Intel® Vtune™ Amplifier and Intel® Advisor - hands-on labs

Carlos Rosales-Fernandez

Introduction

In this hands-on session you will use a simple n-body code to explore the capabilities of Intel® Advisor and Intel® VTune™ Amplifier.

As you progress through the exercises you will investigate the code performance, and use different analysis modes to identify performance issues.

You will not have to modify code directly, all code versions are provided.

The code used is C++, but if you use Fortran in your own work the steps would be exactly the same.

Implementation Details

NOTE: This additional info is provided for your own reference, but it is not necessary to run the exercises.

For each particle the position, the velocity, the acceleration and the mass is stored in a C-like structure and for an N particles case, an array of this structure is allocated. This is the simple data-structure which is very close to the physical representation of a particle mass. The file Particle.hpp contains the implementation of such data-structure.

For each particle indexed by i, the acceleration is computed ai = Gmj(ri-rj)/|ri-rj|^3, which value is used to update the velocity and position using the Euler integration scheme. Furthermore the total energy of the particles' group is computed. The file GSimulation.cpp contains the implementation of the algorithm.

The demo consists of several directories, which correspond to the different optimization steps to take to enabling vectorization and OpenMP multi-threading of the code. Each directory has its own makefile to compile and run the test case. To compiler the code type make and the run the simulation type make run.

As benchmark, the simulation starts with 2000 particles and 500 integration steps. One can change the default giving the number of particles and the number of integration steps using the command line argument: ./nbody.x < # of particles> < # of integration>

Try to change the number of particles and observe how the performance changes.

Getting Started

https://github.com/fbaru-dev/nbody-demo

This is an example code based on a simple N-body simulation of a distribution of point masses placed at location r_1,...,r_N with masses m_1,...,m_N. The position of the particles after a specified time is computed using a finite difference method.

To get started, copy the files to a directory of your choosing in the **/projects** area:

```
$ tar xzvf /projects/SDL_Workshop/crosales/SDL_2018/nbody.tar.gz
```

Then change into the **nbody** directory:

```
$ cd ./nbody
```

Intel® advisor

Collect Roofline Data

Start by building version 2 of the code:

```
$ cd ver2
$ make
```

You can check that the make file contains both appropriate ISA flags for KNL and debug flags.

Now collect both survey and trip counts data using the provided **roofline.run** script. You should look inside the script to make sure you understand the configuration and commands used:

```
$ qsub ./roofline.run
```

Once the run is complete you will have a new directory, **adv_res**, which contains the performance data. Make sure your collection has completed by checking that the job is done:

```
$ qstat -u <username>
```

[*Optional*] Generate a portable snapshot if you wish to look at the results in your own machine:

```
$ advixe-cl --snapshot --project-dir ./adv_res --pack --cache-sources \
  --cache-binaries --search-dir src:=./ --search-dir bin:=./ -- nbody_v2
```

Analyze Roofline Data

You can choose to do this step from a login node or from your own system if it has Intel® Advisor installed and you have generated a snapshot. From a login node simply open the collected data in the GUI:

```
$ source /opt/intel/advisor/advixe-vars.sh
$ advixe-gui ./adv_res
```

Follow the steps we used in the presentation to investigate the code performance - look at the summary, the roofline graph, and the Survey report.

Try to answer the following questions:

- What is the execution time?
- What is the vectorization efficiency?
- Are there expensive operations inhibiting performance? (See the *Code Analytics*)
- From the Roofline representation, can you tell what should be the analysis step?

Collect Memory Access Patterns

Using the current nbody build (ver2), submit the provided script, **map.run**, to perform a Memory Access Patterns analysis. The commands in this script are:

```
$ qsub ./map.run
```

Note: you will have to have completed the Survey and Trip Counts analysis before this step

As in the previous case, your output will be stored in the **adv_res** project directory. Make sure your collection has completed by checking that the job is done:

```
$ qstat -u <username>
```

[*Optional*] Generate a portable snapshot if you wish to look at the results in your own machine:

```
$ advixe-cl --snapshot --project-dir ./adv_res --pack --cache-sources \
  --cache-binaries --search-dir src:=./ --search-dir bin:=./ -- nbody_naive
```

Analyze Memory Access Patterns

You can choose to do this step from a login node or form your own system if it has Intel® Advisor installed. From a login node simply open the collected data in the GUI:

```
$ source /opt/intel/advisor/advixe-vars.sh

$ advixe-gui ./adv_res
```

Follow the steps we used in the presentation to investigate the code performance - look at the **Refinement Reports** tab in the GUI.

