INTEL CORPORATION

Accelerating Financial Applications on Intel® Architecture

Nimisha S Raut, Robert Geva, George Raskulinec

4/27/2015

Table of Contents

Introduction

A paper titled *Accelerating Financial Applications on the GPU*, http://dl.acm.org/citation.cfm?id=2458536 March, 2013 (referred to here as the "reference paper", and the original code referred to as the "reference code"), performed a GPU/CPU comparison using four QuantLib*-based workloads:

- Black-Scholes
- Monte-Carlo
- Bonds
- Repo

In order to run on the GPU, high level C++ code abstraction was removed, i.e., flattened, and translated into a set of lower-level structures and functions. The reference paper states "*In order to fairly compare the GPU implementations to a sequential CPU run, the same implementation without the abstraction in QuantLib is written for the CPU.*"

The reference paper compared parallel code, written in CUDA*, to both single and multi-threaded C code. These account for at least 4 differences:

- Hardware (HW)
- Programming models
- Parallel code vs. serial code
- Different programming languages (and of course different compilers).

A less cautious reading may attribute the performance differences only to the hardware, concluding that GPUs run the same algorithm faster than on Intel® Architecture (IA). In this rebuttal paper, we show that the cause for the performance difference is the parallelization approach, not the change in hardware. By effectively utilizing all of the parallel resources on IA; we demonstrate that the performance of the parallel code running on the CPU matches or exceeds that of the parallel code running on the GPU.

Depending on the workload and test configuration, the reference paper reports Nvidia Tesla* K20 GPU speed-up of over 80X-1000X compared to sequential CPU code and over 6X-35X compared to multi-threaded CPU code running on a dual socket Intel® Xeon® processor E5530-based server (Table 1). Actual performance numbers were not provided in the reference paper.

Table 1: Accelerating Financial Applications on the GPU workload comparison between Tesla* K20 and Intel® Architecture. (Source: Reference paper)

Workload	**Tesla* K20 Speed-up over single-threaded Intel® Architecture Code**	**Tesla K20 Speed-up over multi-threaded Intel Architecture Code**	**Comments**
Black Scholes*	>300x	>35x	When running more than 10,000 options
Monte Carlo*	1000x	>140x	When running 50,000 samples. Multi-threaded speed-up is based on graph data.

Bonds	>80x	>5x	When running more than 100,000 bonds. Multi-threaded speed-up is based on graph data.
Repo	>80x	>5x	When running more than 50,000 repos. Multi-threaded speed-up is based on graph data.

To make a more meaningful comparison, this paper incorporates a comprehensive approach by:

- Utilizing all of the parallel IA resources
- Optimizing the data layout
- Applying libraries optimized for lA
- Re-running the tests using the latest Intel and Nvidia HW as of September 2014

Definitions used in this paper:

- Reference Paper - *Accelerating Financial Applications on the GPU*, March, 2013
- Reference Source Code – Original code mentioned/used in *Accelerating Financial Applications on the GPU*, March, 2013.
- Optimized Code – Reference code that was optimized on Intel Architecture.
- Modified Code – Changes made to the reference code to extend test times, e.g. adding additional options, loop iterations, and code restructuring.

QuantLib* Overview

Quantlib, http://quantlib.org/index.shtml, is a free full-featured open source financial library written in C++. It consists of a number of modules with associated classes that enable users to quickly analyze financial Instruments (bonds, options, swaps, stock, etc.) by applying popular stochastic processes (Monte Carlo, Black-Scholes, Hull-White, Libor, etc.) and term structures (zero curve, ForwardRate, yield term, etc.). Many financial organizations have a similar library that provides an equivalent set of concepts and APIs that are used in their application code.

Parallel Computing on Intel® Architecture

The HW architecture support for parallel computing on Intel Architecture, as well as other CPUs, is inherently different to that on the GPU and therefore requires different algorithm design and different expression of the parallel constructs. The principles of parallel HW support are:

a. Multiple cores, supporting multiple execution threads
b. Vector units
c. Memory hierarchy, especially the cache architecture

An efficient parallel algorithm design takes these principles into account as the program is mapped onto the parallelism in the HW. In addition they must:

- Be cache efficient
- Utilize the vector units
- Lay out the data efficiently for utilization of translation lookaside buffers(TLBs) , caches, and enable efficient vector access

In general, these principles can be expressed independently of each other. In particular, it is worthwhile to try to express thread-level parallelism and vector-level parallelism at different levels. The key design principle in thread-level parallelism is to maximize the amount of parallel work in order to amortize the run-time overhead of the scheduler. This calls for parallelization at the outmost level possible. The key design principle in vector programming is to minimize the amount of control flow divergence and memory access divergence, so that the vector units do the same work as each other as much as possible. This calls for vectorizing at the innermost level possible. Because the design goals are conflicting, it is useful to have syntax that allows the programmer the flexibility to express thread-level parallelism and vector-level parallelism separately or combined.

There are multiple, well-known paradigms to express thread-level parallelism. For instance, programmers can use native treads. The reference paper used OpenMP* [6]. Other alternatives provide different tradeoffs. For example, Intel® Threading Building Blocks (Intel® TBB) [7] is designed to provide better support for nested parallelism. For the purpose of the comparisons presented in this paper, the tradeoffs don't manifest; therefore, we continue to use OpenMP. Explicit vector programming is relatively new and is a part of the OpenMP 4.0 standard.

Vectorization / SIMD

Vectorization is the process of transforming a sequence of scalar operations acting on single data elements at a time (Single Instruction Single Data – SISD), to an operation acting on multiple data elements at once (Single Instruction Multiple Data – SIMD). Every Intel® processor core has a dedicated vector unit. Over the years Intel has continued to invest in both the size and capabilities of the vector unit [8]. An example of the potential speed-up is shown in Figure 1.

