



OPTIMIZING A STENCIL USING VTUNE HPC AND MEMORY ANALYSES

How to optimize Iso3DFD kernel

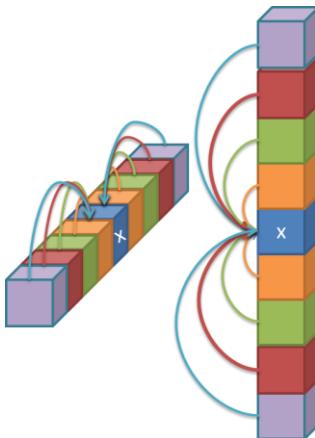
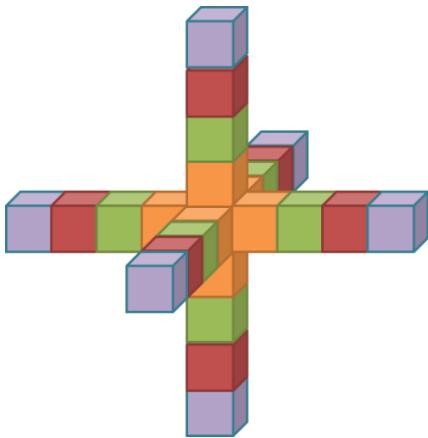
AGENDA

- What is iso3DFD
- Brief introduction to Vtune Amplifier and where to start ?
- Dev00
- Dev01 – Threading
- Dev02 – Memory optimization
- Dev03 – Cache blocking
- Dev04 - Vectorization
- Dev05 – First touch and NUMA effect

WHAT IS ISO3DFD ?

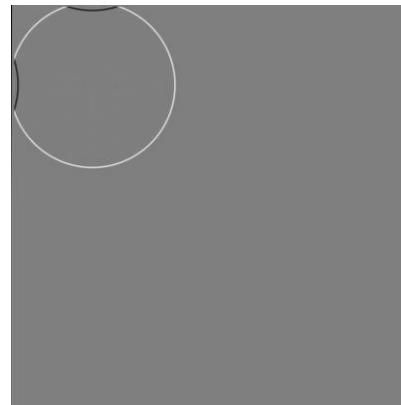
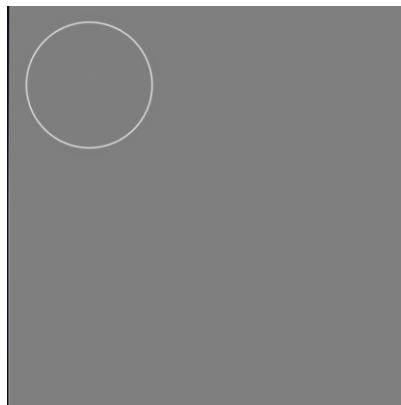
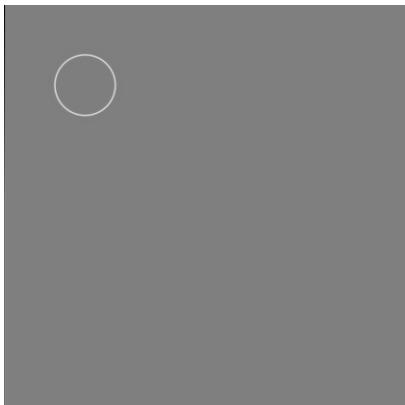
HOW DOES A STENCIL WORK ?

- For computing $P_{t+1}(x,y,z)$, we need to use all neighbors in the 3 dimensions of $P_t(x,y,z)$.
- The stencil looks like a 3D cross in Iso3DFD



ISO3DFD 2D CUT

- This is a 2D cut of our 3D volume
- We create a perturbation and look at the pressure for 4 different time steps
- We see that there is no boundary condition



ISO3DFD - A WAVE PROPAGATION KERNEL

```
For (int iz=0; iz<n3; iz++)
For (int iy=0; iy<n2; iy++)
For (int ix=0; ix<n1; ix++) {
    int offset = iz*dimn1n2 + iy*n1 + ix;
    float value = 0.0;

    value += ptr_prev[offset]*coeff[0];
    for(int ir=1; ir<= 8 ; ir++) {
        value += coeff[ir] * (ptr_prev[offset + ir] + ptr_prev[offset - ir]);
        value += coeff[ir] * (ptr_prev[offset + ir*n1] + ptr_prev[offset - ir*n1]);
        value += coeff[ir] * (ptr_prev[offset + ir*dimn1n2] + ptr_prev[offset -
ir*dimn1n2]);
    }
    ptr_next[offset] = 2.0f* ptr_prev[offset] - ptr_next[offset] + value*ptr_vel[offset];
}
```

- 3D Finite Difference
- Acoustic isotropic, pressure only scheme,
- 16th order in space 2nd order in time
- No boundary conditions
- OpenMP (no MPI)

WHICH HARDWARE

- For the rest of the presentation we are using a 2 sockets system
 - Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
 - Each processor has 18 cores
 - Total of 36 cores
 - Haswell system with AVX2 enabled

BRIEF INTRODUCTION TO VTUNE AMPLIFIER AND WHERE TO START ?

VTUNE IS A VERY POWERFUL TOOL

Where is my application...

Spending Time?

Function - Call Stack	CPU Time
algorithm_2	3.560s
do_xform ←	3.560s
algorithm_1	1.412s
BaseThreadInitTh	0.000s

- Focus tuning on functions taking time
- See call stacks
- See time on source

Wasting Time?

Line	MEM_LOAD... LLC_MISS
475 float rx, ry, rz =	
476 float param1 = (AA 30,000	
477 float param2 = (AA	
478 bool neg = (rz < 0	

- See cache misses on your source
- See functions sorted by # of cache misses

Waiting Too Long?

Wait Time	Wait Count
176.504s	18,277
84.681s	5,499
84.612s	5,489

- See locks by wait time
- Red/Green for CPU utilization during wait

WHICH ANALYSIS SHOULD I RUN FIRST?

- Historically we've encourage users to start with Hotspots or Advanced hotspots

The screenshot shows the Intel VTune Amplifier interface. On the left, there's a navigation tree with categories like Algorithm Analysis, Compute-Intensive Application Analy, Microarchitecture Analysis, Platform Analysis, and Custom Analysis. Under 'Advanced Hotspots', which is selected and highlighted in red, there are sub-options: Basic Hotspots, Advanced Hotspots (selected), Concurrency, Locks and Waits, Memory Consumption, HPC Performance Characterization, General Exploration, Memory Access, TSX Exploration, TSX Hotspots, SGX Hotspots, CPU/GPU Concurrency, System Overview, GPU Hotspots, Disk Input and Output, and Custom Analysis. The main panel displays the 'Advanced Hotspots' configuration. It includes a 'Copy' button, a 'Start' button (which is currently 'Start Paused'), and a 'Choose Target' button. The configuration area has sections for 'CPU sampling interval, ms:' (set to 1), 'Event mode:' (set to 'All'), and checkboxes for 'Analyze user tasks, events, and counters' and 'Analyze OpenMP regions'. At the bottom, there's a 'Details' section with the message: 'Events configured for CPU: Intel(R) Core(TM) Processor code named Haswell'.

WHICH ANALYSIS SHOULD I RUN FIRST ?

- But recently we added new analyses dedicated to HPC

HPC Performance Characterization: Metrics suitable for HPC Applications. It gives a high level picture without too low level details.

Memory Access: Interesting metrics about memory (Bandwidth, latency, NUMA, etc)

ISO3DFD DEVOO - A VERY UNOPTIMIZED VERSION

DEVOO_UNOPTIMIZED (WE SHOULD SAY SUB_OPTIMIZED 😊)

- We created a sub-optimized version of iso3DFD:
 - No parallelization
 - No vectorization
 - Sub-optimal memory accesses
- Let's see how Vtune can highlight the problems in this code
- 1st step, lets have an idea about the main bottlenecks
 - Run the HPC Performance Characterization

HPC PERFORMANCE CHARACTERIZATION ON DEVOO

The screenshot shows the HPC Performance Characterization interface with the following details:

- Elapsed Time:** 315.981s
- SP GFLOPS:** Not supported for this CPU.
- CPU Utilization:** 0.7% (Average CPU Usage: 0.267 Out of 36 logical CPUs)
 - Serial Time (outside parallel regions): 315.956s (100.0%)
 - Parallel Region Time: 0.024s (0.0%)
 - CPU Usage Histogram
- Memory Bound:** 72.9% of Pipeline Slots
 - Cache Bound: 27.9% of Clockticks
 - DRAM Bound: 44.9% of Clockticks
 - NUMA: % of Remote Accesses: 0.0%
 - Bandwidth Utilization Histogram
- FPU Utilization:** Not supported for this CPU.
- Collection and Platform Info**

It doesn't look good:

- Less than 1% of CPU utilization (it looks like there is no threading)
- 73% of memory bound, we are spending some significant time waiting for data

We are using Haswell and unfortunately Flops are not available. We'll see how to solve this problem later

CPU WITH FLOPS COUNTERS AVAILABLE

Memory Bound : 73.0% of Pipeline Slots

Cache Bound : 2.2% of Clockticks
DRAM Bound : 67.2%  of Clockticks
NUMA: % of Remote Accesses : 0.0%
Bandwidth Utilization Histogram

FPU Utilization : 0.2%

SP FLOPs per Cycle : 0.149 Out of 64 
Vector Capacity Usage : 6.2% 
FP Instruction Mix:
% of Packed FP Instr. : 0.0%
% of 128-bit : 0.0%
% of 256-bit : 0.0%
% of 512-bit : 0.0%
% of Scalar FP Instr. : 100.0% 

FP Arith/Mem Rd Instr. Ratio: 0.801

FP Arith/Mem Wr Instr. Ratio: 53.713

Top Loops/Functions with FPU Usage by CPU Time

This section provides information for the most time consuming loops/functions with floating point operations.

Function	CPU Time 	FPU Utilization 	Vector Instruction Set 	Loop Type 
[Loop at line 41 in iso_3dfd]	135.459s	0.2% 		Body
loop at line 41 in iso_3dfd	132.997s	0.2% 		Body

This example was obtained on Skylake where flops counters are available



CPU UTILIZATION IN DETAILS

CPU Utilization : 0.7%

Average CPU Usage : 0.267 Out of 36 logical CPUs

Serial Time (outside parallel regions) : 315.956s (100.0%) 

Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most metric includes the Wait time of the master thread, it may significantly exceed the aggregated CPU time in the table.

