This work was sponsored by the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organization for Scientific Research, NWO). We are grateful to Barry Koren for suggesting the use of the Van Leer scheme and providing us with a 'Van Leer'-implementation on a single patch. We wish particularly to acknowledge the help of Eric van der Maarel for his patience during the numerous explanations of original code fragments.

References

1. P.W. Hemker, H.T.M. van der Maarel, and C.T.H. Everaars. BASIS: A data structure for adaptive multigrid computations. Technical Report NM-R9014, CWI, August 1990.
2. P.W. Hemker and S.P. Spekreijse. Multiple grid and Osher's scheme for the efficient solution of the steady Euler equations. *Appl. Numer. Math.*, 2:475–493, 1986.

3. B. Koren. Private communication, 1992.
4. B. van Leer. Flux-vector splitting for the Euler equations. In *Proceedings Eighth International Conference on Numerical Methods in Fluid Dynamics*, volume 170 of *Lecture Notes in Physics*, pages 507–512, 1982.
5. H.T.M. van der Maarel. Adaptive multigrid for the steady Euler equations. *Comm. Appl. Numer. Methods*, 8(10):749–760, 1992.
6. H.T.M. van der Maarel, P.W. Hemker, and C.T.H. Everaars. EULER: An adaptive Euler code. Technical Report NM-R9015, CWI, August 1990.
7. S. Osher and F. Solomon. Upwind difference schemes for hyperbolic systems of conservation laws. *Math. Comput.*, 38:339–374, 1982.