# Analysis and Optimization of Financial Analytics Benchmark on Modern Multi- and Many-core IA-Based Architectures

Mikhail Smelyanskiy,
Jason Sewall,
Dhiraj D. Kalamkar,
Nadathur Satish,
Pradeep Dubey
Intel Labs

Nikita Astafiev,
Ilya Burylov,
Andrey Nikolaev,
Sergey Maidanov,
Shuo Li,
Sunil Kulkarni
Intel Software and Services Group

Charles H. Finan
Intel Data Center Group
Ekaterina Gonina
UC Berkeley

***Abstract*—In the past 20 years, computerization has driven explosive growth in the volume of financial markets and in the variety of traded financial instruments. Increasingly sophisticated mathematical and statistical methods and rapidly expanding computational power to drive them have given rise to the field of computational finance. The wide applicability of these models, their computational intensity, and their real-time constraints require high-throughput parallel architectures.**

**In this work, we have assembled a financial analytics workload for derivative pricing, an important area of computational finance. We characterize and compare our workload's performance on two modern, parallel architectures: the Intel® Xeon® Processor E5-2680, and the recently announced Intel® Xeon Phi™[1] (formerly codenamed 'Knights Corner') coprocessor. In addition to analysis of the peak performance of the workloads on each architecture, we also quantify the impact of several levels of compiler and algorithmic optimization.**

**Overall, we find that large caches on both architectures, out-of-order cores on Intel® Xeon® processor, and large compute and memory bandwidth on Intel® Xeon Phi™ coprocessor deliver high level of performance on financial analytics.**

## I. Introduction

To evaluate developments in computer architecture and assess the capabilities of new compilers, languages, and programming tools, computing disciplines have traditionally used workload benchmarks that are representative of their domains and stress different components of a computing environment.

Over the past few decades, the financial industry has required ever more computation to power sophisticated models of financial instruments, yet computational finance has relatively few such benchmarks. This is perhaps due to the relative secrecy of these financial models; interested parties in the world of finance are rarely willing to divulge the details of their models, so it is difficult to identify common ground among them.

There has been some effort to develop such benchmarks. Premia [1] was developed in the 1990s for option pricing, hedging and financial model calibration. More recently, STAC [2] was proposed for testing compute-intensive analytic workloads such as risk management and pricing, real-time/near-real-time model calibration, and strategy backtesting. While other published work has examined the performance of individual financial workloads on several architectures (for example, [3] and [4]), to our knowledge, no attempt has been made to quantify and analyze performance of a diverse set of financial workloads on the IA-architecture. This paper aims to rectify that; to that end, we make the following contributions:

- The definition, mapping, and analysis of a representative finance benchmark for option pricing on two modern multi- and many-core IA architectures: the Intel® Xeon® Processor E5-2680 (codenamed 'Sandy Bridge', hereafter SNB-EP) and the Intel® Xeon Phi™ (formerly codenamed 'Knights Corner') coprocessor (KNC).
- Novel algorithms for tiling and vectorizing two of these benchmarks: binomial tree and Crank-Nicolson.
- Quantification of the extent of the 'Ninja Gap' [5] — the performance gap between naïvely-written C/C++ code and fully optimized, peak-performance code.
- The first examination of the performance characteristics of the forthcoming KNC on financial workloads.

Modern IA-based multi-core architectures deliver hundreds of gigaflops (Gflops) of double-precision performance and tens of gigabytes per second (GB/s) in memory bandwidth. Backed up by an aggressive out-of-order execution unit and large last-level caches, they deliver competitive single-thread and parallel performance on a wide spectrum of applications, from medical imaging [6] to graph problems [7] and beyond. The forthcoming KNC delivers a considerable improvement in both compute and memory bandwidth performance to meet the fast-growing demands of modern throughput workloads.

The rest of the paper is organized as follows: Sec. II reviews derivative pricing and its associated numerical methods and introduces the six kernels analyzed in this paper. Sec. III introduces the two hardware configurations used in our studies, as well as our optimization methodology. Sec. IV provides results and analysis of each kernel, and we provide qualitative

[1]Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

| Valuation Models | Pricing Methods | Exercise Styles | Key Math Kernels |
|---|---|---|---|
| Analytical | Black-Scholes | European | Transcendental Math Functions |
| | Roll-Geske-Whaley | American | |
| Lattice | Binomial | European & American | Numerical linear algebra<br>N-point Stencil |
| | Trinomial | | |
| Finite-Difference | Crank-Nicolson | European | |
| | Projected SOR | American | |
| Monte Carlo | Standard MC | European | RNG<br>Brownian Bridge<br>Var Reduction |
| | Longstaff & Schwartz | American | |

Fig. 1: Pricing methods in derivative pricing

discussion of the results and conclusion in Sec. V.

## II. Derivative Pricing

An *option* is a derivative financial instrument that specifies a contract between two parties for a future transaction on an *underlying*: an asset, such as a stock or bond. The holder of an option has the right— but not an obligation — to buy (*call*) or sell (*put*) the asset by a certain date (the *expiry* date) at a certain price (the *strike* price). The *payoff* of an option describes the value of the option as a function of the underlying asset at the time of (or before) expiry.

The two most popular types of options are *European*-style options, which can be exercised only at expiry, and *American*-style options, exercised at any time up to expiry. As an example, the payoff of a European put depends on the price of the underlying asset at expiry $T$ and strike price $K$; call this future price $S_T$. Then, the put payoff $P(S_T)$ is expressed as $P(S_T, T) = \max(K - S_T, 0)$.

There are many methods for pricing options; they fall into four categories, as shown in Fig. 1. *Analytical techniques* provide closed-form solutions to underlying mathematical model of an option — examples of such techniques are the Black-Scholes [8] equation for pricing European options. While closed-form solutions are computationally efficient, such solutions are rarely available, and other numerical methods are needed to price options.

The three most common techniques are *lattice methods*, *finite difference methods* and *Monte Carlo methods*. Lattice methods form a lattice of all possible price paths of the underlying asset, and compute an expectation value that represents the price of the option. Finite difference methods compute solutions to the Black-Scholes equation by discretizing the differentials; the solution is then marched backwards. The limitation of both lattice and finite difference methods is their high computational and space complexity, which scales exponentially with the number of underlyings. As a result, these methods are used only for problems with a small number of underlyings ($\leq 3$). For the most complex options, Monte Carlo approaches are employed. This technique uses stochastic integration to simulate random price paths for each underlying.

Methods for pricing options reply on a relatively small set of low-level mathematical techniques. Examples of these are shown in the right column of Fig. 1. The focus of this paper is on several representative methods and mathematical kernels, shown by the shaded boxes — each of these methods is further described below.

### A. Black-Scholes

The Black-Scholes equation [8] is a second-order partial differential equation for the evolution of an option price over time:

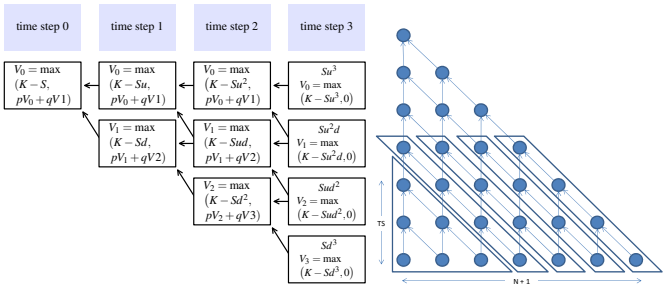
$$\frac{\partial V}{\partial t} + \frac{\sigma^2 S^2}{2}\frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \qquad (1)$$

This equation describes how the option value $V(S, t)$ depends on the stock price $S$ and time $t$. $\sigma$ is the volatility of the underlying, and $r$ is the interest rate. Both $\sigma$ and $r$ are considered given, while $V$ is the quantity being computed (or approximated). For European options, a closed-form solution exists for this PDE — we describe this in Sec. IV-A1.

### B. 1D binomial tree

The binomial option pricing method models the price evolution of an option over the option's validity period using a recombining tree.

For American options (where options may be exercised any time until expiry), there is no known closed-form solution that predicts price over time; the binomial option method provides a very close approximation. As shown in Fig. 2a, the binomial model represents the price evolution of an option's underlying as a binomial tree of all possible prices at equally-spaced time steps from the current date. It assumes that at each step, each node price can only increase by $u$ or decrease by $d$ with probabilities $P_u$ and $P_d$, respectively. Hence, each node has two children, and the process repeats over $N$ time steps. This results in a tree: the root of the tree represents the current price, and each level below represents all possible prices at specific points in the future.




(a) Binomial tree for four time steps (b) Tiling scheme for binomial tree reduction

Fig. 2: Binomial tree computation patterns

The value at a leaf node can be directly calculated as $\max(S_n - K, 0)$ for a call option and $\max(K - S_n, 0)$ for a put option, where $K$ is the exercise price and $S_n$ is the stock price at the leaf. The tree is then walked backwards and the

option price at each intermediate node at time $t$ is computed as the average of its prices from $t+1$. In the case of American options, the maximum of the computed option price at each node and the early exercise payoff is also taken. This process repeats until the entire tree is reduced to a single node; this contains the expected option price at the current date.

While this method is most useful for pricing American options, it can be also used to price European options. As the two computations are very similar in the binomial tree model, we have opted to focus on European options.

### C. Crank-Nicolson

Crank-Nicolson is a finite difference method that can be used for option pricing. The following focuses on American options, which do not have explicit formulas and for which numerical methods are essential. Option prices are modeled using a lattice $u_j^n$, where $n$ represents discretized time from 0 to maturity, and $j$ represents the discretized price of underlying asset. Each option is valued using an average of explicit and implicit methods [9] using the equations ($\forall\, j,\, n$):

$$u_j^{n+1/2} = (1-\alpha)u_j^n + \frac{\alpha}{2}(u_{j+1}^n + u_{j-1}^n) \qquad \text{(explicit)}$$

$$u_j^{n+1/2} = (1+\alpha)u_j^{n+1} - \frac{\alpha}{2}(u_{j-1}^{n+1} + u_{j+1}^{n+1}) \qquad \text{(implicit)}$$

where $\alpha$ is the ratio of the temporal discretization to the square of the price discretization.

While the explicit half step can be analytically computed, the implicit step is solved using iterative methods — such methods iteratively update the solution, stopping when a desired accuracy is reached. Gauss-Seidel Successive Over-Relaxation (GSOR) is one such method, and has been shown to converge faster than traditional methods [10]. Here a system of linear equations of the form $Au^{n+1} = b^{n+1/2}$ ($A$ is a $j \times j$ matrix, $b$ a vector of length $j$) is solved with the following equation (for iteration $k$):

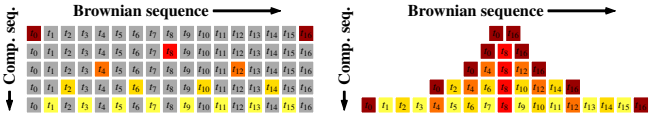
$$u_j^{n+1,k+1} = \frac{1}{1+\alpha}(b_j^{n+1/2} + \frac{\alpha}{2}(u_{j-1}^{n+1,k+1} + u_{j+1}^{n+1,k})) \quad (2)$$

For American options, these options can be exercised at any timestep — this is done if the payoff values at each timestep $g_j^n$ exceeds $u_j^n$. As such, a maximum of these two values is taken to be the next $u_j^n$ value. This modification of GSOR for pricing American options is called Projected SOR [11].

### D. Monte Carlo options pricing

For many types of financial derivatives (such as American options) the closed-form solution to Black-Scholes (Eq. (1)) cannot applied due to uncertain, time-varying factors. In such scenarios, Monte Carlo integration techniques can be applied [12].

These techniques operate by discretizing the time interval $[0, T)$ into $P$ discrete intervals, then integrating the Black-Scholes formula over each subinterval to obtain the next price $S(t_i)$ from $S(t_{i-1})$. Here, $P$ is known as the *path length* for the integration.




(a) Depth-level Brownian bridge (b) Depth-level Brownian bridge compute arrays

Fig. 3: Brownian bridge compute pattern

| | SNB | KNC |
|---|---|---|
| Sockets×Cores×SMT | 2 × 8 × 2 | 1 × 60 × 4 |
| Clock (GHz) | 2.7 | 1.09 |
| Single Precision GFLOP/s$^2$ | 691 | 2127 |
| Double Precision GFLOP/s$^2$ | 346 | 1063 |
| L1 / L2 / L3 Cache (KB) | 32 / 256 / 20,480 | 32 / 512 / - |
| DRAM | 128 GB | 4 GB GDDR |
| Bandwidth from STREAM [14] | 76 GB/s | 150 GB/s |
| PCIe™ Bandwidth | 6 GB/s | |
| Compiler Version | Intel® v13.0.030 | |
| MPI Version | Intel® v4.0.3 | |

TABLE I: System configuration.

The resulting technique typically consumes $P$ normally-distributed random numbers $Z_0$, $Z_1$, …, $Z_{p-1}$ and estimates the value of the option with error that is $O(P^{-1/2})$.

Monte Carlo methods are valuable in a benchmarking environment for their ability to stress random-number generation facilities and mathematical functions.

### E. Brownian bridge

The 'random walk' of investment fame [13] is closely related to the concept of *Brownian motion*. In these processes, the variation of a variable over time is modeled by applying a random perturbation over a series of discrete time steps; such variables classically represent the motion of a particle in space, but have applications in more abstract domains — such as the evolving value of an asset or investment.

Conceptually, Brownian motion is the Wiener process; given the initial value $v(0)$, successive $v(t_i)$ are computed by adding scaled, normally-distributed samples $Z_i$:

$$v(t_i) = v(t_{i-1}) + \sqrt{t_i - t_{i-1}}Z_i \quad (3)$$

While the above process generates increasing $t_i$, the Brownian bridge technique allows individual steps in a Brownian motion to be computed more flexibly; it is based on a property of Brownian motion that given two points $v(r)$ and $v(t)$ in a Brownian motion, if there are no points $v(s) \forall s \in (r, t)$, any such $v(s)$ can computed based on $v(r)$ and $v(t)$. So, for example, when both the initial value $v(0)$ *and* final value $v(t_k)$ are known, the Brownian motion between these points can be reconstructed in arbitrary detail through the bridge process. The properties of the Wiener process that give rise to the Brownian bridge technique are lucidly described in [12].

Brownian bridge is a valuable candidate in a benchmark, as it stresses the ability of a computing environment to deal with indirection and large working sets.

## III. Analysis Testbed

### A. Hardware Platforms

In our experiments, we have used the following systems:

**Intel® Xeon® Processor E5-2680 "SNB-EP":** This is an x86-based multi-core server architecture featuring a super-scalar, out-of-order micro-architecture supporting 2-way hyper threading. In addition to scalar units, it has a 256 bit-wide SIMD unit that executes the AVX instruction set. Separate multiply and add ports allow for the execution of one multiply and one addition instruction (each 4-wide, double-precision) in a single cycle.

**Intel® Xeon Phi™ coprocessor "KNC"[3]:** This features many in-order cores on a single die; each core has 4-way hyper-threading support to help hide memory and multi-cycle instruction latency. To maximize area and power efficiency, these cores are less aggressive — that is, they have lower single-threaded instruction throughput — than SNB-EP cores and run at a lower frequency. However, their vector units are 512 bits wide and execute 8-wide double-precision SIMD instructions.

KNC is physically mounted on a PCIe card and has dedicated GDDR memory. Communication between the host CPU and KNC is therefore done explicitly through message passing. However, unlike many other coprocessors, it runs a complete Linux-based operating system, with full paging and virtual memory support, and features a shared memory model across all threads and hardware cache coherence. Thus, in addition to common programming models for coprocessors, such as OpenCL*, KNC supports more traditional multiprocessor programming models such as pthreads and OpenMP* API.

Additional details can be found in Tab. I. Note that L1/L2 cache sizes for both SNB-EP and KNC are per core, while L3 cache size for SNB-EP is per chip and shared among all cores. Lastly, in terms of peak compute, KNC is 3.2x faster, compared to SNB-EP ($\frac{60}{16} \times \frac{512}{256} \times \frac{1.09}{2.7}$).

### B. Optimization Methodology

We have have categorized every optimization taken on each benchmark into three levels:

**Basic** optimizations rely on the compiler and require minimal changes to the reference code; only compiler directives such as `#pragma unroll`, for loop unrolling, `#pragma simd` for autovectorization, `#pragma omp` for thread-level parallelism are permitted at this level.[4]

**Intermediate** optimizations require some changes to the reference code; examples of this are outer loop vectorization using `F64vec8` notation (see below), manual loop unrolling, and manual insertion of software prefetches for data structures that do not fit in the cache. Another example is replacing some functions with an optimized library calls from the Intel® Math Kernel Library.

**Advanced** optimizations entail algorithmic restructuring, such as modifying data structure layout to improve SIMD efficiency (for example, to 'transpose' an array-of-structures into a structure-of-arrays, also known as an AOS to SOA transformation), and blocking/tiling to reduce working sets.

To justify the need for intermediate and advanced optimizations, we analyze the basic performance using the Intel® Inspector XE and VTune™ Amplifier XE tools available on both architectures. For some kernels, we also develop intutitive performance models to understand the gap between achievable and deliverable performance.

Outer loop vectorization — such as a loop over a number of options — is often the most natural way to exploit SIMD. However, it frequently requires data reformatting to reduce non-coalesced memory accesses, reduces the amount of coarse-grain parallelism available to threads, and potentially enlarges the working set by the vector width. These are occasional drawbacks; this technique proved useful in several kernels. Outer loop vectorization can be realized in three different ways: 1) Using the `#pragma simd` on the outer loop, 2) using the Intel® CilkPlus array notation extension [15] and 3) replacing scalar types with C++ classes for SIMD operations. These classes simply wrap intrinsic functions for vector operations and provide convenient infix operator syntax — the resulting code appears practically identical to the scalar code. An example is shown in Lis. 3 for KNC; here, `F64vec8` is a vector class, which represents an 8-wide vector of 64-byte double-precision quantities. For, SNB-EP the corresponding vector class is called `F64vec4`. Since one is merely substituted for the other when moving between platforms, we refer to these interchangeably throughout the text.

Listing 1: Black-Scholes reference code

```
1  double sig22 = sig*sig/2;
2  for(int i = 0; i < nopt; ++i) {
3    double qlog  = log(opts[i].S/opts[i].X);
4    double denom = 1/(sig*sqrt(opts[i].T));
5    double d1    = (qlog+(r+sig22)*opts[i].T)*denom
6    double d2    = (qlog+(r-sig22)*opts[i].T)*denom;
7    double xexp  = opts[i].X*exp(-r*opts[i].T);
8    opts[i].call = -xexp*cnd( d2)-opts[i].S*cnd( d1);
9    opts[i].put  =  xexp*cnd(-d2)-opts[i].S*cnd(-d1);
10 }
```

## IV. Derivative Pricing Benchmark Optimizations & Performance Results

This section provides details on the implementation and optimization of each benchmark, as well as performance results and analysis.

### A. Black-Scholes

*1) Reference implementation:* Our reference implementation of the closed-form solution to Black-Scholes model is shown in Lis. 1; it computes call and put prices of `nopt` European-style options and depends on 5 input parameters: the current option price `S`, the strike price `X`, the time to expiration `T`, risk free interest rate `r`, and the implied volatility of the

[2]Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

[3]Evaluation card only and not necessarily reflective of production card specifications

[4]Most pragmas and compilation options behave the same across SNB-EP and KNC.

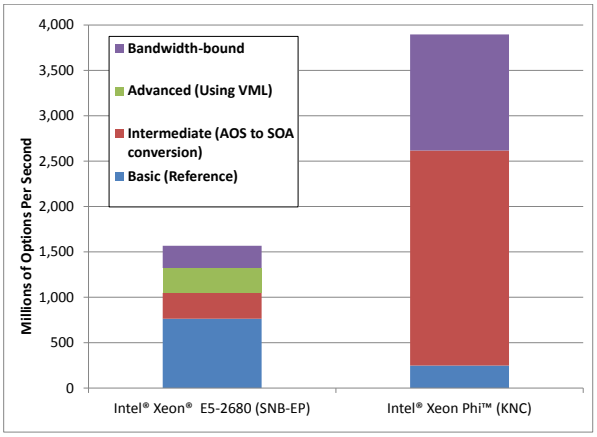



Fig. 4: Performance[5,6] of Black-Scholes

underlying `sig` — we assume that `r` and `sig` are the same for all options. `cnd` is the cumulative normal distribution function. Lis. 1 spends a large fraction of its time in transcendental functions; each option is priced using a sequence of operations that compute the inverse cumulative normal distribution function, followed by math operations involving `exp`, `log`, `sqrt` and division operations. The total computation performed is about 200 ops, while streaming in 24 bytes writing out 16 bytes for each option.

*2) Our implementation:* Our implementation exploits SIMD across options, assigning one option per SIMD lane. To reduce some arithmetic complexity, we also combine call and put options together to take advantage of call/put parity [16]. Furthermore, to take advantage of the fast, vectorized math function implementations in the Intel Short Vector Math Library (SVML), we replace `cnd` with the error function `erf` utilizing the following equivalence: $\mathrm{cnd}(x) = (1+\mathrm{erf}(x/\sqrt{2}))/2$. `erf` is less computationally intensive than `cnd`, while this substitution provides the same accuracy. To further improve SIMD efficiency, we have transposed the data layout (from AOS to SOA, as in [5]).

*3) Performance:* The performance for Black-Scholes is shown in Fig. 4; throughout the paper, performance data is organized as a stacked bar-chart that shows the incremental performance improvement for each level of optimization.

On KNC, the reference version is 3x slower than on SNB-EP; this is due to the fact that the reference data is in AOS format, which requires gathering (scattering) data spread across as many as vector length cachelines. This makes both SNB-EP and KNC implementation instruction- or compute-bound. However, with only a vector length of 4 and superscalar execution, on SNB-EP the overhead of AOS format is less pronounced. In contrast, on KNC, gathering (scattering) data across 8 cachelines for each access to the input (output) data

[5] 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 information go to `http://www.intel.com/performance`

[6] 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 SSE3 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

Listing 2: Reference code for binomial tree options pricing

```
1 for(int o=0; o < noptions; o++)
2 {
3   for(int i = N; i > 0; i--)
4     for(int j = 0; j <= i - 1; j++)
5       opt[o].Call[j] = puByDf*opt[o].Call[j+1]+pdByDf*
            opt[o].Call[j];
6   opt[o].callResult = opt[o].Call[0];
7 }
```

results in more than 10x increase in the number of instructions. This problem is mitigated on KNC by applying an AOS to SOA data conversion — performance improves by 10x. Using the Intel VML is more efficient on SNB-EP than on KNC, where it shows no benefit over SVML. The highly-tuned transcendental math functions are unrolled and inlined by the autovectorizing compiler in SVML, which outperforms the VML version which has a larger cache footprint and requires algorithmic restructuring of both code and data. This is not always the case, as for other workloads VML is faster than SVML.

We use five double-precision parameters to price each option: three for input and two for output. Assuming streaming stores (available on both architectures), the bandwidth-bound performance is $B/40$ options per second, where $B$ is STREAM bandwidth — found in Tab. I. SNB-EP achieves 84% of the bound, while KNC achieves 60% and is thus more compute-bound.

### B. 1D binomial tree

*1) Reference implementation:* The core compute kernel for binomial option pricing for European Options is shown in Lis. 2. The outer loop iterates over options, two inner loops compute price of the option, $o$. `puByDf` and `pdByDf` are the scaled probabilities $P_u$ and $P_d$, respectively (see Sec II-B). The procedure starts from the leaves and moves backward in time; each time step, the `Call` array is updated with values computed at a previous step — eventually, `Call[0]` contains the option price. This kernel requires $\approx 3N(N+1)/2$ floating point computations. The compiler is able to autovectorize and unroll the `j` loop of the reference code on both SNB-EP and KNC; however, the resulting code contains an unaligned load for `Call[j+1]`. Additionally, there is a small loss of SIMD efficiency at the end of the `j` loop when `i` is not a multiple of the SIMD width.

*2) Our implementation:* To improve SIMD efficiency and avoid unaligned memory accesses, we compute one option per SIMD lane — thus, on SNB-EP processors we process 4 double-precision options, and on KNC, 8. This requires vectorization of the outer loop, which is accomplished using the `F64vec8` classes.

We also propose a new tiling scheme to tune the problem based on register file size, cache size, or both. The scheme, shown in Lis. 3, reduces both the working set and instruction overhead of the binomial tree loop. To tile for registers, we pick a tile size `TS` such that the `Tile` array may be allocated in a processor's register file. We then perform the computation in two steps: first, we copy the first `TS` values from the `Call` array

Listing 3: Tiling Algorithm for Binomial Tree Reduction

```
F64vec8 Call[NUM_STEPS+1] = ...;
F64vec8 Tile[TS], m1, m2, m3;
for(m = N; m >= TS; m -= TS) {
  ...
  for(i = TS; i <= m; i++) {
    m1 = Call[i];
    for(j = TS-1; j >= 0; j--) {
      m2 = puByDf * m1 + pdByDf * Tile[j];
      Tile[j] = m1;
      m1 = m2;
    }
    Call[i-TS] = m1;
  }
}
return Call[0];
```

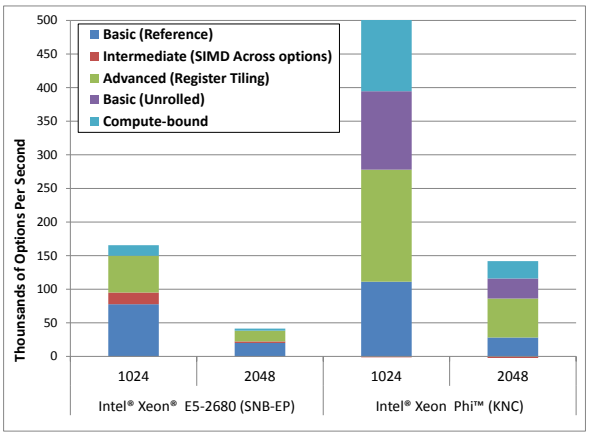



Fig. 5: Performance[5,6] of Binomial Tree European Options Pricing using 1024/2048 time steps (maximum width of the binomial tree)

into `Tile` array and reduce it within register file (see the lower-triangular portion in Fig. 2b). Secondly, we read successive values from the `Call` array, reduce it by `TS` time steps, and store it back to `Call[i-TS]` (the shaded trapezoidal portion). Thus, for `TS` time steps, we read each call value only once and store it back once; the rest of the computation happens entirely within the register file; this significantly increases the arithmetic intensity of the code.

A second-level of tiling can be done similarly, save that `Tile` is now chosen to reside in cache rather in the register file. The inner loop in Lis. 3 is further unrolled and the register move eliminated.

*3) Performance:* The performance for the binomial tree workload is shown in Fig 5. In the reference code with basic optimizations (autovectorization of the inner loop, OpenMP pragmas over outer loop), we see that KNC is 1.4x faster than SNB-EP.

Applying the intermediate-level optimization of SIMD across options hardly improves performance on either platform, but when combined with register tiling, performance increases by more than 2x on both architectures. This is due to the fact that register tiling eliminates extra load/store instructions.

Loop unrolling has little effect on the superscalar SNB-EP, which dynamically exposes a large amount of instruction-level parallelism (ILP). KNC, on the other hand, with its in-order pipeline, sees as high as 1.4x speedup.

Listing 4: Brownian bridge reference code

```
for(int s = 0; s < sim_n; ++s) {
  src[0] = 0.0;
  src[1] = r[i++] * last_sig;
  for(int d = 0; d < bridge_depth; ++d) {
    dst[0] = src[0];
    for(int c = 0; c < (1 << d); ++c) {
      dst[2*c+1] = src[c]  *w_l[d][c]
                 + src[c+1]*w_r[d][c]
                 + sig[d][c]*r[i++];
      dst[2*c+2] = src[c+1];
    }
    swap(src, dst);
  }
  for(int c = 0; c < 1<<bridge_depth+1; ++c)
    sim[s][c] = dst[c];
}
```

The upper bar shows the upper bound for achievable performance based on the minimum flops to price one option; SNB-EP comes within 10% and KNC comes within 30% of this performance bound. Overall, KNC is 2.6x faster than SNB-EP for both 1K and 2K time steps, which is commensurate with the difference in their peak flops (see Tab. I).

### C. Brownian bridge

*1) Reference implementation:* The reference code for Brownian bridge is shown in Lis. 4; this is the depth-level variation (see Fig. 3) similar to what is found in [12]. Each simulation has $2^{\texttt{bridge\_depth}}+1$ steps (that is, `1<<bridge_depth+1` ), and the coefficients for each quantity to be computed at level `d` are stored in `w_l[d]`, `w_r[d]`, and `sig[d]` — these are constant and depend only on the length of the simulation. The input array `r` is a stream of normally-distributed random numbers, and the solution for each simulation `s` is stored in `sim[s]`.

A key challenge for this code is keeping the working set that is ping-ponged between the buffers `src` and `dst` in cache.

*2) Our implementation:* Vectorization of the outermost loop cannot be automatically applied because of the way random numbers are consumed across iterations; to utilize SIMD on both architectures, minor modifications are needed to ensure that random numbers are loaded in vector-width chunks, and then `F64vec8` can be introduced to achieve the vertical vectorization with one simulation per SIMD lane; this is an intermediate-level optimization.

For small- to modest-length sequence generation, the working set (the buffers `src` and `dst`, and the coefficients `w_l` , `w_r`, `s`) fit in cache on both architectures, even after the vertical vectorization. However, each sequence step consumes a random number and the problem can become bandwidth-bound when the random numbers are streamed from main memory. It is possible to reduce bandwidth usage and improve overall performance by interleaving the computation of random numbers with the construction of the bridge; a chunk of numbers small enough to fit into lowest-level cache (LLC) is generated and then consumed from LLC by the bridge construction. This process continues, alternating until the bridge construction is complete and alleviating the memory bandwidth bottleneck. We consider this an advanced-level optimization.

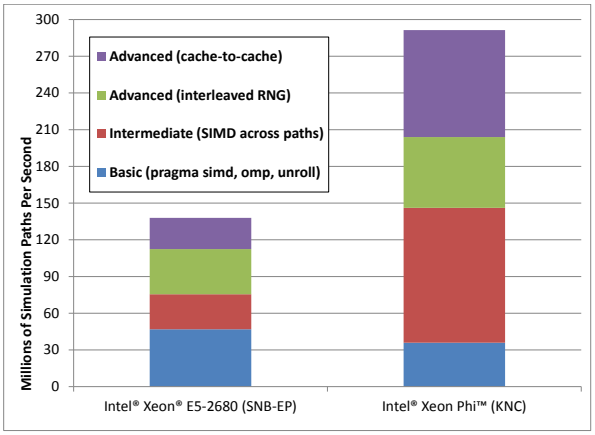



Fig. 6: Performance[5,6] of 64-step, double-precision Brownian bridge for KNC and SNB-EP

Even when interleaving random number generation and bridge construction, the results of each constructed bridge is often written out to memory. In the event that the computed Brownian sequence is to be used immediately and discarded, the sequence can also be divided into chunks and left in LLC for the next compute stage, thus further reducing amount of memory traffic. This is an advanced optimization, and applies only in situations where it is acceptable to immediately use the simulation results without streaming them back out to memory.

*3) Performance:* In Fig. 6, we show the performance of the Brownian bridge algorithm at different levels of optimization, on various architectures. The graph demonstrates the number of double-precision, 64-step simulations that can be performed per second. At the basic level of optimization, SIMD has not been brought to bear on the problem; KNC is 25% slower than SNB-EP. With intermediate optimizations, the performance on KNC improves over SNB-EP— at this point, both architectures are memory bandwidth-bound, and the performance of KNC exceeds that of SNB-EP by the difference their memory bandwidths. The advanced optimizations allow both architectures to become compute-bound. KNC is 2x faster than SNB-EP which is less than the difference in compute flops between both architectures; this is due to the lack of FMAs in the core compute.

Normally-distributed random numbers are a crucial part of any Brownian motion calculation; while the timings in Fig. 6 do not account for the time taken for random number generation, they cover both the case where numbers are streamed from memory and the case where there are generated into the cache and used directly. Tab. II gives timings for normally-distributed random numbers to give an idea of what the combined computation costs would be.

### D. Monte Carlo options pricing

*1) Reference implementation:* Lis. 5 shows a listing of a reference implementation of European options pricing. For each of `nopt` options, we compute `npath` paths based on the input per-option strike prices `X[o]`, stock prices `S[o]`, and option years `T[o]`, and we store the output in `result[o]` and `confidence[o]`. We assume the input `vol` and `mu` are constant

Listing 5: European options pricing reference code

```
1  for(int o = 0; o < nopt; o++) {
2    const double v_rt_t = sqrt(T[o]) * vol;
3    const double mu_t   =      T[o]  * mu;
4    double       v0     = 0, v1 = 0, result;
5    for(int p = 0; p < npath; ++p) {
6      const double r = STREAM ? (m_r[p] : NormalRNG());
7      res = max(0, S[o]*exp(v_rt_t*r + mu_t)-X[o]);
8      v0 += res;
9      v1 += res*res;
10   }
11   result    [o] += v0;
12   confidence[o] += v1;
13 }
```

(`vol` is volatility, while `mu` is derived from the risk-free interest rate and volatility).

Typically, `npath` $\gg$ `nopt` — the inner loop dominates the performance of this algorithm. In particular, the transcendental function call `exp` accounts for the bulk of the runtime of the code.

The code consumes one random number per path step. Two practical implementations are considered. In the first, if `STREAM` is true, random numbers are streamed from memory (`m_r`), and the same set of numebrs is used for all options. In the second, if `STREAM` is false, they are dynamically computed by `NormalRNG` (which is also vectorized), while the new set of random numbers is generated for each option., When streaming, the instruction overhead of computing the double-precision `exp` is high enough such that code remains compute-bound.

*2) Our implementation:* The inner loop in the European options pricing code shown in Fig. 5 consists of 3 multiplications, 4 adds, a `max` operation, and an `exp` call. Peak performance is achieved through the following basic optimization tools; we apply autovectorization to the innermost loop, resulting in a 4x performance gain due to SIMD on SNB-EP and 8x on KNC. Note that the autovectorizer also handles the reduction of `v0` and `v1` across iterations.

With the help of `#pragma unroll`, the compiler unrolls the loop, eliminates back-to-back dependencies, and exposes more ILP to hardware.

European option pricing is easily made to achieve maximum throughput on both SNB-EP and KNC; only a handful of compiler pragmas are needed to parallelize across cores, take advantage of SIMD, and achieve high IPC.

*3) Performance:* Random number generation is an essential part of any Monte Carlo integration technique, and it is possible to utilize the wide vector units and many cores present on both SNB-EP and KNC to generate the normally-distributed random numbers crucial to this and other financial workloads. We use the Intel® MKL Mersenne twister[17] (2203 variant) as the basis for our random number generation (this is ultimately transformed into the appropriate normal distribution).

Tab. II, shows the performance of European options pricing and random number generation on both architectures; the first row demonstrates the number of double-precision options of path length 256k that can be performed per second, assuming that pre-generated random numbers are streamed from memory. The second row shows the performance when random numbers are generated along with the path integration

| | SNB-EP | KNC |
|---|---|---|
| options/sec (stream RNG) | 29,813 | 92,722 |
| options/sec (comp. RNG) | 5,556 | 16,366 |
| normally-dist. DP RNG/sec | 1.79e9 | 5.21e9 |
| uniform DP RNG/sec | 13.31e9 | 25.134e9 |

TABLE II: Performance[5,6] of double-precision European options pricing (path length 256k) on SNB-EP and KNC, along normally-distributed double-precision random number generation performance.

Listing 6: Crank-Nicolson algorithm

```
1  int oldloops  = 10000, loops;
2  double omega  = 1.0; double domega = 0.05;
3  double alpha  = 0.73;
4  double alpha1 = 1.0-alpha; alpha2  = alpha/2;
5  for(int j = jmin; j <= jmax; ++j)
6    U[j] = u_payoff(j*dx, 0);
7  for(int n = 1; n <= nmax; ++n) {
8    double tau = n*dt;
9    for(int j=jmin+1; j<=jmax-1; ++i) {
10     G[j] = u_payoff(j*dx, tau);
11     B[j] = alpha1*U[j]+alpha2*(U[j+1]+U[j-1]);
12   }
13   U[jmin] = G[jmin] = u_payoff(xmin, tau);
14   U[jmax] = G[jmax] = u_payoff(xmax, tau);
15   loops = GSOR(B, U, G, omega);
16   if(loops > oldloops) {
17     omega   += domega;
18     oldloops = loops;
19   }
20 }
```

— here, the random-number generation process dominates the performance. The third and fourth rows show the raw performance of the random-number generation algorithm for normal- and uniform-distributed quantities, respectively. For all four of these workloads, both architectures are compute-bound, and the wider SIMD width and FMA advantage of KNC is demonstrated.

### E. Crank-Nicolson

*1) Reference implementation:* As described in Sec. II-C, the Crank-Nicolson method computes option prices on a lattice; the basic algorithm (based on the description in [9] and [11]) is shown in Fig. 6.

The runtime of the loop containing the explicit method is very small. Apart from the explicit method, computation of the payoff values `G` also occurs in this loop. The cost of computing payoffs is dominated by the cost of math operations, primarily exponent. The exponent occurs inside

Listing 7: Implicit GSOR algorithm. Dependencies across the `u` array prevent auto-vectorization.

```
1  extern double alpha; // global variable
2  int GSOR(double *B, double *U, double *G, double om) {
3    double error, coeff = 1/(1+2*alpha);
4    int loops = 0;
5    do {
6      ++loops;
7      error = 0;
8      for(int j = jmin+1; j < jmax; ++j) {
9        double y   = coeff*(B[j]+alpha*(U[j-1]+U[j+1]));
10       y          = max(G[j], G[j]+om*(y-U[j]));
11       double err = y - U[j];
12       error     += err*err;
13       U[j]       = y;
14     }
15   } while(error > epsilon);
16   return loops;
17 }
```

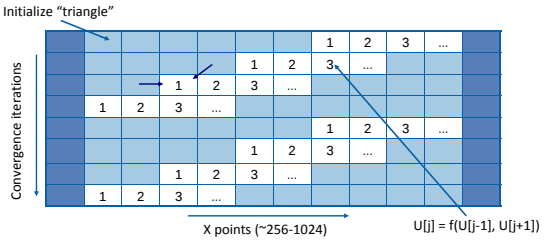



Fig. 7: Vectorization of GSOR in Crank Nicolson. Computation of matrix elements shown with the same number are done is SIMD. There is a prologue and epilogue triangular regions that must be separately vectorized.

lines 5 and 9 in Listing 6, where `u_payoff` is called. `alpha` is a global variable denoting Autovectorizion works well on this loop, generating SVML instrinsics. Typically, this is only about 10% of overall time and does not benefit much from further optimizations.

The implicit method is computed using a GSOR iterative solver. This code (shown in Fig. 7), is not easily vectorized since both the inner `j`-loop over asset prices and the outer do-while convergence loop both have dependencies. The dependence pattern of this code is as shown in Fig. 7; vectorizing requires algorithmic and data-structure changes.

*2) Our implementation:* Here, we focus on the implicit GSOR computation. We parallelize the computation across different options using OpenMP pragmas. To vectorize a single option computation, we first unroll the convergence loop by a factor of the vector width. Since this can increase the number of iterations in the convergence loop — we now check for convergence every 4 or 8 iterations, as opposed to every iteration in original code — this optimization can not be performed by the compiler. We then observe the pattern of the dependencies in the convergence and space loops (shown using the arrows in Fig. 7), and manually vectorize the code as shown in Fig. 7. This requires initial prologue and epilogue triangular regions plus a steady state trapezoidal region. Furthermore, vectorization requires irregular accesses to the `B`, `G` and `U` arrays in our algorithm (Fig. 7). As a final optimization step, we reorder the `B`, `G` and `U` arrays so that accesses during GSOR are consecutive. Note that the working set of the algorithm fits comfortably into L2 caches for the problem sizes we consider.

An alternative approach would be to vectorize over different options, which would require that all three arrays `B`, `G` and `U` to be replicated, as in binomial tree, for example. This would increase the working set beyond cache capacity, require tiling (similar to binomial tree), and reduce the amount of thread-level parallelism available for a small number of options. Our approach — which exploits SIMD parallelism within options and thread-level parallelism (TLP) across options — has both a small working set and scales well even for small number of options.

*3) Performance:* Performance for the Crank-Nicolson method is shown in Fig 8 as American option pricing performance (in double precision) in thousands of options per

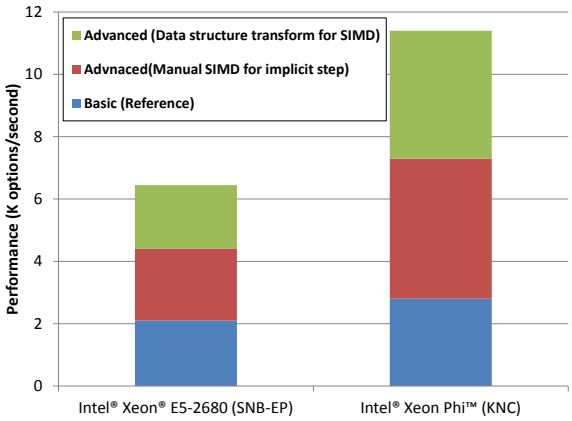



Fig. 8: Performance[5,6] of Crank-Nicolson American options pricing using 256 underlying prices and 1000 time steps.

second under various optimizations. The lowest bar for both SNB-EP and KNC shows the performance of the reference algorithm. Since most of the time is spent in GSOR code that is not vectorized, KNC is only 1.3x faster than SNB-EP.

With unrolling and vectorization, the performance improves to about 4.4K options/second for SNB-EP and 7.3K options/second for KNC. This speedup from SIMD is smaller than the theoretical peak for both architectures; this is because the data structures are irregularly accessed and need to be gathered from different cache lines. Once we perform a data structure transformations, performance increases to 6.4K options/second on SNB-EP and 11.4K options/second on KNC— the gain due to SIMD on the two architectures is about 3.1X and 4.1X respectively.

The remaining difference with respect to the peak of 4x and 8x SIMD scaling is due to the overhead of data structure transformations (the cost of physically rearranging the `B`, `G` and `U` arrays for contiguous access), as well as the fact that a small portion of the computation (the explicit step) was autovectorized in the reference code.

## V. Conclusion

In this work, we have analyzed the performance of two IA-based architectures, the Intel® Xeon® Processor E5-2680 (SNB-EP) and the forthcoming Intel® Xeon Phi™ coprocessor (KNC), on a small but representative set of compute kernels for derivative pricing in computational finance. We have investigate the Ninja performance gap between basic optimizations done by the compiler alone and more advanced algorithmic optimizations that result in peak-performance code.

On average, the Ninja gap is 1.9x for SNB-EP and 4x for KNC. Wide-issue, out of order cores of SNB-EP are more forgiving to extra instruction overhead compared to smaller and less aggressive KNC cores. Nevertheless, for three of the kernels — binomial tree, Monto Carlo option pricing, and Crank-Nicolson — the reference implementation on KNC achieves higher performance than on SNB-EP. For Black-Scholes, the key optimization which bridges the Ninja gap is converting input data from AOS, which is limited by gather overhead, to a SIMD-friendly SOA format. If the data is already provided in SOA format by the previous stage of computation, the compiler will generate SIMD-friendly code on KNC. Similarly, for Brownian Bridge the key benefit comes from outer loop vectorization across options. Once RNG is modified to load data in vector-width chunks, the compiler will produce SIMD-friendly code that achieves bandwidth-bound performance on KNC. The other two kernels, binomial tree and Crank-Nicolson, require algorithmic changes to achieve their best performance. Such changes may not be readily obvious to a programmer, but they demonstrate that highly optimized codes achieve close to the hardware potential on both architectures. In particular, the best-optimized code on KNC achieves on average 2.5x on compute bound kernels and 2x on bandwidth-bound kernels, which is commensurate with the compute and bandwidth differences between the architectures.

## References


[1] C. Martini and A. Zanette, "Premia: Overview version 9," Aug. 2012. [Online]. Available: https://www.rocq.inria.fr/mathfi/Premia/index.html

[2] "STAC benchmark domains," Aug. 2010. [Online]. Available: http://www.stacresearch.com/benchmarkdomains.pdf

[3] NVIDIA, "Computational finanace on the GPU," in *GPU Technology Conference*, Sept. 2009.

[4] V. Agarwal, L.-K. Liu, and D. Bader, "Financial modeling on the cell broadband engine," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 –12.

[5] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the Ninja performance gap for parallel computing applications?" in *Proceedings of the 39th International Symposium on Computer Architecture*, ser. ISCA '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 440–451.

[6] D. D. Kalamkar, J. D. Trzaskoz, S. Sridharan, M. Smelyanskiy, D. Kim, A. Manduca, Y. Shu, M. A. Bernstein, B. Kaul, and P. Dubey, "High performance non-uniform FFT on modern x86-based multi-core systems," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, may 2012, pp. 449 –460.

[7] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, may 2012, pp. 378 –389.

[8] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81, no. 3, pp. 637–654, 1973.

[9] J. Kerman, "Numerical Methods for Option Pricing: Binomial and Finite-difference Approximations," Master's thesis, Courant Institute of Mathematical Sciences, New York University, 2002.

[10] P. Wilmott, S. Howison, and J. Dewynne, *The Mathematics of Financial Derivatives: A Student Introduction*. Cambridge University Press, 1995.

[11] ——, *The Mathematics of Financial Derivatives: a student introduction*. Cambridge University Press, 2002.

[12] P. Glasserman, *Monte Carlo Methods in Financial Engineering*, ser. Applications of mathematics. New York: Springer Science, 2004, vol. 53.

[13] B. Malkiel, *A Random Walk Down Wall Street*. W. W. Norton & Company, Inc., 1973.

[14] J. McCalpin, "STREAM: Sustainable memory bandwidth in high performance computers," Aug. 2012. [Online]. Available: https://www.cs.virginia.edu/stream/

[15] Intel, "A quick, easy and reliable way to improve threaded performance," Aug. 2012. [Online]. Available: http://software.intel.com/en-us/articles/intel-cilk-plus/

[16] J. C. Hull, *Options, Futures, and Other Derivatives*, 6th ed. Upper Saddle River, N.J.: Prentice-Hall, 2006.

[17] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.