Function	Module	Serial CPU Time 
[Loop at line 41 in iso_3dfd]	iso3dfd_dev00_unoptimized_cpu_avx2.exe	42.194s
[Loop at line 41 in iso_3dfd]	iso3dfd_dev00_unoptimized_cpu_avx2.exe	41.279s
__do_softirq	vmlinux	0.320s
_spin_unlock_irqrestore	vmlinux	0.080s
clear_page_c_e	vmlinux	0.077s
[Others]		0.118s

Parallel Region Time : 0.024s (0.0%)

Estimated Ideal Time : 0.002s (0.0%)

OpenMP Potential Gain : 0.023s (0.0%)

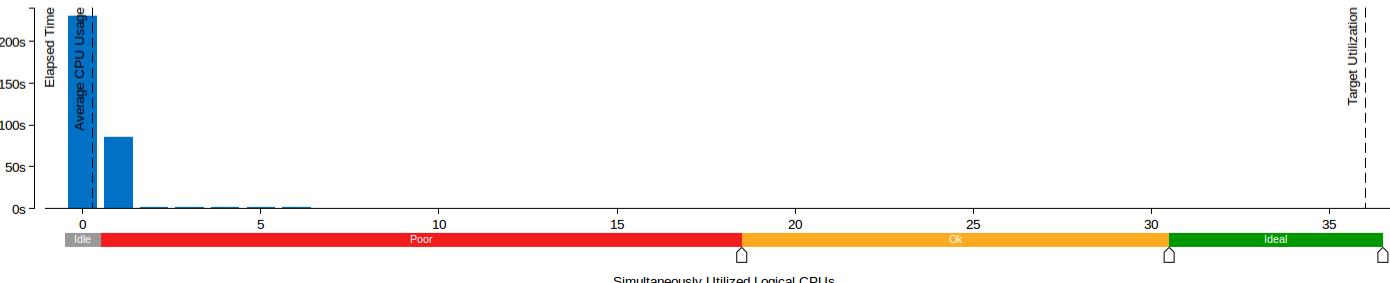
Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed

OpenMP Region	OpenMP Potential Gain  (%) 	OpenMP Region Time 
main\$omp\$parallel:36@:unknown:128:134	0.023s 0.0%	0.024s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.



■ HPC Characterization indicates that CPU utilization is very low

- Lets focus on it first
- We see that only 1 thread is used instead of 36 !
- It looks like we need to implement some threading

LETS LOOK AT THE CODE OF DEVOO

Adding parallelism can be done here

```
void iso_3dfd_it(float *ptr_next, float *ptr_prev, float *ptr_vel, float
*coeff, const int n1, const int n2, const int n3, const int
num_threads, const int n1_Tblock, const int n2_Tblock, const int
n3_Tblock){
    int dimn1n2 = n1*n2;//This value will be used later
    for(int ix=0; ix<n1; ix++) {
        for(int iy=0; iy<n2; iy++) {
            for(int iz=0; iz<n3; iz++) {
                ... // compute the stencil
            }
        }
    }
}
```

Single time step iteration so no dependency inside those loops, threading is then possible

ISO3DFD DEV01 - ADDING THREADING WITH OPENMP

ADDING A SINGLE LINE OF OPENMP TO RUN ON ALL CORES

```
void iso_3dfd_it(float *ptr_next, float *ptr_prev, float *ptr_vel, float
*coeff, const int n1, const int n2, const int n3, const int
num_threads, const int n1_Tblock, const int n2_Tblock, const int
n3_Tblock){
    int dimn1n2 = n1*n2;//This value will be used later
    #pragma omp parallel for default(shared)
    for(int ix=0; ix<n1; ix++) {
        for(int iy=0; iy<n2; iy++) {
            for(int iz=0; iz<n3; iz++) {
                ... // compute the stencil
            }
        }
    }
}
```

WHAT IS THE NEXT STEP?

- We can iterate and run Vtune again to see what is the next bottleneck
 - Run another **HPC Performance Characterization**
 - Look for the next bottleneck
 - Try to optimize it
 - Iterate again
- Other tools can be used in the iteration process
 - **Intel® Advisor** is better suited for vectorization (FPU utilization)
 - **Performance Snapshot** can also be used instead of HPC characterization

HPC PERFORMANCE CHARACTERIZATION ON DEV01

HPC Performance Characterization HPC Performance Characterization viewpoint (change) ?

Collection Log Analysis Target Analysis Type Summary Bottom-up

Elapsed Time [?]: 15.186s

SP GFLOPS [?]: Not supported for this CPU.

CPU Utilization [?]: 68.9% ↘

Average CPU Usage [?]: 24.815 Out of 36 logical CPUs
Serial Time (outside parallel regions) [?]: 0.132s (0.9%)

Parallel Region Time [?]: 15.054s (99.1%)
Estimated Ideal Time [?]: 11.359s (74.8%)
OpenMP Potential Gain [?]: 3.695s (24.3%) ↘

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain [?] (%) [?]	OpenMP Region Time [?]
iso_3dfd\$omp\$parallel:36@unknown:39:56	3.676s ↘ 24.2% ↘	15.034s
main\$omp\$parallel:36@unknown:128:134	0.019s 0.1%	0.020s

CPU Usage Histogram

Memory Bound [?]: 58.3% ↘ of Pipeline Slots

Cache Bound [?]: 32.0% ↘ of Clockticks
DRAM Bound [?]: 26.4% ↘ of Clockticks
NUMA: % of Remote Accesses [?]: 44.6% ↘
Bandwidth Utilization Histogram

FPU Utilization [?]: Not supported for this CPU.

- Threading is already much better
- We went from 316s to 15.2s (=~20X speedup)
- It looks like some improvement on threading can still be obtained
- The biggest bottleneck right now comes from memory

MEMORY ACCESS ANALYSIS IN VTUNE

Memory Access Memory Usage viewpoint (change) [?](#)

Collection Log Analysis Target Analysis Type Summary Bottom-up P

Elapsed Time [?](#): 17.374s

CPU Time [?](#):

515.074s

Memory Bound [?](#):

56.29% of Pipeline Slots

L1 Bound [?](#):

2.2% of Clockticks

L2 Bound [?](#):

0.0% of Clockticks

L3 Bound [?](#):

27.0% of Clockticks

DRAM Bound [?](#):

27.1% of Clockticks

DRAM Bandwidth Bound [?](#):

0.0% of Elapsed Time

Memory Latency:

Remote / Local memory Ratio [?](#): 0.661

Loads:

327,307,818,940

Stores:

17,299,718,976

LLC Miss Count [?](#):

3,583,414,992

Local DRAM Access Count [?](#):

1,592,847,784

Remote DRAM Access Count [?](#):

1,052,431,572

Remote Cache Access Count [?](#):

924,027,720

Average Latency (cycles) [?](#):

31

Total Thread Count:

36

Paused Time [?](#):

0s

Bandwidth Utilization Histogram

Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

Bandwidth Domain: DRAM_GDDR

The histogram shows the distribution of the bandwidth used ordered by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types as the Selection area to group data and see all functions exercised during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum utilization (DRAM and In-memory) bandwidth.

Elapsed Time

Average Bandwidth

Observed Maximum

Top Functions with High Bandwidth Utilization

The section shows the top functions, sorted by LLC Misses that were exercising when bandwidth utilization was high for the domain selected in the Histogram area.

Latency Histogram

The histogram shows a distribution of loads per latency (in cycles).

Latency

Loads

Collection and Platform Info [?](#)

This section provides information about the collection, including result set size and collection platform data.

4:36 PM
11/21/20

Lets look at the bottom up view and sort it by CPU Time

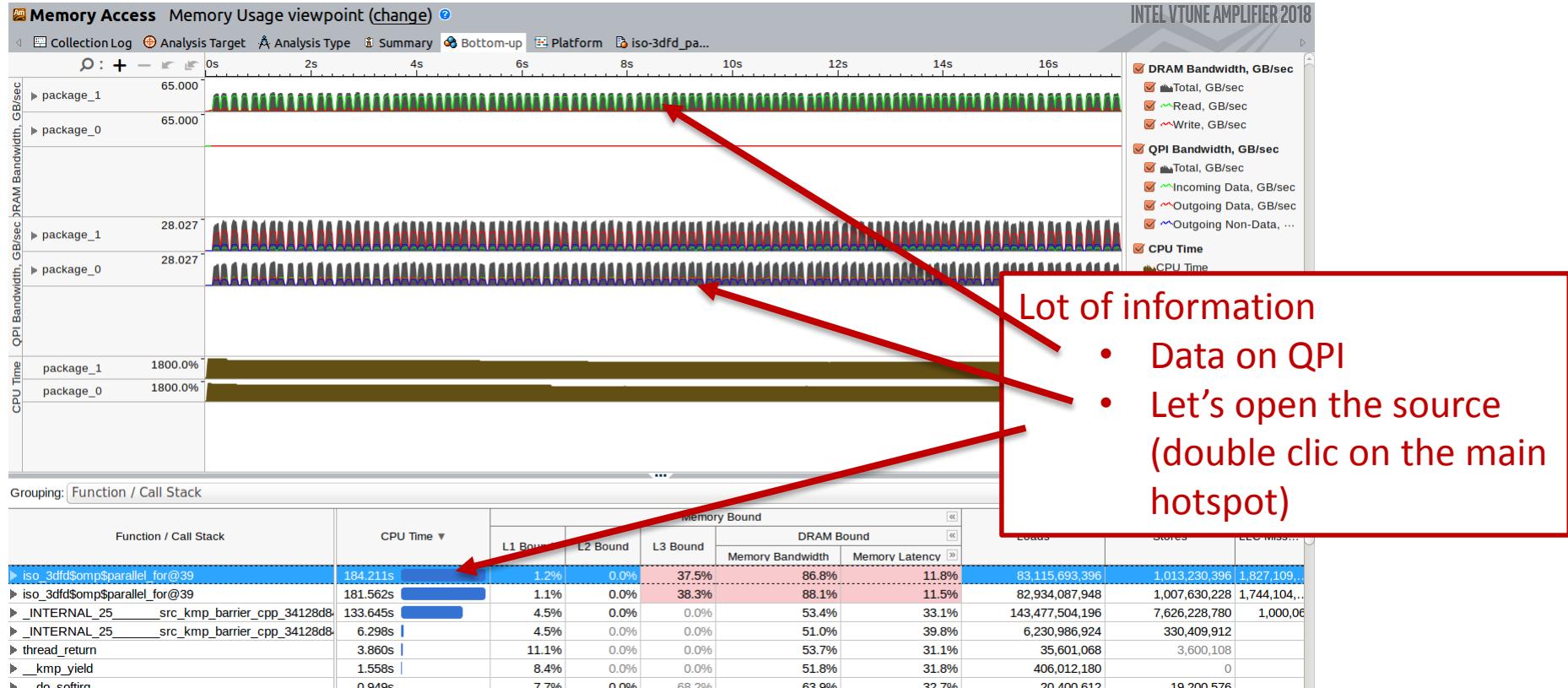
© 2017 Intel Corporation. All rights reserved. Intel and the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

For more complete information about compiler optimizations, see our [Optimization Notice](#).

intel Software

BOTTOM UP VIEW OF MEMORY ACCESS ANALYSIS



Lot of information

- Data on QPI
- Let's open the source (double clic on the main hotspot)

LET'S SEE THE SOURCE REPORT

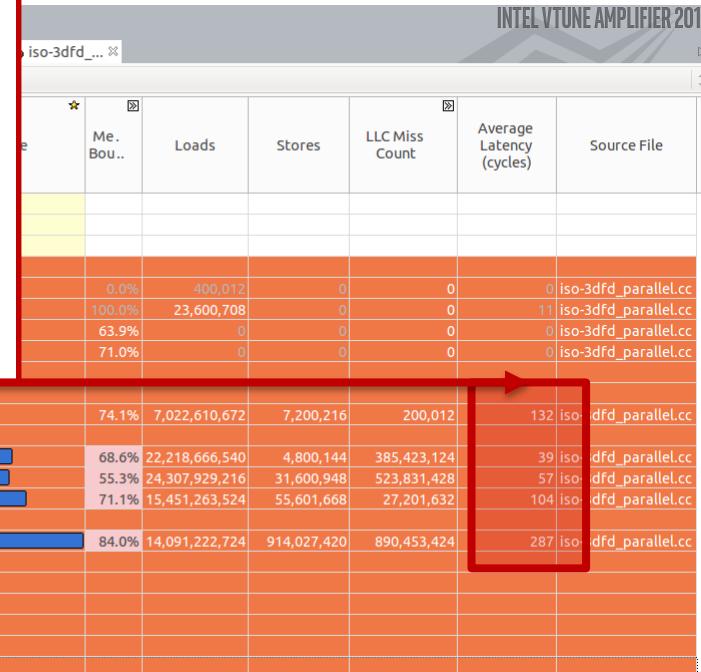
Average Latency is very high
for those accesses. It looks like prefetcher
can't efficiently predict our memory
accesses