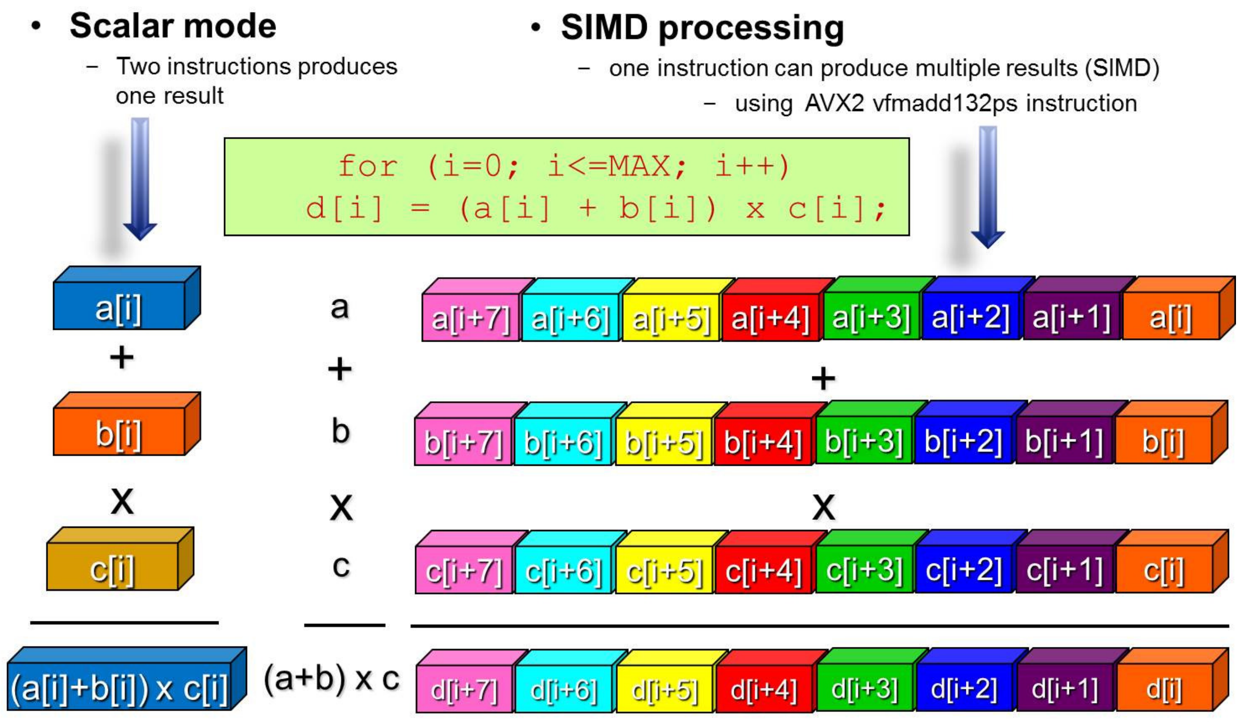

Scalar mode
Two instructions produces one result
SIMD processing
one instruction can produce multiple results (SIMD)
using AVX2 vfmadd132ps instruction
for (i=0; i<=MAX; i++)
d[i] = (a[i] + b[i]) x c[i];
a[i]
+
b[i]
x
c[i]
(a[i]+b[i]) x c[i]
a
+
b
x
c
(a+b) x c
a[i+7] a[i+6] a[i+5] a[i+4] a[i+3] a[i+2] a[i+1] a[i]
b[i+7] b[i+6] b[i+5] b[i+4] b[i+3] b[i+2] b[i+1] b[i]
c[i+7] c[i+6] c[i+5] c[i+4] c[i+3] c[i+2] c[i+1] c[i]
d[i+7] d[i+6] d[i+5] d[i+4] d[i+3] d[i+2] d[i+1] d[i]

Figure 1: Intel® Advanced Vector Extensions 2 vectorization example.

For Intel processors that support Intel® Advanced Vector Extensions 2 (Intel®AVX2), each core has one 256-bit vector unit and supports floating point fused multiply-add (FMA) instructions. Thus, each core can perform an addition and a multiplication on 8 single precision or 4 double precision floating point operations in 1 instruction. Note that for the Intel® Xeon Phi™ coprocessor, the vector register is doubled to 512 bits and can perform 16 single and 8 double precision operations per instruction.

Vector Loops in OpenMP* 4.0

The syntax `#pragma omp simd` can be applied to a `for loop` to indicate that the loop is intended to execute in vectors. The `#pragma omp simd` syntax admits the following optional clauses:

- **Vectorlength:** Determine the number of iterations that the compiler has to chunk together and vectorize. Can be used to ensure that data dependencies are handled correctly or to optimize for performance.
- **Private, firstprivate, lastprivate:** Same as the OpenMP definition and use of these clauses.
- **Linear:** Indicate that a variable is a linear induction variable. The variable has to be declared outside the loop and be incremented by a `loop-` invariant amount in each loop iteration. The compiler privatizes the variable and provides a distinct value for each iteration.
- **Reduction:** The reduction clause is similar to the one previously available in OpenMP for parallel loops. It means that a pair, (operation, variable), are used as a reduction. The only operation allowed on that variable is the one specified in the reduction clause. Unlike the

linear variable, the reduction does not have to execute in every loop iteration and the increment amount does not have to be loop invariant. On the other hand, the value of the reduction is not available during the loop, it is only available after the loop.

SIMD-Enabled Functions

SIMD-enabled functions add modularity to vector loops. When called from a vector loop, multiple consecutive instances of the function execute in chunks, as if they were compiled as a part of the body of the vector loop. The ability to write vector code outside of the scope of the vector loops allows modular programming and independent deployment, such as in libraries that are deployed in binary format, rather than in header files. The syntax `#pragma` declare `simd` designates a function as a SIMD-enabled function. The attribute admits optional clauses:

- **Vectorlength:** Determine the vector length, which is the number of consecutive invocations of the function that the compiler chunks together and vectorizes across.
- **Uniform:** Indicate that a formal parameter is uniform, meaning that it has the same value within each vectorlength consecutive invocations of the function. Uniform arguments are passed in scalar registers, whereas varying arguments (the default) are passed in vector registers.
- **Linear:** Indicate that a variable has a linear increment from one invocation to the next, and optionally indicate what that increment is. The default increment is one.

Multiple `#pragma` declare vector lines are possible for a single SIMD-enabled function. For each vector attribute, the compiler generates one vector variant of the function. The vector attribute and its associated clauses are considered part of the function signature. If the function declaration is inconsistent with the function prototype with respect to these clauses, then the behavior is undefined.

Test Methodology

All testing was performed on the latest available hardware as of September, 2014, shown in Table 2. The software configuration is shown in Table 3.

Table 2: Hardware test configuration.

Platform	Details
GPU: Tesla* K40c	2880 cores, 0.745 GHz, 12 GB GDDR5, Base Clock: 745 MHz
CPU: Intel® Xeon® processor E5-2697 v3	2.6 GHz, 2 Socket, 14 core/Socket, 30MB L3 Cache, 64 GB RAM. Intel® Turbo Boost Technology and Intel® Hyper-Threading Technology were enabled. This Intel Xeon processor supports 256-bit vector length with Intel® AVX2 instructions.
Coprocessor: Intel® Xeon Phi™ coprocessor 7120P	61 cores, 1.238 GHz, 16GB ECC, Intel Turbo Boost Technology was disabled

	Supports 512-bit vector length. Instruction set includes Fuse Multiply Add support.

Table 3: Software test configuration.

Software	Details
Host OS	Red Hat* Enterprise Linux* version 6.5, kernel 2.6.32-431
CUDA*	5.5
Intel® C++ Compiler	CPU binaries built with 15.0.0 Phi binaries built with 14.0.2 20140120 (2013 sp1.2.144)
Coprocessor: Intel® Xeon Phi™ coprocessor 7120P	Driver 6720-16, Flash 2.1.03.0383

Note: Intel® Xeon Phi™ coprocessor 7120P testing completed prior to Intel® C++ Compiler 15.0.0 availability.

The reference source code was downloaded from http://sourceforge.net/projects/quantlib-gpu/. This paper compares the performance between the handwritten CUDA code and the multi-threaded CPU code optimized for Intel hardware by applying the stepwise optimization framework. All tests were run multiple times and the results were averaged.

Additional test details:

- The input data generation and data transfer between device and host were not included in the test time measurements.
- All Intel Xeon Phi coprocessor tests were run natively.
- All tests should complete in the seconds range for accurate/repeatable results and to amortize the PCI Express* transfer times of the coprocessor and accelerator cards, which were not included in these results. All workloads that complete in less than 1 second were extended by adding additional options, test iterations, or code restructuring. These details are discussed below in the appropriate sections.

Stepwise Parallelization Framework

The following parallelization methodology for Intel Architecture was applied to all workloads [5].

1. **Leverage optimized tools and libraries:** Profile the workload using Intel® VTune™ Amplifier to identify hotspots. Use Intel C++ compiler to generate optimal code and apply optimized libraries such as Intel® Math Kernel Library (Intel® MKL), Intel TBB, and OpenMP when appropriate.
2. **Scalar, serial optimization:** Maintain the proper precision, type constants, and use appropriate functions and precision flags.
3. **Vectorization[8]:** Utilize SIMD features in conjunction with data layout optimizations Apply cache-aligned data structures, convert from *arrays of structures* to *structure of arrays*, and minimize conditional logic.
4. **Thread Parallelization:** Profile thread scaling and affinitize threads to cores. Scaling issues typically are a result of thread synchronization or inefficient memory utilization.

