A HIGH-PERFORMANCE BROWNIAN BRIDGE FOR GPUS: LESSONS FOR BANDWIDTH BOUND APPLICATIONS

JACQUES DU TOIT

Abstract. We present a very flexible Brownian bridge generator together with a GPU implementation which achieves close to peak performance on an NVIDIA C2050. The performance is compared with an OpenMP implementation run on several high performance x86-64 systems. The GPU shows a performance gain of at least 10x. Full comparative results are given in Section 8: in particular, we observe that the Brownian bridge algorithm does not scale well on multicore CPUs since it is memory bandwidth bound. The evolution of the GPU algorithm is discussed. Achieving peak performance required challenging the "conventional wisdom" regarding GPU programming, in particular the importance of occupancy, the speed of shared memory and the impact of branching.

Contents

1. Introduction and Software Requirements

The Brownian bridge algorithm (see e.g. [2]) is a popular method for constructing sample paths of a Brownian motion. The procedure may be summarised as follows. Fix two times $t_0 < T$ and let $X = (X_t)_{t_0 \le t \le T}$ denote a Brownian motion on the interval $[t_0, T]$. Let $(t_i)_{1 \le i \le N}$ be any set of time points satisfying $t_0 < t_1 < \ldots < t_N < T$ for some $N \ge 1$. Our aim is to simulate values for $\{X_{t_i}\}_{1 \le i \le N}$ and X_T by using a set of standard Normal random numbers $Z_0, Z_1, \ldots, Z_N$. We assume that the value $X_{t_0} = x$ is always known (often $x = 0$), and we always set $X_T = x + \sqrt{T - t_0} Z_0$. The Brownian bridge algorithm then uses interpolation to fill in the remaining values $\{X_{t_i}\}_{1 \le i \le N}$. Given any two points X_{t_i} and X_{t_k} which are known, a third point X_{t_j} for $t_i < t_j < t_k$ can be computed as

$$X_{t_j} = \frac{X_{t_i}(t_k - t_j) + X_{t_k}(t_j - t_i)}{t_k - t_i} + Z_j \sqrt{\frac{(t_k - t_j)(t_j - t_i)}{t_k - t_i}}. \tag{1}$$

The algorithm is therefore iterative. Given the known starting value $X_{t_0} = x$ and the final value $X_T = x + \sqrt{T - t_0} Z_0$, a third point X_{t_i} for any $1 \le i \le N$ can be computed. Given the three points X_{t_0}, X_T, X_{t_i} a fourth point X_{t_j} for any $j \ne i$ can be computed

by interpolating between its nearest neighbours. The process continues until all the points have been generated.

If the Brownian motion is multidimensional, the algorithm can still be used with minor changes. Each X_{t_i} and Z_i becomes a vector and correlation is introduced by setting

$$X_{t_j} = \frac{X_{t_i}(t_k - t_j) + X_{t_k}(t_j - t_i)}{t_k - t_i} + CZ_j\sqrt{\frac{(t_k - t_j)(t_j - t_i)}{t_k - t_i}}. \tag{2}$$

where C is a matrix such that CC' gives the desired covariance structure of the Brownian motion. When the Brownian bridge is used to solve stochastic differential equations, it is more appropriate to produce scaled increments of the form $(X_{t_{i+1}} - X_{t_i})/(t_{i+1} - t_i)$. It turns out that this is somewhat easier than producing the Brownian sample path points X_{t_i}. We will not discuss scaled increments further, however timings for the increments generators are giving in Section 8.

1.1. **Bridge Construction Orders.** The Brownian bridge algorithm is not fully specified until we state which points X_{t_j} are interpolated from which points X_{t_i} and X_{t_k}. For example, with $N = 12$ and a set of time points $\{t_i\}_{1\le i\le 12}$ we could construct a bridge in the order

$$T \quad t_6 \quad t_3 \quad t_9 \quad t_1 \quad t_4 \quad t_7 \quad t_{11} \quad t_2 \quad t_5 \quad t_8 \quad t_{10} \quad t_{12} \tag{3}$$

