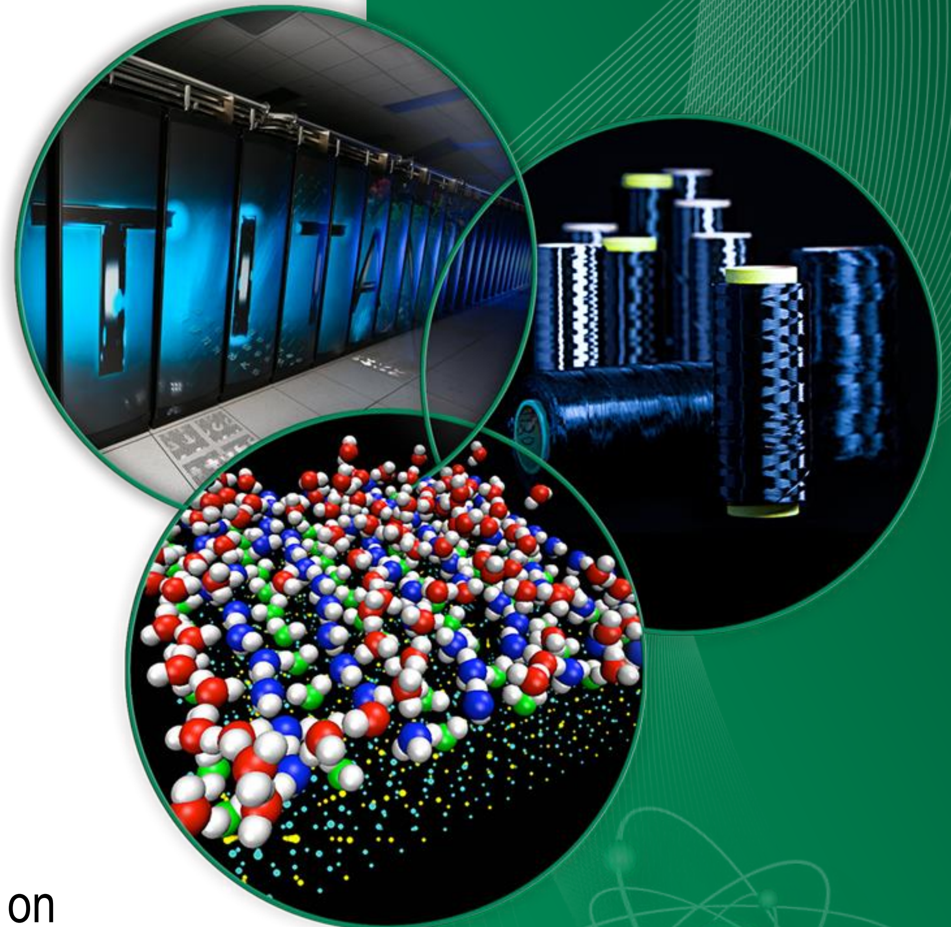
HSLOT: The HERCULES Scriptable Loop Transformations Engine

Christos Kartsaklis (ORNL)

Eunjung Park (UD)

John Cavazos (UD)



November 17, 2014

WOLFHPC: Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing

In conjunction with SC14: The International Conference for High Performance Computing, Networking, Storage, and Analysis

Introduction

- DoE application challenges in accelerator & exascale era
  - Large production-grade simulation codes need porting
  - Reengineering challenges going beyond parameterization
  - Node design to change again; memory types, etc.
- Different tools researched
  - Language extensions, DSL construction/elevation kits, …
- ORNL HERCULES
  - User-definable/extensible compiler infrastructure
  - Compiler-supported analysis/transformation tools

Directive APIs

- Compiler directives
  - Prescriptive or descriptive (hints) instructions to the compiler
  - `#pragma NAMESPACE` (C) and `!$NAMESPACE` (Fortran)
  - Original code annotated with directives, but unchanged otherwise
- Many directive-based parallelization APIs
  - HPF, OpenMP, OmpSs, OpenACC, hiCUDA, …
  - Often standardized & adopted by many compilers
  - Elegant, annotating:

```
#pragma omp parallel for reduction(+:r)
for (i=0 ; i<n ; i++) { ...; r+=...; }
```

- Compilers offer their own optimization directives:
  - The user may know more than what the compiler can deduce
  - GCC `Loop_Optimize`, Cray Fortran Compiler (`SHORTLOOP`, `FUSION`, etc.), Intel (`ivdep`, `loop_count`, etc.)

HSLOT

- **H**ERCULE**S** **Lo**op **T**ransformations Engine
  - A loop transformations directives API & a supporting compiler
- A new loop transformations directives API
  - About 20 different types of transformation clauses, each with its own configuration options
  - Loop nest/stack & iteration space oriented
  - Simplify codegen mainly, not optimal transformation selection
- HSLF90: An Open64-based directives-implementing compiler
  - Non-validating (trust the user) model
  - Minimal legality checking (VHO D/F & tree constraints)
  - HSLOT happily coexists with –O3 strategies

Programming Model

Original program

```
do i = 1, N
   do j = 1, K
   … ! body 1
   end do
end do
do r = 1, N*K
   … ! body 2
end do
```

Tag Loops

```
!$hslot loop(L1)
do i = 1, N
!$hslot loop(L2)
   do j = 1, K
   … ! body 1
   end do
end do
!$hslot loop(L3)
do r = 1, N*K
   … ! body 2
end do
```

Specify Transformations

```
!$hslot loop(L1)
!$hslot collapse(L1, L2, LC), &
        fuse([LC,L3],same,L3)
do i = 1, N
!$hslot loop(L2)
   do j = 1, K
   … ! body 1
   end do
end do
!$hslot loop(L3)
do r = 1, N*K
   … ! body 2
end do
```

```
L3: do q = 1, N*K
        … ! body 1
        … ! body 2
    end do
```

- Identify loops of interest and tag them with a unique ID
- Provide the list of transformations, passing these and new IDs around
- Transformation directives are always processed in the order that they appear in the text (left to right, top to bottom).

Transformation Steps (1/2)

```
L1: do i = 1, N
L2:   do j = 1, K
        … ! body 1
      end do
    end do
L3: do r = 1, N*K
      … ! body 2
    end do
```

collapse(L1,L2,LC)

```
LC: do q1 = 1, N*K
      … compute i & j
      … ! i-only specifics
      … ! body 1
      … ! i-only specifics
    end do

L3: do r = 1, N*K
      … ! body 2
    end do
```

- Loops L1 & L2 invalidated
- ID LC refers to the collapsed product

Transformation Steps (2/2)

```
LC: do q1 = 1, N*K
      … compute i & j
      … ! i-only specifics
      … ! body 1
      … ! i-only specifics
    end do

L3: do r = 1, N*K
      … ! body 2
    end do
```

```
fuse([LC,L3],same,LC)
```

```
LC: do q2 = 1, N*K
      … compute i & j
      … ! i-only specifics
      … ! body 1
      … ! i-only specifics
      … ! body 2
    end do
```

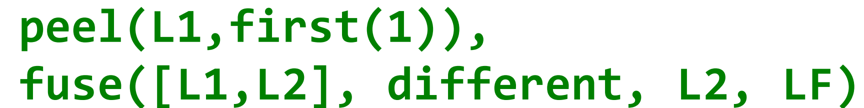
- Loops `LC` & `L3` invalidated
- ID `LC` now refers to the new loop

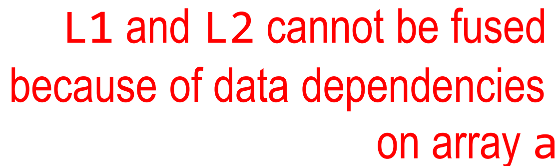
Example1: Merging Non-fusible Loops

- Combining g `peel` and `fuse` to deal with data dependencies

```
$hslot loop(L1)
do i=2,n-1
 S1: b(i) = … * a(i-1) + a(i) + a(i+1)
end do

$hslot loop(L2)
do j=2, n-1
 S2: a(j) = b(j)
end do
```






```
do k=2,n-1
 S1: b(k) = …* a(k-1) +a(k)+ a(k+1)
 S2: a(k) = b(k)
end do
```

```
b(2) = … * a(1) + a(2) + a(3)
a(2) = b(2)
b(3) = … * a(2) + a(3) + a(4)
a(3) = b(3)
```

```
    b(2) = … * a(1) + a(2) + a(3)
LF: do k1=3,n-1
     S1: b(k1) = … * a(k1-1) + a(k1) + a(k+1)
     S2: a(k1-1) = b(k1-1)
    end do ! here k1=(n-1)+1
    do k2=k1, n-1    ! trip count is 0
     S1: b(k2) = …
    end do
L2: do k2=(k1-1), n-1 !
     S2: a(k2) = b(k2)
    end do
```

Example2: Determining the Unroll Factor Dynamically

- Using `fork` for testing trip count at runtime and unrolling accordingly

```
!$hslot loop(L)
do i=1,N
  … ! body
end do
```

```
fork(L,multiple(4),LMOD4,LMOD3),
unroll(LMOD4,4),
fork(LMOD3,multiple(3),LMOD3,L),
unroll(LMOD3,3)
```

```
LMOD4: if (MOD(N,4).eq.0) then
         do i=1,N,4
           … ! body @ iteration i
           … ! body @ iteration i+1
           … ! body @ iteration i+2
           … ! body @ iteration i+3
         end do
       else
LMOD3:   if (MOD(N,3).eq.0) then
           do i=1,N,3
             … ! body @ iteration i
             … ! body @ iteration i+1
             … ! body @ iteration i+2
           end do
         else
L:         do i=1,N
             … ! body
           end do
         endif
       endif
```

Transformations

- Common & unique/customizable set of transformations
  - Emerged by talking to users and by trying to recreate transformed code found in literature.

# of Input Loops Processed	*Transformation Primitives*
Single Loop	`unroll`, `split`, `fork`, `peel`, **`nest`**, `openmp`, **`fission`**, `align`, `gpurr`, `hoistsa`, `ivarep`, `normalize`, `stride`
Double Loops	`collapse`, **`fuse`**, `swpnest`, `interch`
Multiple Loops	**`tile`**, **`specialize`**

Fuse

- Fuses n arbitrary loops together
  - Controlling insertion point (3rd argument)
  - Manage uneven trip counts (2nd argument)

```
!$hslot loop(L1)
do I=1,N1
  DX(I)=DA*DX(I)
end do
call bar();
!$hslot loop(L2)
do J=1,N2
  DY(J)=DY(J)*DX(J)
end do
```

`fuse([L1,L2],same,L2,LF)`

```
     call bar();
     N=N1
LF:  do P=1,N
       I=P
       DX(I)=DA*DX(I)
       J=P
       DY(J)=DY(J)*DX(J)
     end do
```

`fuse([L1,L2],different, L2,LF,[L1R,L2R])`

```
     call bar();
     NMIN=MIN(N1,N2)
LF:  do P=1,NMIN
       I=P
       DX(I)=DA*DX(I)
       J=P
       DY(J)=DY(J)*DX(J)
     end do
L1R: do I=NMIN+1,N1
       DX(I)=DA*DX(I)
     end do
L2R: do J=NMIN+1,N2
       DY(J)=DY(J)*DX(J)
     end do
```

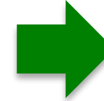
Nest

- Different strategies for creating loop nests

```
!$hslot loop(L), &
 nest(L, simple(J,1,M,2), LNEW)
do I = 1, N
  DX(I) = DA*DX(I)
end do
```

This variable must not exist

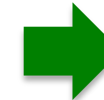
simple



```
do J = 1, M, 2          ! LNEW
  do I = 1, N           ! L
    DX(I) = DA*DX(I)
  end do
end do
```

```
!$hslot loop(L), &
 nest(L, interleave(J,4), LNEW)
do I = 1, N
  DX(I) = DA*DX(I)
end do
```

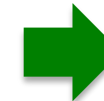
interleave



```
do J = 1, 4             ! LNEW
  do I = 1+J, N, 4      ! L
    DX(I) = DA*DX(I)
  end do
end do
```

```
!$hslot loop(L), &
 nest(L, pick([L1,L2]), LNEW)
do I = 1, N
  … ! block
end do
!$hslot loop(L1)
do J = 1, K
  …
end do
!$hslot loop(L2)
do Q = 1, M
  …
end do
```

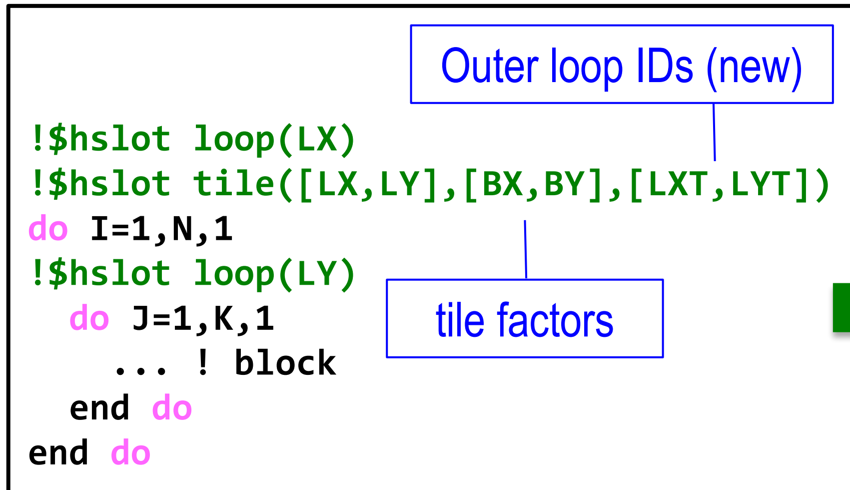
pick



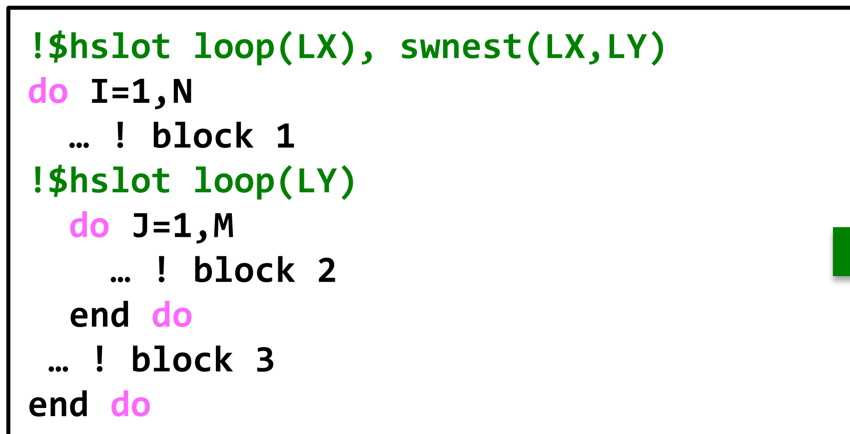
```
do J = 1, K             ! LNEW
  do Q = 1, M
    do I = 1, N         ! L
      … ! block
    end do
  end do
end do
do J = 1, K             ! L1
  …
end do
do Q = 1, M             ! L2
  …
end do
```

Tile & swnest (1/2)

- `tile` - blocking for a multi-level memory hierarchy, simple loop nest expected
- `swnest` – helper tool for swapping nests




```
LXT: do I0=1,N,BX
LYT:   do J0=1,K,BY
LX:      do I=I0,min(I0+BX-1,N),1
LY:        do J=J0,min(J0+BY-1,K),1
             ... ! block
           end do
         end do
       end do
     end do
```



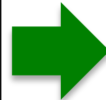

```
LY: do J=1,M
LX:   do I=1,N
        … ! block 1
        … ! block 2
        … ! block 3
      end do
    end do
```


Oak Ridge National Laboratory
MANAGED BY UT-BATTELLE FOR THE U.S. DEPARTMENT OF ENERGY

Tile & swnest (2/2)

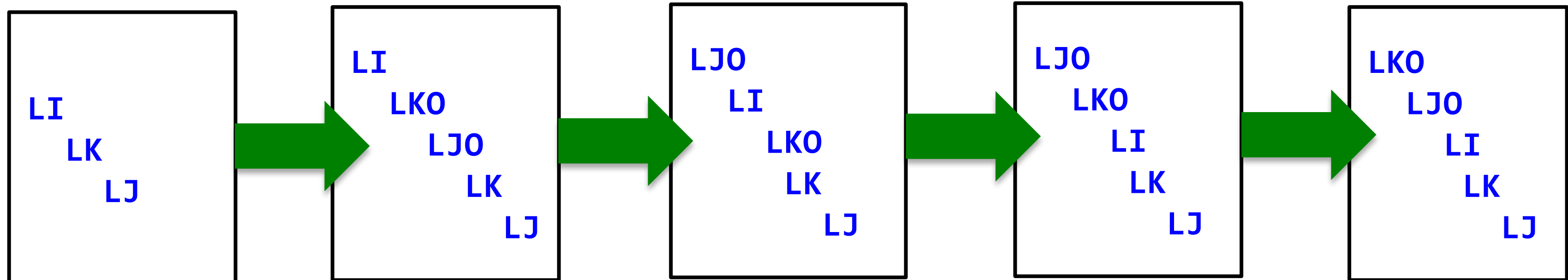
```
tile([LK,LJ],[KBF,JBF],[LKO,LJO]),
swnest(LI,LJO), swnest(LI,LKO), swnest(LJO,LKO)
```

```
!$hslot loop(LI)
do I=1,N
!$hslot loop(LK)
  do K=1,N
    Z = B(K,I)
!$hslot loop(LJ)
    do J=1,N
      C(J,I) = C(J,I) + Z*A(J,K)
    end do
  end do
end do
```



```
LKO: do K0=1,N,KBF
LJO:   do J0=1,N,JBF
LI:      do I=1,N
LK:        do K=K0,MIN(K0+KBF-1,N)
             Z = B(K,I)
LJ:          do J=J0,MIN(J0+JBF-1,N)
               C(J,I) = C(J,I) + Z*A(J,K)
             end do
           end do
         end do
       end do
     end do
```

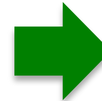
- Transforming a naive matrix multiplier by tiling for reuse (factors as variables – target knowledge necessary).



Specialize

- Performs a conditional or unconditional constant propagation
- User specifies variable names & constants

```
!$hslot specialize([L1,L2], true,          &
                   [BETA,ALPHA],[1.0,0.0])
do J=1,N
!$hslot loop(L1)
  do I=1,M
    C(I,J)=BETA*C(I,J)
  end do
  do L=1,K
    TEMP=ALPHA*B(L,J)
!$hslot loop(L2)
    do I=1,M
      C(I,J)=C(I,J)+TEMP*A(I,L)
    end do
  end do
end do
```



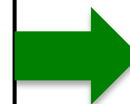
```
do J=1,N
  if ( (BETA  .eq. 1.0) .and. &
       (ALPHA .eq. 0.0)) then
    do I=1,M
      C(I,J)=1.0*C(I,J)
    end do
    do L=1,K
      TEMP=0.0*B(L,J)
      do I=1,M
        C(I,J)=C(I,J)+TEMP*A(I,L)
      end do
    end do
  else
    … ! Original body
  end if
end do
```

Can be simplified and/or eliminated

Fission

- Loop fission using a nested loop statements bookmark approach
- Loop body dissected around these bookmarks, hints control which new nest instance statements end up in
- For every body statement, if bookmark encountered: before (dump queue, add), after (add, dump queue), both (dump queue, add, dump), none (add)

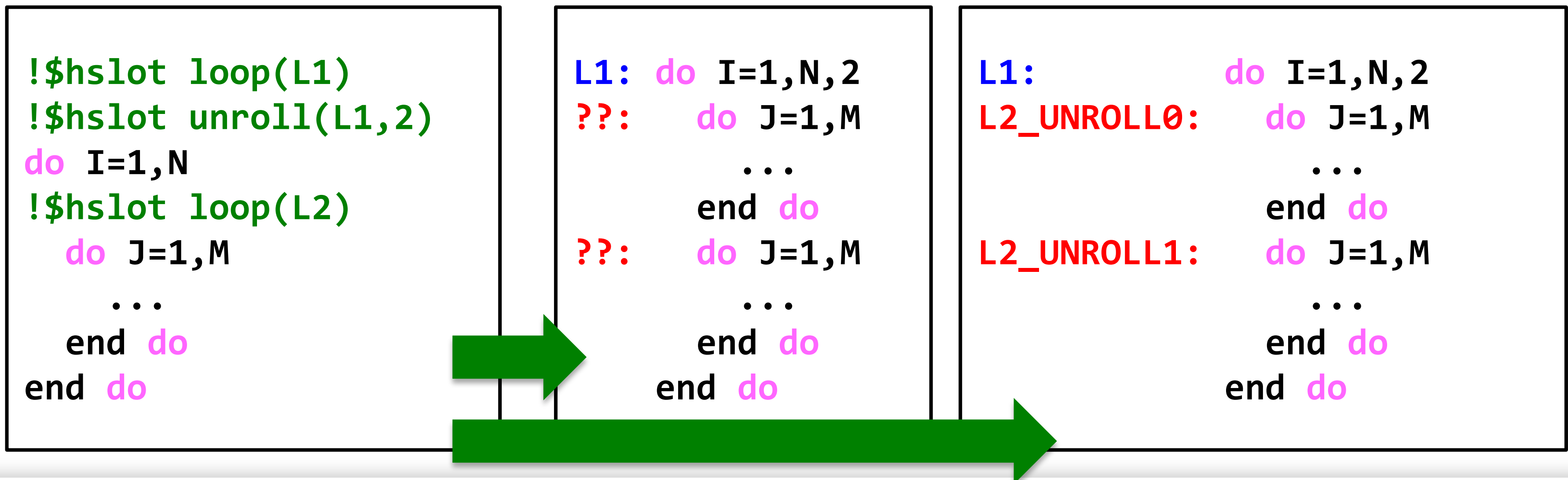
```
!$hslot loop(L)
!$hslot fission(L, [L1,L2], [both,none],
                   [LI1,LI2])
do I=1,N
  DX(I)=DA*DX(I)
!$hslot loop(L1)
  do J=1,K
    … ! block 1
  end do
!$hslot loop(L2)
  do M=1,K
    … ! block 2
  end do
  … ! block 3
end do
```



```
      do I=1,N
        DX(I)=DA*DX(I)
      end do
L1I:  do I=1,N
L1:     do J=1,K
          … ! block 1
        end do
      end do
L2I:  do I=1,N
L2:     do M=1,K
          … ! block 2
        end do
        … ! block 3
      end do
```

Programming Model - ROC

- Problem: what happens to tagged loops' IDs during duplication?
- Re-Tag on Copy (ROC): automatically assign new identifiers during copying of loops – predefined patterns:
  - Unroll: `_UNROLLx`, where x is the body copy (0-based).
  - Fork: `_ELSE` for the "else" branch
  - Peel: `_PREx` and `_PSTx` for the first/last iter peeled (0-based)
  - Specialize: `_SPEC` for the specialized part



HSLF90 Tag Tracking / Debugging

- HSLF90 tracks tag lifetime
- What the user has declared
  - Report on attempts to use undeclared tags
- What was once a valid tag, then destroyed
  - Report on outdated tag uses

```
hslf90 -c fork-unroll-test.f90
... // transformed source dump
slot: (V) finished with 'fork(L,multiple(4),LMOD4,L)'.
slot: (W) ignoring unnecessary normalization of 'LMOD4'
slot: (V) finished with 'normalize(LMOD4,LMOD4)'.
... // transformed source dump
slot: (V) finished with 'unroll(LMOD4,4)'.
slot: (E) reason: alias 'LX' not bound to a loop
slot: (E) transformations stack trace:
slot: (E)   fork(LX,multiple(3),LMOD3,L)
```

Early Results (1/2)

- 11 programs from Polybench Fortran V2.1
  - Manually optimized with HSLOT (EunJung, UD)
- Dual Xeon E5-2650 system (8/16; 20MB LLC)
- HSLF90 is modified Open64 Fortran 4.2.4 (openf90)
- Comparing baseline (–O3) to HSLOT directives + –O3
  - HSLOT is VHO, so no interference with LNO but interesting to see effects of heavily/radically modified loops & LNO
- Serial & threaded versions (8, 16 & 32 counts)

Early Results (2/2)

Kernel	Config	Serial	8 Thr.	16 Thr.	32 Thr.	Transformation Choice/Count
2mm	NI,NJ,NK,NL=1024	1.21x	2.18x	1.76x	4.03x	peel(2), fuse(3), openmp
3mm	NI,NJ,NK,NL,NM= 1024	1.15x	1.29x	1.43x	2.05x	peel(2), fuse(3), openmp
correlation	N,M=512	0.62x	-	-	-	fission(2), unroll(1), peel(2), fuse(2)
doitgen	NQ,NR,NP=128	1.04x	8.24x	8.29x	16.5x	unroll, openmp
dynprog	LEN=50, TSTEPS=10000	1.17x	1.74x	1.39x	1.23x	unroll(2), fuse, openmp
fdtd-2D	NX,NY=1000	1.08x	-	-	-	peel(3), fuse(2)
gemm	NI,NJ=1024	1.05x	6.65x	7.58x	14.56x	unroll, fuse(2), openmp
jacobi-1D	N=10000, TSTEPS=100	1.14x	-	-	-	peel(2), fuse, unroll
jacobi-2D	N=9000, TSTEPS=10	1.32x	3.36x	3.41x	2.23x	unroll(2), fuse(4), peel(2), openmp
mvt	N=4000	2.48x	4.38x	4.48x	8.29x	unroll(2), openmp(2)
seidel-2D	N=3200, TSTEPS=4	1.06x	7.41x	13.52x	26.96x	unroll, fuse, openmp
GMEAN		1.15x	2.47x	2.57x	3.47x	

Thank you!

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Split

- Different strategies for splitting the iteration space

```
$hslot loop(L1)
$hslot split(L, ???, L1, L2)
do I=1,N
  DX(I) = DA*DX(I)
end do
```

```
L1: do I0=1,MOD(N,4)
      DX(I0) = DA*DX(I0)
    end do
L2: do I=MOD(N,4)+1,N
      DX(I) = DA*DX(I)
    end do
```

```
L1: do I=1,N,2
      DX(I) = DA*DX(I)
    end do
L2: do I=2,N
      DX(I) = DA*DX(I)
    end do
```

```
L1: do I=1,N/2
      DX(I) = DA*DX(I)
    end do
L2: do I=(N/2)+1,N
      DX(I) = DA*DX(I)
end do
```

Peel & openmp

```
peel(L, first(2), last(1))
```

```
$hslot loop(L)
do I=1,N
  C(I) = BETA*C(I)
end do
```

```
C(1)=BETA*C(1)
C(2)=BETA*C(2)
do I=3,N-1
  C(I) = BETA*C(I)
end do
C(N)=BETA*C(N)
I=N
```

```
openmp(for,L)
```

The loop index variable will be `lastprivate`, arrays become `shared` and everything else is set to `firstprivate`.

```
!$omp parallel do lastlocal(I), &
      shared(C), firstprivate(BETA,N)
do I=1,N
  C(I) = BETA*C(I)
end do
```