An Array API for FDM

Intro

- Move towards ultrascale computing
- Uniform mapping
- Hierarchical memory
  - (Processor -> Core LM, Accelerator LM, Core SM, Accelerator/Processor SM)
- Message Passing and Partitioned Global Address Space
- CUDA and Hybrid models
- Problem with porting

Array API

- Linear indices -> Hierarchical memory
  - Don’t have to rely on compilers
- Problem space -> Linear Array
  - Reusable (generics)
  - Collective operations

Burgers equation (Finite difference method)

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = \nu \nabla^2 \vec{u},$$

$$\frac{\partial (u)}{\partial t} + (u) \cdot \nabla (u) = \nu \nabla^2 (u)$$

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}.$$

$$u_i^{n+1} = u_i^n - \frac{1}{2\Delta x} u_i^n (u_{i+1}^n - u_{i-1}^n) + \frac{\nu}{(\Delta x)^2} (u_{i+1}^n + u_{i-1}^n - 2u_i^n).$$

In 3D

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = \nu\frac{\partial^2 u}{\partial x^2} + \nu\frac{\partial^2 u}{\partial y^2} + \nu\frac{\partial^2 u}{\partial z^2} \quad (5)$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = \nu\frac{\partial^2 v}{\partial x^2} + \nu\frac{\partial^2 v}{\partial y^2} + \nu\frac{\partial^2 v}{\partial z^2} \quad (6)$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = \nu\frac{\partial^2 w}{\partial x^2} + \nu\frac{\partial^2 w}{\partial y^2} + \nu\frac{\partial^2 w}{\partial z^2}$$

In 3D

$$
\begin{aligned}
u_{i,j,k}^{n+1} &= u_{i,j,k}^{n} - \frac{1}{2\Delta x} u_{i,j,k}^{n} (u_{i+1,j,k}^{n} - u_{i-1,j,k}^{n}) \\
&+ \frac{\nu}{(\Delta x)^2} (u_{i+1,j,k}^{n} + u_{i-1,j,k}^{n} - 2u_{i,j,k}^{n}) \\
&- \frac{1}{2\Delta y} v_{i,j,k}^{n} (u_{i,j+1,k}^{n} - u_{i,j-1,k}^{n}) \\
&+ \frac{\nu}{(\Delta y)^2} (u_{i,j+1,k}^{n} + u_{i,j-1,k}^{n} - 2u_{i,j,k}^{n}) \\
&- \frac{1}{2\Delta z} w_{i,j,k}^{n} (u_{i,j,k+1}^{n} - u_{i,j,k-1}^{n}) \\
&+ \frac{\nu}{(\Delta z)^2} (u_{i,j,k+1}^{n} + u_{i,j,k-1}^{n} - 2u_{i,j,k}^{n}),
\end{aligned}
$$

In Code

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
   for (int k = 0; k < P; k++) {
      up(i,j,k) = un(i,j,k)
        - un(i,j,k)
          * ( un(i+1,j,k) - un(i-1,j,k) )
          / (2 * deltax)
        + ... ;
      vp(i,j,k) = ...;
      wp(i,j,k) = ...;
   }
}
```

Multiarray API

