Auto-tuning Towards to Post Moore’s Era: Adapting a new concept from FLOPS to BYTES

Takahiro Katagiri

(Information Technology Center, Nagoya University)

Collaborative work with
Masaharu Matsumoto and Satoshi Ohshima
(Information Technology Center, The University of Tokyo)

First International Workshop on Deepening Performance Models for Automatic Tuning (DPMAT)
Date: September 7th (Wed), 2016, 14:00-14:30
Place: Room IB-014, IB Building (Integrated Building), Higashiyama Campus, Nagoya University
Outline

• Background

• An Auto-tuning (AT) Language: *ppOpen-AT* and Adapting AT to an FDM code

• Performance Evaluation with the FX100

• Conclusion
Outline

• Background
• An Auto-tuning (AT) Language: \textit{ppOpen-AT} and Adapting AT to an FDM code
• Performance Evaluation with the FX100
• Conclusion
Symposium on “New Frontiers of Computer & Computational Science towards Post Moore Era”

Source: http://www.cspp.cc.u-tokyo.ac.jp/p-moore-201512/

● Planned by:
  Prof. Satoshi Matsuoka @TITECH

● December 22th, 2015@U.Tokyo
  ● Hosted by:
    ITC, U.Tokyo and GSIC, TITECH
  ● Co-hosted by:
    ITC, Hokkaido U., ITC, Kyusyu U.,
    AICS, RIKEN, JST CREST, JHPCN

● Targets:
  ● Hardware
  ● System Software
  ● Algorithm & Applications
    (PI: kengo Nakajima@ITC, U.Tokyo)

Co-located with SC16 in Salt Lake City,
Post-Moore's Era Supercomputing
(PMES) Workshop!

Japanese:
「ポストムーアに向けた計算機科学・計算科学の新展開」シンポジウム

New Frontiers of Computer & Computational Science towards Post Moore Era

概要

2015年12月22日（火）10:00開会

協賛：東京大学, 一橋大学, 立命館大学, 東京工業大学, 三菱電機

協賛：東京大学

協賛：北九州大学, 九州大学, 理化学研究所

協賛：計算科学技術研究飛躍拡大戦略

協賛：科学技術振興機構CREST, ポストモーア技術

協賛：日本高性能計算振興機構

1960年代にコンピュータの発展が進んだ「ムーアの法則」（LSIの増大は性能を指数的に増加する）によるCPU性能の指数的成長がこの数十年間持続され、科学技術の発展、社会インフラの充実に大きな貢献し、
AT Technologies in Post Moore’s Era

- It is expected that Moore’s Low is broken around end of 2020.
  - End of “One-time Speedup”: Many Cores, Wiring miniaturization to reduce power.
    → It cannot increase FLOPS inside node.
- However, memory bandwidth inside memory can increase by using “3D Stacking Memory” Technologies.
- 3D Stacking Memory:
  - It can increase bandwidth for Z-Direction (Stacking distance), and keeping access latency (→ High performance)
  - It can be low access latency for X-Y directions.
- Access latency between nodes goes down, but bandwidth can be increased by optical interconnection technology.
- We need to take care of new algorithms with respect to ability of data movements.

Reconstruction of algorithms w.r.t.
- Increase memory bandwidth
- Increase local memory amounts (caches)
- Non-Uniform memory accesses latency

Issues in AT Technologies.
- Hierarchical AT Methodology
- Reducing Communication Algorithm.
- New algorithms and code (algorithm) selection utilizing high bandwidth ability.
  - Rethink classical algorithms.
  - Non-Blocking Algorithms.
  - From explicit method to implicit method.
  - Out of Core algorithms (Out of main memory)
Development Flow of HPC Software

1. Phase of Specification

2. Phase of Programming

3. Phase of Optimization

4. Phase of Database and Knowledge of Discovery for Tuning

Increase of Number of Cores, Programming Models, and Code Optimizations (Architecture Kinds)

Database for Tuning Knowledge

Code Generation

Compile and Run

Analyzing of Results

Target Computers

Increase of Number of Cores, Programming Models, and Code Optimizations (Architecture Kinds)
A Motivating Example (An simulation based on FDM)

This indicates:
- Significant variation in hybrid MPI/OpenMP execution.

-> Tuning of codes in each execution is time-consuming to find the best execution.

Hybrid MPI/OpenMP
Xeon Phi : 8 Nodes
(1920 Threads)

Total Execution Time

Types of hybrid MPI/OpenMP

P8T240, P16T120, P32T60, P64T30, P128T15, P240T8, P480T4
**ppOpen-AT System** (Based on FIBER 2),3),4),5) 

Library Developer

User Knowledge

**ppOpen-AT** Directives

**ppOpen-APPL /*

Before Release-time

Automatic Code Generation

Selection

This user benefited from AT.

**ppOpen-AT** Auto-Tuner

Library Call

Auto-tuned Kernel Execution

Library User

 Execution Time

:Target Computers

Auto-tuned: Target Computers

Before Release-time
Outline

• Background

• An Auto-tuning (AT) Language: *ppOpen-AT* and Adapting AT to an FDM code

• Performance Evaluation with the FX100

• Conclusion
Target Application

**Seism3D:**
Simulation for seismic wave analysis.

- Developed by Professor T.Furumura at the University of Tokyo.
  - The code is re-constructed as ppOpen-APPL/FDM.

- Finite Differential Method (FDM)
- 3D simulation
  - 3D arrays are allocated.

- Data type: Single Precision (real*4)
Flow Diagram of ppOpen-APPL/FDM

1. Initialization
2. Velocity Derivative (def_vel)
3. Velocity Update (update_vel)
4. Velocity PML condition (update_vel_sponge)
5. Velocity Passing (MPI) (passing_vel)
6. Stress Derivative (def_stress)
7. Stress Update (update_stress)
8. Stress PML condition (update_stress_sponge)
9. Stress Passing (MPI) (passing_stress)
10. Stop Iteration?
   - NO
   - YES: End

**Space difference by FDM.**
\[
\frac{d}{dx} \sigma_{pq}(x, y, z) = \frac{1}{\Delta x} \sum_{m=1}^{M/2} c_m [\sigma_{pq} \{x + (m + \frac{1}{2})\Delta x, y, z\} - \sigma_{pq} \{x - (m - \frac{1}{2})\Delta x, y, z\}],
\]

**Explicit time expansion by central difference.**
\[
\frac{u_p^{n+1}}{2} = \frac{u_p^{n-1}}{2} + \frac{1}{\rho} \left( \frac{\partial \sigma_{xp}^n}{\partial x} + \frac{\partial \sigma_{yp}^n}{\partial y} + \frac{\partial \sigma_{zp}^n}{\partial z} + f_p^n \right) \Delta t, \quad (p = x, y, z)
\]
AT WITH LOOP TRANSFORMATION
Target Loop Characteristics

• Triple-nested loops

```c
!$omp parallel do
  do k = NZ00, NZ01
    do j = NY00, NY01
      do i = NX00, NX01
        <Codes from FDM>
      end do
    end do
  end do
!$omp end parallel do
```

OpenMP directive to the outer loop (Z-axis)

Loop lengths are varied according to problem size, the number of MPI processes and OpenMP threads.

The codes can be separable by loop split.
What is separable codes?

- Variable definitions and references are separated.
- There is a flow-dependency, but no data dependency between each other.

$d_1 = ...$
$d_2 = ...$

... = ... $d_1$ ...
... = ... $d_2$ ...
...
... = ... $d_k$ ...

Variable definitions

Variable references

Split to Two Parts

$d_1 = ...$
$d_2 = ...$

... = ... $d_1$ ...
... = ... $d_2$ ...
...

... = ... $d_k$ ...
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX

RL = LAM (I,J,K)
RM = RIG (I,J,K)
RM2 = RM + RM

RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL

QG = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)

SXX (I,J,K) = ( SXX (I,J,K)+ (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
SYY (I,J,K) = ( SYY (I,J,K)+ (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
SZZ (I,J,K) = ( SZZ (I,J,K) + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG

RMAXY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
RMAXZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
RMAYZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))

SXY (I,J,K) = ( SXY (I,J,K) + (RMAXY*(DXVY(I,J,K)+DYVY(I,J,K)))*DT )
SXZ (I,J,K) = ( SXZ (I,J,K) + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )
SYZ (I,J,K) = ( SYZ (I,J,K) + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )

END DO
END DO
END DO
Loop Collapse
– One dimensional

Merit: Loop length is huge.
This is good for OpenMP thread parallelism.

```
DO KK = 1, NZ * NY * NX
  K = (KK-1)/(NY*NX) + 1
  J = mod((KK-1)/NX,NY) + 1
  I = mod(KK-1,NX) + 1
  RL = LAM(I,J,K)
  RM = RIG(I,J,K)
  RM2 = RM + RM
  RMAXY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
  RMAXZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
  RMAYZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
  RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
  QG = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
  SXX(I,J,K) = ( SXX(I,J,K) + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
  SYY(I,J,K) = ( SYY(I,J,K) + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
  SZZ(I,J,K) = ( SZZ(I,J,K) + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
  SXZ(I,J,K) = ( SXZ(I,J,K) + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
  SYZ(I,J,K) = ( SYZ(I,J,K) + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
END DO
```
Loop Collapse
– Two dimensional

DO KK = 1, NZ * NY
  K = (KK-1)/NY + 1
  J = mod(KK-1,NY) + 1
  DO I = 1, NX
    RL = LAM (I,J,K)
    RM = RIG (I,J,K)
    RM2 = RM + RM
    RMAXY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
    RMAXZ = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I+1,J,K) + 1.0/RIG(I,J,K+1) + 1.0/RIG(I+1,J,K+1))
    RMAZY = 4.0/(1.0/RIG(I,J,K) + 1.0/RIG(I,J+1,K) + 1.0/RIG(I,J,K+1) + 1.0/RIG(I,J+1,K+1))
    RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
    QG = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXX (I,J,K) = ( SXX (I,J,K) + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
    SYY (I,J,K) = ( SYY (I,J,K) + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
    SZZ (I,J,K) = ( SZZ (I,J,K) + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
    SXY (I,J,K) = ( SXY (I,J,K) + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
    SXZ (I,J,K) = ( SXZ (I,J,K) + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
    SYZ (I,J,K) = ( SYZ (I,J,K) + (RMAZY*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
  ENDDO
END DO

Merit: Loop length is huge.
This is good for OpenMP thread parallelism.

This I-loop enables us an opportunity of pre-fetching
Loop Split with Re-Computation

Re-computation is needed. ⇒ Compilers do not apply it without directive.
DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
    RL = LAM (I,J,K)
    RM = RIG (I,J,K)
    RM2 = RM + RM
    RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
    QG = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXX (I,J,K) = ( SXX (I,J,K) + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
    SYY (I,J,K) = ( SYY (I,J,K) + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
    SZZ (I,J,K) = ( SZZ (I,J,K) + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
ENDDO; ENDDO; ENDDO

DO K = 1, NZ
DO J = 1, NY
DO I = 1, NX
    STMP1 = 1.0/RIG(I,J,K)
    STMP2 = 1.0/RIG(I+1,J,K)
    STMP4 = 1.0/RIG(I,J,K+1)
    STMP3 = STMP1 + STMP2
    RMAXY = 4.0/(STMP3 + 1.0/RIG(I,J+1,K) + 1.0/RIG(I+1,J+1,K))
    RMAXZ = 4.0/(STMP3 + STMP4 + 1.0/RIG(I+1,J,K+1))
    RMAYZ = 4.0/(STMP3 + STMP4 + 1.0/RIG(I,J+1,K+1))
    QG = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
    SXY (I,J,K) = ( SXY (I,J,K) + (RMAXY*(DXVY(I,J,K)+DYVX(I,J,K)))*DT )*QG
    SXZ (I,J,K) = ( SXZ (I,J,K) + (RMAXZ*(DXVZ(I,J,K)+DZVX(I,J,K)))*DT )*QG
    SYZ (I,J,K) = ( SYZ (I,J,K) + (RMAYZ*(DYVZ(I,J,K)+DZVY(I,J,K)))*DT )*QG
END DO; END DO; END DO;
ppOpen-AT Directives
: Loop Split & Collapse with data-flow dependence

!$omp parallel do private(k,i,j,STMP1,STMP2,STMP3,STMP4,RL,RM,RM2,RMAXY,RMAXZ,RMAYZ,RLTHETA,QG)
  DO K = 1, NZ
    DO J = 1, NY
      DO I = 1, NX
        RL = LAM(I,J,K); RM = RIG(I,J,K); RM2 = RM + RM
        RLTHETA = (DXVX(I,J,K)+DYVY(I,J,K)+DZVZ(I,J,K))*RL
        QG = ABSX(I)*ABSY(J)*ABSZ(K)*Q(I,J,K)
        SXX(I,J,K) = ( SXX(I,J,K) + (RLTHETA + RM2*DXVX(I,J,K))*DT )*QG
        SYY(I,J,K) = ( SYY(I,J,K) + (RLTHETA + RM2*DYVY(I,J,K))*DT )*QG
        SZZ(I,J,K) = ( SZZ(I,J,K) + (RLTHETA + RM2*DZVZ(I,J,K))*DT )*QG
        STMP1 = 1.0/RIG(I,J,K); STMP2 = 1.0/RIG(I+1,J,K); STMP4 = 1.0/RIG(I,J,K+1)
        STMP3 = STMP1 + STMP2
        RMAXY = 4.0/(STMP3 + 1.0/RIG(I+1,J+1,K) + 1.0/RIG(I+1,J+1,K))
        RMAXZ = 4.0/(STMP3 + STMP4 + 1.0/RIG(I+1,J,K+1))
        RMAYZ = 4.0/(STMP3 + STMP4 + 1.0/RIG(I,J+1,K))
      END DO;
    END DO;
  END DO;
!$omp end parallel do

Specify Loop Split and Loop Fusion

Re-calculation is defined.

Loop Split Point

Using the re-calculation is defined.
AT Effect (update_stress) (Accumulated time for 2000 steps)

Xeon Phi (KNC) Cluster (8 Nodes)

- Without AT
- With AT

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Without AT</th>
<th>With AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>P8T240</td>
<td>113.41</td>
<td>54.81</td>
</tr>
<tr>
<td>P16T120</td>
<td>41.05</td>
<td>38.12</td>
</tr>
<tr>
<td>P32T60</td>
<td>47.26</td>
<td>32.02</td>
</tr>
<tr>
<td>P64T30</td>
<td>42.21</td>
<td>32.40</td>
</tr>
<tr>
<td>P128T15</td>
<td>218.49</td>
<td>39.16</td>
</tr>
<tr>
<td>P240T8</td>
<td>217.49</td>
<td>46.76</td>
</tr>
<tr>
<td>P480T4</td>
<td>231.38</td>
<td>45.37</td>
</tr>
</tbody>
</table>

- Speedups

- Best SW:6
- Best SW:5
- Best SW:4

AT WITH CODE SELECTION
Original Implementation (For Vector Machines)

Fourth-order accurate central-difference scheme for velocity. (def_stress)

call ppoHFDM_pdiffx3_m4_OAT( VX,DXVX, NXP,NYP,NZP,NXP0,NXP1,NYP0,...)
call ppoHFDM_pdiffy3_p4_OAT( VX,DYVX, NXP,NYP,NZP,NXP0,NXP1,NYP0,...)
call ppoHFDM_pdiffz3_p4_OAT( VX,DZVX, NXP,NYP,NZP,NXP0,NXP1,NYP0,...)
call ppoHFDM_pdiffy3_m4_OAT( VY,DYVY , NXP,NYP,NZP,NXP0,NXP1,NYP0,... )
call ppoHFDM_pdiffx3_p4_OAT( VY,DXVY , NXP,NYP,NZP,NXP0,NXP1,NYP0,... )
call ppoHFDM_pdiffz3_p4_OAT( VY,DZVY, NXP,NYP,NZP,NXP0,NXP1,NYP0,...)
call ppoHFDM_pdiffx3_p4_OAT( VZ,DXVZ, NXP,NYP,NZP,NXP0,NXP1,NYP0,...)
call ppoHFDM_pdiffy3_p4_OAT( VZ,DYVZ, NXP,NYP,NZP,NXP0,NXP1,NYP0,...)
call ppoHFDM_pdiffz3_m4_OAT( VZ,DZVZ, NXP,NYP,NZP,NXP0,NXP1,NYP0,...)

if( is_fs .or. is_nearfs ) then
  call ppoHFDM_bc_vel_deriv( KFSZ,NIFS,NJFS,IFSX,IFSY ,IFSZ,JFSX,JFSY,JFSZ )
end if

Explicit time expansion by leap-frog scheme. (update_stress)
Original Implementation (For Vector Machines)

```fortran
subroutine OAT_InstallppohFDMupdate_stress(..)
  !$omp parallel do private(i,j,k,RL1,RM1,RM2,RLRM2,DXVX1,DYVY1,DZVZ1,...)
    do k = NZ00, NZ01
      do j = NY00, NY01
        do i = NX00, NX01
          RL1  = LAM (I,J,K); RM1  = RIG (I,J,K); RM2  = RM1 + RM1; RLRM2 = RL1+RM2
          DXVX1 = DXVX(I,J,K); DYVY1 = DYVY(I,J,K); DZVZ1 = DZVZ(I,J,K)
          D3V3  = DXVX1 + DYVY1 + DZVZ1
          SXX (I,J,K) = SXX (I,J,K) + (RLRM2*(D3V3)-RM2*(DZVZ1+DYVY1) ) * DT
          SYY (I,J,K) = SYY (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVX1+DZVZ1) ) * DT
          SZZ (I,J,K) = SZZ (I,J,K) + (RLRM2*(D3V3)-RM2*(DXVX1+DYVY1) ) * DT
          DXVYDYVX1 = DXVY(I,J,K)+DYVX(I,J,K); DXVZDZVX1 = DXVZ(I,J,K)+DZVX(I,J,K)
          DYVZDZVY1 = DYVZ(I,J,K)+DZVY(I,J,K)
          SXY (I,J,K) = SXY (I,J,K) + RM1 * DXVYDYVX1 * DT
          SXZ (I,J,K) = SXZ (I,J,K) + RM1 * DXVZDZVX1 * DT
          SYZ (I,J,K) = SYZ (I,J,K) + RM1 * DYVZDZVY1 * DT
        end do
      end do
    end do
  end do
  return
end
```

Explicit time expansion by leap-frog scheme.

Input and output for arrays in each call -> Increase of B/F ratio: ~1.7
The Code Variants (For Scalar Machines)

- **Variant1** (IF-statements inside)
  - The followings include inside loop:
    1. Fourth-order accurate central-difference scheme for velocity.
    3. Explicit time expansion by leap-frog scheme.

- **Variant2** (IF-free, but there is IF-statements inside loop for process of model boundary.)
  - To remove IF sentences from the variant1, the loops are reconstructed.
  - The order of computations is changed, but the result without round-off errors is same.
  - **[Main Loop]**
    1. Fourth-order accurate central-difference scheme for velocity.
    2. Explicit time expansion by leap-frog scheme.
  - **[Loop for process of model boundary]**
    1. Fourth-order accurate central-difference scheme for velocity.
    3. Explicit time expansion by leap-frog scheme.
Fourth-order accurate central-difference scheme for velocity.

Stress tensor of $S_{xx}$, $S_{yy}$, $S_{zz}$

Process of model boundary.

Variant 1 (For Scalar Machines)

Explicit time expansion by leap-frog scheme

$B/F$ ratio is reduced to $0.4$

IF sentences inside – it is difficult to optimize code by compiler.
Variant2 (IF-free)

Fourth-order accurate central-difference scheme for velocity.

Explicit time expansion by leap-frog scheme.

Stress tensor of Sxx, Syy, Szz

Fourth-order accurate central-difference scheme for velocity.

Explicit time expansion by leap-frog scheme.

Win-win between B/F ratio and optimization by compiler.
Loop for process of model boundary

Variant 2 (IF-free)

Process of model boundary.

Fourth-order accurate central-difference scheme for velocity

Explicit time expansion by leap-frog scheme.

if (K == KFSZ(I,J)+1) then
  DZVX0 = ( VX(I,J,KFSZ(I,J)+2) - VX(I,J,KFSZ(I,J)+1) ) / DZ
  DZVY0 = ( VY(I,J,KFSZ(I,J)+2) - VY(I,J,KFSZ(I,J)+1) ) / DZ
else if (K == KFSZ(I,J)-1) then
  DZVX0 = ( VX(I,J,KFSZ(I,J) ) - VX(I,J,KFSZ(I,J)-1) ) / DZ
  DZVY0 = ( VY(I,J,KFSZ(I,J) ) - VY(I,J,KFSZ(I,J)-1) ) / DZ
end if

DXV1 = DXVX0
DYV1 = DYVY0
DZV1 = DZVZ0
D3V = DXVX1 + DYVY1 + DZV1
DXVDYVX1 = DXVX0 + DYVY0
DXVZDVX1 = DXVX0 + DZV1
DYVZDVY1 = DYVY0 + DZV1

if (K == KFSZ(I,J)+1) then
  KK = 2
else
  KK = 1
end if

SXX(I,J,K) = SSXX(I,J,KK) & + (RLRM2*(D3V3)-RM2*(DZV1+DYVY1)) * DT
SYY(I,J,K) = SSYY(I,J,KK) & + (RLRM2*(D3V3)-RM2*(DXVX1+DZV1)) * DT
SZZ(I,J,K) = SSZZ(I,J,KK) & + (RLRM2*(D3V3)-RM2*(DXVX1+DYVY1)) * DT
SXV(I,J,K) = SSXY(I,J,KK) & + RM1 * DEXV1DVYX1 * DT
SZX(I,J,K) = SSXZ(I,J,KK) & + RM1 * DXVZDVX1 * DT
SYZ(I,J,K) = SSYZ(I,J,KK) & + RM1 * DYVZDVY1 * DT
end do
end do
end if
Code selection by ppOpen-AT and hierarchical AT

Upper Code

Program main
....
!OAT$ install select region start
!OAT$ name pphoFD M update_vel_select
!OAT$ select sub region start
call pphoFD M_pdiffx3_p4( SXX,DXSXX,NXP,NYP,NZP,...)
call pphoFD M_pdiffy3_p4( SYY,DYSYY, NXP,NYP,NZP,...)
...
if( is_fs .or. is_nearfs ) then
  call pphoFD M_bc_stress_deriv( KFSZ,NIFS,NJFS,IFSX,...)
end if
call pphoFD M_update_vel( 1, NXP, 1, NYP, 1, NZP)
!OAT$ select sub region end

!OAT$ install select region start
Call pphoFD M_update_vel_Intel( 1, NXP, 1, NYP, 1, NZP)
!OAT$ select sub region end

!OAT$ install select region end

With Select clause, code selection can be specified.

Lower Code

subroutine pphoFD M_pdiffx3_p4(....)
....
!OAT$ install LoopFusion region start
....

!OAT$ install LoopFusion region start
!OAT$ name pphoFD M update_vel
!OAT$ debug (pp)
!$omp parallel do private(i,j,k,ROX,ROY,ROZ)
do k = NZ00, NZ01
  do j = NY00, NY01
    do i = NX00, NX01
      ....
....

subroutine pphoFD M_update_vel(....)
Call tree graph by the AT

Main Program

Start

update_vel_select

Velocity PML condition (update_vel_sponge)

Velocity Passing (MPI) (passing_vel)

update_stress_select

Stress PML condition (update_stress_sponge)

Stress Passing (MPI) (passing_stress)

Stop iteration?

NO

YES

End

: auto-generated codes

Velocity Derivative (def_vel)

Velocity Update (update_vel)

Selection

Velocity Update (update_vel_Scalar)

Selection

Velocity Update (update_vel_IF_free)

Selection

Stress Derivative (def_stress)

Stress Update (update_stress)

Selection

Stress Update (update_stress_Scalar)

Selection

Stress Update (update_stress_IF_free)
Execution Order of the AT

We can specify the order via a directive of ppOpen-AT. (an extended function)
# The Number of AT Candidates (ppOpen-APPL/FDM)

<table>
<thead>
<tr>
<th>Kernel Names</th>
<th>AT Objects</th>
<th>The Number of Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. update_stress</td>
<td>・Loop Collapses and Splits : 8 Kinds ・Code Selections : 2 Kinds</td>
<td>10</td>
</tr>
<tr>
<td>2. update_vel</td>
<td>・Loop Collapses, Splits, and re-ordering of statements: : 6 Kinds ・Code Selections: 2 Kinds</td>
<td>8</td>
</tr>
<tr>
<td>3. update_stress_sponge</td>
<td>・Loop Collapses : 3 Kinds</td>
<td>3</td>
</tr>
<tr>
<td>4. update_vel_sponge</td>
<td>・Loop Collapses : 3 Kinds</td>
<td>3</td>
</tr>
<tr>
<td>5. ppoHFDMP_diffx3_p4</td>
<td>Kernel Names : def_update, def_vel ・Loop Collapses : 3 Kinds</td>
<td>3</td>
</tr>
<tr>
<td>6. ppoHFDMP_diffx3_m4</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>7. ppoHFDMP_difff3_p4</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>8. ppoHFDMP_difff3_m4</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>9. ppoHFDMP_difffz3_p4</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>10. ppoHFDMP_difffz3_m4</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>11. ppoHFDMP_ps_pack</td>
<td>Data packing and unpacking ・Loop Collapses : 3 Kinds</td>
<td>3</td>
</tr>
<tr>
<td>12. ppoHFDMP_ps_unpack</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>13. ppoHFDMP_pv_pack</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>14. ppoHFDMP_pv_unpack</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

- **Total**: 54 Kinds
- **Hybrid MPI/OpenMP**: 7 Kinds
- **54 × 7 = 378 Kinds**
Outline

• Background

• An Auto-tuning (AT) Language: *ppOpen-AT* and Adapting AT to an FDM code

• Performance Evaluation with the FX100

• Conclusion
FX100
### FX100 (ITC, Nagoya U.), The Fujitsu PRIMEHPC FX100

<table>
<thead>
<tr>
<th>Contents</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whole System</td>
<td></td>
</tr>
<tr>
<td>Total Performance</td>
<td>3.2 PFLOPS</td>
</tr>
<tr>
<td>Total Memory Amounts</td>
<td>90 TiB</td>
</tr>
<tr>
<td>Total #nodes</td>
<td>2,880</td>
</tr>
<tr>
<td>Inter Connection</td>
<td>The TOFU2 (6 Dimension Mesh / Torus)</td>
</tr>
<tr>
<td>Local File System Amounts</td>
<td>6.0 PB</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2880 Nodes (92,160 Cores)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Contents</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td></td>
</tr>
<tr>
<td>Theoretical Peak Performance</td>
<td>1 TFLOPS (double precision)</td>
</tr>
<tr>
<td>#Processors (#Cores)</td>
<td>32 + 2 (assistant cores)</td>
</tr>
<tr>
<td>Main Memory Amounts</td>
<td>32 GB</td>
</tr>
<tr>
<td>Processor</td>
<td></td>
</tr>
<tr>
<td>Processor Name</td>
<td>SPARC64 XI-fx</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.2 GHz</td>
</tr>
<tr>
<td>Theoretical Peak Performance</td>
<td>31.25 GFLOPS</td>
</tr>
</tbody>
</table>
Comparison with the FX10 (ITC, U. Tokyo) and FX100 (ITC, Nagoya U.)

<table>
<thead>
<tr>
<th></th>
<th>FX10</th>
<th>FX100</th>
<th>Ratios (FX100/FX10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node FLOPS</td>
<td>236.5 GFLOPS</td>
<td>2 TFLOPS (single precision)</td>
<td>8.44x</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>85 GB/S</td>
<td>480 GB/S</td>
<td>5.64x</td>
</tr>
<tr>
<td>Networks</td>
<td>5 GB/S x2</td>
<td>12.5 GB/S x2</td>
<td>2.5x</td>
</tr>
</tbody>
</table>
Execution Details

- ppOpen-APPL/FDM ver.0.2
- ppOpen-AT ver.0.2
- The number of time step: 2000 steps
- The number of nodes: 8 node
- Target Problem Size (Almost maximum size with 8 GB/node)
  - \( \text{NX} \times \text{NY} \times \text{NZ} = 512 \times 512 \times 512 / 8 \text{ Node} \)
  - \( \text{NX} \times \text{NY} \times \text{NZ} = 256 \times 256 \times 256 / \text{node} \) (\(!=\) per MPI Process)
- Target MPI Processes and Threads on the Xeon Phi
  - 1 node of the Ivy Bridge with 2 HT (Hyper Threading)
  - \( \text{PXY} \): \( X \) MPI Processes and \( Y \) Threads per process
  - \( \text{P8T32} \): Minimum Hybrid MPI-OpenMP execution for ppOpen-APPL/FDM, since it needs minimum 8 MPI Processes.
    - \( \text{P16T16} \)
    - \( \text{P32T8} \)
    - \( \text{P64T4} \)
    - \( \text{P128T2} \)
    - \( \text{P256T1} \): pure MPI
- The number of iterations for the kernels: 100
NUMA affinity

• Sparc64 XI-fx is a NUMA.
• 2 sockets: 16 cores + 16 cores
• NUMA affinity
  – Memory allocation
    • “Local allocation” is used.
    • plm_ple_memory_allocation_policy=localalloc
  – CPU allocation
    • P8 and P16:
      plm_ple_numanode_assign_policy=simplex
    • More than P32:
      plm_ple_numanode_assign_policy=share_band
RELATED WORK
The **explicit** function of loop transformations (loop splits and collapses) and code selection at simultaneously is unique!

<table>
<thead>
<tr>
<th>AT Language / Items</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
<th>#7</th>
<th>#8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppOpen-AT</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vendor Compilers</td>
<td>Out of Target</td>
<td>Limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformation</td>
<td>Recipe</td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recipes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X language</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADAPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atune</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PEPPHER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xevolver</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

2. Obtaining loop length in run-time.  
3. Loop split with increase of computations, and loop collapses to the split loops.  
4. Re-ordering of inner-loop sentences.  
5. Code selection with loop transformations (Hierarchical AT descriptions)  
6. Algorithm selection.  
7. Code generation with execution feedback.  
8. Software requirement.
Outline

• Background
• An Auto-tuning (AT) Language: *ppOpen-AT* and Adapting AT to an FDM code
• Performance Evaluation with the FX100
• Conclusion
Conclusion

- Evaluation performance between the FX10 with a conventional memory and the FX100 with a 3D stacking memory.
- We observe effective execution in the FX100 with respect to hardware ability for memory transfer performance to that of the FX10.
- According to results of profiler, loading time for floating point data from cache is dramatically reduced in the FX100 compared to the FX10.
Future work

• Auto-tuning for dynamic scheduling and its chunk size.
  – schedule(dynamic, chunk_size) in OpenMP
  – There is a case to speedup by using it in the FX10.
    • Code selection for description with dynamic or static, and the size of chunk_size.

• OpenACC optimization
  – Code selection for description of “Gang”, “Vector”, “Worker” and its size of resources.


• Performance Modeling
  – Performance model (Black-Box model) for hierarchical AT.
  – D-spline model (Tanaka, Fujii, et.al@Kogakuin U.)
  – Surrogate model (Wang, et.al@National Taiwan U.)
Thank you for your attention!

Questions?

http://ppopenhpc.cc.u-tokyo.ac.jp/