5. **Scale from Intel Xeon processor to Intel Xeon Phi coprocessor:** Apply optimized Extended Math Unit(EMU) functions and additional memory optimizations including data blocking.

Black-Scholes*

Black-Scholes is a closed-form financial derivative used for European option pricing valuation. The Black-Scholes model was developed in 1973 by Fisher Black, Robert Merton, and Myron Scholes and is still widely used today in quantitative finance. Robert Merton was the first one to publish a closed-form solution to the Black-Scholes Equation and for European options, also known as the Black-Scholes-Merton Formula. As mentioned in the reference paper, this application is from the European Option test in the QuantLib test suite. European options can be exercised only on the expiration date. This version is a single-precision 32-bit floating point implementation.

Intel® Architecture Optimizations

The reference OpenMP version was threaded, but no additional optimizations were made. Thus, many of the performance resources on the Intel Architecture were not utilized. By applying the stepwise parallelization methodology the following key optimizations were applied.

1. Vectorization: As mentioned earlier, the vector unit can provide significant performance speed-up by operating on multiple vector operands in a single instruction. The critical loop in the reference code would not vectorize primarily due to the use of a switch statement and data dependencies. The switch statement was converted to an if/else and #pragma omp simd was applied to the critical loop.
2. Data layout: Convert array of structures to structures of arrays. Provides excellent cache locality, alignment, and unit stride access for the critical vector loop.

   Utilize Intel TBB memory allocator for memory allocation. The benefits are two–fold: memory is allocated on cache-aligned boundaries and each thread has its own memory pool, which removes reliance on the global heap and improved NUMA locality.

Note: From a performance perspective, steps 1 and 2 are tightly coupled.

3. Precision consistency: Use precision-appropriate transcendental functions and constants. For example, expf() is used for single precision and exp is used for double precision scenario. Also, constants are defined with suffix “f” (example; 0.1f) for single precision workloads.
4. EMU functions[4]: Used the single precision math function, exp2f(), implemented in the Intel Xeon Phi coprocessor hardware. The exp2f() function requires a conversion to base 2 by multiplying by a constant M_LOG2E.
5. Makefile precision flags. Applied the following flags: *-fno-alias, -fimf-precision=low, -fimf-domain-exclusion=31, -fimf-accuracy-bits=11, -no-prec-div and -no-prec-sqrt*. Validation code was added to compare reference code results with precision results within a small error tolerance, 9^-5. Note that all performance data was collected with and without the precision flags and the results are included in the graphs.

Table 4 compares single-threaded reference performance and the optimized results on Intel Architecture. As you can see, by using the Intel compiler and applying the optimizations listed above, the single-threaded code performance improved a minimum of 19.68X[1] compared to the un-optimized reference code.

Table 4[1]: Single-threaded reference performance results vs. optimized results.

Options, Iterations	**Reference Intel® Xeon® processor E5-2697 v3 Intel® Architecture code (1 thread) (Sec)**	**Optimized Intel Xeon processor E5-2697 v3 Intel Architecture code without precision flags (1 thread) (Sec)**	**Speed-Up[1]**
5M, 1 Iteration	1.063	0.054	19.68X
10M, 1 Iteration	2.137	0.104	20.55X
50M, 1 Iteration	21.28	0.938	22.69X

All time in seconds. (Source: Intel Measured)

Reference Test Configuration Comparison

The reference paper test code computed 5 million options, and the authors state that the speed-up for all K20 implementations is greater than 35X vs. the OpenMP implementation when running more than 10,000 options. Table 5 presents the kernel run-time values of the reference CUDA code and optimized Intel Architecture version for 5 million options. The original CUDA code takes 3.7 mSec. Optimal performance was achieved on Intel Xeon processor E5-2697 v3 when running 20 threads. Because Intel® Turbo Boost technology [1] is enabled, the active core frequency automatically increased from 2.6 GHz to 2.9 GHz providing optimal performance. The Intel Xeon Phi coprocessor takes 6.9 mSec. SW stack initialization times, for example, for OpenMP initialization, take longer on Intel Xeon Phi coprocessors which, to a large extent, explain its performance numbers.

Table 5[1]: Reference CUDA* results vs. optimized Intel® Architecture results for 5 million options, 1 iteration.

Options	**Reference Tesla* K40 (mSec)**	**Optimized Intel® Xeon® processor E5-2697 v3 (20 threads) with precision flags (mSec)**	**Optimized Intel Xeon processor E5-2697 v3 (20 threads) without precision flags (mSec)**	**Optimized Intel® Xeon Phi™ coprocessor 7120P (244 threads) with precision flags (mSec)**	**Optimized Intel Xeon Phi coprocessor 7120P (244 threads) without precision flags (mSec)**
5M	3.7	4.9	5.3	6.9	21.83

(Source: Intel Measured)

Test Modifications

The runtimes presented in Table 5 include only the computation time but do not include initialization time and transfer times (host to device and device to host). The runtimes indicate that the performance on the host, which has no transfer time penalty, is equivalent to the GPU performance.

Running tests that complete in 5 mSec could easily be performed on the host CPU, eliminating additional programming techniques, proprietary libraries, and additional HW resources, i.e., a co-processor or accelerator.

To amortize the transfer time overhead and to focus exclusively on algorithmic-scaling comparisons, the test times were extended beyond 1 second by increasing the number of options from 5 million to 15 million. Adding additional options was limited by the memory configurations on the Tesla K40 and Intel Xeon Phi coprocessor. Additionally, an iteration loop was added to call the same test multiple times with the same input data. Identical changes were implemented in the CUDA version.

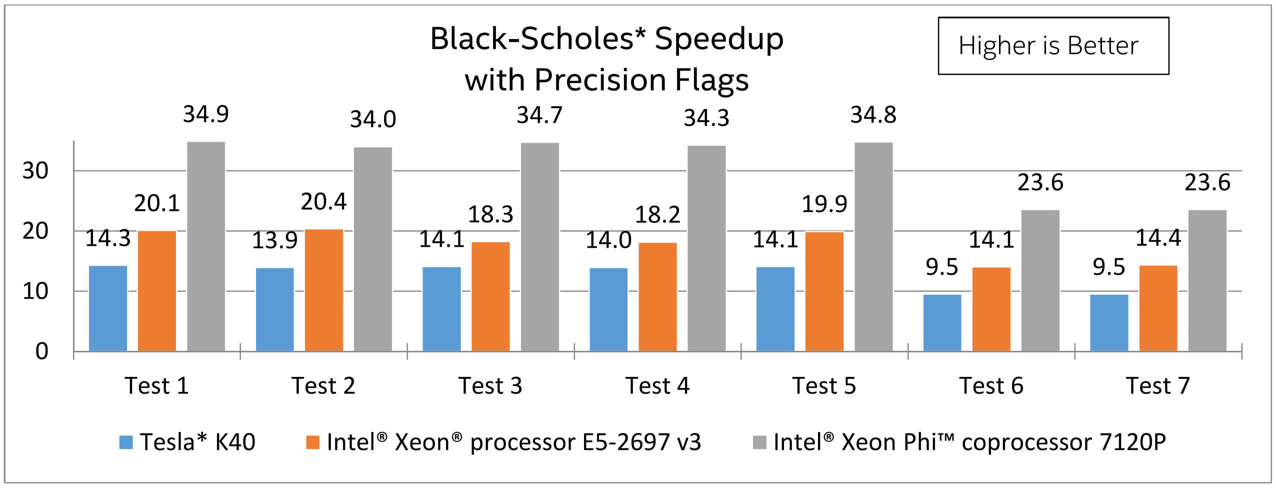
Modified Test Configuration Comparison

Table 6 shows the results for the CUDA version and the optimized Intel Architecture version for multiple options and multiple iterations. Different combinations of options and iterations were executed to study the performance behavior of both CUDA and Intel Architecture versions and to ensure test input data did not favor one platform over the other.