Try to answer the following questions:

- What is the stride distribution?
- What is the recommendation given by Intel® Advisor?
- Is there an alternative to the current data layout that may help?

Changing the Code

The MAP analysis should have pointed you to a problem with the data structures in the code. Let' snow build version 3, which changes the default Array of Sructures implementation to Structure of Arrays in the hope of improving performance.

You should still be inside the version 2 directory, so move to the version 3 directory and build the new binary :

```
$ cd ../ver3
$ make
```

Now collect roofline data (survey and tripcounts) again, since you have a new binary:

```
$ qsub roofline.run
```

As with the previous version, once the collection completes you will see a new directory called **adv_res**. Make sure execution is actually complete before moving forward.

Analyze Roofline Data for Version 3

You can choose to do this step from a login node or from your own system if it has Intel® Advisor installed and you have generated a snapshot. From a login node simply open the collected data in the GUI:

```
$ source /opt/intel/advisor/advixe-vars.sh
$ advixe-gui ./adv_res
```

Follow the steps we used in the presentation to investigate the code performance - look at the summary, the roofline graph, and the Survey report.

Did the new Structure of Arrays implementation improve performance?

- What is the vectorization efficiency?
- What is the main performance issue in the current version?
- From the advisor output, can you tell what should be the analysis step?

Collect Dependencies Data

Well, that was a surprise, wasn't it?

Looks like we have introduced data dependencies of some type that are preventing vectorization (or at least the compiler thinks so)

Let's run a dependencies analysis to see if those dependencies are true or just assumed:

```
$ qsub deps.run
```

As with the previous version, once the collection completes you will see a new directory called **adv_res**. Make sure execution is actually complete before moving forward.

Analyze Dependencies

You can choose to do this step from a login node or form your own system if it has Intel® Advisor installed. From a login node simply open the collected data in the GUI:

```
$ source /opt/intel/advisor/advixe-vars.sh

$ advixe-gui ./adv_res
```

Follow the steps we used in the presentation to investigate the code performance - look at the **Refinement Reports** tab in the GUI.

Try to answer the following questions:

- Are there any true dependencies?
- What are the dependency types?
- Can you think of a way to resolve them? (you can see the fixes in version 4)

Performance of the nbody Optimized Version

At this point you should be familiar with the process of using Intel® Advisor.

If you are curious about the performance once the dependencies are fixed, build version 4 and collect roofline data (survey and tripcounts) again.

If you compare this output to the original version 2 data you should observe the following traits:

- Higher vectorization efficiency
- Better looking roofline representation
- Significant performance improvements overall

Intel® Vtune™ Amplifier's application performance snapshot

A Simple APS Report

To run this simple exercise simply go back to the version 2 directory for the nbody test and submit the **aps.run** job script:

```
$ qsub ./aps.run
```

Feel free to inspect the submission script, it simply sets up the environment for **aps** and uses **aprun** to launch the data collection.

Once the job has completed (make sure it is not active in the queue) you can process the data collected in the new **aps_res** directory:

- Setup your environment on the login node to be able to use aps:

```
$ source  /opt/intel/vtune_amplifier/apsvars.sh
```

- Produce text and html reports with the following command:

```
$ aps -report=./aps_res
```

Analysis of the APS Report

Use either HTML or text output from APS to answer the following questions:

- What is the main performance bottleneck in this code?
- What percentage of floating point operations a packed instructions (vectorized instructions)?
- What would be your next step in order to investigate the performance issue?

Note: While APS offers possible explanations for the low CPU utilization and high proportion of back-end stalls, it does not pinpoint the cause. Instead, it points us at VTune™ Amplifier to look into them.

Intel® Vtune™ Amplifier

Looking Into System Level Performance

In this exercise you will analyze the performance of a new nbody implementation (version 7), which is an improvement over the previous versions we have considered.

Version 7 uses OpenMP* threads to parallelize the code and get rid of the extremely low CPU utilization reported by APS.