```
1 /** Elemental addition, multiplication and subtracti
2   function _+_ ( a:E, b:E ) : E;
3   function _*_ ( a:E, b:E ) : E;
4   function _-_ ( a:E, b:E ) : E;
5   function -_ ( a:E ) : E;
6 /** Mapped addition, multiplication and subtraction
7   function _+_ ( a:MA, b:MA ) : MA;
8   function _*_ ( a:MA, b:MA ) : MA;
9   function _-_ ( a:MA, b:MA ) : MA;
0   function -_ ( a:MA ) : MA;
1
2 /** Relating the mapped and elemental operations. */
3   axiom binaryMap ( a:MA, b:MA, i,j,k:in
4     assert get( a+b, i,j,k )
        == get(a,i,j,k) + get(a,i,j,k);
      assert get( a*b, i,j,k )
        == get(a,i,j,k) * get(a,i,j,k);
      assert get( a-b, i,j,k )
        == get(a,i,j,k) - get(a,i,j,k);
  }
  axiom unaryMap ( a:MA, i,j,k:int ) {
    assert get( -a, i,j,k ) == - get(a,i,j
  }
```

Multiarray API

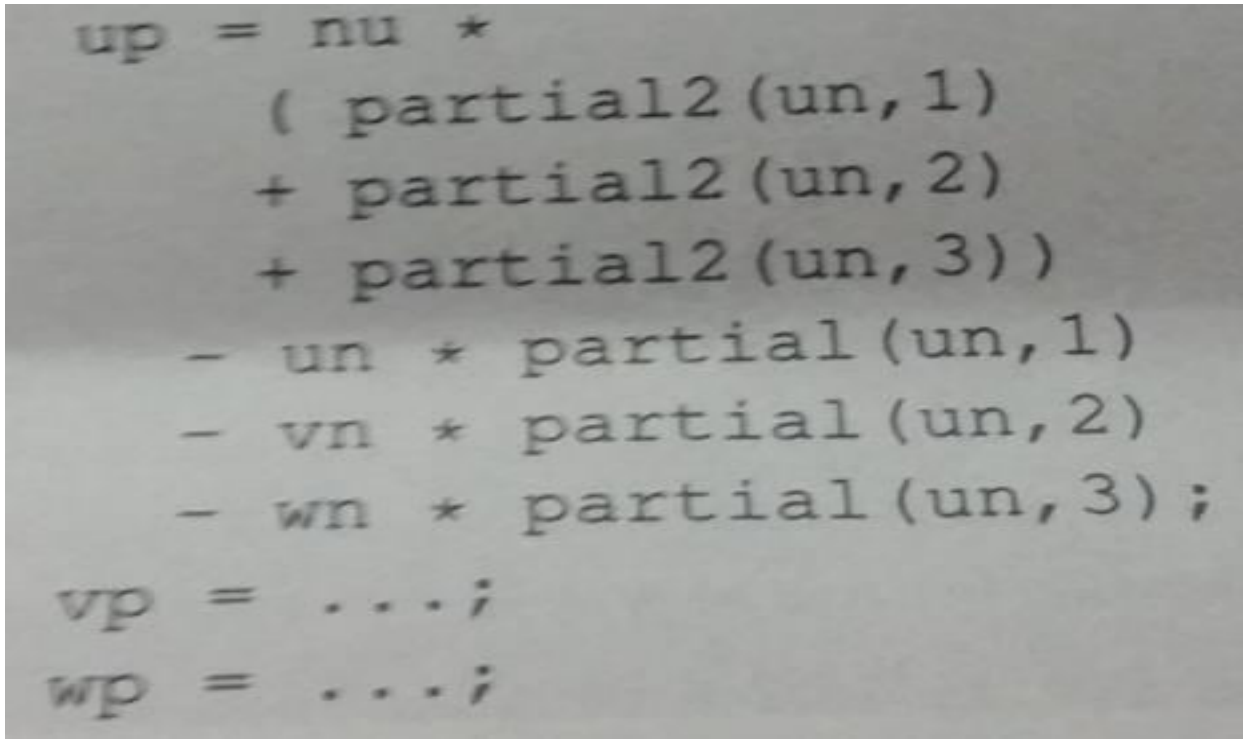
```
function shift (a:MA, dir:int, d:int)
```

```
axiom multiarrayShiftAxiom
  ( a:MA, d:int, i,j,k:int ) {
  assert get( shift(a,1,d), i,j,k)
    == get(a, (Lx+i+d)%Lx,j,k);
  assert get( shift(a,2,d), i,j,k)
    == get(a, i,(Ly+j+d)%Ly,k);
  assert get( shift(a,3,d), i,j,k)
    == get(a, i,j,(Lz+k+d)%Lz);
};
```

Multiarray API

```
function partial1 ( a:MA, dir:int ) : MA {
  return (shift(a,dir,1) - shift(a,dir,-1))
    / (2 * deltax );
};
function partial2 ( a:MA, dir:int ) : MA {
  return
    (shift(a,dir,-1) - 2*a + shift(a,dir,1))
    / (deltax * deltax );
};
```

Multiarray API



```
up = nu *
     ( partial2(un,1)
     + partial2(un,2)
     + partial2(un,3))
   - un * partial(un,1)
   - vn * partial(un,2)
   - wn * partial(un,3);
vp = ...;
wp = ...;
```

Linear array API