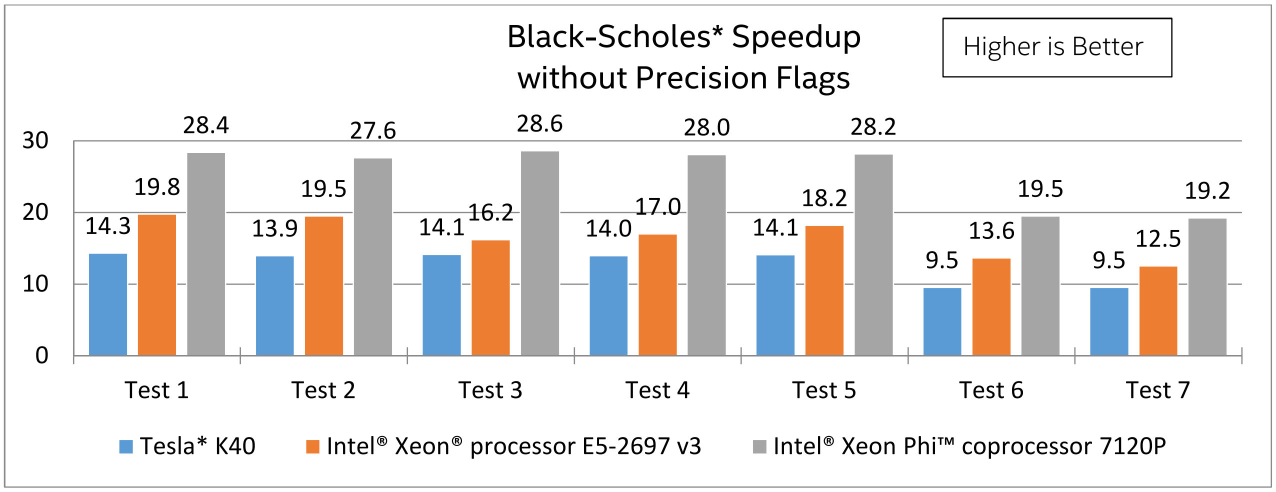
Table 6[1]: CUDA* results vs. optimized Intel® Architecture results for multiple options and iterations.

Test Number	Options	Iterations	Modified Tesla* K40 (Sec)	Optimized Intel® Xeon® processor E5-2697 v3 1T without precision flags (Sec)	Optimized Intel Xeon processor E5-2697 v3 56T with precision flags (Sec)	Optimized Intel Xeon processor E5-2697 v3 56T without Precision flags (Sec)	Optimized Intel® Xeon Phi™ coprocessor 7120P 244T with precision flags (Sec)	Optimized Intel Xeon Phi coprocessor 7120P 244T without precision flags (Sec)
1	5 M	1024	3.22	46.064	2.29	2.33	1.32	1.623
2	5 M	2048	6.42	89.530	4.39	4.591	2.63	3.240
3	10 M	1024	6.40	90.431	4.95	5.583	2.603	3.161
4	10 M	2048	12.80	178.821	9.84	10.521	5.22	6.378
5	15 M	512	4.79	67.567	3.39	3.716	1.94	2.399
6	15 M	1024	9.60	91.500	6.51	6.708	3.88	4.694
7	15 M	2048	19.18	182.867	12.72	14.577	7.75	9.509

All times are in seconds. (Source: Intel Measured)



Graph 1[1]: Speed-up compared to single-threaded performance on the Intel® Xeon® processor E5-2697 v3. (Source: Intel Measured)



Graph 2[1]: Speed-up compared to single-threaded performance on the Intel® Xeon® processor E5-2697 v3. (Source: Intel Measured)

Graphs 1 and 2 show the speed-up on the Tesla K40, the Intel Xeon processor E5-2697 v3, and the Intel Xeon Phi coprocessor 7120P compared to the single-thread performance on the Intel Xeon processor E5-2697 v3. The results of the Intel Xeon processor E5-2697 v3 and the Intel Xeon Phi coprocessor 7120P as compared to the Nvidia Tesla K40 remain consistent regardless of the test option/iteration count. With precision flags, the Intel Xeon processor E5-2697 v3 outperforms K40 by a minimum of 1.29X[1] and the Intel Xeon Phi coprocessor 7120P outperforms K40 by a minimum of 2.44X[1]. Without precision flags, the Intel Xeon processor E5-2697 v3 outperforms K40 by a minimum of 1.22X[1] and the Intel Xeon Phi coprocessor 7120P outperforms K40 by a minimum of 1.98X[1].

Monte-Carlo*

Monte Carlo is a popular simulation mathematical model used to evaluate complex instruments, portfolios, and investments by applying statistical computing to model the uncertainty of underlying stock variable changes. The reference code uses single-precision, floating-point results to price a single option with 250 time steps.

Intel® Architecture Optimizations

Similar to Black-Scholes, the reference version of Monte-Carlo was threaded without any further Intel Architecture optimizations underutilizing the available Intel Architecture parallel resources. The additional key optimizations were made:

1. Vectorization: The inner loop used the path value of the previous iteration to compute the current path value in the current iteration. This is referred to as a "read-after-write" dependency and may lead to incorrect results. This loop type cannot be vectorized. Upon further inspection, it became clear that this dependency could be removed by introducing a scalar to store intermediate results and then assigning the final value to the path variable. After making this change and applying `#pragma omp simd`, the loop successfully vectorized.
2. Intel MKL: Used this library to generate a vector of random numbers in a single API call [2].
3. Data blocking: Random numbers are generated once and shared across all options. Due to the size of random numbers, they cannot all be processed simultaneously and fit in the data cache. Instead, a data blocking technique was implemented to process blocks of random numbers, filling the cache one block at a time, across all options.
4. Precision consistency: Applied precision-appropriate transcendental functions and defined constants as single precision.
5. EMU functions: The Intel Xeon Phi coprocessor also has a set of single-precision math functions implemented in the hardware. The `exp2f()` function was used where appropriate.
6. Data layout: Use cache-aligned arrays.
7. Makefile precision flags. Applied the following flags: *-fno-alias, -fimf-precision=low, -fimf-domain-exclusion=31, -fimf-accuracy-bits=11, -no-prec-div and -no-prec-sqrt*. Note that all performance data was collected with and without the precision flags and included in Graphs 3 and 4.

Table 7 compares single-threaded reference performance and the optimized results on Intel Architecture. By using the Intel compiler and applying the optimizations listed above, the single-threaded performance improved a minimum of 13.4X[1] compared to the reference code.

Table 7[1]: Reference Intel® Architecture results vs. optimized Intel Architecture results.