To get started move into the **ver7** directory:

```
$ cd ../ver7
```

And compile this version of the code to get a new **nbody.x** executable:

```
$ make
```

Feel free to open the source code and explore the changes, although that is not strictly necessary for this exercise.

Analysis of OpenMP* Version (ver7)

For the next step you will submit a job that analyzes the execution of nbody.x using 64 threads and a much larger workload than the default.

This is to avoid a trivial bottleneck due to synchronization issues if the workload is too small.

You can inspect the submission script **vtune.run** for details. To run simply execute the usual command:

```
$ qsub ./vtune.run
```

This will perform and **hpc-performance** analysis, which contains data regarding CPU microarchitecture, memory subsystem, and OpenMP* synchronization.

Results will be saved to the **vtune_hpc** directory. Make sure your job is no longer in the queue before you try to open the results!

[Optional] Archive the data and copy it over to your own system (requires a working installation of VTune™ Amplifier):

```
$ amplxe-cl -r ./vtune_hpc -archive
$ cp ./*.cpp ./*.hpp ./vtune_hpc
$ zip -r ./vtune_hpc.zip ./vtune_hpc
```

Analysis of OpenMP* Version (ver7)

Setup the environment for VTune™ Amplifier on the login node:

```
$ source /opt/intel/vtune_amplifier/amplxe-vars.sh
```

Open the result using the GUI :

```
$ amplxe-gui ./vtune_hpc
```

Look first at the summary, then click on the top time consuming function. This will take you to the bottom-up view, where you can review its characteristics.

Can you tell what the main bottleneck for this part of the code is from the provided statistics (you may have to scroll to the right to see all the columns)?

Proceed to the next slide for some answers.

Analysis of OpenMP* Version (ver7)

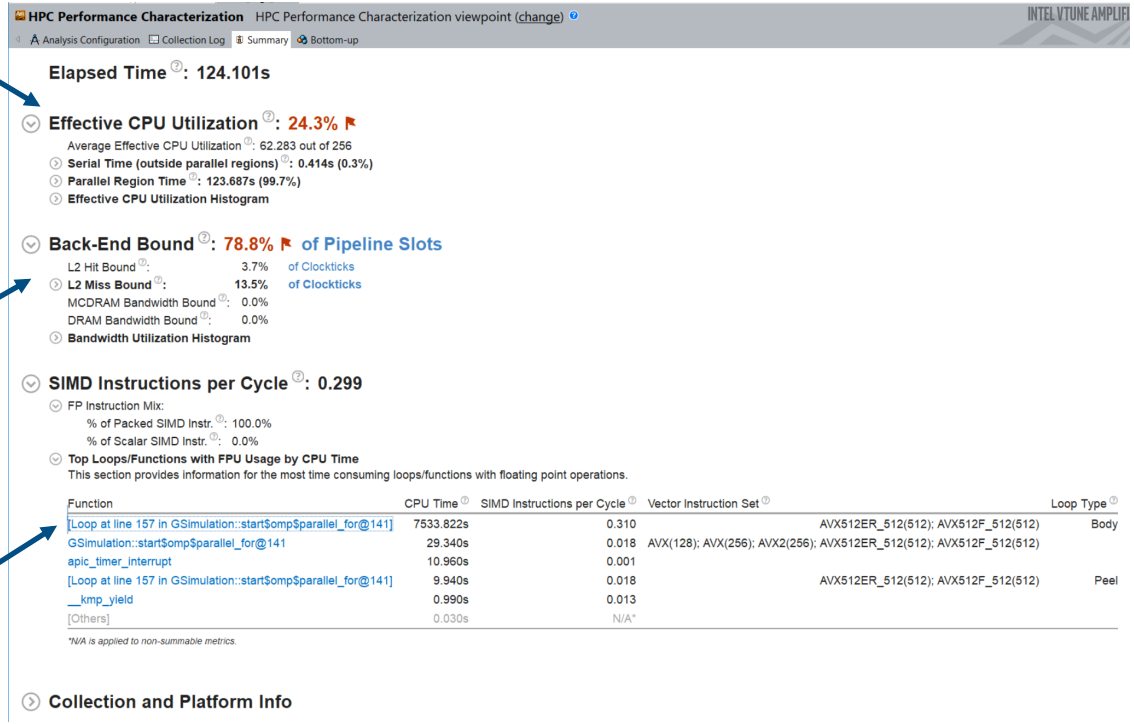
CPU utilization is 24%, but this is OK

- We used only 64 out or 256 logical CPUs
- The effective average usage is actually 62 / 64 (or 97%)

Code is heavily back-End bound

- 80% of pipeline slots are stalled
- No significant DRAM/MCDRAM usage.

