ABSTRACT

The STAC-A2™ benchmark is an emerging standard designed to evaluate the speed, scalability and quality of computational platforms for performing financial risk analytics in the capital markets industry. The problem posed by the benchmark is the computation of several types of Greeks for an exotic option under an American exercise model. We recently reported record-setting performance for a STAC-A2 benchmark solution developed for an IBM® POWER8® S824 server. We explain the high performance of our solution in terms of the architecture, scalability and high memory bandwidth provided by POWER8 based systems. Developing the benchmark application also led us to investigate and perfect several techniques that are generally applicable to the simulation of complex options and their sensitivities. We describe several of these techniques in detail, along with the performance impacts we observed when compared with other approaches. We focus on two areas in particular, namely cache-efficient data management for Monte Carlo simulation of American-exercise options, and a parallel implementation of the Longstaff-Schwartz algorithm.

Keywords

POWER8, STAC-A2, Heston model, Longstaff-Schwartz, Matrix transpose, Parallel SVD, OpenPOWER

1. INTRODUCTION

Pricing and sensitivity analysis of options are important problems in computational finance. The Securities Technology Analysis Center (STAC®), through the STAC Benchmark Council™, has chosen the computation of exotic option sensitivities as the basis of a benchmark to evaluate the performance, scalability and quality of platforms for computational finance. The STAC-A2™ benchmark has rapidly become an industry standard in this area, with leading financial firms looking to these results as they evaluate new hardware platforms. The STAC-A2 benchmark specification does not restrict the technologies used in the solution. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Therefore we only provide a brief operational overview

2. BENCHMARK OVERVIEW

The STAC-A2 benchmark, as well as other implementations, have been introduced in many references. Therefore we only provide a brief operational overview
Table 1: Key STAC-A2 benchmark workloads. Items marked * are specific to our solution.
industry-standard DRAM through a high-speed link to the processor. Each of the 16 Centaurs also includes 16 MB of level-4 cache, for a system-level total of 256MB of L4. POWER8 is specified to provide up to 192 GB/second of memory bandwidth per processor module, and others have reported achieving 85% of this figure with the STREAMS triad benchmark [6].

The system is virtualized as a single dedicated partition using the IBM PowerVM® hypervisor. We run an unmodified version of Red Hat® Enterprise Linux® 7. We disable the transparent hugepage support provided by recent Linux kernels in favor of large (64KB) pages. Although using huge (16MB) pages does improve warm run performance slightly, allocating and coalescing huge pages inordinately penalizes kernels in favor of large (64KB) pages. Although using hugepages is a common practice in POWER8 little-endian operation, and little-endian environment we find to provide significantly better performance than current open-source alternatives.

POWER8 based Linux systems offer a choice between traditional big-endian operation, and little-endian environments providing compatibility and simplified porting for applications developed on Intel X86 systems. Although the results reported here are for a big-endian environment, several little-endian Linux distributions are also now available for the S824 and other POWER8 based Linux servers.

4. PROGRAM STRUCTURE

STAC-A2 is a high-level specification expressed as an R-language reference model. Each benchmark participant develops a complete, unique and proprietary solution to the problem optimized for their particular hardware and software environment. Here we describe our approach to the problem. Because other solutions are largely undisclosed we can not provide detailed comparisons.

Figure 1 illustrates the conceptual flow of our application. We use the AES-acceleration instructions introduced in POWER8 to implement the high-quality ARS5 [22] cryptographic random number generator (RNG). The RNG process includes converting uniform variates to unit-normal variates and correlation between assets. Monte Carlo simulation is implemented directly from the reference [6], and the raw path data is post-processed to create \( O(assets^2) \) paths \( \times \) timesteps arrays of asset-basket values under various conditions. All arrays are generated simultaneously from a single set of random numbers. The amount of data is quadratic in the number of assets because Cross Gamma and Correlation Vega measure sensitivities to changes in price or correlation respectively between all pairs of assets (combination \( \binom{n}{2} \) is \( O(N^2) \)). Table 1 details the number of arrays and the total data storage required for each workload. Many of the arrays directly correspond to scenarios, while in other cases the scenarios are generated on-the-fly during pricing by combining arrays. Further details are beyond the scope of this paper, but are not required to understand the algorithms and performance results presented.

