Automatically Generating High Performance Source Code for a Many-Core CPU or Accelerator from a Simplified Input Code Expression

Jagan Jayaraj, Pei-Hung Lin, Paul R. Woodward,
and Pen-Chung Yew
Laboratory for Computational Science & Engineering
University of Minnesota

Jan. 27, 2011

The Problem:
Single-Chip & Single-Node Performance

Overall system performance begins at a node.

- ***Scalability is not enough. 1000 x 0.01 = 10.***
- **Getting to exascale will be immensely easier if we can bump up the typical single core performance from 1% – 5% of peak to 25 – 50% of peak.**

We must overcome:

- **Small ratio off-chip memory bandwidth / peak flops**
- **Need for strong scaling, doing less per time step on each core while still running at 25 – 50% of peak.**

Our Approach for Achieving High Single-Chip & Single-Node Performance

Small ratio off-chip memory bandwidth / peak flops:

- **The only way to address this in general is with a sufficiently large on-chip cache.**
- **Making optimal use of this resource requires, in our experience, extreme code restructuring.**
- **This requires assistance of a precompilation tool.**

Need for strong scaling, doing less per time step on each core while still running at 25 – 50% of peak:

- **Must make full use of vector SIMD units.**
- **Vectors must be short (cache is small), hence *aligned*.**
- **Multiple cores on single chip must *cooperatively* update a single subdomain to minimize messaging.**

Quantifying the Trade-off between Memory Bandwidth and On-Chip Cache Capacity

Example of our present multifluid PPM code.

(Ignore cost of reading the instructions.)

1) **One entire time step update on chip:**
   3900/120 flops/byte = 130 flops/word (9 MB)
2) **Single 1-D pass update on chip:**
   1300/120 flops/byte = 43.3 flops/word (256 KB)
3) **PPMinterp for single variable on chip:**
   34/32 flops/byte = 4.25 flops/word (64 KB)
4) **RiemannStates on chip:**
   79/140 flops/byte = 2.26 flops/word (64 KB)
5) **Fluxes on chip:**
   162/208 flops/byte = 3.12 flops/word (64 KB)

A modest cache buys a factor of 10 in mem. Bandwidth!!


How does the code look?

What did the translator do

Inlining

- ✓ **Necessary for pipelining**

Pipelining

- ✓ **Controls the explosion of temporaries**

Temporary array space reduction

- ✓ **Fit the whole computation into the L-1/L-2 cache**

Prefetching

- ✓ **Overlap computation w/ communication to a node's own main memory**

Fusing (Fusing IFs, not loops)

- ✓ **Move pipelined code blocks to reduce branching**

Unrolling

- ✓ **Reduce branching.**

Relies on directives and annotations

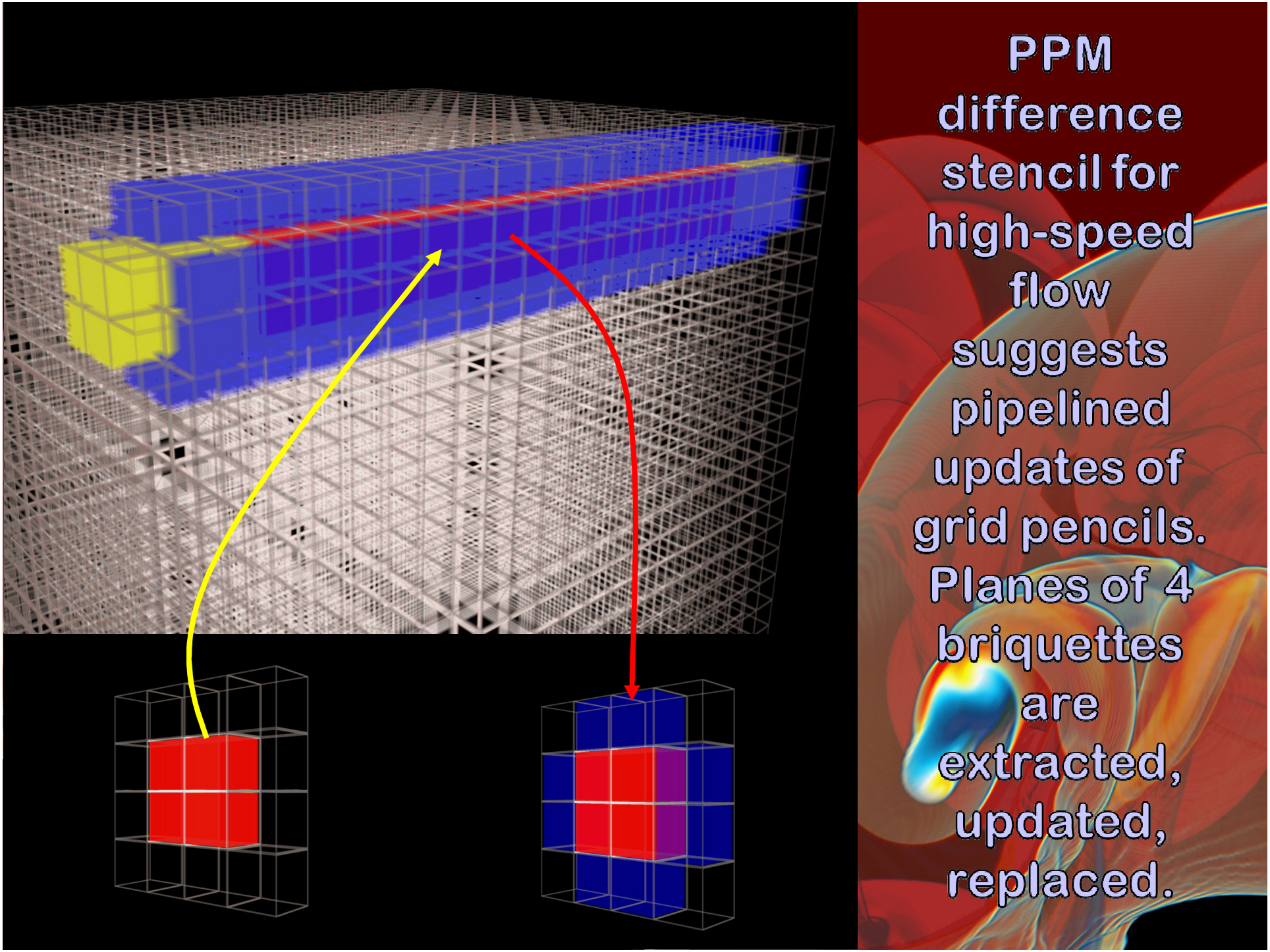
Fully Pipelined Processing of Grid Briquettes