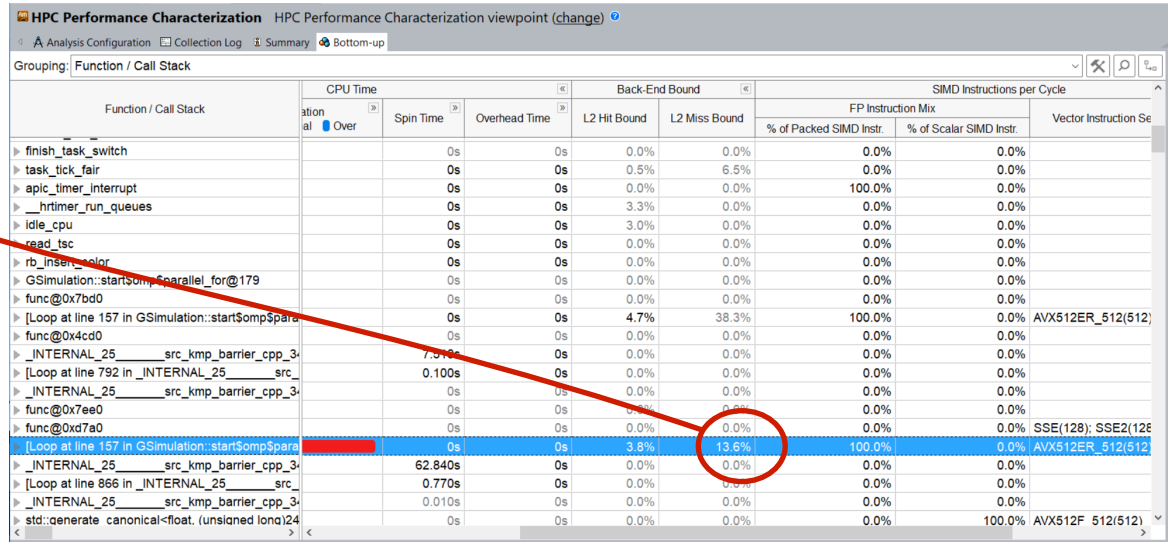
Click on the top time-consuming function to se the bottom-up view presented on the next slide.



Analysis of OpenMP* Version (ver7)

Code is running reasonably well, but there is a significant number of L2 cache misses that are probably causing the pipeline stalls.

Blocking for L2 would improve reuse and reduce this misses. This is implemented on version 8 of the code (next).



Analysis of OpenMP* Version (ver8)

You can verify what improvement can be achieved by blocking this code using version 8. To get started move into the **ver8** directory:

```
$ cd ../ver8
```

You can inspect the submission script **vtune.run** for details, it is exactly the same you just used for version 7. To run simply execute the usual command:

```
$ qsub ./vtune.run
```

Results will be saved to the **vtune_hpc** directory. Make sure your job is no longer in the queue before you try to open the results!

[Optional] Archive the data and copy it over to your own system (requires a working installation of VTune™ Amplifier):

```
$ amplxe-cl -r ./vtune_hpc -archive
$ cp ./*.cpp ./*.hpp ./vtune_hpc
$ zip -r ./vtune_hpc.zip ./vtune_hpc
```

Analysis of OpenMP* Version (ver8)

Open the result using the GUI :

```
$ amplxe-gui ./vtune_hpc
```

Look first at the summary, then click on the top time consuming function. This will take you to the bottom-up view, where you can review its characteristics.

Answer the following questions:

- What is the speedup of the code compared to the previous version?
- Has the fraction of pipeline stalls gone down as expected?
- Has the fraction of L2 cache misses gone down as expected?

Proceed to the next slide for some answers.