Paths, Options	Reference Intel® Architecture code; Intel® Xeon® processor E5-2697 v3 (1 thread) (Sec)	Optimized Intel Architecture code without precision flags; Intel Xeon processor E5-2697 v3 (1 thread) (Sec)	Speed-Up[1]

50k, 1 Option	2.077	0.155	13.4X
500k, 1 Option	20.704	1.022	15.05X

All times in seconds. (Source: Intel Measured)

Reference Test Configuration Comparison

The reference paper stated all Tesla K20-accelerated results using at least 2,000 samples show a speed-up of at least 18X over the multi-core CPU implementation. Table 8 presents test times using 50,000 samples and single option. The reference CUDA code takes 46.9 mSec. The Intel Xeon processor E5-2697 v3 outperforms the GPU by a minimum of 1.82X[1] when running 16 threads. Intel Turbo Boost Technology increases the CPU core frequency to 2.9 GHz.

Table 8[1]: Reference CUDA* results vs. optimized Intel® Architecture code for a single option.

Options	Samples	Reference Tesla* K40 (mSec)	Optimized Intel® Xeon® processor E5-2697 v3 (16 threads) with precision flags (mSec)	Optimized Intel Xeon processor E5-2697 v3 (16 threads) without precision flags (mSec)	Optimized Intel® Xeon Phi™ coprocessor (244 threads) with precision flags (mSec)	Optimized Intel Xeon Phi coprocessor (244 threads) without precision flags (mSec)
1	50,000	46.9	17.83	25.77	2300	2225

(Source: Intel Measured)

Test Modifications

Test times <100 mSec, excluding transfer times, can easily be computed on the host. Also, to gain a better understanding of the scaling properties of the Monte-Carlo algorithm, the reference code was modified to support multiple options. Code changes were required in both the Intel Architecture and GPU versions to run multiple options. Intel Architecture changes included adding an options loop over existing paths loop. The reference paper mentions: "*An extension of this implementation allows the parallel pricing of multiple options. This is particularly useful when there are not enough samples in each option to fully take advantage of the massive parallelism available on the GPU.*"

However, the reference CUDA code device kernel does not include any code for multiple options implementation. To implement a multiple options version using CUDA, the MonteCarloMultiGPU example provided in the CUDA SDK 5.5 was used. In this version, each thread block in CUDA computes one option for multiple paths using 256 threads. Shared memory in CUDA is on-chip memory and hence faster to access. Additionally, all threads in a thread block can synchronize using shared memory, which was used to store intermediate results and accumulated at the end to calculate the option price.

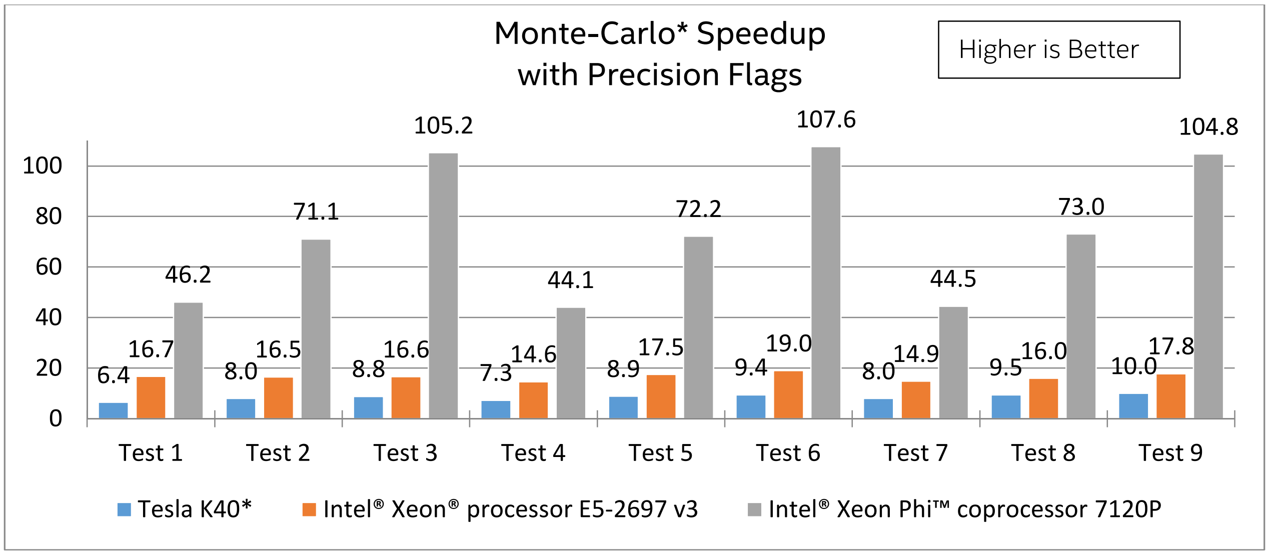
Modified Test Configuration Comparison

Table 9 presents the multiple option results using the modified CUDA and optimized Intel Architecture versions. The reference paper claimed that the best GPU speed-up is achieved at 50k samples. Thus, tests were run from 50k up to 262k samples while varying the option count. The test configuration used in CUDA's 5.5 SDK, 262k samples and 512 options, was also included. As Table 9 shows, Intel Xeon processor E5-2697 v3 outperforms K40 by a minimum of 1.4X[1] and the Intel Xeon Phi coprocessor 7120P outperforms K40 by 2.0X[1] minimum.

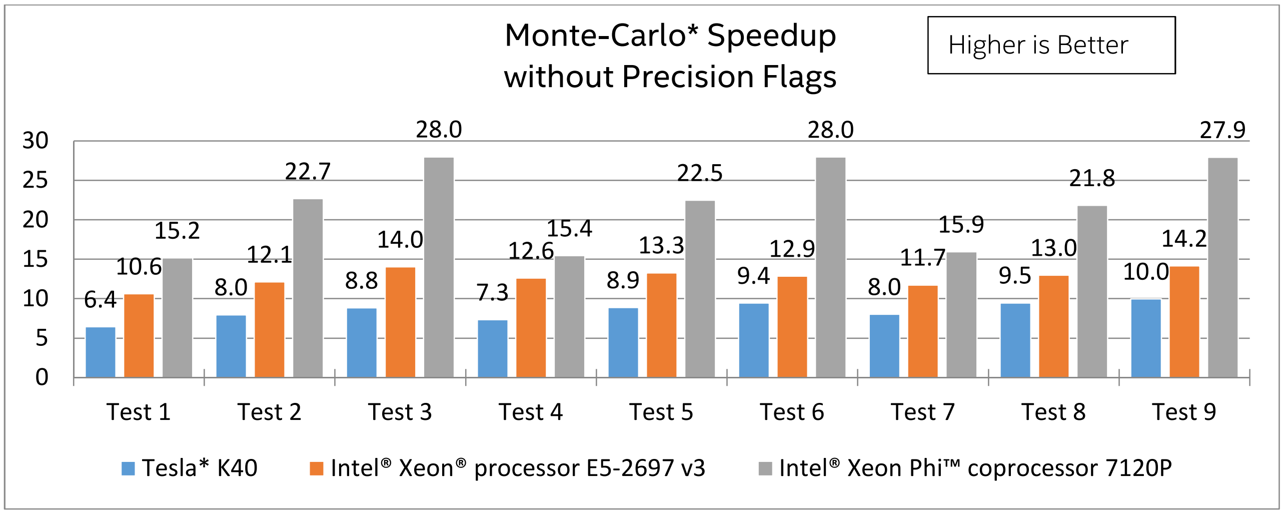
Table 9[1]: CUDA* results vs. optimized Intel® Architecture results for multiple options and paths.