L1 =~ 5 cycles

L2 =~ 12 cycles

L3 =~ 40-60 cycles

```
45     float value = 0.0;
46     value += ptr_prev[offset]*coeff[0];
47     for(int ir=1; ir<=HALF_LENGTH; ir++) {
48         value += coeff[ir] * (ptr_prev[offset + ir] + ptr_pre
49         value += coeff[ir] * (ptr_prev[offset + ir*n1] + ptr
50         value += coeff[ir] * (ptr_prev[offset + ir*dimIn2] +
51     }
52     ptr_next[offset] = 2.0f* ptr_prev[offset] - ptr_next[offs
53 }
54 }
55 }
56 }
57 }
```



PROBLEM CAN BE FOUND WITH ADVISOR

Memory Access Patterns Report						Dependencies Report	Recommendations
ID	Stride	Type	Source	Nested Function	Variable references	Max. Site Footprint	
P3	76800	Constant stride	iso-3dfd_parallel.cc:46		block 0x7fb15681b010 allocated at iso-3dfd_main.cc:183	65MB	
P7	76800	Constant stride	iso-3dfd_parallel.cc:48		block 0x7fb15681b010 allocated at iso-3dfd_main.cc:183	65MB	
P11	76800	Constant stride	iso-3dfd_parallel.cc:49		block 0x7fb15681b010 allocated at iso-3dfd_main.cc:183	65MB	
P15	76800	Constant stride	iso-3dfd_parallel.cc:50		block 0x7fb15681b010 allocated at iso-3dfd_main.cc:183	65MB	
P19	76800	Constant stride	iso-3dfd_parallel.cc:52		block 0x7fb14b851010 allocated at iso-3dfd_main.cc:185, block 0x7fb151036010 allocated at iso-3dfd_main.cc:184	65MB	
P23	76800	Constant stride	iso-3dfd_parallel.cc:52		block 0x7fb151036010 allocated at iso-3dfd_main.cc:184	65MB	
P29		Parallel site information	iso-3dfd_parallel.cc:43				
P56	0	Uniform stride	iso-3dfd_parallel.cc:46			4B	
P65	0	Uniform stride	iso-3dfd_parallel.cc:48			4B	
P71	0	Uniform stride	iso-3dfd_parallel.cc:49			8B	
P81	0	Uniform stride	iso-3dfd_parallel.cc:50			8B	
P91	0	Uniform stride	iso-3dfd_parallel.cc:52			8B	

- Advisor reports many locations where we have a constant stride of 76800 happens ($256 \times 300 = 76800$ which is the size of our 2 first dimensions)
- Constant strides are lines 46,48,49, 50,52

WHERE DOES THE CONSTANT STRIDE HAPPEN ?

```
void iso_3dfd_it(float *ptr_next, float *ptr_prev, float *ptr_vel, float  
*coeff, const int n1, const int n2, const int n3, const int  
num_threads, const int n1_Tblock, const int n2_Tblock, const int  
n3_Tblock){  
    int dimn1n2 = n1*n2;//This value will be used later  
    #pragma omp parallel for default(shared)  
    for(int ix=0; ix<n1; ix++) {  
        for(int iy=0; iy<n2; iy++) {  
            for(int iz=0; iz<n3; iz++) {  
                // Constant stride appears here in the innermost loop  
                int offset = iz*dimn1n2 + iy*n1 + ix;  
                float value = 0.0;  
                value += ptr_prev[offset]*coeff[0];  
                ...  
            }  
        }  
    }  
}
```

Iterations are not in the good order !!!

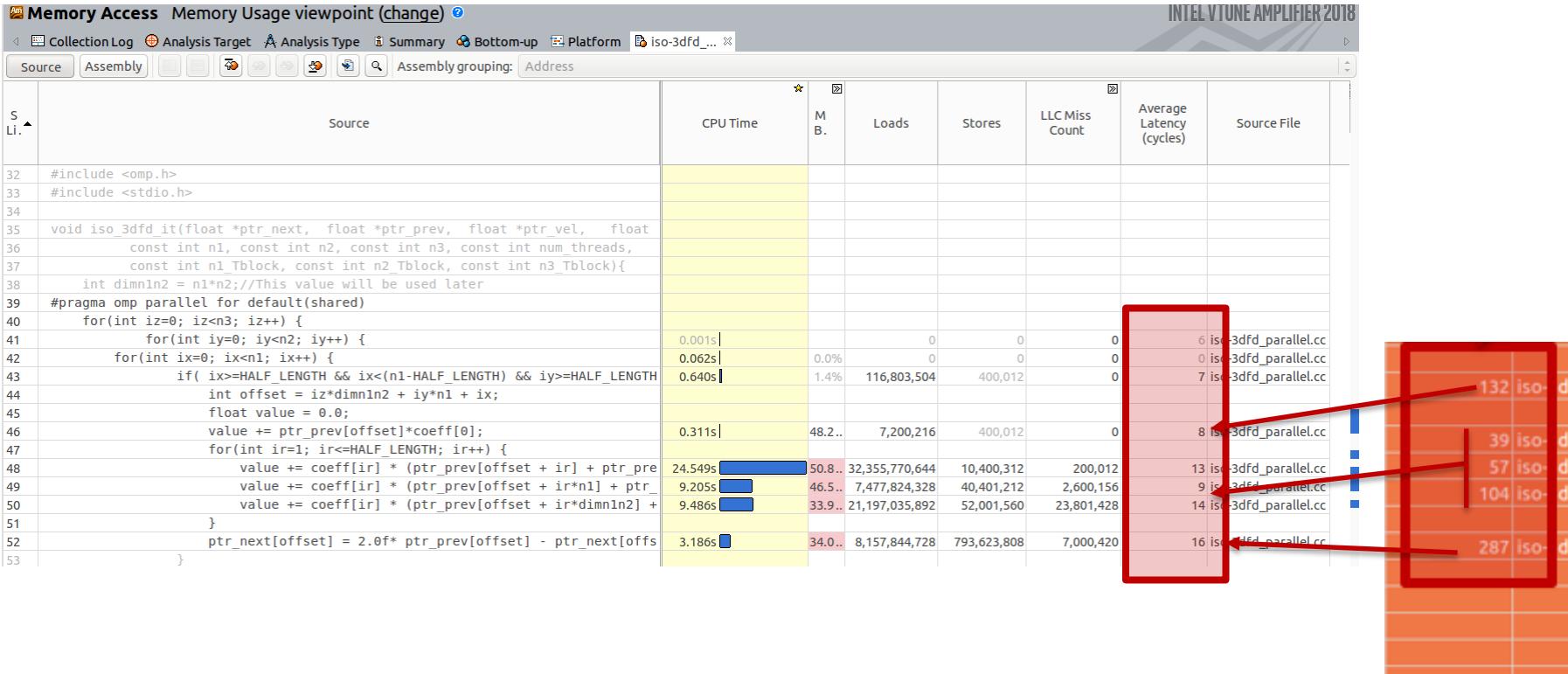
ISO3DFD DEVO2 - RE-ORDERING THE LOOPS

CHANGING LOOP ORDER SOLVES THE PROBLEM

```
void iso_3dfd_it(float *ptr_next, float *ptr_prev, float *ptr_vel, float
*coeff, const int n1, const int n2, const int n3, const int
num_threads, const int n1_Tblock, const int n2_Tblock, const int
n3_Tblock){
    int dimn1n2 = n1*n2;//This value will be used later
    #pragma omp parallel for default(shared)

        for(int iy=0; iy<n2; iy++) {
            for(int iz=0; iz<n3; iz++) {
                for(int ix=0; ix<n1; ix++) {
                    // Constant stride appears here in the innermost loop
                    int offset = iz*dimn1n2 + iy*n1 + ix;
                    float value = 0.0;
                    value += ptr_prev[offset]*coeff[0];
                    ...
                }
            }
        }
}
```

DID WE SOLVE THE LATENCY ISSUES ?



WHAT IS COMING NEXT ?

- We enter the iterating process again
 - Run **HPC Performance Characterization**
 - Detect the strongest bottleneck
 - New: Use **Advisor** if not enough information available
- Remember that **Vtune** is perfect for understanding
 - Threading issues
 - Memory issues
- **Advisor** can help you to
 - Optimize vectorization
 - Understand how far you are from the maximum performance
 - Investigate data access

HPC PERFORMANCE CHARACTERIZATION ON DEVO2

Collection Log Analysis Target Analysis Type Summary Bottom-up

Elapsed Time [?]: 6.613s

SP GFLOPS [?]: Not supported for this CPU.

CPU Utilization [?]: 49.8% ↘

Average CPU Usage [?]: 17.937 Out of 36 logical CPUs

Serial Time (outside parallel regions) [?]: 0.133s (2.0%)

Parallel Region Time [?]: 6.480s (98.0%)

Estimated Ideal Time [?]: 3.628s (54.9%)

OpenMP Potential Gain [?]: 2.852s (43.1%) ↘

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric assumes no runtime overhead.

OpenMP Region	OpenMP Potential Gain [?] (%) [?]	OpenMP Region Time [?]
iso_3dfd\$omp\$parallel:36@unknown:39:56	2.843s ↘ 43.0% ↘	6.468s
main\$omp\$parallel:36@unknown:128:134	0.009s 0.1%	0.012s

CPU Usage Histogram

Memory Bound [?]: 27.7% ↘ of Pipeline Slots

Cache Bound [?]: 24.0% ↘ of Clockticks

DRAM Bound [?]: 2.8% of Clockticks

NUMA: % of Remote Accesses [?]: 31.2%

Bandwidth Utilization Histogram

FPU Utilization [?]: Not supported for this CPU.

- We went from 15.2s to 6.6s ($\approx 2.3X$ additional speedup)
- Threading seems a bit low but the test case might be a bit small
- Let's try to increase the size of data to see if something shows up

ISO3DFD DEV03 - CACHE BLOCKING



MEMORY ISSUES?

Collection Log Analysis Target Analysis Type Summary Bottom-up iso-3dfd_pa...

Elapsed Time [?]: 31.720s

SP GFLOPS [?]: Not supported for this CPU.

CPU Utilization [?]: 61.8% ↗

Average CPU Usage [?]: 22.237 Out of 36 logical CPUs

Serial Time (outside parallel regions) [?]: 0.517s (1.6%)

Top Serial Hotspots (outside parallel regions)

Parallel Region Time [?]: 31.203s (98.4%)

Estimated Ideal Time [?]: 27.163s (85.6%)

OpenMP Potential Gain [?]: 4.040s (12.7%) ↗

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that runtime overhead.

OpenMP Region	OpenMP Potential Gain [?] (%) [?]	OpenMP Region Time [?]
iso_3dfd\$omp\$parallel:36@unknown:39:56	4.035s ↗ 12.7% ↗	31.197s
main\$omp\$parallel:36@unknown:128:134	0.005s 0.0%	0.006s

CPU Usage Histogram

Memory Bound [?]: 49.9% ↗ of Pipeline Slots

Cache Bound [?]: 45.1% ↗ of Clockticks

DRAM Bound [?]: 4.6% of Clockticks

DRAM Bandwidth Bound [?]: 0.0% of Elapsed Time

NUMA: % of Remote Accesses [?]: 56.8%

Bandwidth Utilization Histogram

Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth usage.

Bandwidth Domain: DRAM, GB/sec

Bandwidth Utilization Histogram

Top Functions with High Bandwidth Utilization

Looking at the bottom up analysis might give more details

HPC PERFORMANCE CHARACTERIZATION INDICATES MEMORY BOUND EXECUTION

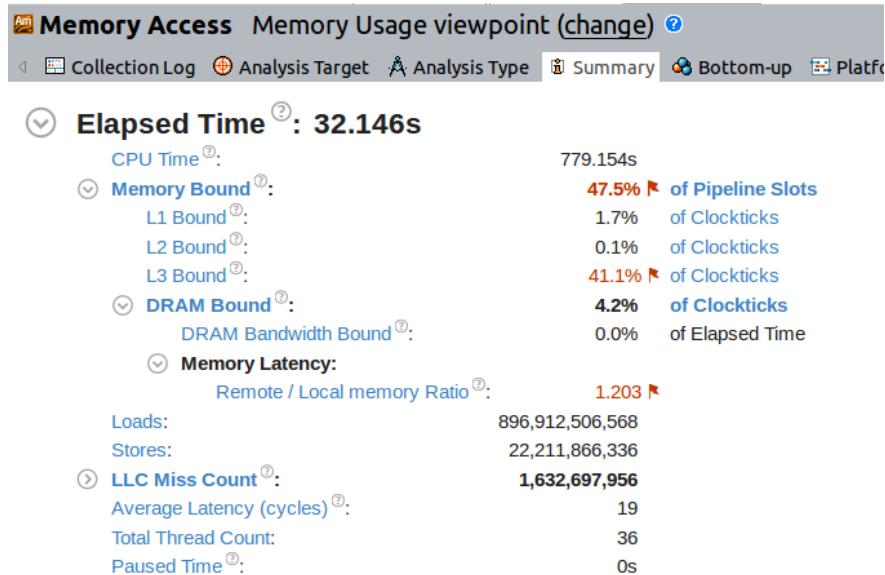
We might have memory issues,
Let's use the memory access
analysis to investigate.

```
S ▲  
31 #include "iso-3dfd.h"  
32 #include <omp.h>  
33 #include <stdio.h>  
34  
35 void iso_3dfd it(flo  
36     const int  
37     const int  
38     int dimnln2 = n1*n2//This value will be used later  
39 #pragma omp parallel for default(shared)  
40     for(int iz=0; iz<n3; iz++) {  
41         for(int iy=0; iy<n2; iy++) {  
42             for(int ix=0; ix<n1; ix++) {  
43                 if( ix>=HALF_LENGTH && ix<(n1-HALF_LENGTH) && iy>=HALF_LENGTH && i  
44                     int offset = ix*dimnln2 + iy*n1 + ix;  
45                     float value = 0.0;  
46                     value += ptr_prev[offset]*coeff[0];  
47                     for(int ir=1; ir<=HALF_LENGTH; ir++) {  
48                         value += coeff[ir] * (ptr_prev[offset + ir] + ptr_prev[offset + ir+1]);  
49                         value += coeff[ir] * (ptr_prev[offset + ir*n1] + ptr_prev[offset + ir*n1+1]);  
50                         value += coeff[ir] * (ptr_prev[offset + ir*dimnln2] + ptr_prev[offset + ir*dimnln2+1]);  
51                     }  
52                     ptr_next[offset] = 2.0f* ptr_prev[offset] - ptr_next[offset] +  
53                 }  
54             }  
55         }
```

Memory Bound	Source File
0.191s	0.0% iso-3dfd_parallel.cc
3.291s	9.8% iso-3dfd_parallel.cc
1.867s	36.0% iso-3dfd_parallel.cc
179.820s	61.2% iso-3dfd_parallel.cc
73.237s	60.1% iso-3dfd_parallel.cc
74.715s	48.6% iso-3dfd_parallel.cc
20.890s	38.9% iso-3dfd_parallel.cc

L3 BOUND?

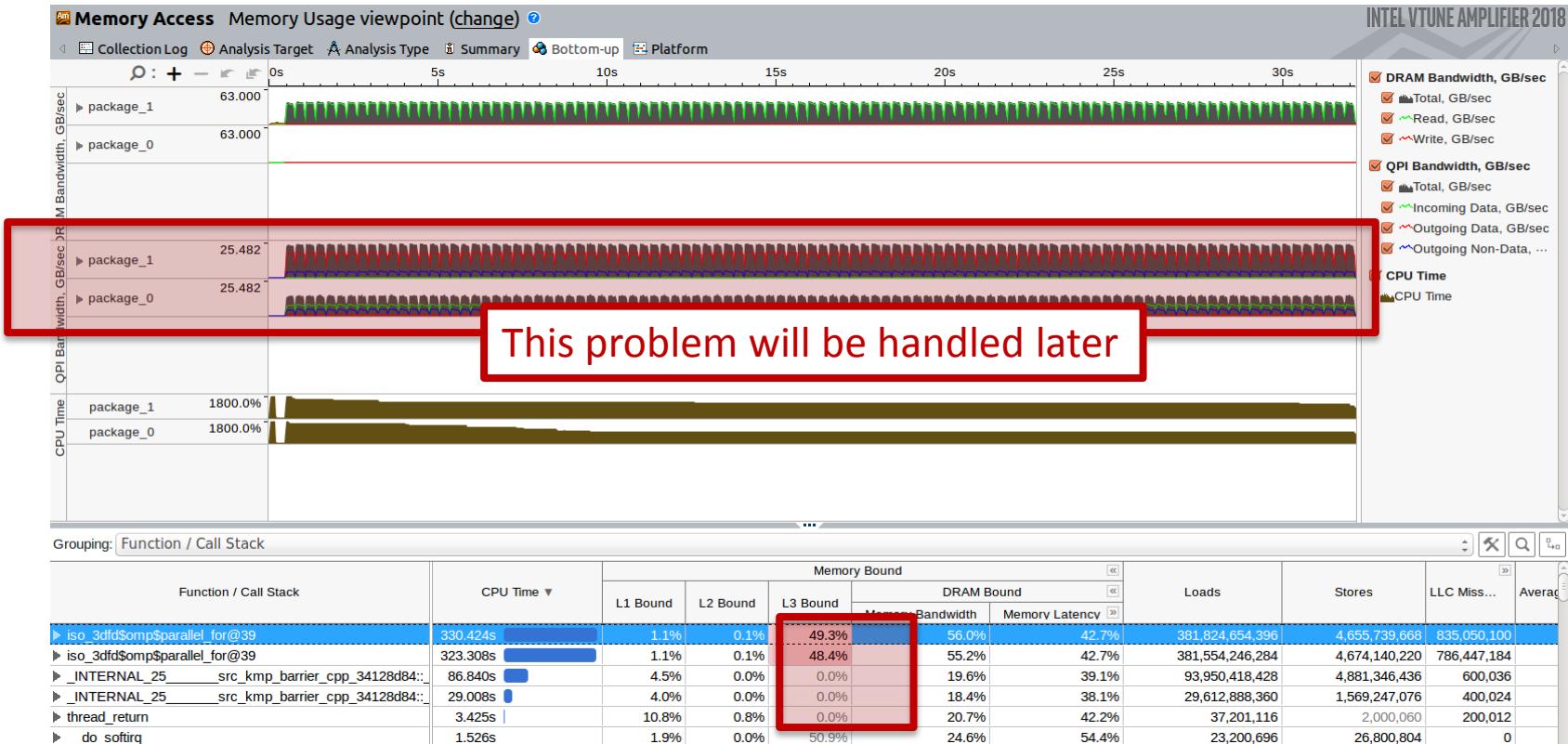
- It looks like we are spending a significant amount of time waiting data from the L3



In an ideal case, we would expect to be L1 bound (meaning that data mainly come from L1)

The bottom up view can give more precise information

MEMORY ACCESS ANALYSIS - BOTTOM UP VIEW



LATENCY IS STILL IMPORTANT ON A BIGGER TESTCASE

S Li. ▲	Source	CPU Time	M B	L1 Cache	L2 Cache	L3 Cache	LL Cache
32	#include <omp.h>						
33	#include <stdio.h>						
34							
35	void iso_3dfd_it(float *ptr_next, float *ptr_prev, float *ptr_vel, float						
36	const int n1, const int n2, const int n3, const int num_threads,						
37	const int n1_Tblock, const int n2_Tblock, const int n3_Tblock){						
38	int dimm1n2 = n1*n2;//This value will be used later						
39	#pragma omp parallel for default(shared)						
40	for(int iz=0; iz<n3; iz++) {						
41	for(int iy=0; iy<n2; iy++) {	0.011s	17.9%	0	0	0	
42	for(int ix=0; ix<n1; ix++) {	0.233s	0.0%	0	0	0	
43	if(ix>=HALF_LENGTH && ix<(n1-HALF_LENGTH) && iy>=HALF_LENGTH	3.198s	9.4%	1,106,833,204	800,024	0	
44	int offset = iz*dimm1n2 + iy*n1 + ix;						
45	float value = 0.0;						
46	value += ptr_prev[offset]*coeff[0];	1.852s	36.1%	102,003,060	800,024	3,400,204	
47	for(int ir=1; ir<HALF_LENGTH; ir++) {						
48	value += coeff[ir] * (ptr_prev[offset + ir] + ptr_pre	108.898s	61.3%	221,545,846,...	56,001,680	22,201,332	
49	value += coeff[ir] * (ptr_prev[offset + ir*n1] + ptr_	67.650s	57.9%	39,676,790,268	92,002,760	17,801,068	
50	value += coeff[ir] * (ptr_prev[offset + ir*dimm1n2] +	69.015s	46.2%	86,685,800,496	160,804,824	749,644,976	
51	}						
52	ptr_next[offset] = 2.0f* ptr_prev[offset] - ptr_next[offs	19.567s	39.7%	32,707,381,192	4,345,330,356	42,002,520	
53	}						

Reminder:

L1 =~ 5 cycles

L2 =~ 12 cycles

L3 =~ 40-60 cycles

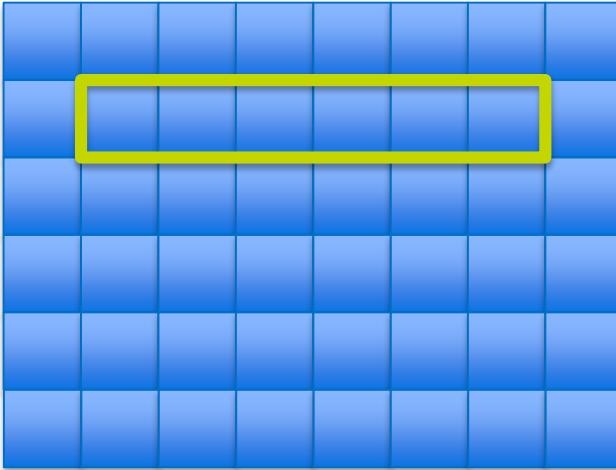
So the number of cycles is still high,
20-40 cycles means that many accesses
are resolved in L3.

CACHE BLOCKING

- We observed that our data mainly come from L3
 - Average number of cycle for accessing data is high (close to L3 latency)
 - Vtune reports the kernel to be memory bound in L3
- We can try to implement Cache Blocking
 - Cache blocking consists in creating blocks in the 3D volume and to compute block by block
 - It increases the chance to benefit from data locality

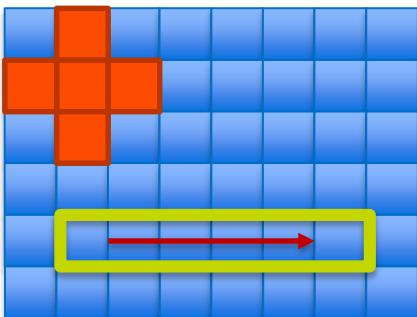
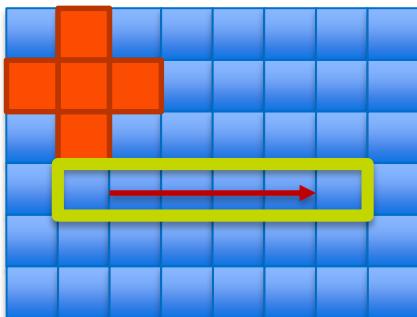
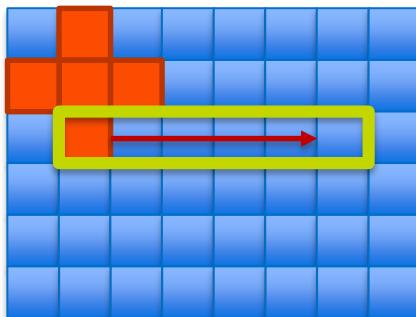
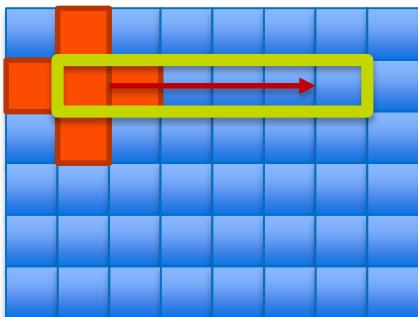
STENCIL WITHOUT CACHE BLOCKING

- We refresh cells following unit stride order



STENCIL WITHOUT CACHE BLOCKING

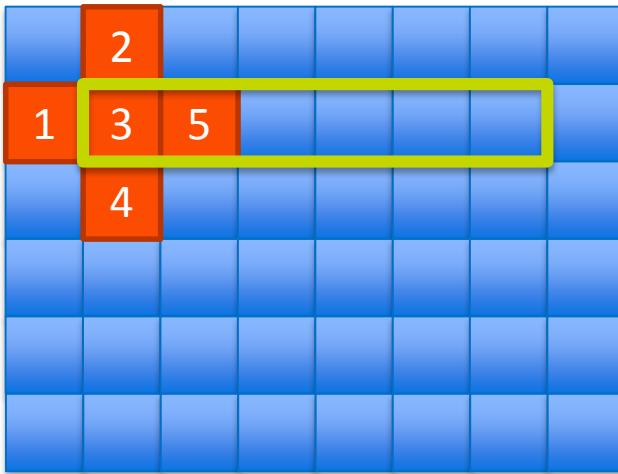
- We refresh cells following unit stride order



- We iterate with 2 for loops
- ```
for(int i=0;i<imax;i++)
 for(int j=0; j<jmax; j++)
 tab[i][j]=...
```

# STENCIL WITHOUT CACHE BLOCKING

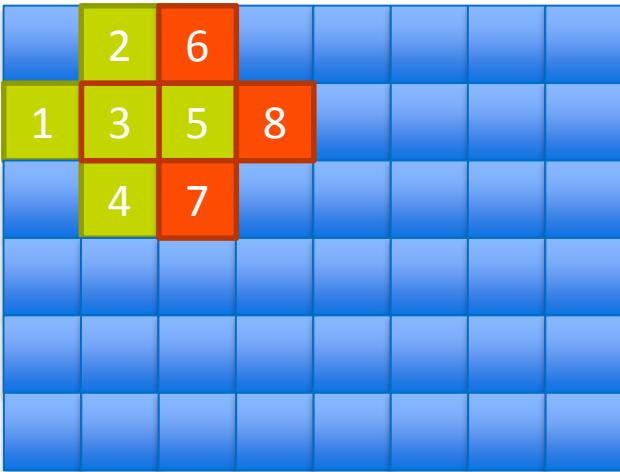
- Let's assume we can store 16 cells in cache (fifo policy)



0 cache evictions  
5 cache misses

# STENCIL WITHOUT CACHE BLOCKING

- Let's assume we can store 16 cells in cache (fifo policy)

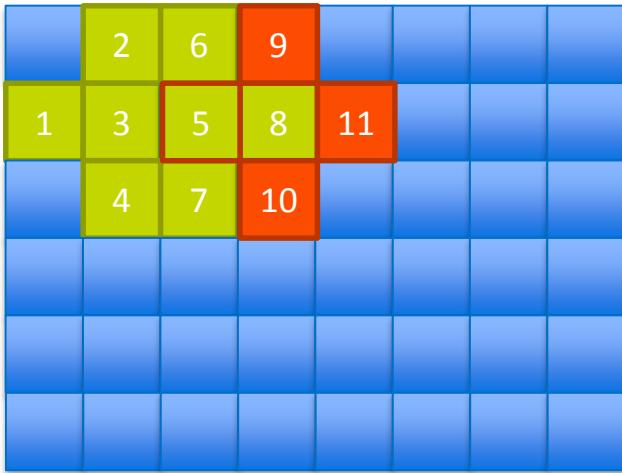


Memory cell still in cache

0 cache evictions  
8 cache misses

# STENCIL WITHOUT CACHE BLOCKING

- Let's assume we can store 16 cells in cache (fifo policy)



Memory cell still in cache

0 cache evictions  
11 cache misses

# STENCIL WITHOUT CACHE BLOCKING

- Let's assume we can store 16 cells in cache (fifo policy)

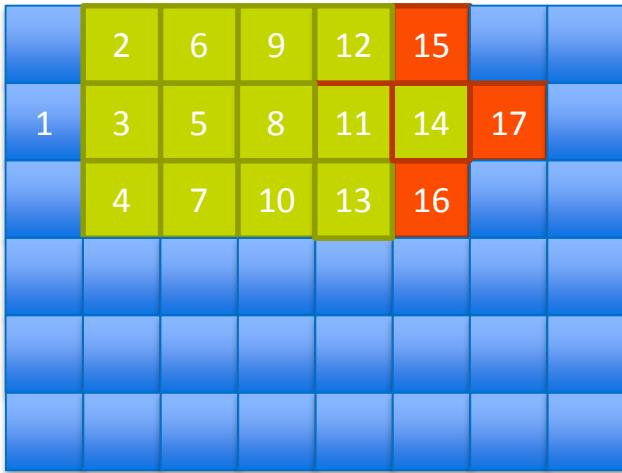


Memory cell still in cache

0 cache evictions  
14 cache misses

# STENCIL WITHOUT CACHE BLOCKING

- Let's assume we can store 16 cells in cache (fifo policy)

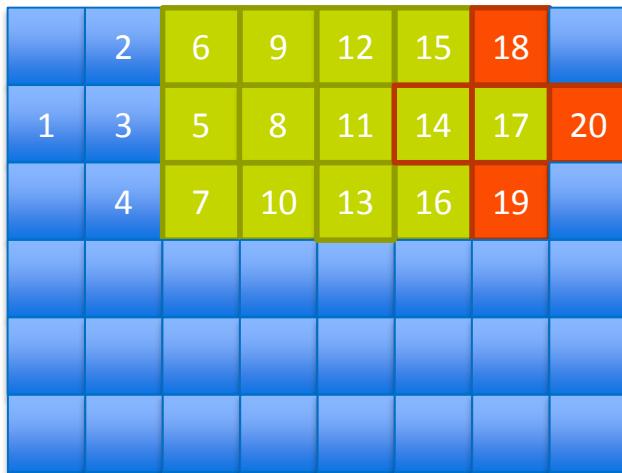


Memory cell still in cache

1 cache evictions  
17 cache misses

# STENCIL WITHOUT CACHE BLOCKING

- Let's assume we can store 16 cells in cache (fifo policy)

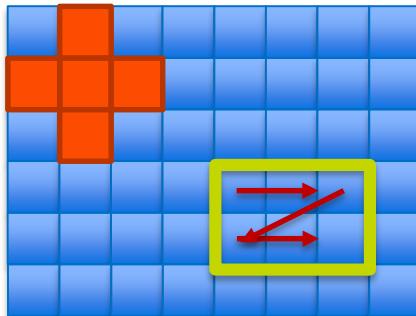
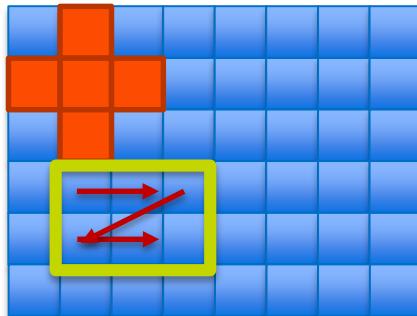
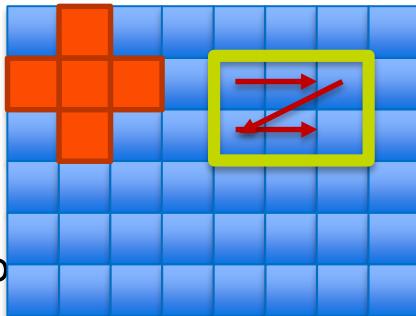
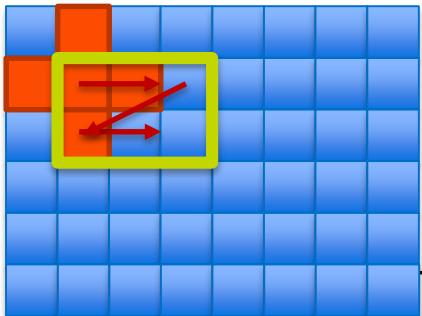


## Memory cell still in cache

4 cache evictions  
20 cache misses

# STENCIL WITH CACHE BLOCKING

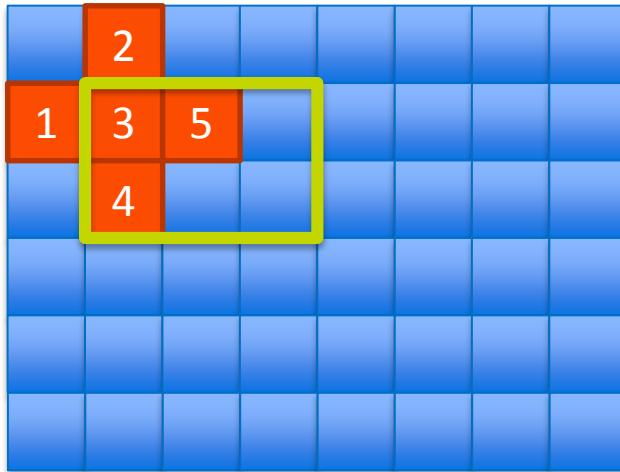
- We create blocks and we iterate using unit strides inside the blocks.



```
for(int bi=0;bi<iMax;i+=Bl)
 for(int bj=0; bj<jMax; j+=BJ)
 for(int i=bi; i<bi+Bl; i++)
 for(int j=bj; j<bj+BJ; j++)
 tab[i][j]=...
```

# STENCIL WITH CACHE BLOCKING

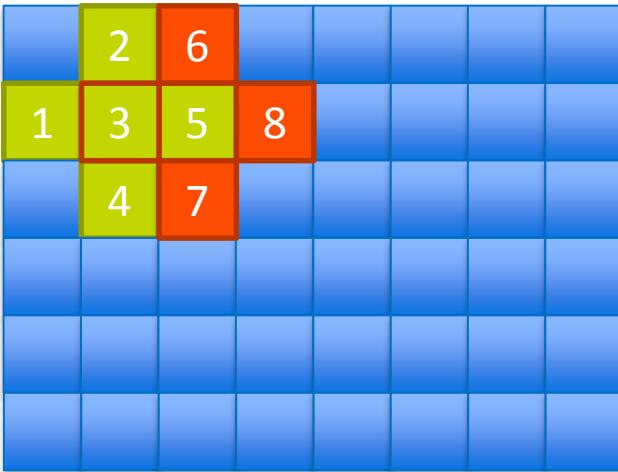
- Let's assume we can store 16 cells in cache (fifo policy)



0 cache evictions  
5 cache misses

# STENCIL WITHOUT CACHE BLOCKING

- Let's assume we can store 16 cells in cache (fifo policy)



Memory cell still in cache

0 cache evictions  
8 cache misses

# STENCIL WITHOUT CACHE BLOCKING

- Let's assume we can store 16 cells in cache (fifo policy)

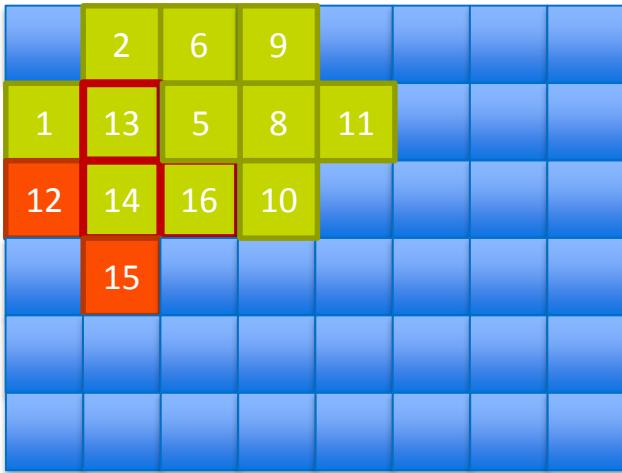


Memory cell still in cache

0 cache evictions  
11 cache misses

# STENCIL WITHOUT CACHE BLOCKING

- Let's assume we can store 16 cells in cache (fifo policy)

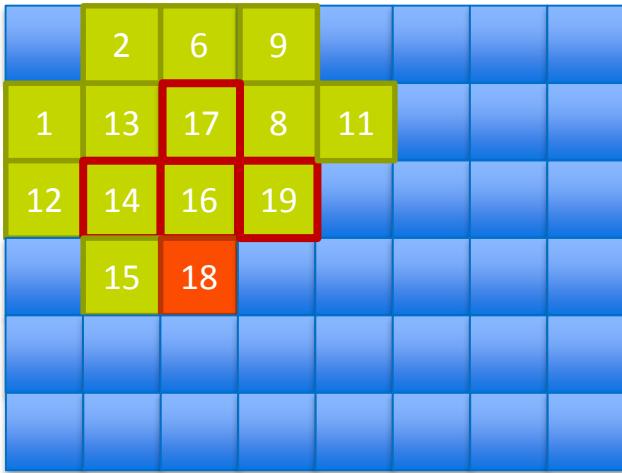


Memory cell still in cache

0 cache evictions  
13 cache misses

# STENCIL WITHOUT CACHE BLOCKING

- Let's assume we can store 16 cells in cache (fifo policy)



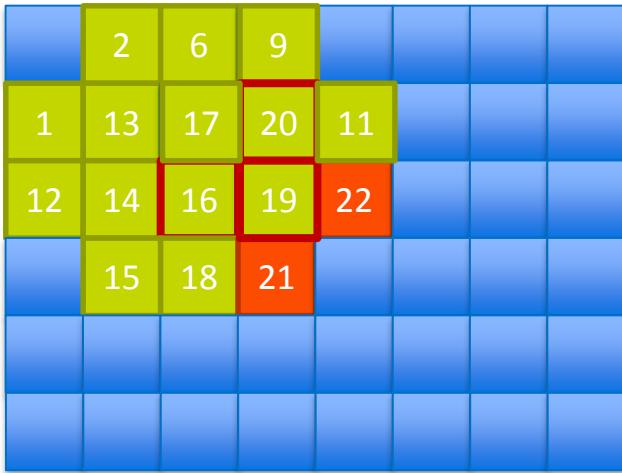
Memory cell still in cache

0 cache evictions

14 cache misses

# STENCIL WITHOUT CACHE BLOCKING

- Let's assume we can store 16 cells in cache (fifo policy)



Memory cell still in cache

0 cache evictions  
16 cache misses

# COMPARISON ?

- In both cases we update 6 elements (center of the stencil)
- Without cache blocking
  - 4 cache eviction
  - 20 cache misses
- With cache blocking
  - 0 cache eviction
  - 16 cache misses

# HOW TO IMPLEMENT CACHE BLOCKING ?

- Need to create an additional for loop for each dimension
  - General principle
  - In practice, you usually don't want to do cache blocking in the unit stride dimension (it breaks prefetching)
  - It adds a significant overhead in code readability
- Other solution, use TBB if you are in C++
  - Use blocked\_range feature of TBB

# BASIC IMPLEMENTATION

```
int nZEnd = nZ - HALF_LENGTH;
int nYEnd = nY - HALF_LENGTH;
int nXEnd = nX - HALF_LENGTH;

#pragma omp parallel for OMP_SCHEDULE OMP_N_THREADS collapse(3) default(shared)
for(int bz=HALF_LENGTH; bz<nZEnd; bz+=nZ_Sblock){
 for(int by=HALF_LENGTH; by<nYEnd; by+=nY_Sblock){
 for(int bx=HALF_LENGTH; bx<nXEnd; bx+=nX_Sblock){
 int izEnd = MIN(bz+nZ_Sblock, nZEnd);
 int iyEnd = MIN(by+nY_Sblock, nYEnd);
 int ixEnd = MIN(bx+nX_Sblock, nXEnd);
 for(int iz=bz; iz<izEnd; iz++) {
 for(int iy=by; iy<iyEnd; iy++) {
 for(int ix=bx; ix<ixEnd; ix++) {
```

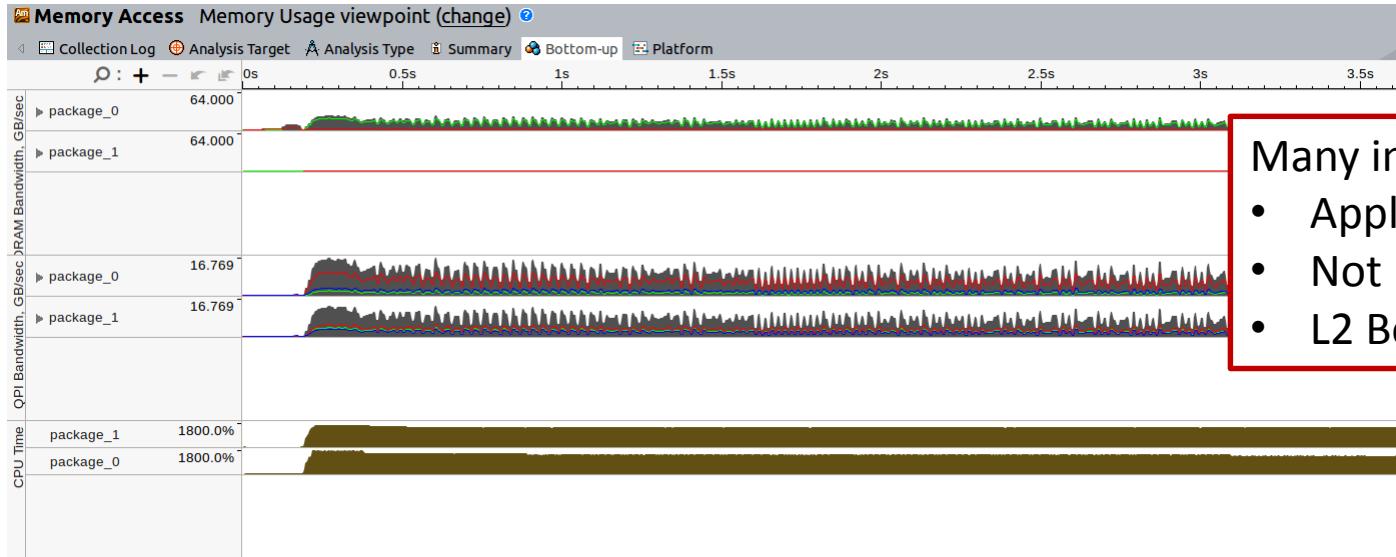
# WHAT IS THE PERFORMANCE OF DEVO3 ?

The screenshot shows the HPC Performance Characterization tool interface. At the top, there's a toolbar with various icons and a menu bar with 'Welcome' and 'hpcperfde... X'. Below that is a navigation bar with tabs: 'Collection Log', 'Analysis Target', 'Analysis Type', 'Summary', and 'Bottom-up'. The main content area displays the following metrics:

- Elapsed Time**: 3.732s
- SP GFLOPS**: Not supported for this CPU.
- CPU Utilization**: 74.7%
- Average CPU Usage: 26.903 Out of 36 logical CPUs
- Serial Time (outside parallel regions): 0.192s (5.2%)
- Parallel Region Time: 3.540s (96.6%)
- CPU Usage Histogram
- Memory Bound**: 11.7% of Pipeline Slots
  - Cache Bound: 10.8% of Clockticks
  - DRAM Bound: 1.2% of Clockticks
  - NUMA: % of Remote Accesses: 0.0%
  - Bandwidth Utilization Histogram
- FPU Utilization**: Not supported for this CPU.
- Collection and Platform Info**

- We went from 6.6s to 3.7s ( $\approx 1.7X$  additional speedup)
- We are back to the small test case to have fair comparison
- Cache blocking seems to significantly improve our memory accesses
- Better CPU utilization due to less memory contention

# WHAT ABOUT THE L3 AND L2 NOW?



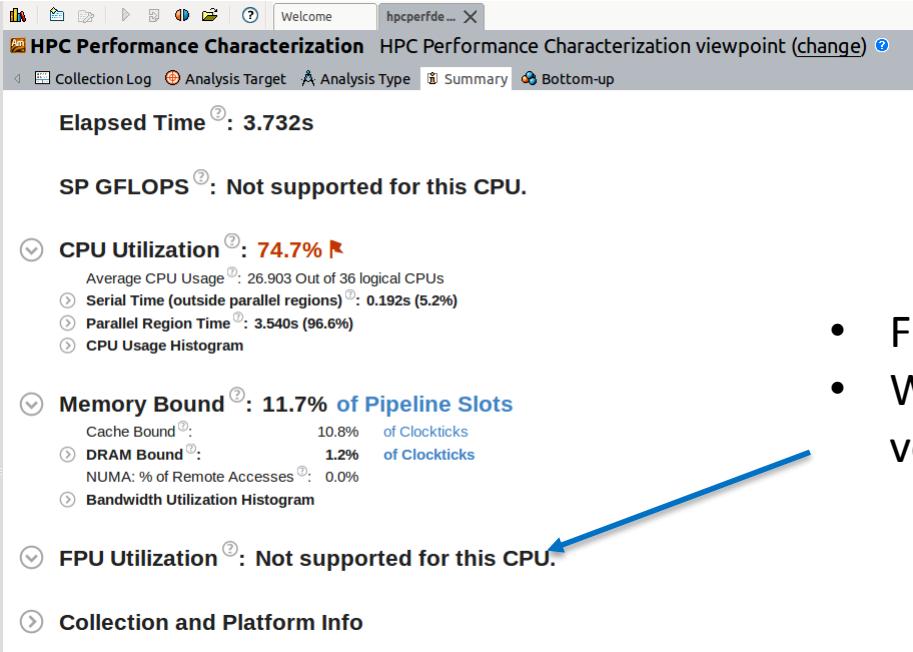
Many interesting points:

- Application runs faster
- Not L3 bounded anymore
- L2 Bound > L3 Bound

Grouping: Function / Call Stack

| Function / Call Stack                       | CPU Time ▾ | Memory Bound     |                |       |            |       |                | Loads | Stor |  |  |
|---------------------------------------------|------------|------------------|----------------|-------|------------|-------|----------------|-------|------|--|--|
|                                             |            | L1 Bound         |                |       | DRAM Bound |       |                |       |      |  |  |
|                                             |            | Memory Bandwidth | Memory Latency |       |            |       |                |       |      |  |  |
| iso_3dfd\$omp\$parallel_for@51              | 48.559s    | 1.7%             | 7.1%           | 2.5%  | 11.9%      | 72.8% | 72,316,969,444 | 1,12  |      |  |  |
| iso_3dfd\$omp\$parallel_for@51              | 48.275s    | 1.6%             | 7.2%           | 2.3%  | 11.4%      | 70.0% | 72,219,366,516 | 1,13  |      |  |  |
| _INTERNAL_25_src_kmp_barrier_cpp_34128d84:: | 4.849s     | 4.7%             | 0.0%           | 0.0%  | 47.1%      | 36.9% | 4,976,149,280  | 28    |      |  |  |
| _INTERNAL_25_src_kmp_barrier_cpp_34128d84:: | 0.264s     | 3.2%             | 0.0%           | 1.1%  | 44.7%      | 36.2% | 303,609,108    | 1     |      |  |  |
| do_sofirn                                   | 0.144s     | 10.5%            | 23.0%          | 14.7% | 21.0%      | 62.9% | 3,600,108      | 1     |      |  |  |

# TIME TO LOOK AT VECTORIZATION?



The screenshot shows the HPC Performance Characterization tool interface. At the top, there's a toolbar with various icons and a menu bar. Below that is a navigation bar with tabs: 'Welcome' (selected), 'hpcperfde...', 'Collection Log', 'Analysis Target', 'Analysis Type' (selected), 'Summary', and 'Bottom-up'. The main content area displays performance metrics:

- Elapsed Time**: 3.732s
- SP GFLOPS**: Not supported for this CPU.
- CPU Utilization**: 74.7% 
  - Average CPU Usage: 26.903 Out of 36 logical CPUs
  - Serial Time (outside parallel regions): 0.192s (5.2%)
  - Parallel Region Time: 3.540s (96.6%)
  - CPU Usage Histogram
- Memory Bound**: 11.7% of Pipeline Slots
  - Cache Bound: 10.8% of Clockticks
  - DRAM Bound: 1.2% of Clockticks
  - NUMA: % of Remote Accesses: 0.0%
  - Bandwidth Utilization Histogram
- FPU Utilization**: Not supported for this CPU. 
- Collection and Platform Info**

- Flops counters not available on this hardware ?
- We can use Intel® Advisor to enable vectorization

# HOW CAN ADVISOR EXTRACT VECTORIZATION INFORMATIONS ?

- Vtune relies on Hardware counter to extract flops
  - No available hardware counter = No flops reported
  - Hardware counters have been broken on our previous platforms
  - Few ms run might result on non-accurate results
- Intel® Advisor uses instrumentation
  - Advisor can extract data from the binary (from internal markers that the compiler put in your executable)
  - Advisor looks for instructions executed and count **real flops** (takes masking into account)
  - It only uses sampling for the timings

# ISO3DFD DEV04 - VECTORIZATION

# USING INTEL® ADVISOR ON DEVO3

The screenshot shows the Intel Advisor interface. At the top, there are tabs for 'Summary', 'Survey & Roofline', and 'Refinement Reports'. A warning message states: 'Some target modules are compiled with optimization disabled. Suggestion: rebuild with version 15.0 or higher of the intel compiler and enable debug information and optimization before rebuilding.' The main area is divided into two sections: 'ROOFLINE' and 'Source'.

**ROOFLINE:** This section displays a table of performance issues. One row is highlighted in orange, indicating a 'Scalar loop. Not vectorized: vector dependence prevents vectorization'. The table includes columns for 'Performance Issues', 'Self Time', 'Total Time', 'Type', and 'Why No Vectorization'.

| Performance Issues                                                                                       | Self Time | Total Time | Type   | Why No Vectorization       |
|----------------------------------------------------------------------------------------------------------|-----------|------------|--------|----------------------------|
| Scalar loop. Not vectorized: vector dependence prevents vectorization<br>No loop transformations applied | 63.932s   | 63.932s    | Scalar | vector dependence          |
| Scalar loop. Not vectorized: vector dependence prevents vectorization<br>No loop transformations applied | 63.462s   | 63.462s    | Scalar | vector dependence prev...  |
|                                                                                                          | 0.100s    | 63.562s    | Scalar | outer loop was not auto... |
|                                                                                                          | 0.090s    | 63.652s    | Scalar | outer loop was not auto... |
|                                                                                                          | 0.080s    | 64.012s    | Scalar | outer loop was not auto... |
|                                                                                                          | 0.070s    | 64.082s    | Scalar | outer loop was not auto... |
|                                                                                                          | 0.069s    | 0.069s     | Scalar | vector dependence prev...  |

**Source:** This section shows the C++ code for 'iso\_3dfd\_parallel.cc:64'. The code implements a 3D finite difference stencil. Several loops are annotated with Intel Advisor's analysis. Lines 64 and 65 are highlighted in orange, both reporting 'Scalar loop. Not vectorized: vector dependence prevents vectorization'. The 'Source' column shows the code, and the 'Total Time' and '% Loop/Function Time' columns show the execution times for these specific lines.

| Line | Source                                                                | Total Time | % | Loop/Function Time | % | Traits |
|------|-----------------------------------------------------------------------|------------|---|--------------------|---|--------|
| 56   | int iyEnd = MIN(by+n2_Tblock, n2End);                                 |            |   |                    |   |        |
| 57   | int ixEnd = MIN(n1_Tblock, n1End-bx);                                 |            |   |                    |   |        |
| 58   | int ix;                                                               |            |   |                    |   |        |
| 59   | for(int iz=bz; iz<izEnd; iz++) {                                      |            |   |                    |   |        |
| 60   | for(int iy=by; iy<iyEnd; iy++) {                                      |            |   |                    |   |        |
| 61   | float* ptr_next = ptr_next_base + iz*dimnln2 + iy*n1 + bx;            |            |   |                    |   |        |
| 62   | float* ptr_prev = ptr_prev_base + iz*dimnln2 + iy*n1 + bx;            |            |   |                    |   |        |
| 63   | float* ptr_vel = ptr_vel_base + iz*dimnln2 + iy*n1 + bx;              |            |   |                    |   |        |
| 64   | for(int ix=0; ix<ixEnd; ix++) {                                       | 0.500s (   |   | 63.932s            |   |        |
|      | [loop in iso_3dfd\$omp\$parallel_for@51 at iso_3dfd_parallel.cc:64]   |            |   |                    |   |        |
|      | Scalar loop. Not vectorized: vector dependence prevents vectorization |            |   |                    |   |        |
|      | No loop transformations applied                                       |            |   |                    |   |        |
|      | [loop in iso_3dfd\$omp\$parallel_for@51 at iso_3dfd_parallel.cc:64]   |            |   |                    |   |        |
|      | Scalar loop. Not vectorized: vector dependence prevents vectorization |            |   |                    |   |        |
|      | No loop transformations applied                                       |            |   |                    |   |        |
| 65   | float value = 0.0;                                                    |            |   |                    |   |        |

- Advisor detects 2 significant loops not vectorized
- Vector dependence prevents vectorization

# DEPENDENCY ANALYSIS ON DEVO3

The screenshot shows the Intel Advisor interface. On the left, a tree view under 'Function Call Sites and Loops' shows several loop nodes. The main panel displays a table of 'Performance Issues' with columns: Performance Issues, Self Time, Total Time, Type, Why No Vectorization?, Vectorized Loops, and FLOPS. One row is highlighted in black, indicating an 'Assumed dependency present'. Below the table, there are sections for 'Recommendations' and 'Why No Vectorization?'.

**Step 1: Look at recommendations**

**Recommendation:** Confirm dependency is real  
There is no confirmation that a real (proven) dependency is present in the loop. To confirm: Run a Dependencies analysis.

**Issue:** Assumed dependency present  
The compiler assumed there is an anti-dependency (Write after read - WAR) or a true dependency (Read after write - RAW) in the loop. Improve performance by investigating

The screenshot shows the Intel Advisor interface with the 'Dependencies Report' tab selected. A table lists site locations and their dependency status. A red box highlights a row where 'No dependencies found' is listed twice. To the right, a large red box contains the 'Step 2: Advisors tells you what to do' section.

**Step 2: Advisors tells you what to do**

**Recommendation:** Enable vectorization  
The Dependencies analysis shows there is no real dependency in the loop for the given workload. Tell the compiler it is safe to vectorize using the `restrict` keyword or a directive:

| Directive                                                  | Outcome                                            |
|------------------------------------------------------------|----------------------------------------------------|
| <code>#pragma simd</code> or <code>#pragma omp simd</code> | Ignores all dependencies in the loop               |
| <code>#pragma ivdep</code>                                 | Ignores only vector dependencies (which is safest) |

**Issue:** Assumed dependency present  
The compiler assumed there is an anti-dependency (Write after read - WAR) or a true dependency (Read after write - RAW) in the loop. Improve performance by investigating the assumption and handling accordingly.

**Enable vectorization**

# ADDING VECTORIZATION ON DEVO3

```
#pragma omp simd
for(int ix=0; ix<ixEnd; ix++) {
 float value = 0.0;
 value += ptr_prev[ix]*coeff[0];
 #pragma unroll(HALF_LENGTH)
 for(int ir=1; ir<=HALF_LENGTH; ir++) {
 value += coeff[ir] * (ptr_prev[ix + ir] + ptr_prev[ix - ir]);// horizontal
 value += coeff[ir] * (ptr_prev[ix + ir*n1] + ptr_prev[ix - ir*n1]);// vertical
 value += coeff[ir] * (ptr_prev[ix + ir*dimn1n2] + ptr_prev[ix - ir*dimn1n2]); // in front / behind
 }
 ptr_next[ix] = 2.0f* ptr_prev[ix] - ptr_next[ix] + value*ptr_vel[ix];
```

# DEVO4 - HPC PERFORMANCE CHARACTERIZATION

**HPC Performance Characterization** HPC Performance Characterization

Collection Log Analysis Target Analysis Type Summary Bottom-up

Elapsed Time : 1.267s

SP GFLOPS : Not supported for this CPU.

CPU Utilization : 66.0%

Average CPU Usage : 23.761 Out of 36 logical CPUs

Serial Time (outside parallel regions) : 0.199s (15.7%)

Parallel Region Time : 1.067s (84.3%)

CPU Usage Histogram

Memory Bound : 39.6% of Pipeline Slots

Cache Bound : 24.1% of Clockticks

DRAM Bound : 13.3% of Clockticks

DRAM Bandwidth Bound : 10.3% of Elapsed Time

NUMA: % of Remote Accesses : 46.2%

Bandwidth Utilization Histogram

- We went from 3.7s to 1.2s (=~3X additional speedup)
- As we are doing more flops, we request also more data.

# ISO3DFD DEV05 - NUMA EFFECT



# DEVO4 - REMOTE MEMORY ACCESSES ?

The screenshot shows the Intel HPC Performance Characterization interface. At the top, it displays "HPC Performance Characterization" twice. Below that is a navigation bar with tabs: "Collection Log", "Analysis Target", "Analysis Type", "Summary" (which is selected), and "Bottom-up".

Elapsed Time <sup>?</sup>: 1.267s

SP GFLOPS <sup>?</sup>: Not supported for this CPU.

CPU Utilization <sup>?</sup>: 66.0%

Average CPU Usage <sup>?</sup>: 23.761 Out of 36 logical CPUs

Serial Time (outside parallel regions) <sup>?</sup>: 0.199s (15.7%)

Parallel Region Time <sup>?</sup>: 1.067s (84.3%)

CPU Usage Histogram

Memory Bound <sup>?</sup>: 39.6% of Pipeline Slots

Cache Bound <sup>?</sup>: 24.1% of Clockticks

DRAM Bound <sup>?</sup>: 13.3% of Clockticks

DRAM Bandwidth Bound <sup>?</sup>: 10.3% of Elapsed Time

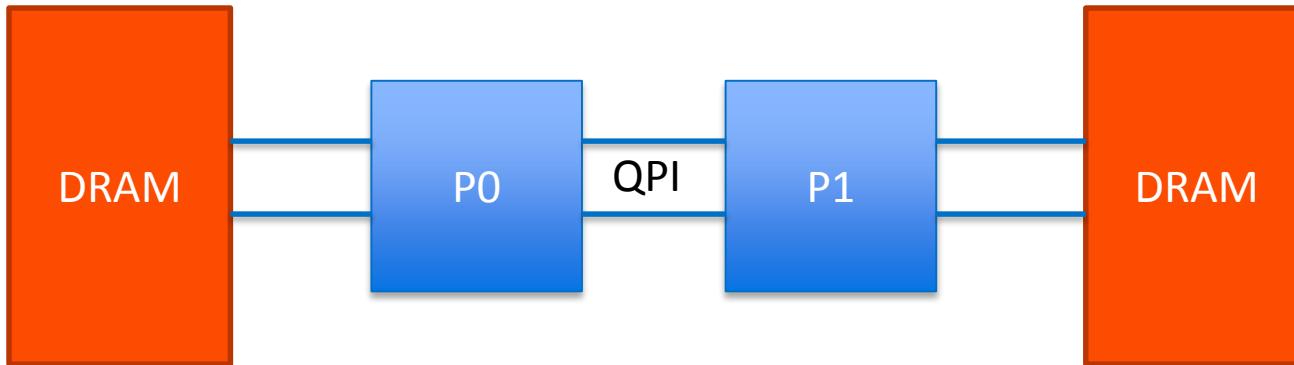
NUMA: % of Remote Accesses <sup>?</sup>: 46.2%

Bandwidth Utilization Histogram

- We have done significant progress
- But memory seems to be a bottleneck again
- Vtune indicates a high ratio of NUMA accesses

# WHAT IS NON UNIFORM MEMORY ACCESS (NUMA)

- Wikipedia: Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor.
- Multi-socket system are sensitive to NUMA



# HOW TO SOLVE NUMA PROBLEMS ?

- Implements First touch policy

```
#pragma omp parallel for schedule(static)
```

```
for(int i=0;i<N;i++){
```

```
 tab[i] = 0;
```

```
 output[i] = 0;
```

```
}
```

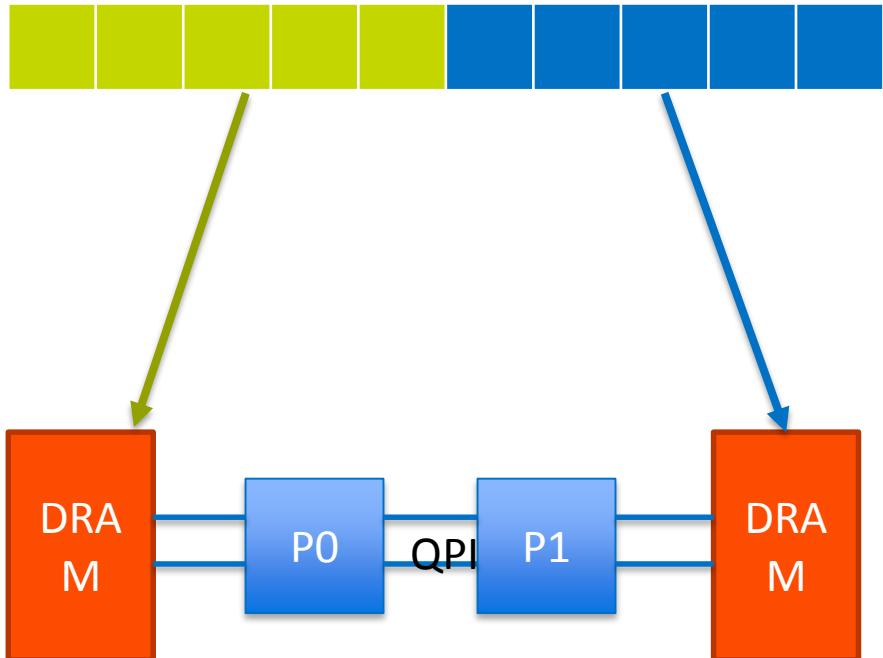
```
//initialize output and tab
```

```
real_init(tab, output);
```

```
#pragma omp parallel for schedule(static)
```

```
for(int i=0;i<N;i++)
```

```
 output[i] = tab[i]*.....
```



# HOW TO SOLVE NUMA PROBLEMS 2 ?

- Use MPI
  - You need to modify your application to handle MPI domain decomposition
  - Use MPI runtime to bind one process per NUMA node (socket)
  - You have to handle communications between your nodes
  - If a process is bind to a socket, it will allocate the memory on the closest NUMA node.

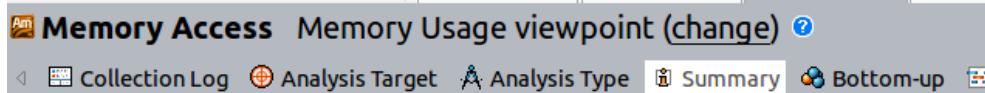
# HOW TO TEST YOUR IDEAL PERFORMANCE ?

- Run your application with a given test case
  - Look at the timing
  - You can also check bandwidth on QPI with Vtune
- Run 2 instances of your application with NUMA CTL
  - Divide the amount of work by 2
  - Bind 1 process on node 0 and assign memory allocation on socket 0
  - Bind 1 process on node 1 and assign memory allocation on socket 1

# HOW TO TEST YOUR IDEAL PERFORMANCE ?

- bin/iso3dfd\_dev05\_cpu\_avx2.exe 256 300 300 36 100
  - real 0m1.348s
  - user 0m39.340s
  - sys 0m0.500s
  - We went from 1.3s to 0.6s  
(=~2.16X additional speedup)
- numactl --cpunodebind=0 --membind=0 bin/iso3dfd\_dev05\_cpu\_avx2.exe 128 300 300 18 100 &  
numactl --cpunodebind=1 --membind=1 bin/iso3dfd\_dev05\_cpu\_avx2.exe 128 300 300 18 100
  - real 0m0.558s
  - user 0m9.296s
  - sys 0m0.338s

# WITH FIRST TOUCH?



## Elapsed Time: 0.954s

CPU Time:

28.748s

Memory Bound:

41.4% of Pipeline Slots

L1 Bound:

5.9% of Clockticks

L2 Bound:

1.0% of Clockticks

L3 Bound:

17.1% of Clockticks

DRAM Bound:

14.3% of Clockticks

DRAM Bandwidth Bound:

0.0% of Elapsed Time

Memory Latency:

Remote / Local memory Ratio: 1.026

Loads:

30,948,128,416

Stores:

2,174,065,220

LLC Miss Count:

56,403,384

Average Latency (cycles):

17

Total Thread Count:

36

Paused Time:

0s

- We went from 1.2s to 0.9s (=~1.3X additional speedup)
- Overall 315 to 0.9s with
  - Threading
  - Vectorization
  - Cache blocking
  - Memory optimizations
- We can get even more if we tune cache blocking parameters

# FIRST TOUCH MEMORY ACCESS

- How to improve ?

Elapsed Time <sup>?</sup>: 0.954s

SP GFLOPS <sup>?</sup>: Not supported for this CPU.

▽ CPU Utilization <sup>?</sup>: 62.8% 

Average CPU Usage <sup>?</sup>: 22.618 Out of 36 logical CPUs

 CPU Usage Histogram

▽ Memory Bound <sup>?</sup>: 41.4%  of Pipeline Slots

Cache Bound <sup>?</sup>: 24.0%  of Clockticks

 DRAM Bound <sup>?</sup>: 14.3%  of Clockticks

NUMA: % of Remote Accesses <sup>?</sup>: 50.6% 

 Bandwidth Utilization Histogram

▽ FPU Utilization <sup>?</sup>: Not supported for this CPU.

 Collection and Platform Info

We might want to specify OMP scheduling

- It might improve threading
- It might improve NUMA accesses

# FIRST TOUCH MEMORY ACCESS

- Export KMP\_AFFINITY=compact

Elapsed Time <sup>②</sup>: 0.754s

SP GFLOPS <sup>②</sup>: Not supported for this CPU.

- We went from 0.9s to 0.7s  
(=~1.3X additional speedup)
- Overall 315 to 0.7s

⌄ CPU Utilization <sup>②</sup>: **80.2%** ↗

Average CPU Usage <sup>②</sup>: 28.876 Out of 36 logical CPUs

⌄ CPU Usage Histogram

⌄ Memory Bound <sup>②</sup>: **50.3%** ↗ of Pipeline Slots

Cache Bound <sup>②</sup>: **29.0%** ↗ of Clockticks

⌄ DRAM Bound <sup>②</sup>: **18.0%** ↗ of Clockticks

NUMA: % of Remote Accesses <sup>②</sup>: **43.1%** ↗

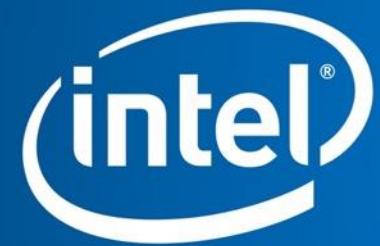
⌄ Bandwidth Utilization Histogram

⌄ FPU Utilization <sup>②</sup>: Not supported for this CPU.

⌄ Collection and Platform Info

# CONCLUSION

- Use Vtune for:
  - Threading (HPC PERFORMANCE CHARACTERIZATION)
  - Memory optimization (MEMORY ACCESS ANALYSIS)
    - Look for Cache Bound, Memory bound, Cache latency
    - Look for data on QPI
- Use Advisor for:
  - Vectorization
  - Profiling data accesses
  - Characterization (Roofline model – not covered here)



Software