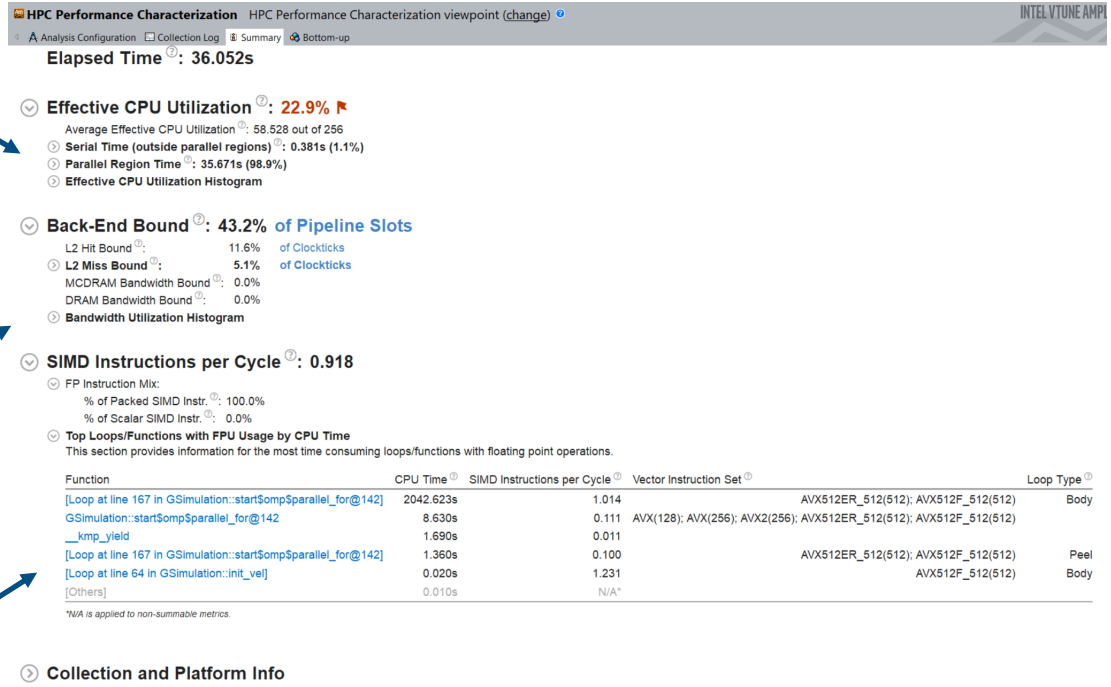
Analysis of OpenMP* Version (ver8)

The new codes should speedup by a factor 3x-4x

Code is now much less back-end bound

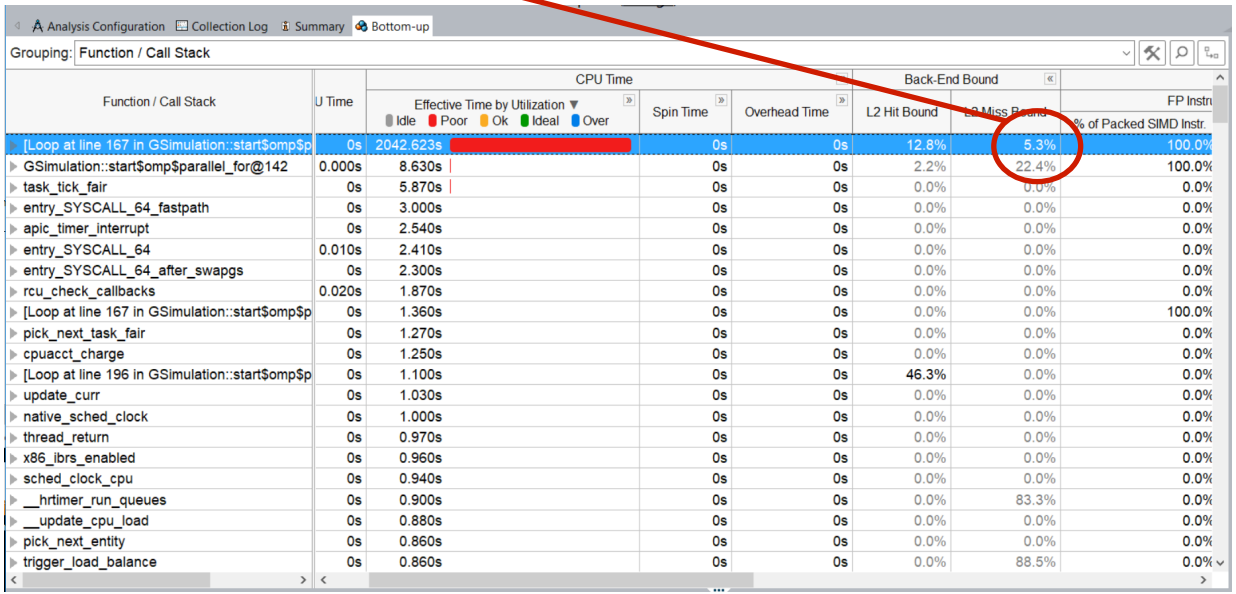
- 40% of pipeline slots are stalled
- Still no significant DRAM/ MCDRAM usage.

Click on the top time-consuming function to se the bottom-up view presented on the next slide.



Analysis of OpenMP* Version (ver8)

L2 Miss Bound column has gone down by a factor 2-3X, while Hit bound has increased by a similar factor.



Function / Call Stack	U Time	CPU Time: Effective Time by Utilization (Idle Poor Ok Ideal Over)	CPU Time: Spin Time	CPU Time: Overhead Time	Back-End Bound: L2 Hit Bound	Back-End Bound: L2 Miss Bound	FP Instr: % of Packed SIMD Instr.
[Loop at line 167 in GSimulation::startompp	0s	2042.623s	0s	0s	12.8%	5.3%	100.0%
GSimulation::startompparallel_for@142	0.000s	8.630s	0s	0s	2.2%	22.4%	100.0%
task_tick_fair	0s	5.870s	0s	0s	0.0%	0.0%	0.0%
entry_SYSCALL_64_fastpath	0s	3.000s	0s	0s	0.0%	0.0%	0.0%
apic_timer_interrupt	0s	2.540s	0s	0s	0.0%	0.0%	0.0%
entry_SYSCALL_64	0.010s	2.410s	0s	0s	0.0%	0.0%	0.0%
entry_SYSCALL_64_after_swapgs	0s	2.300s	0s	0s	0.0%	0.0%	0.0%
rcu_check_callbacks	0.020s	1.870s	0s	0s	0.0%	0.0%	0.0%
[Loop at line 167 in GSimulation::startompp	0s	1.360s	0s	0s	0.0%	0.0%	100.0%
pick_next_task_fair	0s	1.270s	0s	0s	0.0%	0.0%	0.0%
cpuacct_charge	0s	1.250s	0s	0s	0.0%	0.0%	0.0%
[Loop at line 196 in GSimulation::startompp	0s	1.100s	0s	0s	46.3%	0.0%	0.0%
update_curr	0s	1.030s	0s	0s	0.0%	0.0%	0.0%
native_sched_clock	0s	1.000s	0s	0s	0.0%	0.0%	0.0%
thread_return	0s	0.970s	0s	0s	0.0%	0.0%	0.0%
x86_ibrs_enabled	0s	0.960s	0s	0s	0.0%	0.0%	0.0%
sched_clock_cpu	0s	0.940s	0s	0s	0.0%	0.0%	0.0%
__hrtimer_run_queues	0s	0.900s	0s	0s	0.0%	83.3%	0.0%
__update_cpu_load	0s	0.880s	0s	0s	0.0%	0.0%	0.0%
pick_next_entity	0s	0.860s	0s	0s	0.0%	0.0%	0.0%
trigger_load_balance	0s	0.860s	0s	0s	0.0%	88.5%	0.0%

Legal Disclaimer & Optimization Notice

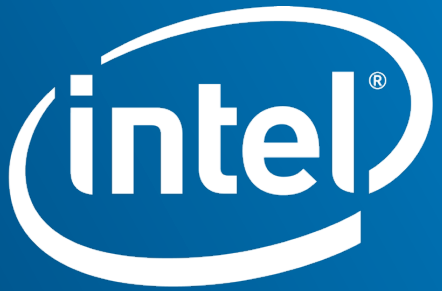
The benchmark results reported above may need to be revised as additional testing is conducted. The results depend on the specific platform configurations and workloads utilized in the testing, and may not be applicable to any particular user's components, computer system or workloads. The results are not necessarily representative of other benchmarks and other benchmark results may show greater or lesser impact from mitigations.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804



Software