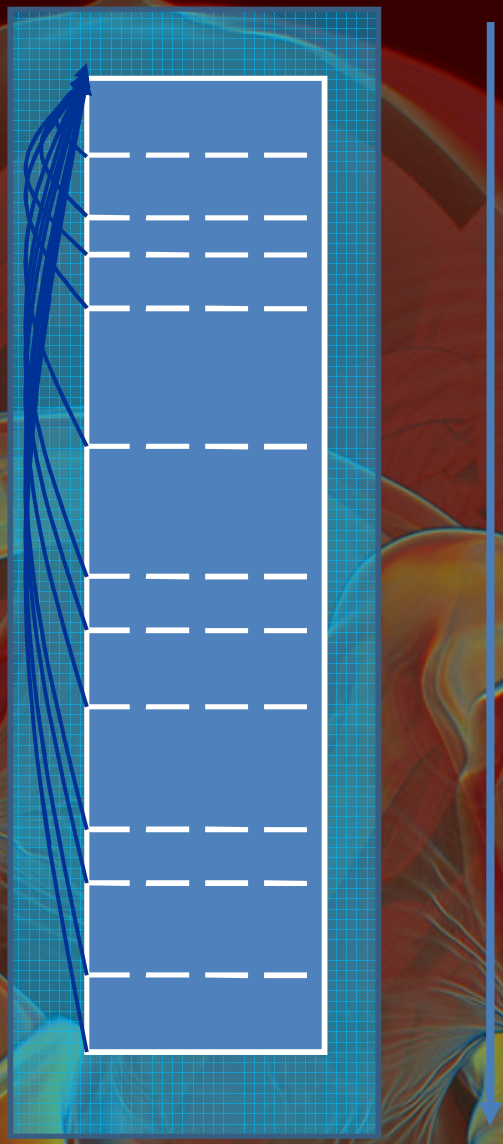
Each uninterruptible unit of work for a CPU core is:

1000 continue

- ✓ **Prefetch the next group of 16 grid briquettes.**
- ✓ **Unpack the previously fetched group of briquettes.**
- ✓ **Construct 32-word vectors from briquette records.**
- ✓ **Perform 32×1447 vector SIMD flops, with all vector operands aligned.**
- ✓ **Result is set of 4 updated grid briquettes.**
- ✓ **Pack new result vectors back into 4 grid briquette records, possibly transposing the contents.**
- ✓ **Write the 4 updated grid briquette records back to main memory.**
- ✓ **If more briquettes in strip, then go to 1000.**

Return to begin next strip of briquettes.


PPM
difference
stencil for
high-speed
flow
suggests
pipelined
updates of
grid pencils.
Planes of 4
briquettes
are
extracted,
updated,
replaced.

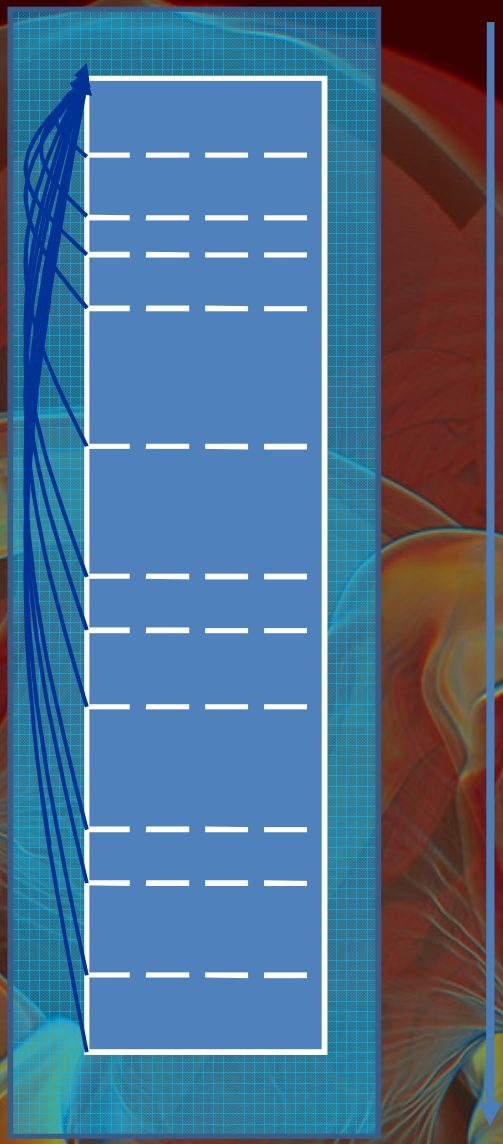


We can view the pipelined code as shown here.

The vertical dimension is the amount of work, which breaks into "phases" indicated by the dashed lines.

A the end of each phase we must, for the first time through, return to the beginning of the loop.

The horizontal dimension represents the number of results computed "simultaneously" in vector mode.

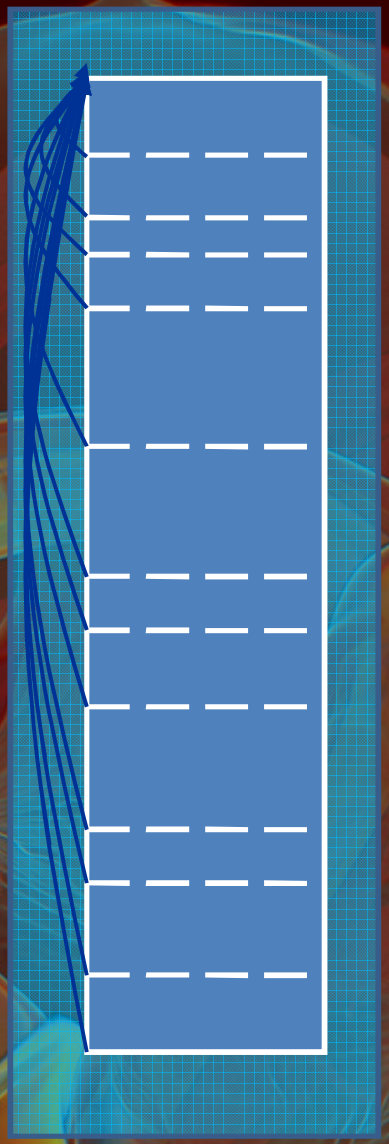


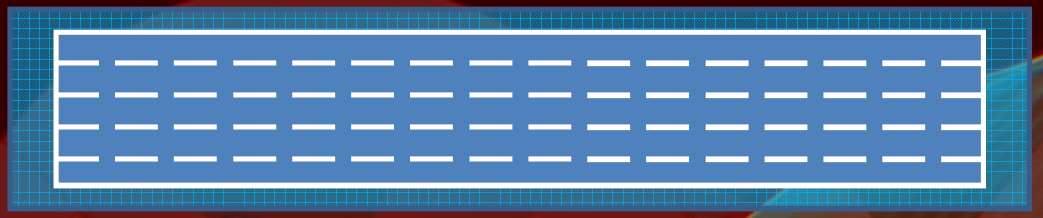
A phase ends, because to go further we need to have done this phase twice.

In the subsequent phase we reuse the results from 2 executions of the previous phase.

Data reuse also occurs within the body of each phase.

Data from main memory only enters at the top of the pipeline and exits at the bottom. This formulation demands a minimum of memory bandwidth.





Here we try to represent the difference between a Cell and a GPU pipe-lined algorithm.

The GPU code (top right) may have several phases, but it must have a greater vector length (width as shown here). The total flops (the area as shown here) must be very much less.

Benefits of Pipelined Operation:

- All accesses to main memory are in "atomic" units of grid briquette records, which are each 480 bytes.
- 16 briquette records trickle into cache while we update previous 4 briquettes in the sequence.
- All operands are 4 quadwords and all are aligned.
- Derivatives are evaluated in direction of the 1-D pass, which enables operand alignment to be preserved.
- Small number of transverse derivatives evaluated using specially constructed, aligned operands.
- Only the minimum amount of data required to update 4 grid briquettes resides in cache.
- Huge amount of on-chip data reuse, 4.82 flops/byte/core or 19.3 flops/word/core.
- Uses less than 10% of available memory bandwidth.

Summary / The tool on completion does :

- Inline all subroutines a single briquette update needs
- Pipeline over briquettes to avoid redundant unpacking of data records and redundant computation
- Fuse code blocks belonging to same pipeline stages
- Unroll outer loops over longitudinal dimension (strip direction)
- Prefetch
- Perform liveness analysis and further reindex temporaries
  - Retain minimal set of transverse grid planes needed for a single briquette update
- Optionally unroll inner loops over transverse direction to produce sequence of quadword SIMD operation

We will NOT handle everything for all codes.

- Programmers will have to annotate their codes, or at least the sections to be translated.
- Programmers will be responsible for the correctness of the directives they use. Incorrect usage is a program bug, not the translator's.
- Programmers will control decomposition of the algorithm into a sequence of calculations each of which will be pipelined.
- Translator will assume that all devices have roughly the same ratio of memory bandwidth to peak performance (which is true today).
- If entire algorithm is pipelined into a single loop, may need to spill too many "vector registers" and read them back later.
- If algorithm broken up into too many separate pipelined loops, on-chip data reuse reduced.
- Programmer experiments, using code translator.

Restrictions:

Simplifies analysis and tool building

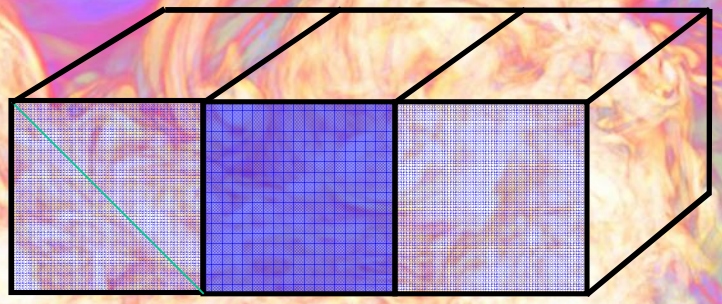
Examples:

- ***Simplify parsing***
  - **Integer lengths in type statements must be just constants => real*8 allowed, but not real*(8) or real*(4+4)**
- ***Simplify analysis***
  - **Unnamed common blocks not supported**
  - **Common block statements must appear all its members have been declared**
  - **Loops bounds must be constants in the transformation region**
  - **Scalar statements not allowed between the pipeline stages**

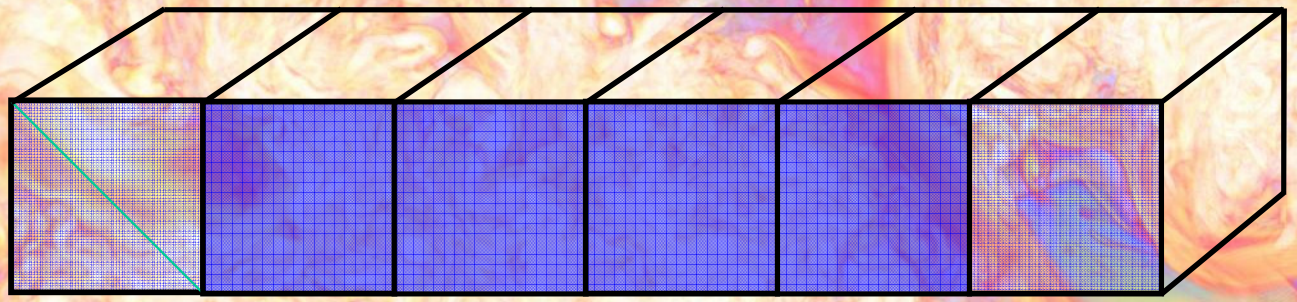
Doesn't impair programmability

1) **Input:**

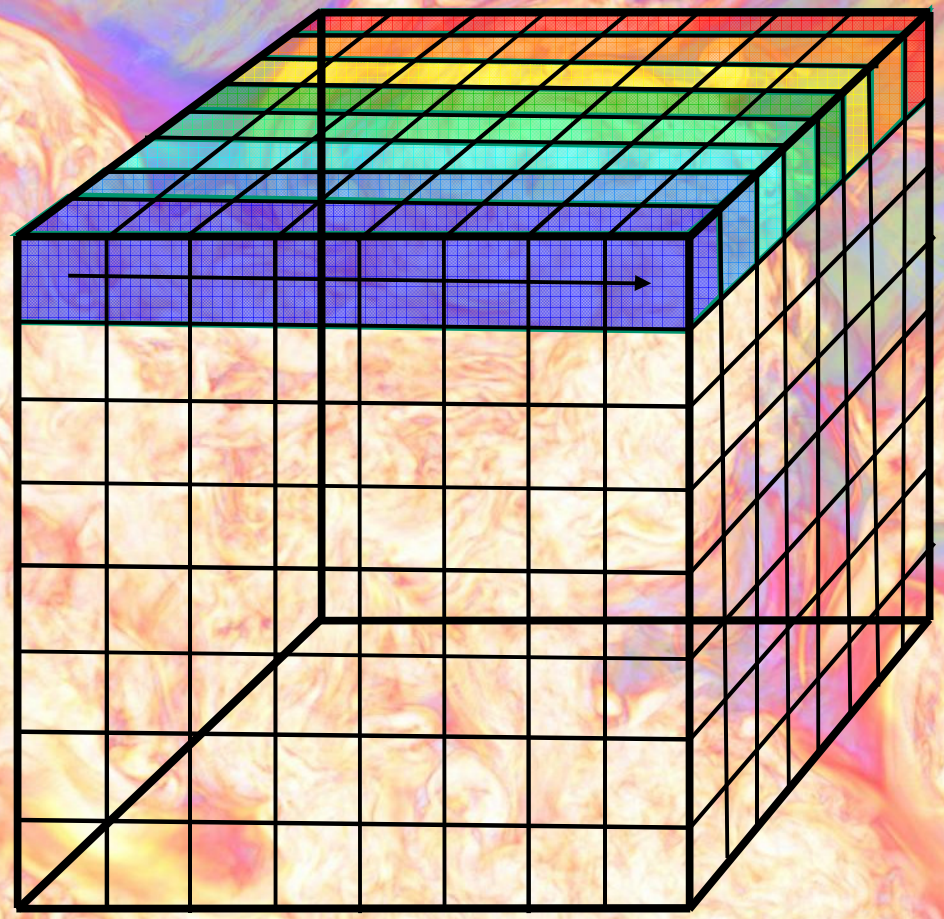
A program updating a sequence of grid briquettes



2) **The translator pipelines it**



8 cores (different colors) simultaneously update 8 strips of sugar cubes.



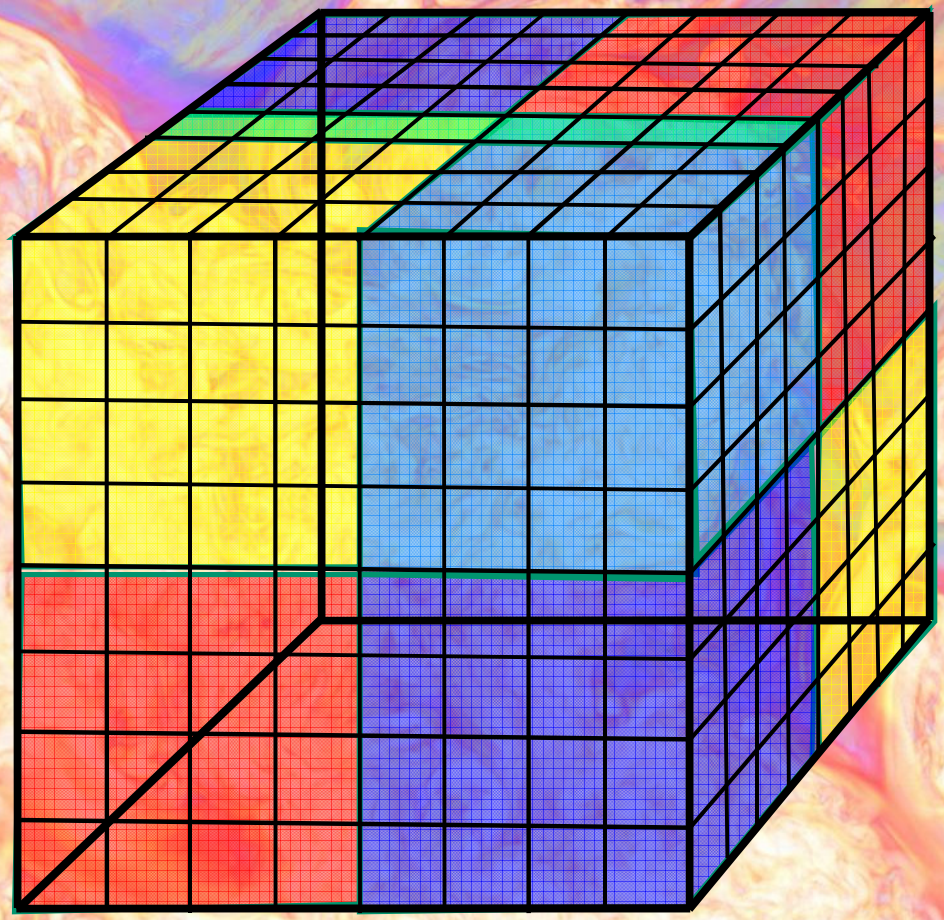
3)