STAC-A2 performance benchmarks measure the latency of computing a large set of Greeks for a particular set of parameters, therefore parallel program structure and load balancing are key to competitive results. We implemented our multithreaded C++ application using direct calls to POSIX threads APIs in order to maximize the potential for experimentation and analysis. A high-level view of the application parallel structure is presented in Figure 2. In (a), the master thread \( T_0 \) performs initialization and then spawns threads \( T_1, \ldots, T_{n-1} \) which perform the Monte Carlo simulation in parallel (b). Each thread in our application simulates more-or-less equal numbers of paths for every array. Simulation data structures are carefully aligned to avoid false sharing, and the granularity of work is typically a set of 16 paths, where 16 is the number of double-precision values in the POWER architecture 128-byte wide cache line. Simulation work is scheduled statically, but threads finishing early can “steal” 16-path quanta of work from other threads.

We do not begin American-exercise pricing until all threads have joined after Monte Carlo simulation. Once joined the threads are typically more-or-less evenly grouped into cohorts of one or more threads; Figure 2(c) illustrates cohorts of 2 threads. Each cohort is fully responsible for pricing one or more scenarios using the Longstaff-Schwartz algorithm as discussed in §5.3. For example, the standalone Delta benchmark for 5 assets requires pricing 10 scenarios. On our 24-core system in SMT4 mode, the 96 threads are grouped into 10 cohorts of 9 or 10 threads and all scenarios are priced in parallel using a static schedule. Once the number of scenarios exceeds the number of available threads we typically treat each thread as a cohort of one. Load balancing is accomplished by having cohorts choose scenarios to price from a global pool. Once pricing completes the master thread computes the final Greeks (Figure 2(d)).

The scheme described above is modified when simulating large numbers of paths, where the storage required to simulate and price all of the scenarios exceeds physical memory. In this case the workload is statically partitioned into multiple phases, where each phase simulates and prices a subset of the scenarios. We will also explain in §6 why it is advantageous to price workloads with large numbers of paths using a single cohort consisting of all threads. We refer to this as the all-threads-cohort (ATC) mode, distinguishing it from the small-threads-cohort (STC) mode where smaller numbers of threads form the cohorts.
transposition requires little or no extra storage, the most cache-efficient algorithm known requires each array element to be copied at least four times during the permutation [15]. We need to transpose the data with minimal extra storage and data movement, while taking advantage of the fact that the path data is “hot” in the caches.

A straightforward copy of a path-pair $P$ into a column-major paths $\times$ timesteps array $A$ is disastrous for performance (Alg. 1). Since the data access stride is at least the number of paths, each column of $A$ is in a unique virtual page (stressing translation mechanisms), and cache lines of $A$ must be brought into the highest level caches multiple times before they are fully populated.

Alg. 1 Insert path-pair $P$ into array $A$ at path-index $p$.

```plaintext
for $t$ in $0, \ldots, \text{timesteps} - 1$ do
    $A_{p,t} \leftarrow P_{2t}; A_{p+1,t} \leftarrow P_{2t+1}$
end for
```

One solution is to use a time-linear buffer to store path data for a small set of paths, then copy the data into path-linear storage once the buffer is full. Recall from [9] that each thread is simultaneously generating paths for each row of every array to amortize the cost of random number generation. Since there are 16 double precision values in a POWER8 cache line, we can allocate an $8 \times (2 \times \text{timesteps})$ buffer for each thread for each array. Once 8 path-pairs have been inserted the copy is done such that each path-linear destination cache line is fully populated whenever touched (Alg. 2). In our system with 96 threads this requires storage for 1536 extra paths per array, which is a reasonable overhead.

Alg. 2 Insert path buffer $B$ into array $A$ at path-index $p$.

```plaintext
for $t$ in $0, \ldots, \text{timesteps} - 1$ do
    for $j$ in $0, \ldots, 7$ do
        $A_{p+2j,t} \leftarrow B_{j,2t}; A_{p+2j+1,t} \leftarrow B_{j,2t+1}$
    end for
end for
```

However in many cases a more efficient solution is to use a blocked, in-place matrix transposition. We pad the destination array if necessary and treat it as a $[\text{paths}/16] \times [\text{timesteps}/16]$ array of $16 \times 16$ blocks. Interleaved path-pairs are deinterleaved and inserted in a way that fully populates destination cache lines, but creates each block as the transpose of its final form (Alg. 3). Once all paths have been inserted a straightforward square matrix transpose of each block completes the operation. Although each datum is still copied twice, the source and destination cache line sets are identical here in the second copy, often making this approach more efficient than the buffered insertion.

Alg. 3 Blocked insertion of (padded) path-pair $P$ into array $A$ at path-index $p$, for block size $S$.

```plaintext
$k \leftarrow p \mod S; t \leftarrow 0$
for $b$ in $0, \ldots, [\text{timesteps}/S] - 1$ do
    for $i$ in $0, \ldots, S - 1$ do
        $A_{p+i,Sb+k} \leftarrow P_{2t}; A_{p+i,Sb+k+1} \leftarrow P_{2t+1}$
        $t \leftarrow t + 1$
    end for
end for
```
Figure 3 illustrates the operation of Alg. 3 for hypothetical \(4 \times 4\) blocks and 10 timesteps. In (A), data for interleaved paths \(a\) and \(b\), \(a_0, b_0, \ldots, a_9, b_9\) is deinterleaved and inserted into a row of blocks. In (B), data for interleaved paths \(c\) and \(d\) is inserted. Once all paths have been inserted, transposing the \(4 \times 4\) blocks brings the path data into the correct orientation (C). Note that this algorithm requires the symmetry of a square block. A similar technique has been suggested in the context of matrix transposition in GPU memories [23].

The transposition of the square blocks can either be done immediately after the last path of a set is inserted, or deferred until the arrays are fully populated. Immediately transposing a row of blocks suffers from large-stride issues, but could be advantageous if the data resides in the L3 cache. Deferring transposition largely eliminates the problems of large strides in this phase because blocks can be transposed block-column by block-column, following the natural layout of column-major memory (Alg. 4).

### Table 2: Relative performance of Monte Carlo simulation for different transpose methods

<table>
<thead>
<tr>
<th>Workload</th>
<th>WS, MB</th>
<th>Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>6.4</td>
<td>a 0.63</td>
</tr>
<tr>
<td>Large Problem</td>
<td>93.5</td>
<td>a 0.46</td>
</tr>
<tr>
<td>Asset Scaling</td>
<td>797</td>
<td>a 0.85</td>
</tr>
<tr>
<td>Path Scaling</td>
<td>6.4</td>
<td>a 0.63</td>
</tr>
</tbody>
</table>

5:126:25,000

5:1260:25,000

5:1260:25,000

3.2

32.0

0.75

0.97

1.07*

0.88

1.02

Table 2 compares the performance of Monte Carlo simulation by approximately 5% for the blocked algorithm. Referring to Figure 3 for blocked insertion, prior to inserting data for \(a_0, \ldots, b_3\), we issue prefetch instructions for the two cache lines that will be populated by \(a_4, \ldots, b_7\), and so on until the final block of a row. For buffered insertion and the block transpose we prefetch the 16 lines to be updated or transposed next.

Table 2 compares the performance of Monte Carlo simulation for the workloads defined in Table 3 and other parameterizations denoted as assets:timesteps:paths (warm runs). Buffered and blocked insertion performance includes optimized prefetch. The table also shows the SMT4 active working set (WS) of a POWER8 core during the simulation, i.e., the storage required to hold 16 paths for every array generated by each of the 4 software threads.

Any technique other than the simple insertion reduces the overhead of transposition such that the final differences between the best performing methods are small, but generally repeatable. We prefer the blocked insertion due to its overall performance and simplicity. Although we have not done exhaustive studies, our data suggests choosing whether or not to defer the final block transpose based on whether the per-core working set appears to fit in the 8MB L3 cache. (Since we don’t control placement of the numerous arrays involved we can’t be sure the working set is actually fully resident). This heuristic is easy to compute from the workload parameters. Although blocked insertion has no memory overhead and provides the best performance for many workloads, there are also cases where the buffered copy performs best and we continue to investigate this area.

### 5.3 Longstaff-Schwartz Pricing

Longstaff and Schwartz published their least-squares Monte Carlo (LSMC) approach to valuing early-exercise options in 2001 [18]. The original reference is a well-written description of this popular numerical algorithm and its mathematics. Next we describe an implementation of the Longstaff-Schwartz algorithm that was motivated by the STAC-A2 benchmark problem but can be generalized to other situations. We speculate NVIDIA used a similar approach [11].

#### 5.3.1 Least Squares Fitting via SVD

The challenge in valuing an early-exercise option is that the holder must continuously decide whether to exercise the option immediately, or in the future. On the exercise date the option holder exercises the option if it is in the money. At a time prior to expiration the holder will exercise an in-the-money option if the future discounted cash flow from holding the option is expected to be less than the current value, and the value of the option is maximized if the exercise happens as soon as this is true. LSMC estimates the future value by least squares regression using a cross section of simulated data, approximating this value as a linear combination of basis functions.

We assume a timestep \(t\) (other than the final timestep). Let \(n\) be the number of basis functions, and \(A\) be the \(\text{paths} \times n\) design matrix where each \(A_{ij}\) is the application of basis function \(j\) to the underlying value on path \(i\) at time \(t\). Let \(b\) be a column vector where each \(b_i\) is the future discounted cash flow along path \(i\). \(A\) and \(b\) are restricted to only consider paths where the option is in-the-money at time \(t\). The regression problem is to find a set of coefficients \(x\) that minimizes \(\|Ax - b\|_2\). Once the coefficients are known the estimated continuation value is computed from the current path value and the basis functions.

Compared to the pseudo-inverse method, the singular value decomposition (SVD) is preferred to implement the LSMC regression due to its superior numerical stability [5]. Let the SVD of \(A\) be \(U\Sigma V^T\), where \(\Sigma\) is diagonal and both \(U\) and \(V\) are unitary. The solution to the regression problem is then \(x = V\Sigma^{-1}U^T b\), where \(\Sigma^{-1}\) is the inverse of \(\Sigma\).

Computing the coefficients \(x\) at time \(t\) requires the future value \(b\) computed at time \(t + 1\) and so on. Thus LSMC is inherently a serial algorithm operating backwards in time. One way of partially parallelizing the algorithm is to notice that the SVD of \(A\) can be computed in advance at every timestep, since the SVD only depends on the path values.
which are all available at the end of Monte Carlo simulation [3]. Finishing the regression can be deferred until the future value is known. However there are issues with scaling this technique. If a thread computes an SVD in advance it must either wait for the future value to be available before continuing, or store the SVD until that thread or another thread can complete the regression. Load balancing is also an issue since any number of threads can be computing SVDs for a number of timesteps in parallel, but the final regression is still ultimately serial in time.

5.3.2 Parallel SVD of Vandermonde Matrices

The computation of the SVD for “tall and skinny” matrices $A_{m \times n}$ ($m \gg n$) is usually by means of the QR decomposition, as in:

$$A_{m \times n} = Q_{m \times n}R_{n \times n}$$

where $Q$ is unitary and $R$ is upper triangular. The QR decomposition is usually handled by Householder reflection or Givens rotation [14]. Since $R = U^T \Sigma V^T$ is the SVD of $R$ then $A = (QU)\Sigma V^T$ is the SVD of $A$.

LSCM has been found to be robust in the choice of basis functions, and the STAC-A2 specification requires using polynomial basis functions. If $p_i$ is the current price on path $i$, then each row of $A$ has the form of:

$$1 \quad p_i \quad p_i^2 \quad \ldots \quad p_i^{n-1}$$

hence $A$ is a row Vandermonde matrix. Although the QR decomposition is parallelizable in general [10], for this type of Vandermonde matrix there is a lesser-known QR factorization method which has excellent data locality and a simple parallel implementation [9]. A derivative of the Lanczos method can be applied to perform the QR factorization, such that $A = Q\Sigma B^T$, where $Q_{m \times n}$ is unitary, $\Sigma_{n \times n}$ is diagonal and positive definite, and $B_{n \times n}$ is lower triangular (hence $B^T$ is upper triangular) with values of one (1.0) at diagonal entries. The technical details of the method can be found in [9]. However we want to point out that the algorithm outlined in the original paper has a few typographical errors. The correct algorithm is outlined in Alg. 5.

In practice we implement Alg. 5 in parallel in two phases. The first phase of this QR decomposition method is the calculation of the “moments” for the in-the-money paths

$$M_j = \sum_{i=0}^{\text{paths}-1} p_i^{j-1} \quad \text{for} \quad j = 1, \ldots, 2n-1$$

used to generate $B$. Each thread first computes and stores partial moments for a set of paths, and then computes the final sums once all partial sums are available. For the benchmark case $B^T$ is a small matrix whose SVD can then be computed in a few microseconds. Since each $Q_{ij}$ only depends on $p$, and the moments, the parallel generation of $Q$ can be deferred until computing the regression coefficients.

Consider the case where the design matrix $A$ is a paths $\times n$ array and the paths are partitioned into sets 1, \ldots, N for $N$ threads. If $B^T = U\Sigma V^T$ is the SVD of $B^T$, it is easy to show that the least squares solutions with given vector $b$ can be computed as:

$$x^T = b^T QU \Sigma^{-1} \Sigma_2^{-1} V^T = b_{\text{paths}} W_{\text{paths} \times n}$$

Note that $(U\Sigma^{-1} \Sigma_2^{-1} V^T)_{n \times n}$ is a small constant matrix that is easily multiplied into $Q_{\text{paths} \times n}$ as $Q$ is generated during a second pass over the path data. Partitioning $b^T$ and $W$

$$x^T = [b^T(1) \ b^T(2) \ \ldots \ b^T(N)] \times \begin{bmatrix} W(1) & W(1)_2 & \ldots & W(1)_n \\ W(2) & W(2)_2 & \ldots & W(2)_n \\ \vdots & \vdots & \ddots & \vdots \\ W(N) & W(N)_2 & \ldots & W(N)_n \end{bmatrix}$$

shows that each of the coefficients $x^T_i$ can be computed as the simple sum of partial coefficients $x^T_i = \sum_{j=1}^{N} b^T(j) W(j)$, where each thread $j$ computes $n$ independent product terms.

The result is the parallel Longstaff-Schwartz algorithm for call options using simple polynomial basis functions of degree $n-1$ sketched as Alg. 6. Given the risk-free rate $r$, time to expiration $T$ and strike price $K$, a partition $p$ of price data $P$ at each timestep is used to compute a partition $f$ of the future value $F$. Once the algorithm terminates the scenario value is simply the average cash flow at time 0, $\|F\|/\text{paths}$. The amount of data exchanged between threads is trivial for reasonable values of $n$: $n+2$ partial moments and $n$ partial coefficients per thread. We actually compute $s$ by row, and $p$ and $f$ are not computed as separate vectors. Instead we use index vectors and other techniques to partition $p$ and $f$ between in- and out-of-the-money paths in place, giving the implementation the character of a sparse vector algorithm.