meaning that X_{t_6} is interpolated between X_{t_0} and X_T; X_{t_3} is interpolated between X_{t_0} and X_{t_6}; X_{t_9} is interpolated between X_{t_6} and X_T; X_{t_1} is interpolated between X_{t_0} and X_{t_3}; X_{t_4} is interpolated between X_{t_3} and X_{t_6}; and so on. However we could equally construct the bridge in the order

$$T \quad t_2 \quad t_4 \quad t_3 \quad t_9 \quad t_1 \quad t_7 \quad t_{12} \quad t_5 \quad t_{10} \quad t_6 \quad t_{11} \quad t_8 \tag{4}$$

where now X_{t_2} is interpolated between X_{t_0} and X_T; X_{t_4} is interpolated between X_{t_2} and X_T; X_{t_3} is interpolated between X_{t_2} and X_{t_4}; X_{t_9} is interpolated between X_{t_4} and X_T; X_{t_1} is interpolated between X_{t_0} and X_{t_2}; and so on. Both construction orders are equally valid. Indeed, any permutation of the times $\{t_i\}_{1\le i\le N}$ will specify a valid bridge construction order. If $\Theta \equiv \{\theta_i\}_{1\le i\le N}$ denotes a permutation of the set $\{t_i\}_{1\le i\le N}$, then for any $\theta_i \in \Theta$ we will have

$$X_{\theta_i} = \frac{X_\ell(r - \theta_i) + X_r(\theta_i - \ell)}{r - \ell} + Z_i\sqrt{\frac{(r - \theta_i)(\theta_i - \ell)}{r - \ell}} \tag{5}$$

where $\ell = \max\{t_0, \theta_j \mid 1 \le j < i,\ \theta_j < \theta_i\}$ is the greatest "known" point smaller than θ_i and $r = \min\{T, \theta_j \mid 1 \le j < i,\ \theta_j > \theta_i\}$ is the smallest "known" point greater than θ_i. Here we mean that a time point s is "known" if the corresponding value X_s has already been computed (and is therefore known) by the time we come to computing X_{θ_i}. We are simply ensuring that when we interpolate X_{θ_i}, we interpolate between its nearest known neighbours. For example in (4) above, we interpolate X_{t_4} between X_{t_2} and X_T and not between X_{t_0} and X_T.

1.2. **Quasi-Random Numbers.** When the Z_i s are drawn from a pseudorandom generator, there is no theoretical reason to prefer one bridge construction order over another since each Z_i is independent from, and identical to, every other Z_i. However the Brownian bridge algorithm is frequently used with *quasi-random* numbers generated from low discrepancy sequences (such as Sobol sequences), and in this case the situation is very different. We refer to [2] for a more detailed discussion about why one would use quasi-random points with a Brownian bridge algorithm, but essentially the idea is that one

"covers" the space of Brownian sample paths more evenly than one would with pseudo-random points. The advantages are exactly analogous to using quasi-random points in a Monte Carlo integration of an $N+1$ dimensional function.

A single $N+1$ dimensional quasi-random point $(Z_0, Z_1, \ldots, Z_N)$ is used to construct an entire sample path. The problem is that, for most quasi-random generators, the lower dimensions $(Z_0, Z_1, \ldots)$ typically display much better uniformity properties than the higher dimensions $(\ldots, Z_{N-1}, Z_N)$. The lower dimensions are therefore more "valuable" and should be used to construct the most important parts of the Brownian motion. For example, if we consider a model which is particularly sensitive to the behaviour of the Brownian motion at time η, then we would ensure that

- time η was one of the interpolation points,
- X_η was constructed using a Z_i from the lower dimensions,
- X_η was interpolated between points which were themselves constructed using Z_i s from the lower dimensions.

This idea maps quite naturally to the bridge construction orders as depicted in (3) and (4) above. If we specify a bridge construction order through a permutation $\Theta \equiv \{\theta_i\}_{1 \le i \le N}$ of the times $\{t_i\}_{1 \le i \le N}$, and we ensure that the most important time points are given by $\theta_1, \theta_2, \ldots$, then we can use Z_0 to construct X_T, use Z_1 to construct X_{θ_1}, use Z_2 to construct X_{θ_2}, and so on. The construction order Θ maps directly onto the dimensions of the quasi-random point so that it is clear which dimension will be used to construct each point X_{θ_i}. For ease of notation we will set $\theta_0 \equiv T$ so that Z_i is used to construct X_{θ_i} for each $0 \le i \le N$.

1.3. **Memory Bandwidth Bound Algorithm.** The multipliers $(t_k - t_j)/(t_k - t_i)$, $(t_j - t_i)/(t_k - t_i)$ and $\sqrt{(t_k - t_j)(t_j - t_i)/(t_k - t_i)}$ in (1) above can be precomputed: there are only N of them, and they do not change once the construction order is fixed. These values can be taken as known when considering (1), they simply have to be fetched from memory. The Brownian bridge is therefore a *memory bandwidth bound application*: there is very little computation relative to the amount of data transferred. Our aim therefore was to achieve peak memory bandwidth (or as near to it as possible) as measured when considering only the movement of Z_i s from main memory and the movement of X_{t_i} s to main memory. Additional data movement (if any) would not be taken into account so that the metric would reflect the ideal world (from a user's point of view) where any extra data traffic could somehow be made to disappear. Users were to have complete freedom to specify any bridge construction order. Helper routines would be provided so that a user wouldn't have to manufacture a full construction order themselves. The algorithms would be implemented on traditional x86-64 architectures and on NVIDIA GPUs using CUDA. Our discussion will focus on the GPU implementation: comparative results with the x86-64 multicore implementations are given in Section 8.

2. Algorithm Design

The fact that a user can specify any bridge construction order means that it is essential to abstract this away, thereby reducing all construction orders to a common format. To achieve this, a two step design was adopted. In the first (called *initialisation*), the user passes in the bridge construction order. This order is then processed and reduced to an execution strategy which is copied to the GPU. As long as the bridge construction order and the time points remain unchanged, the execution strategy remains valid. In the second step (called *generation*), the Brownian sample paths are generated from a set of input Normal random numbers (either pseudorandom or quasi-random). The generation step can be executed several times to generate successive sets of Brownian sample paths.

2.1. **Local Stack.** An efficient implementation of a Brownian bridge algorithm typically requires a local workspace array, which we call the *local stack*. Since previously computed Brownian points X_{θ_i} are used to interpolate new points, it is clear that computed points should be kept as close to the processing units as possible, preferably in hardware cache. Since the hardware cache is too small to hold all the Brownian points, at each time step the algorithm must decide which points to keep and these are stored in the local stack. The algorithm must also determine when a point in the local stack is no longer required, so that it can be replaced by a new point. The idea is that, if the local stack does not grow too large, there is a good chance it will fit in L1 or L2 cache.

A key requirement of the bridge algorithm therefore is that it ensures that the local stack stays small.

2.2. **Initialisation and the Execution Strategy.** It turns out that it is possible to permute a given bridge construction order without changing the actual Brownian points that are computed. To illustrate, consider (3) above. This construction order is equivalent to the following construction order

$$T \quad t_6 \quad t_9 \quad t_3 \quad t_{11} \quad t_7 \quad t_4 \quad t_1 \quad t_{12} \quad t_{10} \quad t_8 \quad t_5 \quad t_2 \tag{6}$$

since X_{t_6} is interpolated between X_{t_0} and X_T ; X_{t_9} is interpolated between X_{t_6} and X_T ; X_{t_3} is interpolated between X_{t_0} and X_6 ; $X_{t_{11}}$ is interpolated between X_{t_9} and X_T ; and so on. The output from this construction order will be identical to the output from (3) as long as the same Z_i s are used to create each bridge point. Therefore in (6) we would use Z_0 to generate X_T ; Z_1 to generate X_{t_6} ; Z_3 to generate X_{t_9} ; Z_2 to generate X_{t_3} ; Z_7 to generate $X_{t_{11}}$; and so on.

It is easy for an arbitrary bridge construction order to use a lot of local stack. The task of the initialisation step is therefore to take the user's construction order and to find an equivalent construction order (along with a permutation of the Z_i s) which uses a minimal amount of stack. This procedure is rather technical and won't be discussed further, but the output is an *execution strategy* which is passed to the generate step. The execution strategy consists of a new bridge construction order and a permutation of the Z_i s.

2.3. **Conventional Wisdom: Occupancy, Shared Memory and Branching.** The conventional wisdom regarding GPU programming is that occupancy is important for memory bandwidth bound applications, that shared memory is fast, and that branching should be avoided. By branching here we do not mean warp divergence: we mean traditional serial branching. On a GPU this is equivalent to an if-then-else statement where all the threads in a warp take the same branch. Avoiding branches, especially in inner loops, is a standard optimisation strategy for CPU code.

To increase occupancy, a kernel should use as few registers as possible and the kernel should be launched with as many threads per streaming multiprocessor (SM) as possible. The idea is that the memory requests from all these threads will saturate the memory bus, and while data is being fetched for some threads (and they can therefore do nothing while waiting for the data to arrive), other threads whose data has arrived can continue executing.

In order to interpolate any Brownian point X_{θ_i} , the following steps have to be carried out:

(a) Determine the left and right neighbours of X_{θ_i} and find their locations in the local stack. These are the points X_ℓ and X_r in (5) above between which X_{θ_i} is interpolated.
(b) Read the left and right neighbours off the local stack.
(c) Determine which random point Z_i to use.

(d) Read the Z_i point from global memory.
(e) Compute X_{θ_i} using (5).
(f) Determine where to store X_{θ_i} in main memory. Clearly points should be stored in the correct order, namely $X_{t_1}, X_{t_2}, \ldots, X_T$.
(g) Store X_{θ_i} to main memory.
(h) Determine where to store X_{θ_i} in the local stack.
(i) Store X_{θ_i} in the local stack.

Steps (a), (c), (f) and (h) are all done during the initialisation step and together constitute the execution strategy. Physically the execution strategy consists of an array of integers which are copied to the GPU and read off by each CUDA thread as it generates a sample path. At each time step a thread would need to read 5 integers in order to compute a new Brownian point: indexes of left and right neighbours in the local stack; index of point Z_i ; storage index of X_{θ_i} in global memory; storage index of X_{θ_i} in the local stack. Given this information, the generate step then simply consists of

(a) Read the left and right neighbours off the local stack.
(b) Read the Z_i point from global memory.
(c) Compute X_{θ_i} using (5).
(d) Store X_{θ_i} to main memory.
(e) Store X_{θ_i} in the local stack.

This is a branchless process and therefore should be very efficient. In addition, the local stack can be stored in shared memory which is very fast. There is no warp divergence (each thread generates a separate Brownian sample path – threads in a warp are doing the same thing at the same time), all accesses to global memory are aligned and fully coalesced, and there are no bank conflicts when accessing shared memory. All in all, the algorithm seems ideally suited the GPU and should perform very well.

2.4. **Test System and Test Problem.** The test system consists of an Intel Core i7 860 running at 2.8GHz with 8GB RAM and a Tesla C2050 with Error Checking and Correction (ECC) on. The system runs 64 bit Linux with CUDA Toolkit v4.0. The basic test problem consists of generating 1,439,744 one dimensional Brownian sample paths each with 64 time steps, and the construction order is a standard bisection order such as that given by (3). All computations are carried out in *single precision*: it is harder to saturate the memory bus with 4 byte transfers than with 8 byte transfers. In total then the test problem consists of moving 351MB of data (the Z_i s) onto the compute cores, and then moving 351MB of data (the X_{t_i} s) back out to global memory. All performance measurements were obtained through NVIDIA's Compute Visual Profiler with all performance counters enabled. If an algorithm introduced local memory bus traffic (values in L1 cache spilling to L2 cache or to global memory), this was noted. Only the generate step was timed - the initialisation step was ignored.

The peak memory bandwidth of the C2050 is given as 144GB/s. However this is the figure with ECC turned off. When ECC is turned on, the peak bandwidth of the card is reduced to slightly less than 120GB/s (see [1]). The target is therefore to achieve as close to 120GB/s overall transfer rate as possible.

All performance figures quoted below are for the optimal launch configuration that was found, as measured by kernel execution time. Occupancy figures are for that launch configuration.

3. First GPU Strategy

Recall that the execution strategy consists of an array of integers which contain indexes into the local stack, the array of Z_i s and the bridge storage (output) array. Since these

integers are fixed for the duration of the generate step, and each thread in a warp will access the same element at the same time (broadcast access), it seems obvious to store them in constant memory. The first GPU implementation was therefore as follows.

To conserve space, the execution strategy used 8 bit unsigned integers to store the read and write indexes into the local stack and 16 bit unsigned integers to store the Z_i read index and the X_{θ_i} write index. The total size of the execution strategy for each bridge point was therefore 2×16 bit $+$ 3×8 bit for a total cost of 7 bytes. These values were read from constant memory through 16 bit and 8 bit broadcasts.

The stack was held in local memory and L1 caching was turned off, so that the stack physically resided in the L1 cache. Since the same physical hardware is used for both L1 cache and shared memory, this should be equivalent to placing the stack in shared memory.

3.1. **Parallelisation and Multidimensional Brownian Motions.** When the Brownian motion is one dimensional, it is clear that each CUDA thread can create a Brownian sample path independently of every other CUDA thread. In this case one would have each thread create several Brownian paths, so that each thread block does a reasonable amount of work. However if the Brownian motion is multidimensional we must compute the matrix-vector product CZ_i. The way this was implemented was to put C in constant memory and have threads read the values of each Z_i vector into shared memory. Threads would then synchronise and each thread would compute its row of the matrix-vector product CZ_i. Accesses to C followed a broadcast access pattern. The bridge point X_{θ_i} could then be computed as before, independently of all other CUDA threads. This approach meant that two `__syncthreads()` calls had to be issued at each time step.

3.2. **Performance.** The kernel used 20 registers, had an occupancy of 100% and a performance of 29GB/s.

4. Second GPU Strategy

Frequently heard advice for bandwidth bound applications is to use vector data types so that each thread transfers and processes more data. We adjusted the algorithm from Section 3 to allow each thread to read the Z_i s and write the X_{θ_i} s as either `float2`, `float3` or `float4`. Now each thread would process 2, 3 or 4 Brownian sample paths at once, so that the amount of local stack required would increase by a factor of 2, 3 or 4. Apart from that, the algorithm was unchanged: the execution strategy was read from constant memory, there were two synchronisations per time step, and the matrix C was held in constant memory.

4.1. **Performance.** The kernels used between 20 and 34 registers, had occupancy ranging between 100% and 54%, and generally gave poor performance. Although global memory traffic increased, this was due to the much larger local stacks which spilled to L2 cache. The stacks were held in local memory, and thus resided in L1 cache. To increase occupancy, each thread block was launched with many threads. However as there were no restrictions on registers or shared memory which would force the runtime to only place one or two blocks per SM, the runtime placed several blocks on each SM, exhausting the L1 cache and causing the stacks to spill to L2 and global memory. Running the kernel with fewer blocks and fewer threads alleviated the spilling, but lead to worse performance since there were now fewer transactions to global memory for Z_i s and X_{θ_i} s.

In all, each kernel ran slower than the kernel from Section 3, meaning the effective performance was less than 29GB/s.

5. Third GPU Strategy

Since the normal running of the algorithm could not generate enough memory instructions to saturate the memory bus, we introduced *explicit data prefetching.* This idea exploits the fact that the GPU compute cores do not stall on a data load instruction: they only stall once the register used for the load (i.e. the register into which the data from global memory is loaded) becomes the argument of an operation. Therefore if one thread issues several load operations on several different registers, and a programmer takes care not to use those registers until later in the program, then the loads will happen asynchronously while that thread carries on executing.

We changed the algorithm from Section 3 so that each thread prefetched P Normal random numbers before starting a path calculation. Then when each point X_{θ_i} in the path is calculated, the corresponding Normal number is used and a new Normal random number corresponding to $X_{\theta_{i+P}}$ is loaded to the same register. This way, P Normal fetches are always in flight. With this strategy it is also necessary to unroll the main compute loop P times to ensure that registers are handled correctly.

5.1. **Performance.** After some experiments, we found that the optimal value was $P = 8$. This gave a kernel using 34 registers with an occupancy of 58% and a performance of 58GB/s.

6. Fourth GPU Strategy

At this point it was clear that a comprehensive re-thinking of the algorithm was needed. The fundamental problem seemed to be the level of indirection that is placed between the threads and all the memory operations. Before any memory is accessed, each thread first has to fetch the index at which the access is to occur, and must then use the index to access the data. Each memory operation therefore requires *two* trips to memory: one to fetch the index, and another to access the data using the index. This was slowing the whole process down. Constant memory is simply not fast enough to deliver the indexes at a rate which saturates the global memory bus. Synthetic experiments where each thread "computed" the indexes on-the-fly during execution came close to saturating the bus. The bottleneck therefore appeared to be fetching the indexes.

Indeed, moving the indexes from constant memory into shared memory showed no improvement. Shared memory, just as constant memory, is too slow to provide the indexes at a rate which saturates the global memory bus. Since it is impossible to compute the indexes on-the-fly, a way around the bottleneck had to be found.

Prefetching data seemed to provide some hope. However if we wish to prefetch Normal random numbers, then we should also be prefetching their indexes (since the index is needed to effect the load). And if we are prefetching those indexes, perhaps it makes sense to prefetch all the indexes.

Therefore we changed the algorithm as follows:

(a) Threads no longer cooperated to generate a multidimensional Brownian sample path. Each thread would compute all the dimensions of each sample path. The matrix C was moved to registers so that the matrix-vector product CZ_i could be performed on data held in registers. All the synchronisation barriers were removed.
(b) Since Z_i values were no longer read into shared memory, the local stack was moved from local memory into shared memory.
(c) The execution strategy was moved from constant memory to shared memory. However since shared memory is much too small to hold it, the execution strategy would have to reside in global memory and sections of it would be copied into shared memory as needed.

(d) The indexes defining the execution strategy were prefetched. At each step, 5 indexes are required, and two full steps worth of indexes were prefetched, meaning that 10 indexes were prefetched before the first Brownian point X_T was computed.
(e) In addition to prefetching the indexes, P Normal random numbers were also prefetched from global memory.
(f) The rest of the algorithm remained the same: left and right neighbours were read off the stack, a new Brownian point was computed, and it in turn was stored to the stack and to global memory.

6.1. **Performance.** The changes above led to an explosion in the complexity of the code. Although the ideas are relatively simple, when written down the code takes some time to understand. Not only must the prefetch registers be treated properly, but exceptional care must be taken with the corner cases and cleanup code which results from copying sections of the execution strategy into shared memory. This is compounded by the prefetching, since careful book keeping must be done to know which sections to copy. In summary, turning the idea into a robust implementation is not simple. The resulting kernel used 42 registers per thread, had an occupancy of 45% and ran at 85GB/s.

GPU Strategy	Registers	Occupancy	Performance	% of Peak
First	20	100%	29GB/s	24.1%
Second	20 to 34	100% to 54%	<29GB/s	<24%
Third	34	58%	58GB/s	48.3%
Fourth	42	45%	85GB/s	70.8%
Fifth without prefeching	24	43%	79GB/s	65.8%
Fifth with prefeching	33	58%	102GB/s	85%

TABLE 1. Comparison of the different GPU strategies.

7. Fifth GPU Strategy

The performance of the previous GPU kernel is only 70% of theoretical peak, and there is not much more scope for prefetching. The most likely bottleneck in the algorithm is the traffic into and out of the local stack, and the associated fetching of indexes. At each time step we read two values from the stack and write one value to it, which means fetching three indexes. The indexes also reside in shared memory, which means at each time step we load three values from shared memory, then use those values to load two more values and finally to store a value to shared memory. This traffic is too much for the shared memory bus to handle: it becomes the bottleneck in the program.

Returning to the initialisation step, we considered whether it was possible to reduce the amount of local stack traffic. The answer turns out to be yes, but only by introducing some branches in the innermost loop of the generate function. Consider the situation in (5) where X_ℓ and X_r are the left and right neighbours respectively of the Brownian point X_{θ_i}. By carefully tuning the execution strategy, it is possible to ensure that the set $\{X_\ell, X_r, X_{\theta_i}\}$ contains both the left and right neighbours of the Brownian point $X_{\theta_{i+1}}$. This cannot be done for every point $X_{\theta_{i+1}}$, but it can be done for many of them, and so it is necessary to introduce some flags to identify when it holds. Correspondingly, the main compute loop of the generate step must contain some branches. The branches when computing the point $X_{\theta_{i+1}}$ look roughly as follows:

Are the left and right neighbours of $X_{\theta_{i+1}}$ in the set $\{X_\ell, X_r, X_{\theta_i}\}$?

✓ Yes.
- Identify which of the points X_ℓ , X_r and X_{θ_i} are the neighbours and use them to compute $X_{\theta_{i+1}}$.

× No.
- Read the left and right neighbours from the local stack and use them to compute $X_{\theta_{i+1}}$.

Usually programmers would avoid branches such as this, especially in inner loops, but in this case it proves highly effective.

7.1. **Performance.** Without adding any prefetching, the kernel outlined above uses 24 registers, has an occupancy of 43% and runs at 79GB/s. Once the prefetching in Section 6 is added in, the kernel uses 33 registers, has an occupancy of 58% and runs at 102GB/s. The same code in double precision uses 38 registers, has an occupancy of 33% and runs at 115.66GB/s.

8. Summary of Results and Conclusions

In summary, we state the performance results for both the Brownian sample path generator and the scaled Brownian increments generator as measured on the system detailed in Section 2.4:

- Brownian sample paths generator
  - **Single Precision**: runtime is 10.01ms at 102GB/s achieved global memory throughput. This is 1.9x faster than the time taken to generate the Normal random numbers.[1]
  - **Double Precision**: runtime is 17.12ms at 115.6GB/s achieved global memory throughput. This is 2.2x faster than the time taken to generate the Normal random numbers.[2]
- Scaled Brownian increments generator
  - **Single Precision**: runtime is 9.09ms at 109GB/s achieved global memory throughput. This is 2.09x faster than the time taken to generate the Normal random numbers. [1]
  - **Double Precision**: runtime is 17.01ms at 116.3GB/s achieved global memory throughput. This is 2.2x faster than the time taken to generate the Normal random numbers. [2]

The algorithm from Section 7 (without prefetching) was coded up in C, parallelised with OpenMP and run on a number of different x86-64 systems. The results are given in Table 2.

The three CPU systems represent a range in performance, from the popular Intel Core i7 desktop processor to the top end Intel Xeon X5680 aimed at the server market. The AMD machine is a dual socket system featuring two Magny Cours chips, and is similar to the Phase 2 nodes in the UK HECToR supercomputer. The Xeon machine is also a dual socket system with two Westmere chips. Observe the scaling of the CPU code. The Core i7 shows limited scaling past 2 threads. In addition, with more than 2 threads the performance becomes highly variable (the values shown here are averages). The Xeon shows virtually no scaling past 8 threads. The AMD is the only system that shows scaling up to full hardware capacity. That said, the performance of the AMD system is not particularly exciting, being around 70% slower than the Xeon.

[1] Using the NAG GPU MRG32k3a generator, single precision
[2] Using the NAG GPU MRG32k3a generator, double precision

Generators		Intel Core i7 860, 4 cores @ 2.8GHz (with hyperthreading)					GPU speedup
		1 Thread	2 Threads	3 Threads	4 Threads	8 Threads	
Bridge	float	1212.5ms	1142.6ms	1189.3ms	787.2ms	720.3ms	72.0x
	double	1771.1ms	930.5ms	704.4ms	803.6ms	593.4ms	34.7x
Bridge Incs	float	1170.5ms	613.2ms	857.1ms	639.1ms	605.0ms	67.1x
	double	1452.8ms	782.6ms	742.1ms	653.5ms	549.7ms	32.3x
Generators		AMD Opteron 6174, 24 cores @ 2.2GHz (dual socket Magny Cours, 2 × 12 cores)					GPU speedup
		1 Thread	8 Threads	12 Threads	16 Threads	24 Threads	
Bridge	float	2402.6ms	1019.6ms	676.1ms	452.4ms	355.4ms	35.5x
	double	2948.8ms	900.3ms	576.4ms	454.2ms	328.4ms	19.2x
Bridge Incs	float	2173.6ms	637.2ms	434.4ms	321.1ms	257.0ms	28.6x
	double	2694.1ms	804.9ms	555.3ms	418.4ms	298.8ms	17.6x
Generators		Intel Xeon X5680, 12 cores @ 3.33GHz (dual socket Westmere, 2 × 6 cores with hyperthreading)					GPU speedup
		1 Thread	8 Threads	12 Threads	16 Threads	24 Threads	
Bridge	float	1392.1ms	192.4ms	195.6ms	171.2ms	171.9ms	17.2x
	double	1463.8ms	205.7ms	222.7ms	193.6ms	227.6ms	11.4x
Bridge Incs	float	1207.4ms	172.6ms	168.8ms	146.0ms	207.6ms	16.2x
	double	1308.1ms	183.2ms	184.9ms	166.2ms	211.8ms	9.8x

TABLE 2. Benchmark figures for Tesla C2050 vs. several high performance CPUs.

The behaviour in Table 2 is typical for memory bandwidth bound applications. Throwing additional compute cores at the problem does not improve performance: the speed of the actual memory hardware must be increased. Here GPUs have a distinct advantage due to the fast GDDR5 graphics memory.

8.1. **Conclusions.** In the process of producing the Brownian bridge generators, we have learned a number of lessons regarding the "conventional wisdom" of programming GPUs:

- Higher occupancy does not necessarily mean higher performance for memory bandwidth bound applications.
- Explicit prefetching of data into registers can boost performance significantly.
- Shared memory is not as fast as one might think. In particular, inserting a layer of indirection whereby an index is fetched from shared memory, and then that index is used to fetch another value from shared memory, can slow things down a lot.
- Aggressive use of registers is a good way to boost performance, even for bandwidth bound applications, since registers are the fastest memory on the GPU.
- Judicious use of branching can increase performance, even when the branch is in the innermost compute loop.
- GPUs can be very effective at accelerating memory bandwidth bound applications, which often do not scale well on traditional multicore platforms.

8.2. **Acknowledgments.** The Numerical Algorithms Group wishes to thank Professor Mike Giles, whose work on a GPU Brownian bridge routine was invaluable in making the present implementation. NAG would also like to acknowledge the advice and feedback from two senior quantitative analysts who gave valuable insights into how a Brownian bridge algorithm is used in production.

8.3. **Access to Software.** Users who wish to obtain the Brownian bridge routines should contact NAG either through the website www.nag.co.uk, or via email at infodesk@nag.co.uk.

Both GPU and CPU (single threaded) implementations are available in the NAG Numerical Routines for GPUs [3]. The routine documentation is available at [3]. Example programs (including documentation) showing how to use a GPU Sobol generator together with a GPU Brownian bridge in order to create a GPU Monte Carlo pricing application is available at [4].

As well as featuring in the NAG Numerical Routines for GPUs, both serial and multi-threaded implementations of this very flexible Brownian bridge algorithm will also feature in future releases of NAGs CPU Libraries and the NAG Toolbox for MATLAB. Early releases may be made available upon request.

References

[1] CUDA Optimization : Memory Bandwidth Limited Kernels + Live Q&A by Tim Schroeder. NVIDIA GPU Computing Webinar Series.
http://developer.download.nvidia.com/CUDA/training/Optimizing_Mem_limited_kernels.mp4
[2] Glasserman, P. (2004). *Monte Carlo Methods in Financial Engineering.* Springer.
[3] NAG Numerical Routines for GPUs routine documentation.
http://www.nag.co.uk/numeric/GPUs/doc.asp.
[4] Demos using NAG Numerical Routines for GPUs.
http://www.nag.co.uk/numeric/GPUs/gpu_demo_applications.

Numerical Algorithms Group Ltd
E-mail address: jacques@nag.co.uk

[3] http://www.nag.co.uk/numeric/gpus/index.asp