```
1  /** Get the element at the index position i.
2  function get ( a:A, i:int ) : E;
```

```
1  /** Elemental addition, multiplication and subtraction. */
2    function _+_ ( a:E, b:E ) : E;
3    function _*_ ( a:E, b:E ) : E;
4    function _-_ ( a:E, b:E ) : E;
5    function -_ ( a:E ) : E;
6  /** Mapped addition, multiplication and subtraction. */
7    function _+_ ( a:A, b:A ) : A;
8    function _*_ ( a:A, b:A ) : A;
9    function _-_ ( a:A, b:A ) : A;
10   function -_ ( a:A ) : A;
11
12 /** Relating the mapped and elemental operations. */
13 axiom binaryMap ( a:A, b:A, i:int ) {
14   assert get( a+b, i ) == get(a,i) + get(a,i);
15   assert get( a*b, i ) == get(a,i) * get(a,i);
16   assert get( a-b, i ) == get(a,i) - get(a,i)
17 }
18 axiom unaryMap ( a:A, i:int ) {
19   assert get( -a, i ) == - get(a,i);
```

Linear array API

```
1  /** Shifts the grp-sized groups of data d positions circularly to the left within each seg-sized segment. */
2  function shiftSegmentGroups ( a:A, seg:int, grp:int, d:int) : A
3    guard seg % grp == 0 && getSize(a) % seg == 0 && abs(d) <= seg;
4
5  axiom shiftSegmentGroupsDefinitionAxiom ( a:A, seg:int, grp:int, d:int, j:int ) {
6    var size = getSize(a);
7    assert size % seg == 0 && seg % grp == 0 && abs(d) <= seg && 0 <= j && j < size;
8      // local index within a segment
       var si = j % seg;
       // normalize actual shift value to perform within a segment
       var sh = (grp * (seg + d)) % seg;
       // new index within segment after the local shift
       var ni = ( seg+si-sh ) % seg;
       // obtain the global position of ni within the whole array
       var ind = idiv(j, seg)*seg + ni;
     assert get(shiftSegmentGroups(a,seg,grp,d),ind) == get(a,j);
);
```

Multiarray Library

```
function get( a:MA, i,j,k:int ) : E {
  return get(a, i*Ly*Lz + j*Lz + k );
}
```

Multiarray Library

```
 1 function shift ( a:MA, dir:int, d:int ) : MA (     m
 2   var seg =                                        n
 3     if dir == 1 then Lx * Ly * Lz
 4     else if dir == 2 then Ly * Lz
 5     else /* dir == 3 */       Lz;
 6     end end;
 7   var grp = seg /
 8     if dir == 1 then Ly * Lz
 9     else if dir == 2 then Lz
10     else /* dir == 3 */    1;
11     end end;
12   return shiftSegmentGroups( a, seg, grp, d );
13 }
```

Implementations

- CPU C++
  - C++ arrays
- Cuda
  - Linear structure on device
- CudaBuffer
  - Buffer created at start

Runtime results

Cpu C++	1	10	100	1000
50	0.268	2.900	26.198	264.116
63	0.535	5.794	53.620	530.304
79	1.058	11.516	103.989	1032.465
100	2.176	21.374	220.488	2181.002
126	4.671	51.581	482.747	4642.034
159	10.156	128.887	1176.986	10124.613
200	28.517	958.223	6532.079	32186.134
252	776.532	2387.521	13636.333	120525.580

Cuda	1	10	100	1000
50	0.290	3.151	29.479	286.206
63	0.310	3.408	31.190	311.993
79	0.370	3.939	36.294	363.476
100	0.487	4.848	49.076	492.145
126	0.747	7.557	75.908	749.141
159	1.325	14.401	132.114	1327.804
200	3.227	31.472	310.946	3098.445
252	6.448	62.974	638.149	6358.311

CudaBuffer	1	10	100	1000
50	0.052	0.577	5.057	50.619
63	0.095	1.045	9.552	94.447
79	0.162	1.777	16.178	162.277
100	0.290	2.934	29.526	292.908
126	0.563	5.694	57.514	567.513
159	1.133	12.439	114.399	1134.132
200	2.346	26.430	233.351	2477.200
252	5.108	49.518	500.261	5032.021

Runtime results