5.3.3 Discussion

We originally implemented Longstaff-Schwartz in parallel following [9], but found that performance for pricing a scenario did not improve for cohorts of more than two threads!
Alg. 6 Parallel Longstaff-Schwartz (call option) for polynomial basis functions, executed by each thread.

\[
\begin{align*}
    d & \leftarrow e^{-r T / \text{timesteps}} \quad \triangleright \text{Per-timestep discount} \\
    \text{for } t \text{ in } \text{timesteps} - 1, \ldots, 0 \text{ do} \\
    \quad p & \leftarrow P_{\text{firstpath: lastpath}, t} \quad \triangleright \text{Price vector at time } t \\
    \quad \text{if } t = \text{timesteps} - 1 \text{ then} \\
    \quad \quad f & \leftarrow d \times \max(p - K, 0) \quad \triangleright \text{Initial future value} \\
    \quad \quad \text{else} \\
    \quad \quad \quad \hat{p} & \leftarrow p \quad \triangleright \text{In-the-money paths} \\
    \quad \quad \quad f & \leftarrow f \times p > K \quad \triangleright \text{and future value} \\
    \quad \quad \quad M_j & \leftarrow |\hat{p}|^{j-1}, \text{for } j = 1, \ldots, 2n - 1 \quad \triangleright \text{Moments} \\
    \quad \quad \text{Thread Join; Exchange partial moments} \\
    \quad \quad \text{SVD and QR using } M \text{ and } \hat{p} \text{ producing } W \\
    \quad \quad x_i^T & \leftarrow f \times W, \text{for } i = 1, \ldots, n \triangleright \text{Partial coefficients} \\
    \quad \quad \text{Thread Join; Exchange partial coefficients} \\
    \quad \quad \hat{s} & \leftarrow [\hat{p} \hat{p}^2 \cdots \hat{p}^{n-1}] \times x \quad \triangleright \text{Least-squares fit} \\
    \quad \quad f & \leftarrow d \times \begin{cases} p - K \text{ for } p - K > \hat{s} & \triangleright \text{New fut. val.} \\
    f & \text{otherwise} \end{cases} \\
\end{align*}
\]

end for

In contrast, for the Baseline case (25K paths) Alg. 6 performance continues to improve up to 16 or 20 SMT4 threads per cohort, at which point synchronization overheads become excessive. Thus our approach is very effective for problems with fewer scenarios to price than available threads.

Once the number of paths exceeds about 500K very significant performance advantages begin to accrue from pricing all scenarios serially using a cohort of all threads (ATC mode), primarily due to eliminating the imbalance of the variations in the times required to price individual scenarios. ATC mode naturally load balances scenario pricing since the threads move together in lock step through each timestep in every scenario, and statistically do equal amounts of work if the paths are partitioned equally. ATC mode also benefits from better cache utilization and ideal NUMA locality. The more threads per cohort, the smaller the per-thread partitions \(p\) and \(f\), increasing the chances that data remains L3 cache-resident. In ATC mode a thread prices the same portion of path data that it created during simulation. Thus the majority of data will be held in memory allocated “close” to each core-pinned thread, in contrast to single-thread cohorts where each thread processes a cross section of data created by every other thread.

ATC mode improves the end-to-end performance of the Path Scaling workload by over 20% when simulating 28M paths. As future work we would like to reduce the synchronization overhead such that we could gain the benefits of ATC mode for the Large Problem (100K paths), whose size is more representative of real-world problems.

6. PERFORMANCE ANALYSIS

6.1 Core- and System-Level Scalability

The STAC-A2 benchmark includes a scaling experiment, and the audited results for the POWER8 S824 are summarized in Table 3 (data courtesy STAC). The table lists relative speedups observed when scaling the Baseline problem from a single thread on a single core to all threads on 24 cores. SMT4 mode provides an approximate 2X speedup over SMT1 at each scale. Performance rolls off in SMT8 mode, which is not unusual for an HPC application.

System-level scalability is an important factor in the end-to-end performance of parallel applications like STAC-A2. Table 3 also shows that this POWER8 based system achieves (44.3/2.0)/24 = 92% of the theoretical scalability when scaling the workload from 1 to 24 SMT4 cores. The scalability of POWER8 stands in contrast to a recent STAC-A2 result for a 2-socket Intel Xeon E5-2699 v3 based system that showed only 73% of theoretical scalability for the same workload when scaling from 1 to 36 cores with Intel Hyper-Threading Technology enabled.

6.2 Impact of Memory bandwidth

When the S824 benchmark audit set a new record for Path Scaling by a wide margin, we were interested if we could explain this result in terms of memory bandwidth. The POWER8 processor in the S824 system has eight memory channels per socket, each supporting a single CDIMM. The number of populated memory channels has a linear impact on available memory bandwidth regardless of CDIMM size. We created full (1.0, 1TB), half (0.5, 512GB) and quarter (0.25, 256GB) bandwidth configurations by populating 8, 4 and 2 slots per socket respectively with 64GB CDIMMs to test this hypothesis. (Note full bandwidth is also available starting at 256GB with smaller CDIMMs.)

Figure 4 lists the performance of different runs, each at three different levels of available memory bandwidth normalized to the run with maximum memory bandwidth. The Baseline and Large Problem workloads are standard. For asset scaling we use 65 assets. For path scaling we look at 1M, 8M and 16M paths, for both the small-threads-cohort (STC) and the all-threads-cohort (ATC) modes. Scaling beyond 8M paths or 65 assets requires more than 256GB of memory and so does not provide comparison points for lower levels of memory capacity/bandwidth. The 16M paths can be run with 512GB of memory but not with 256GB.

The Asset Scaling problem is purely compute bound and available memory bandwidth appears to have negligible impact on performance. This workload is completely domi-
nated by the cache-contained $O(assets^2)$ triangular matrix multiplication used to correlate random numbers, and the $O(assets^2)$ Monte Carlo simulation process.

As the number of paths increases from 25K (Baseline) to 100K (Large) to millions (Path Scaling), there is increasing impact on performance from reducing memory bandwidth.

Since the parallel Longstaff-Schwartz algorithm requires three passes over the price data for each timestep, memory bandwidth is critical as cache capacities are exceeded with larger number of paths. The ATC version significantly reduces the cache pressure by parallelizing each scenario across all the threads, lowering the cache requirement per thread. Consequently, it is less memory bound and sees smaller performance reduction for given bandwidth reduction compared to the STC version. Even so, memory bandwidth remains very important to top performance on the Path Scaling workload.

The POWER8 S824 currently still holds the record in the STAC-A2 Path Scaling benchmark, in part because of its high bandwidth. The only comparable reported throughput on this benchmark is for a 4-socket Intel Xeon E7-8890 v3 based server which has system memory bandwidth roughly equivalent to the S824 [5] - however, it requires 2X the number of processors (3X the number of cores) as the POWER8 based system for the similar result.

7. CONCLUSIONS

STAC-A2 is a well-rounded HPC benchmark that stresses a system at scale. It includes dense matrix and sparse vector algorithms, CPU-bound Monte Carlo simulation and memory-intensive American-exercise pricing. In this paper, we discuss different algorithmic optimization and demonstrate their benefits on the IBM POWER8 S824 platform. We also discuss different system characteristics of the platform and their importance to various aspects of the benchmark. Our results demonstrate that POWER8 systems are highly competitive platforms for computational finance.

IBM, NVIDIA and other major technology companies recently announced the formation of the OpenPOWER™ foundation, an organization dedicated to system designs centered on the POWER microprocessor. OpenPOWER systems include unique support for heterogeneous computing including the Coherent Attached Processor Interface (CAPI) and the NVIDIA NVLINK™ interconnect. We expect even more interesting optimization opportunities for heterogeneous financial workloads with future OpenPOWER systems.

8. ACKNOWLEDGMENTS

We would like to thank Kenneth Hill of the University of Florida and Julien Demouth of NVIDIA for their technical assistance and insights.

9. REFERENCES


All URLs listed above were valid as of October 15, 2015.