Test Number	Samples	Options	Modified Tesla* K40 (Sec)	Optimized Intel® Xeon® processor E5-2697 v3 1T without precision flags (Sec)	Optimized Intel Xeon processor E5-2697 v3 56T with precision flags (Sec)	Optimized Intel Xeon processor E5-2697 v3 56T without precision flags (Sec)	Optimized Intel® Xeon Phi™ coprocessor 7120P 244T with precision flags (Sec)	Optimized Intel Xeon Phi coprocessor 7120P 244T without precision flags (Sec)
1	50 K	256	3.94	25.394	1.518	2.388	0.55	1.675
2	50 K	512	6.38	50.775	3.070	4.187	0.714	2.241
3	50 K	1024	11.49	101.521	6.125	7.238	0.965	3.630
4	100 K	256	6.94	50.772	3.470	4.025	1.152	3.288
5	100 K	512	11.44	101.546	5.815	7.649	1.406	4.517
6	100 K	1024	21.49	203.051	10.677	15.799	1.887	7.262
7	262144	256	16.604	133.029	8.955	11.335	2.987	8.348
8	262144	512	28.120	266.077	16.660	20.522	3.643	12.198
9	262144	1024	53.308	532.054	29.895	37.577	5.079	19.058

All time in seconds. (Source: Intel Measured)



Graph 3[1] Speed-up compared to the single threaded performance on the Intel Xeon processor E5-2697 v3. (Source: Intel Measured)



Graph 4[1] Speed-up compared to single threaded performance on the Intel Xeon processor E5-2697 v3. (Source: Intel Measured)

Graphs 3 and 4 show the speed-up of the Tesla K40, the Intel Xeon processor E5-2697 v3, and the Intel Xeon Phi coprocessor 7120P compared to the single thread performance on the Intel Xeon processor E5-2697 v3. The Intel Xeon processor E5-2697 v3 and the Intel Xeon Phi coprocessor 7120P consistently outperform the K40 regardless of the test option/iteration count. With precision flags, the Intel Xeon processor E5-2697 v3 outperforms K40 by a minimum of 1.69X[1] and the Intel Xeon Phi coprocessor outperforms K40 by a minimum of 5.56X[1] . Without precision flags, the Intel Xeon processor E5-2697 v3 outperforms K40 by a minimum of 1.37X[1] and the Intel Xeon Phi coprocessor 7120P outperforms K40 by a minimum of 1.99X[1].

Bonds

A bond is a form of loan between an issuer, such as a corporation or a government, and a holder or investor for a pre-determined period of time at a fixed interest rate. The bond issuer is obligated to pay the holder interest at specified intervals and/or pay back the principle at the maturity date. The bonds workload uses double-precision, floating-point results. The reference paper claims a significant speed-up over the OpenMP implementation. As the reference paper does not mention any test times, the graph data was interpreted to get the speed-up. The graph data shows that the K20 speed-up over the OpenMP implementation is greater than 5X for more than 500,000 bonds.

Intel® Architecture Optimizations

The bonds scenario is a complex workload presenting little vectorization benefit. Some of the vectorization challenges:

- Switch cases and if-else statements lead to vector inefficiencies.
- if-else statement execution does not follow a pattern that would provide hints to the compiler to generate optimal code.

- Critical loop counts are low (<10).
- Data access could not be aligned due to nested structures.

The following optimizations were successfully applied:

1. Data layout: The Intel TBB memory allocator was used for memory allocation.
2. Existing arrays were defined as static and cache aligned.

Reference Test Configuration Comparison

Table 10 presents test times using 1 million bonds. The reference CUDA code takes 826 mSec. The Intel Xeon processor E5-2697 v3 outperforms the GPU by up to 1.4X[1]. The Intel Xeon Phi coprocessor 7120P takes 1265 mSec.

Table 10[1]: Reference CUDA* results vs. optimized Intel® Architecture code for 1 million bonds.

Bonds	Reference Tesla* K40 (mSec)	Optimized Intel® Xeon® processor E5-2697 v3 (56 threads) with precision flags (mSec)	Optimized Intel Xeon processor E5-2697 v3 (56 threads) without precision flags (mSec)	Optimized Intel® Xeon Phi™ coprocessor 7120P (244 threads) with precision flags (mSec)	Optimized Intel Xeon Phi coprocessor 7120P (244 threads) without precision flags (mSec)
1 Million	826	585	609	1105	1265

(Source: Intel Measured)

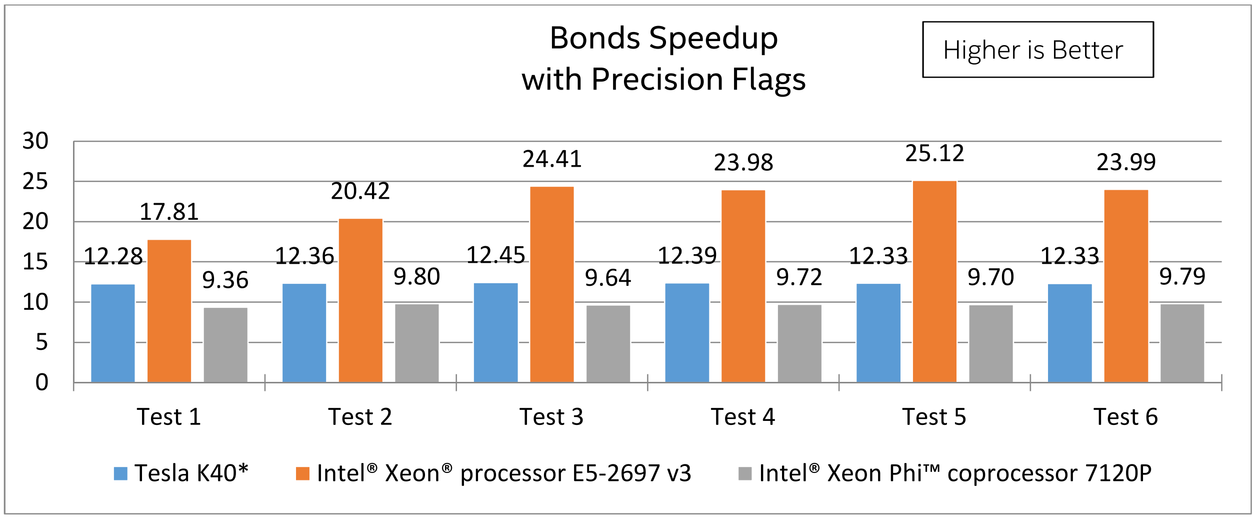
Modified Test Configuration Comparison

The reference code executes up to 1 million bonds, but the test times are below the one second range. To profile algorithmic scaling, test times were extended to the seconds range by increasing the number of bonds to 15 million. The only modification to the reference CUDA code was to increase the number of bonds. Table 11 shows the results.