OpenMP, or any equivalent threading model, is used to have multiple CPU cores cooperatively update a single grid brick.

Each core updates strips of grid briquettes (sugar cubes).

There is now a barrier synchronization among the core threads at the end of each grid brick update.

8 cores simultaneously update each of the 8 bricks in succession.



4)

At the beginning of each grid brick update, we receive 1 message in each of the 3 grid directions.
At the end of each grid brick update, we send 1 message in each of the 3 grid directions.

This messaging strategy allows all the worker threads to just keep right on working without ever stopping.

```
c     Output arrays from Fluxes:
c
      dimension       dvoll(nssq,1-nghostcells:nx+nghostcells)
      dimension      dmassl(nssq,1-nghostcells:nx+nghostcells)
c
c     Output arrays from CellUpdate:
c
      dimension       rhonu(nssq,1-nghostcells:nx+nghostcells)
c
c     Scratch arrays:
c
      dimension    thing01(nssq,1-nghostcells:nx+nghostcells)
      dimension    thing02(nssq,1-nghostcells:nx+nghostcells)
      dimension    thing03(nssq,1-nghostcells:nx+nghostcells)
c
c     Scratch arraylets:
c
      dimension    thngy01(nssq), thngy02(nssq), thngy03(nssq)
      dimension    thngy04(nssq), thngy05(nssq), thngy06(nssq)
```

Inside the principal routine, scratch storage on the stack is laid out as above in the Fortran-W version. This version is intended to be easy to write and maintain.

```
cPPM$ ELIMINATE REDUNDANT ITERATIONS
      do icget = icube-nghostcubes,icube+nghostcubes
c
c       write (6,*) 'icget =',icget
cPPM$ PREFETCH BEGIN
cPPM$ DOUBLEBUFFER D
        mycube = 1 + (icget-1)*incx - incy - incz
        do kcube = 1,nsugarcubes
        mycube = mycube + incz
        micube = mycube
        do jcube = 1,nsugarcubes
        micube = micube + incy
        do j = 1,nsugarcubed
          d(j,jcube,kcube) = dd(j,micube)
        enddo
        enddo
        enddo
c
        do i = 1,nsugar*2
        xlcube(i) = xxl(i,icget)
        enddo
cPPM$ PREFETCH END
```

In this principal Fortran-W routine, we do a loop on icube, and on each iteration we fetch a briquette record and the necessary ghost briquettes needed to enable the updating of the variables in this grid briquette. This data is unpacked and rearranged once we have it in the on-chip cache memory.

```
      call ppmintrf0vec (rho,
     &           unsmth,
     &           rhol,rhor,drho,rho6,
     &           dal,absdal,dasppm,damnot,alsmth,alunsm,
     &           thngy01,thngy02,thngy03,thngy04,thngy05,thngy06,
     &           sixth,crterr,ferrfc,small,
     &          adds,amults,recips,cvmgms,sqrts,rsqrts,exps,
     &          ifdebug,time,myrank,mythread,mbrick,
     &          iold,icube,jbq,kbq,ipass,
     &          MyBrickX,MyBrickY,MyBrickZ,
     &          NXBricks,NYBricks,NZBricks)
```

Once we have unpacked the grid briquette records and rearranged their contents in cache-resident arrays on the stack, we call a routine that takes this data and operates upon it using a very simple Fortran expression, which is the advantage of Fortran-W. We pass a temporary workspace on the stack to this routine in the form of the arrays thngy0x.

```
      subroutine ppmintrf0vec (a,
     &           unsmth,
     &           al,ar,da,a6,
     &           dal,absdal,dasppm,damnot,alsmth,alunsm,
     &           thngy1,thngy2,s,almon,armon,unsmooth,
     &           sixth,crterr,ferrfc,small,
     &          adds,amults,recips,cvmgms,sqrts,rsqrts,exps,
     &          ifdebug,time,myrank,mythread,mbrick,
     &          iold,icube,jbq,kbq,ipass,
     &          MyBrickX,MyBrickY,MyBrickZ,
     &          NXBricks,NYBricks,NZBricks)
```

From the perspective of the routine ppmintrf0vec, written in Fortran-W, our single grid briquette, with its ghost cells, constitutes the entire grid of the problem. We exploit the fact that in a parallel code, the updating of a subset of the problem domain follows the same algorithm and code expression as the update of the entire domain. This is a sort of self-similarity.

```
parameter (nsugar=nnsugar)
parameter (nsugarcubes=nnsugarcubes)
parameter (nbdy=nnbdy)
parameter (nbdy1=nbdy+nsugar-1)
parameter (nghostcubes=(nbdy1/nsugar))
parameter (nghostcells=nghostcubes*nsugar)
parameter (n=nsugar)
parameter (nx=nsugar)
parameter (ny=nsugar)
parameter (nz=nsugar)
parameter (nyy=ny*nsugarcubes)
parameter (nzz=nz*nsugarcubes)
parameter (nssq=nyy*nzz)
```

Here nsugar, the number of grid cells on each side of each grid briquette, has the value 4.

nbdy is the number of ghost cells required on each end of a 1-D grid strip for a 1-D pass of this algorithm. In this case it is also 4, which is especially felicitous.

The permitted formats make use of parameter statements like these. The programmer can choose whatever parameter names are desired, and set them, as in this example, using the C preprocessor, but they must evaluate to integer constants. When handed to the PPM code translator, the grid briquettes must be cubes, here called sugar cubes.

```
cPPM$  LOWERBOUND   1-nbdy
cPPM$  UPPERBOUND   nx+nbdy
cPPM$  LONGITUDINAL  LOOP
 do 7000   i = 4-nbdy,n+nbdy-3
!DEC$ VECTOR ALWAYS
c!DEC$ VECTOR ALIGNED
      do jk = 1,nssq
      al(jk,i) = alunsm(jk,i)
      ar(jk,i) = alunsm(jk,i+1)
      almon(jk) = 3. * a(jk,i)  -  2. * ar(jk,i)
      armon(jk) = 3. * a(jk,i)  -  2. * al(jk,i)
      if (((a(jk,i) - al(jk,i)) * (a(jk,i) - ar(jk,i))) .ge. 0.)  then
        al(jk,i) = a(jk,i)
        ar(jk,i) = a(jk,i)
        almon(jk) = a(jk,i)
        armon(jk) = a(jk,i)
      endif
      if (((ar(jk,i) - al(jk,i)) * (almon(jk) - al(jk,i))) .gt. 0.)
     &      al(jk,i) = almon(jk)
      if (((ar(jk,i) - al(jk,i)) * (armon(jk) - ar(jk,i))) .lt. 0.)
     &      ar(jk,i) = armon(jk)
da(jk,i) = ar(jk,i) - al(jk,i)
      a6(jk,i) = 6. * (a(jk,i)  -  .5 * (al(jk,i) + ar(jk,i)))
      enddo
7000  continue
```

The programmer communicates to the translator how this loop is to be inserted into a fully pipelined translation by means of the extents on the outer loop over i. The outer loop must run over the dimension considered as X in this routine – the direction of this 1-D pass.

```
!DEC$ VECTOR ALWAYS
c!DEC$ VECTOR ALIGNED
      do jk = 1,nssq
      rhol(jk,i16m03) = alunsm(jk,i16m03)
      rhor(jk,i16m03) = alunsm(jk,i16m02)
      almon(jk) = 3. * rho(jk,i16m03)  -  2. * rhor(jk,i16m03)
      armon(jk) = 3. * rho(jk,i16m03)  -  2. * rhol(jk,i16m03)
      if (((rho(jk,i16m03) - rhol(jk,i16m03))
     &   * (rho(jk,i16m03) - rhor(jk,i16m03))) .ge. 0.)  then
        rhol(jk,i16m03) = rho(jk,i16m03)
        rhor(jk,i16m03) = rho(jk,i16m03)
        almon(jk) = rho(jk,i16m03)
        armon(jk) = rho(jk,i16m03)
      endif
      if (((rhor(jk,i16m03) - rhol(jk,i16m03))
     &   * (almon(jk) - rhol(jk,i16m03))) .gt. 0.)
     &     rhol(jk,i16m03) = almon(jk)
      if (((rhor(jk,i16m03) - rhol(jk,i16m03))
     &   * (armon(jk) - rhor(jk,i16m03))) .lt. 0.)
     &     rhor(jk,i16m03) = armon(jk)
drho(jk,i16m03) = rhor(jk,i16m03) - rhol(jk,i16m03)
      rho6(jk,i16m03) = 6. * (rho(jk,i16m03)
     &           -  .5 * (rhol(jk,i16m03) + rhor(jk,i16m03)))
```

The Fortran-I intermediate form translation of the Fortran-W loop is unrolled 4 times and placed in a pipelined position in a single routine into which all subroutines have been inlined. The outer array indices are barrel shifted on each trip through the pipeline, so that precious space in the on-chip memory is optimally utilized.

```
!DEC$ VECTOR ALWAYS
c!DEC$ VECTOR ALIGNED
      do jk = 1,nssq
      rhol(jk,i16m03) = alunsm(jk,i16m03)
      rhor(jk,i16m03) = alunsm(jk,i16m02)
      almon(jk) = 3. * rho(jk,i16m03)  -  2. * rhor(jk,i16m03)
      armon(jk) = 3. * rho(jk,i16m03)  -  2. * rhol(jk,i16m03)
      if (((rho(jk,i16m03) - rhol(jk,i16m03))
     &   * (rho(jk,i16m03) - rhor(jk,i16m03))) .ge. 0.)  then
        rhol(jk,i16m03) = rho(jk,i16m03)
        rhor(jk,i16m03) = rho(jk,i16m03)
        almon(jk) = rho(jk,i16m03)
        armon(jk) = rho(jk,i16m03)
      endif
      if (((rhor(jk,i16m03) - rhol(jk,i16m03))
     &   * (almon(jk) - rhol(jk,i16m03))) .gt. 0.)
     &     rhol(jk,i16m03) = almon(jk)
      if (((rhor(jk,i16m03) - rhol(jk,i16m03))
     &   * (armon(jk) - rhor(jk,i16m03))) .lt. 0.)
     &     rhor(jk,i16m03) = armon(jk)
drho(jk,i16m03) = rhor(jk,i16m03) - rhol(jk,i16m03)
      rho6(jk,i16m03) = 6. * (rho(jk,i16m03)
     &           -  .5 * (rhol(jk,i16m03) + rhor(jk,i16m03)))
```

In the Fortran-I expression, each line of code is a 16-wide SIMD operation. When fully transformed, the outer indices are compressed to the minimum size of circular buffer of 16-element grid planes needed for each variable residing on chip. The loops on the index jk also all go away to give a simple, but enormous stream of SIMD instructions.

Implementation

- ✓ Built using ANTLR parser generator
  - Currently, supports FORTRAN77 input
  - Parser generates AST
  - Transformations implemented in multiple passes
  - Each pass modifies AST
  - Symbol tables aid the process
- ✓ and StringTemplate template engine
  - Generates high performance FORTRAN output

We have a whole another back-end tool chain to generate architecture specific ports.

- C with (SIMD) intrinsics for different architectures.
- Current ports include Cell, Altivec (Power7), SSE

Results

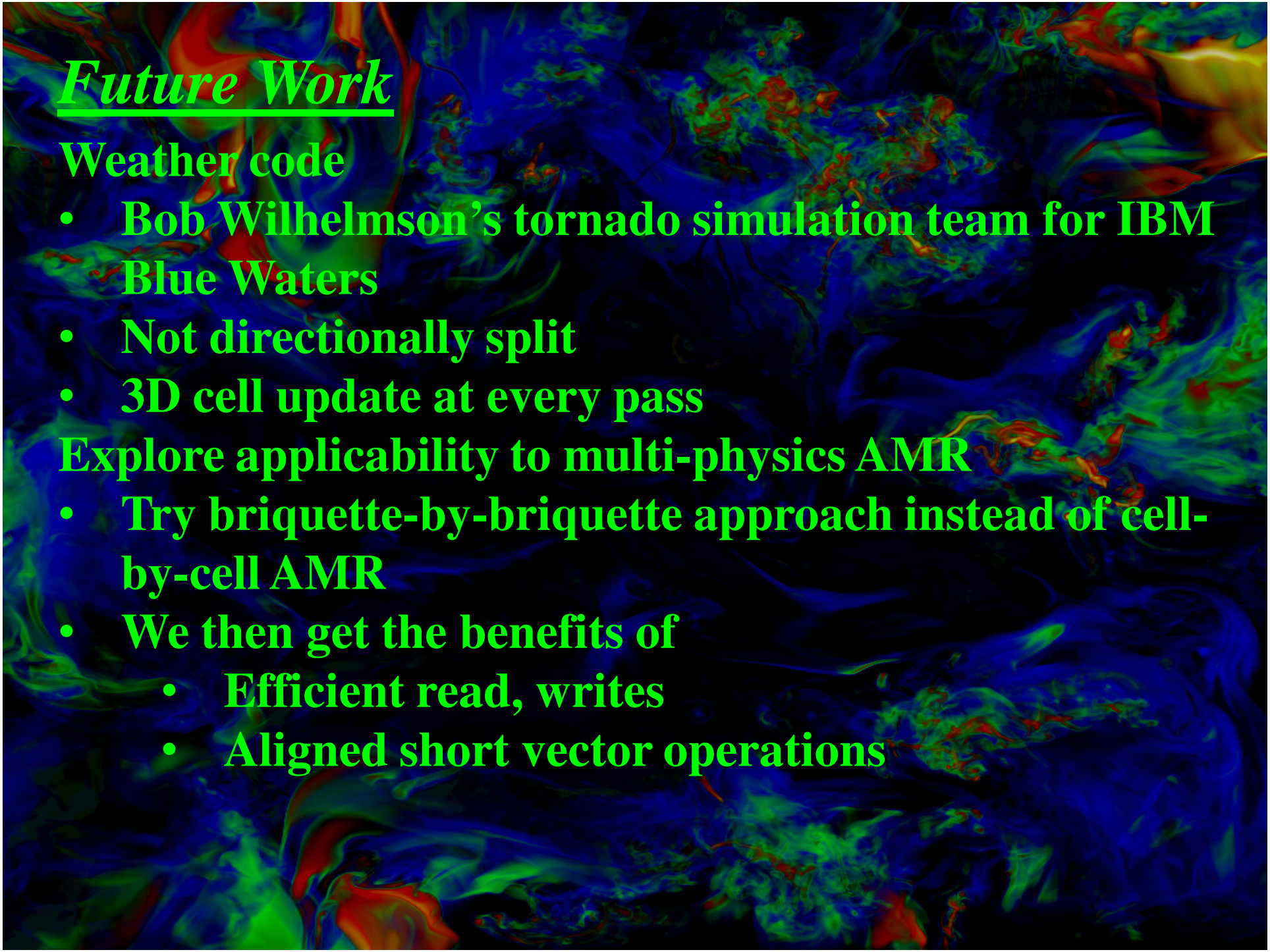
Reciprocals counted as 3 flops

Multifluid PPM

- **Nehalem cluster**
  - ✓ **5.5 Gflop/s/core (23% of 32-bit peak)**
- **Cell (pure 32-bit)**
  - ✓ **4.5 Gflop/s/SPU on 1 Cell processor**
  - ✓ **3.4 Gflop/s/SPU on 1440 Cell processors**
- **Cell (Mixed 32-bit and 64-bit)**
  - ✓ **2.04 Gflop/s/SPU on 24 Cell processors in our lab's RR tri-blades**
  - ✓ **1.37 Gflop/s/SPU on 7168 Cell processors**

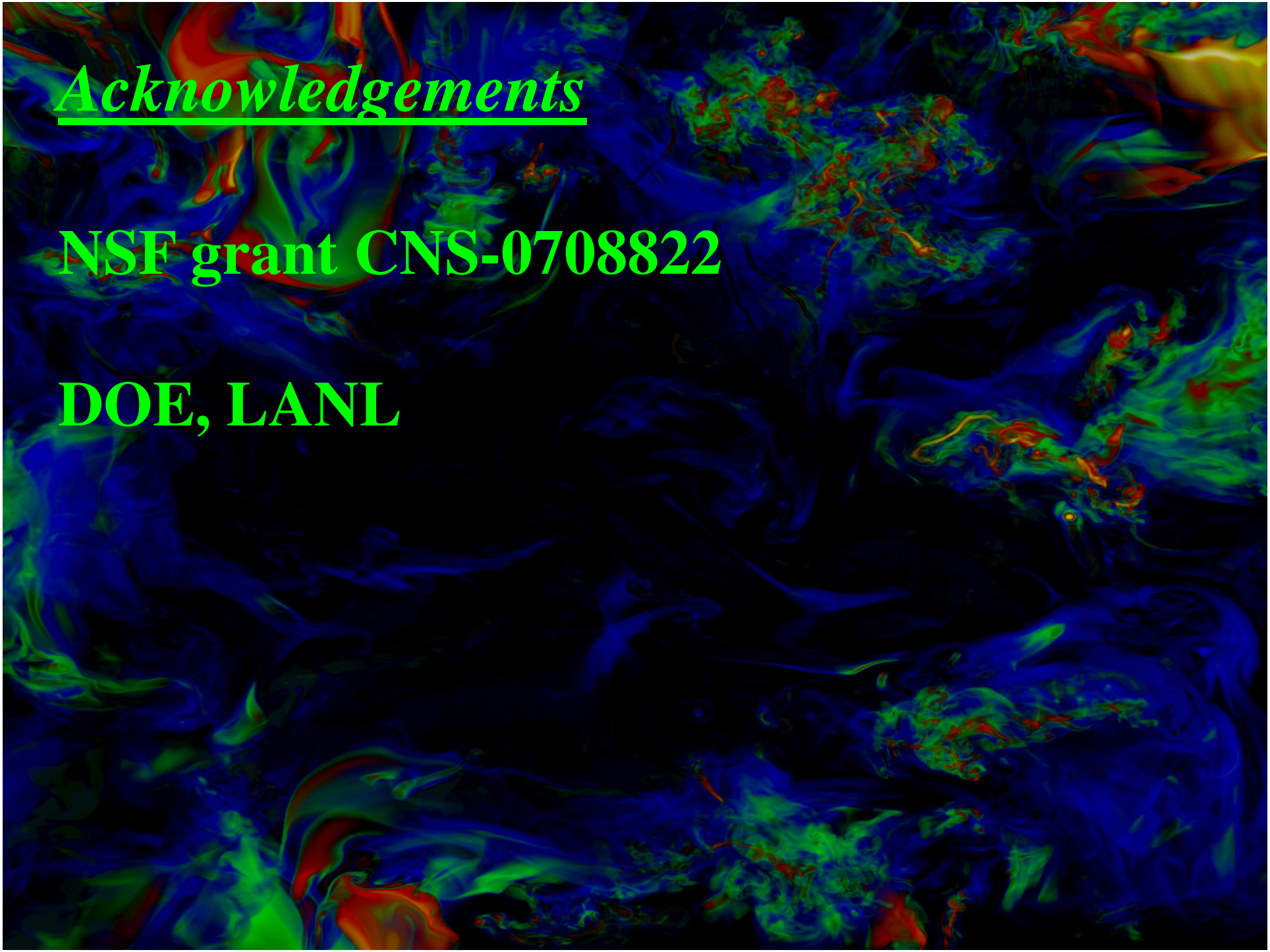
PPM advection

- ✓ **5.64 Gflop/s/core on our Nehalem cluster**
- ✓ **6.28 Gflop/s/core self-reported**
- ✓ **Working with Intel folks to boost it up to 50% of 32-bit peak**


Future Work
Weather code
• Bob Wilhelmson's tornado simulation team for IBM Blue Waters
• Not directionally split
• 3D cell update at every pass
Explore applicability to multi-physics AMR
• Try briquette-by-briquette approach instead of cell-by-cell AMR
• We then get the benefits of
• Efficient read, writes
• Aligned short vector operations

Conclusions

- Pipelining enables high computational intensity and execution speed (23% of 32-bit peak on Nehalem).
- Granularity is dramatically reduced, because there is now very much more work in each episode in order to make it efficient. Allows efficient scaling to over a million cores.
- Code is readable, but you would never agree to write it this way, and even if you did, you could not maintain it in this form with a reasonable level of effort
- Need automatic code translators to convert simple expression to high performance expression and platform specific code expression (C with SIMD intrinsics, CUDA)
- Tools come with reasonable restrictions to insure reasonable tool development efforts.



Acknowledgements

NSF grant CNS-0708822

DOE, LANL

References

1) P.R. Woodward, J. Jayaraj, P.-H. Lin, G.M. Rockefeller, C.L. Fryer, Guy Dimonte, W. Dai, R.J. Kares, "Simulating Rayleigh-Taylor (RT) instability using PPM hydrodynamics @ Scale on Roadrunner," Proceedings NECDC 2010 conference, Los Alamos National Laboratory, Oct. 2010. Preprint available at www.lcse.umn.edu/NECDC2010
2) Woodward, P.R., J. Jayaraj, P.-H. Lin, and P.-C. Yew, "Moving Scientific Codes to Multicore Microprocessor CPUs," *Computing in Science & Engineering*, special issue on novel architectures, Nov. 2008, p. 16-25. Preprint available at www.lcse.umn.edu/CiSE.
3) Woodward, P.R., J. Jayaraj, P.-H. Lin, and W. Dai, "First Experience of Compressible Gas Dynamics Simulation on the Los Alamos Roadrunner Machine," *Concurrency and Computation: Practice and Experience*, June 2009, vol. 21, p. 2160-2175.