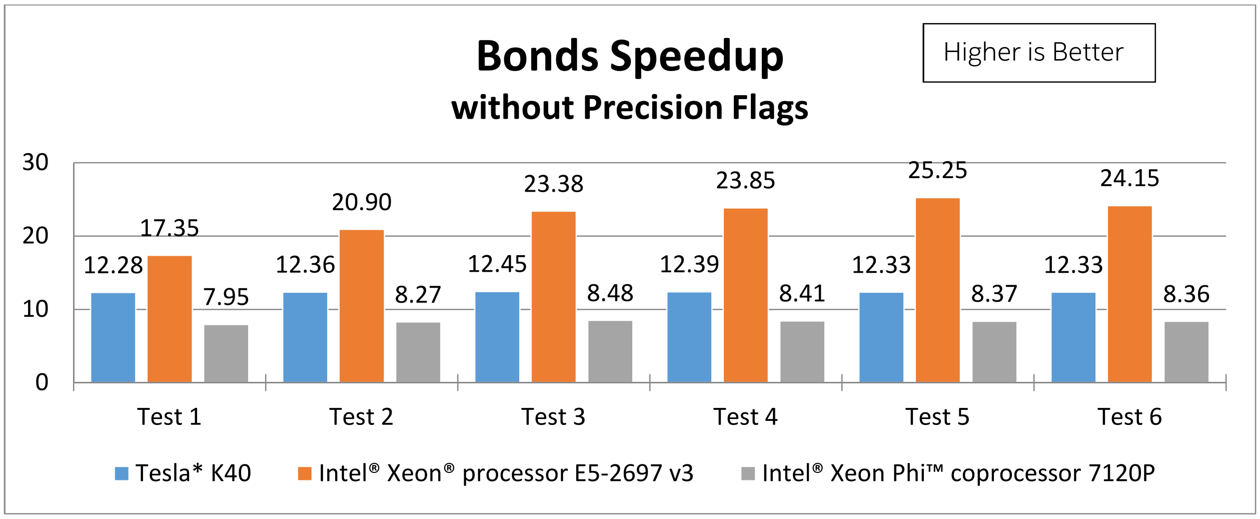
Table 11[1]: Modified CUDA* results vs. optimized Intel® Architecture code.

Test Number	Bonds	Modified Tesla* K40 (Sec)	Optimized Intel® Xeon® processor E5-2697 v3 1T without precision flags (Sec)	Optimized Intel Xeon processor E5-2697 v3 56T with precision flags (Sec)	Optimized Intel Xeon processor E5-2697 v3 56T without precision flags (Sec)	Optimized Intel® Xeon Phi™ coprocessor 244T with precision flags (Sec)	Optimized Intel Xeon Phi coprocessor 244T without precision flags (Sec)
1	2 M	1.646	20.218	1.135	1.165	2.161	2.544
2	5 M	4.102	50.694	2.483	2.426	5.172	6.129
3	8 M	6.561	81.666	3.346	3.493	8.474	9.625
4	10 M	8.187	101.423	4.230	4.252	10.430	12.059
5	12 M	9.833	121.265	4.827	4.802	12.500	14.484
6	15 M	12.294	151.577	6.318	6.277	15.477	18.133

All time in seconds. (Source: Intel Measured)



Graph 5[1]: Speed-up compared to single threaded performance on the Intel® Xeon® processor E5-2697 v3. (Source: Intel Measured)



Graph 6[1]: Speed-up compared to single threaded performance on the Intel® Xeon® processor E5-2697 v3. (Source: Intel Measured)

Graphs 5 and 6 show the speed-up of the Tesla K40, the Intel Xeon processor E5-2697 v3, and the Intel Xeon Phi coprocessor 7120P compared to the single thread performance on the Intel Xeon processor E5-2697 v3. As mentioned earlier, the Bonds scenario presents fewer opportunities to utilize the vector units. Despite this, the Intel Xeon processor E5-2697 v3 outperforms the K40 by as much as 1.8X[1]. Conversely, the K40 outperforms the Intel Xeon Phi coprocessor 7120P by up to 1.4X[1], primarily due to the lack of vectorization benefit on the Intel Xeon Phi coprocessor.

Repo

A repo is a form of short-term borrowing. Government securities are sold to investors and are repurchased by the seller within a short time at a greater price. The price difference can be viewed as interest and is called the "repo rate." The experiments compare the results of varying the repo purchase date, repo sale date, the repo rate, underlying fixed-rate bond rate, and underlying fixed-rate bond date. These variables affect whether or not the buyer and/or seller gains or loses money as a result of the agreement. Repo is a double-precision scenario. The reference paper claims a significant speed-up over the OpenMP implementation. Because the reference paper does not mention any test times, the graph data was interpreted to get the speed-up. The graph data shows that the K20 speed-up compared to the OpenMP implementation is over 6X for more than 100,000 repos.

Intel® Architecture Optimizations

The repo code is similar to the bond code and presents similar vectorization challenges, as listed below:

- Switch cases and if-else statements lead to vector dependence.
- if-else statement execution does not follow a pattern that would provide hints to the compiler to generate optimal code.
- Critical loop counts are low (<10).
- Data is not aligned due to nested structures.

The following optimizations were successfully applied:

1. Data layout: The Intel TBB memory allocator was used for memory allocation.
2. Existing arrays were defined as static and cache aligned.

Reference Test Configuration Comparison

The reference paper runs up to 2 million repos. Table 12 presents test times using 2 million repos. The original CUDA code takes 876 mSec and the Intel Xeon processor E5-2697 v3 takes 1027 mSec. The K40 outperforms the Intel Xeon processor E5-2697 v3 by 1.18X. Without vectorization, the Intel Xeon Phi coprocessor 7120P takes 1801 mSec.

Table 12[1]: Reference CUDA* results vs. optimized Intel® Architecture code for 2 million repos.

Repos	Reference Tesla* K40 (mSec)	Intel® Xeon® processor E5-2697 v3 (56 threads) with precision flags (mSec)	Intel Xeon processor E5-2697 v3 (56 threads) without precision flags (mSec)	Intel® Xeon Phi™ coprocessor 7120P (244 threads) with precision flags (mSec)	Intel Xeon Phi coprocessor 7120P (244 threads) without precision flags (mSec)
2 Million	867	1064	1027	1489	1801

(Source: Intel Measured)

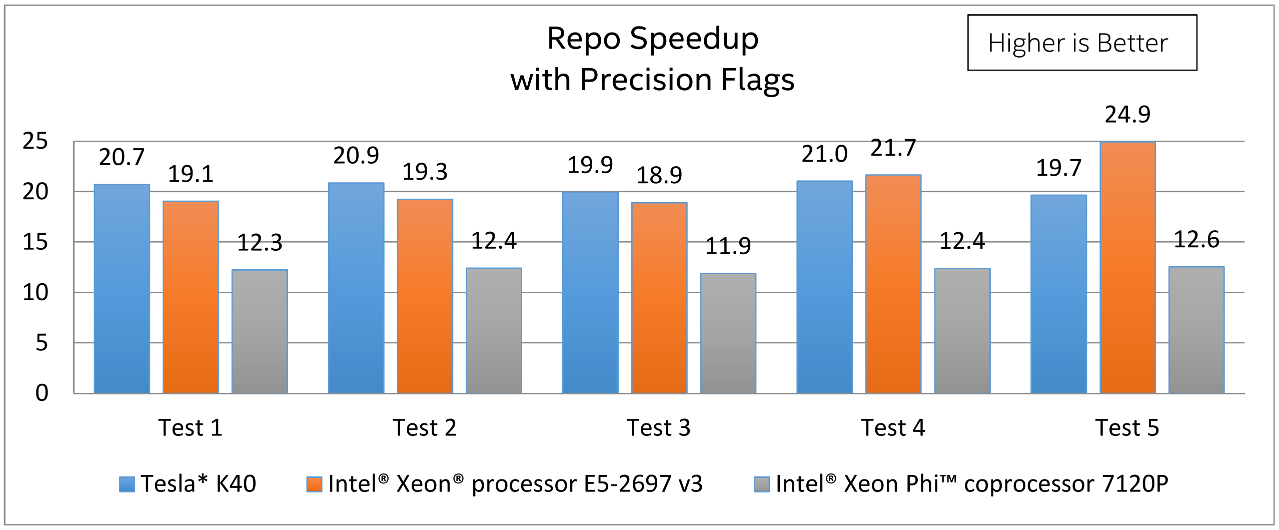
Modified Test Configuration Comparison

Table 13 presents the results for the CUDA version and optimized Intel Architecture version for varying number of repos. To extend the test times beyond 1 second additional repos were added by changing the `#define`. K40* could run a limit of 10 million repos without generating an error.

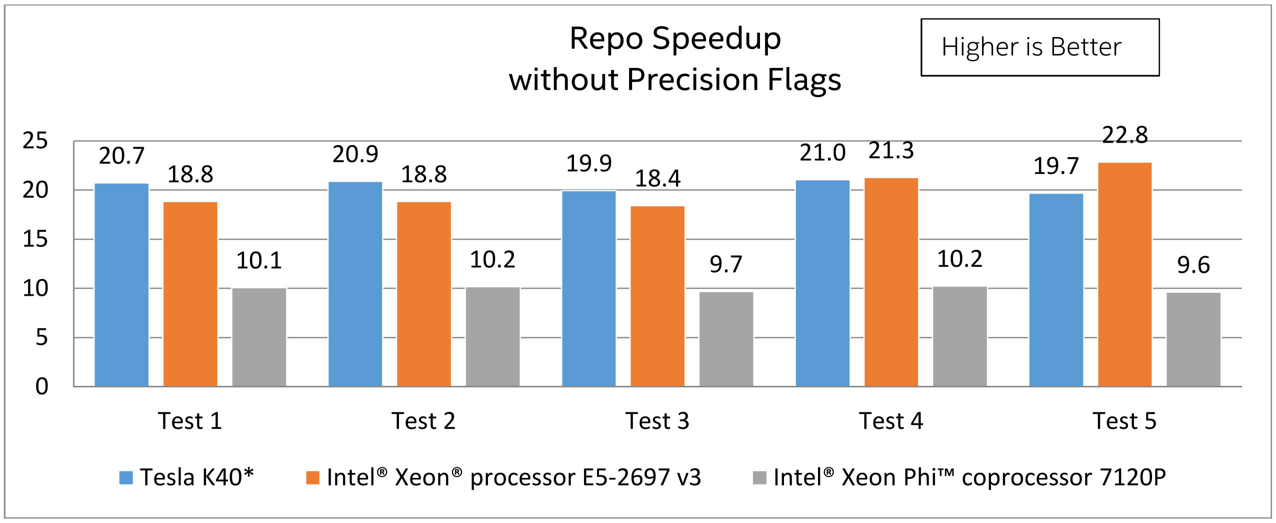
Table 13[1]: Modified CUDA* results vs. optimized Intel® Architecture code.

Test Number	Repos	Modified Tesla* K40 (Sec)	Optimized Intel® Xeon® processor E5-2697 v3 1T without precision flags (Sec)	Optimized Intel Xeon processor E5-2697 v3 56T with precision flags (Sec)	Optimized Intel Xeon processor E5-2697 v3 56T without precision flags (Sec)	Optimized Intel® Xeon Phi™ coprocessor 7120P 244T with precision flags (Sec)	Optimized Intel Xeon Phi coprocessor 7120P 244T without precision flags (Sec)
1	3 M	1.299	26.926	1.413	1.430	2.198	2.672
2	4 M	1.729	36.090	1.873	1.917	2.903	3.546
3	5 M	2.155	42.974	2.272	2.335	3.614	4.440
4	6 M	2.587	54.450	2.514	2.561	4.392	5.320
5	8 M	3.463	68.089	2.730	2.982	5.420	7.086

All times in seconds. (Source: Intel Measured)



Graph 7[1]: Speed-up compared to single-threaded performance on the Intel® Xeon® processor E5-2697 v3. (Source: Intel Measured)



Graph 8[1]: Speed-up compared to single-threaded performance on the Intel® Xeon® processor E5-2697 v3. (Source: Intel Measured)

Graphs 7 and 8 show the speed-up of the Tesla K40, the Intel Xeon processor E5-2697 v3, and the Intel Xeon Phi coprocessor 7120P compared to the single-thread performance on the Intel Xeon processor E5-2697 v3. Although there are challenges in vectorizing this workload on Intel Architecture, the Intel Xeon processor E5-2697 v3 performance is equivalent to the Tesla K40—in some cases slightly ahead and others slightly behind. The Tesla K40 outperforms Intel Xeon Phi coprocessor 7120P by up to 2X[1], due to the lack of vectorization.

Conclusions

The reference paper compared GPU/CPU performance on four Quantlib-based financial workloads. The reference paper GPU performance claims and the optimized Intel Architecture vs. K40 performance are presented in Table 14.

Table 14: Comparison between the reference paper claims to the modified test results discussed in this document.

Workload	Reference paper claim: Tesla* K20 performance gains vs. multi-threaded Intel® Architecture code	Actual results[1]: Intel® Xeon® processor E5-2697 v3 vs. Tesla* K40	Actual results[1]: Intel® Xeon Phi™ coprocessor 7120P vs. K40	Comments: Key IA performance optimizations
Black-Scholes*	>35X	Intel Xeon processor E5-2697 v3 outperforms K40 by a minimum of 1.15X	Intel Xeon Phi coprocessor 7120P outperforms K40 by 1.98X minimum.	Utilize vector units, proper data alignment and cache locality.
Monte-Carlo*	>140X	Intel Xeon processor E5-2697 v3 outperforms K40 by a minimum of 1.37X	Intel Xeon Phi coprocessor 7120P outperforms K40 by 1.99X minimum.	Utilize vector units, Intel® MKL for RND generation, proper data alignment and cache locality.
Bonds	>5X	Intel Xeon processor	K40 outperforms Intel	Proper data

		E5-2697 v3 outperforms K40 by a minimum of 1.4X	Xeon Phi coprocessor 7120P by up to 1.55X.	alignment and cache locality.
Repo	>5X	Equivalent	K40 outperforms Intel Xeon Phi coprocessor 7120P by up to 2.04X	Proper data alignment and cache locality.

(Source: Intel Measured)

None of the reference paper performance speedups can be substantiated. Utilizing both thread and data parallel resources combined with efficient memory utilization provided significant speedup over the reference IA code.

Since the Intel Xeon processor and the Intel Xeon Phi coprocessor use the same programming model, most of the optimizations made for the processor are applicable on the coprocessor, thereby achieving significant performance improvement for both the architectures.

By applying the stepwise parallelization methodology to utilize all of the parallel resources on Intel Architecture, both the Intel Xeon processor E5-2697 v3 and/or the Intel Xeon Phi coprocessor 7120P outperform the Tesla K40 for a majority of the workloads. For workloads that cannot utilize all of the parallel resources, the Intel Xeon processor E5-2697 v3 performance exceeds or is comparable to K40 demonstrating the flexibility of Intel Architecture.

References

1. Intel® Turbo Boost Technology:
   http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html
2. Intel® MKL User Guide:
   https://software.intel.com/sites/default/files/managed/4a/d6/mkl_11.2.1_lnx_userguide.pdf
3. Intel C++ Compiler version 15:
   https://software.intel.com/en-us/compiler_15.0_ug_c
4. Intel® Xeon Phi™ Coprocessor Vector Microarchitecture
   https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture
5. IA programming model:
   https://software.intel.com/en-us/articles/case-study-achieving-high-performance-on-monte-carlo-european-option-using-stepwise
6. OpenMP* 4.0 :
   http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
7. Intel® Threading Building Blocks:
   http://www2.thu.edu.tw/~emtools/VMC/Intel%20Threading%20Building%20Blocks.pdf
   https://software.intel.com/en-us/tbb_4.3_U2_doc

8. A Guide to Vectorization with Intel® C++ Compilers : https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf
9. Reference paper:
   http://dl.acm.org/citation.cfm?id=2458536

Performance Disclaimer: