GPU implementation of finite difference solvers

Mike Giles, Endre László, István Reguly
Oxford e-Research Centre
mike.giles@maths.ox.ac.uk,
lasen@digitus.itk.ppke.hu,
istvan.reguly@oerc.ox.ac.uk

Jeremy Appleyard, Julien Demouth
NVIDIA
jappleyard@nvidia.com,
jdemouth@nvidia.com

ABSTRACT

This paper discusses the implementation of one-factor and three-factor PDE models on GPUs. Both explicit and implicit time-marching methods are considered, with the latter requiring the solution of multiple tridiagonal systems of equations.

Because of the small amount of data involved, one-factor models are primarily compute-limited, with a very good fraction of the peak compute capability being achieved. The key to the performance lies in the heavy use of registers and shuffle instructions for the explicit method, and a non-standard hybrid Thomas/PCR algorithm for solving the tridiagonal systems for the implicit solver.

The three-factor problems involve much more data, and hence their execution is more evenly balanced between computation and data communication to/from the main graphics memory. However, it is again possible to achieve a good fraction of the theoretical peak performance on both measures. The high performance requires particularly careful attention to *coalescence* in the data transfers, using local shared memory for small array transpositions, and padding to avoid shared memory bank conflicts.

Computational results include comparisons to computations on Sandy Bridge and Haswell Intel Xeon processors, using both multithreading and AVX vectorisation.

Categories and Subject Descriptors

G.1.3 [**Numerical Analysis**]: Numerical linear algebra; G.4 [**Mathematical Software**]: Parallel and vector implementations

General Terms

Algorithms, Performance

Keywords

Computational finance, GPU computing, vectorisation, tridiagonal equations

1. INTRODUCTION

Monte Carlo simulation are naturally parallel, and so excellent performance has been achieved on GPUs using well-established random number generation libraries from NVIDIA [10] or NAG [8]. Previous research on Monte Carlo methods has addressed the challenges of the least-squares regression in the Longstaff-Schwartz algorithm for American options [4] and the parallel reductions required in local-vol surface adjoint computations [3].

This paper addresses the challenges in implementing financial finite difference methods. There is already a considerable literature on the use of GPUs for a wide range of finite difference methods (for example see [13]) in engineering and science, in areas such as computational fluid dynamics, electromagnetics and acoustics. The particular focus here is on

- the percentage of peak performance achievable for financial applications which involve relatively few operations per word;
- the best way to implement the tridiagonal solutions which are a key part of the implicit time-marching methods commonly used in finance;
- the data transposition techniques used to maximise the achieved memory bandwidth for 3D applications.

2. ONE-FACTOR MODELS

2.1 Explicit time-marching

A standard approximation of a 1D PDE, such as the Black-Scholes PDE, leads to an explicit finite difference equation of the form

$$u_j^{n+1} = u_j^n + a_j\, u_{j-1}^n + b_j\, u_j^n + c_j\, u_{j+1}^n, \quad j = 0, 1, \ldots J-1$$

with $u_{-1} = u_J = 0$. Here n is the timestep index which increases as the computation progresses, so u_j^{n+1} is a simple combination of the values at the nearest neighbours at the previous timestep. All of the numerical results are for a grid of size $J = 256$ which corresponds to a fairly high resolution grid in financial applications. Additional parallelism is achieved by solving multiple one-factor problems at the same time, with each one having different model constants, or a different financial option payoff.

Each individual calculation is sufficiently small so that it can be performed within a single thread block, with all data held in the registers or local shared memory of a single SMX unit within an NVIDIA Kepler GPU, as illustrated in Figure 1 [12]. In this way, a single *kernel* written in CUDA [9] can

978-1-4799-7027-8/14 $31.00 © 2014 IEEE
DOI 10.1109/WHPCF.2014.10

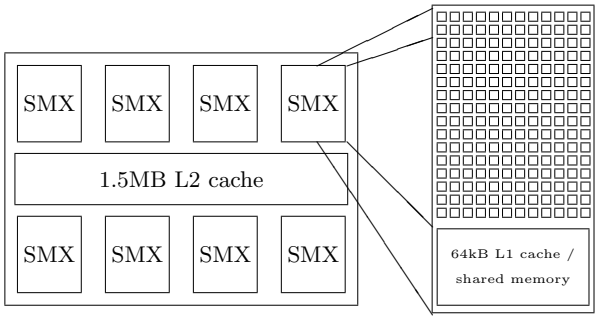

Figure 1: Kepler GPU with 8 SMX units

carry out all of the timesteps required, returning the final result to the global graphics device memory.

The first GPU implementation `explicit1` uses blocks of 256 threads, with each thread responsible for the computations of one particular grid point. Shared memory is used so that threads may obtain the data from neighbouring grid points, but this then requires the use of thread synchronisation to ensure that one set of shared memory reads/writes is completed before the next takes place:

```
__shared__ float u[258];
...
u[0] = 0.0f; u[257] = 0.0f;
i = threadIdx.x + 1
utmp = u[i];

for (int n=0; n<N; n++) {
  utmp = utmp + a*u[i-1] + b*utmp + c*u[i+1];
  __syncthreads();
  u[i] = utmp;
  __syncthreads();
}
```

To improve the performance, it is necessary to avoid the cost of moving data to/from shared memory, and also the cost of thread synchronisation. In Kepler GPUs, the 256 threads are divided into 8 *warps* of 32 threads, with each warp acting in lockstep with all of its threads performing the same computation at the same time but with different data, except that some threads may be masked by conditional predicates. In the second implementation, `explicit2`, each warp handles a different 1D problem, with each thread handling the data for 8 grid points.

No synchronisation is now required between the warps, and the data transfer between neighbouring threads within a warp is preformed using *shuffle* commands, a new hardware capability introduced in the Kepler generation of GPUs [12].

```
for (int n=0; n<N; n++) {
  um = __shfl_up(u[7], 1);
  up = __shfl_down(u[0], 1);

  for (int i=0; i<7; i++) {
    u0   = u[i];
    u[i] = u[i] + a[i]*um + b[i]*u0 + c[i]*u[i+1];
    um   = u0;
  }
  u[7] = u[7] + a[7]*um + b[7]*u[7] + c[7]*up;
}
```

Table 1: One-factor results on a K40 GPU. The timings are for 50000 (explicit) or 2500 (implicit) timesteps, for 2048 options each on a grid of size 256.

	single prec.		double prec.	
	msec	GFlop/s	msec	GFlop/s
explicit1	224	700	258	610
explicit2	52	3029	107	1463
implicit1	19	1849	57	892
implicit2	23	1979	63	951
implicit3	9	1605	22	687

The performance achieved by `explicit1` and `explicit2` is documented in Table 1 on an NVIDIA Tesla K40 GPU clocked at 875 MHz, that is capable of achieving 5.04 TFlops in single precision or 1.68 TFlops in double precision, which is higher than in the datasheet [11] which gives values for the base clock of 745 MHz. The theoretical bandwidth is 288GB/s but the measured bandwidth on a standard test is 235 GB/s. To obtain accurate timing results, the codes performed 2048 separate option calculations, each using 50000 timesteps. `explicit2` achieves 2.5 - 4 times the performance of `explicit1`, and approximately 60% of the SP peak and 87% of the DP peak performance.

The results in the table are for the execution of the GPU kernel which performs the calculations. This includes the transfer from the CPU to the GPU of the parameters required for the calculation, but does not include the transfer back of the results, which are taken as a single data point (for the at-the-money option value). However, this adds less than 0.1 msec to the execution time, and so is negligible.

2.2 Tridiagonal solution algorithms

Before discussing the implementation of implicit time-marching approximations, we first review a number of different algorithms for solving tridiagonal systems of linear equations. This builds on a substantial body of prior research [1, 14, 16, 18, 19], but the particular hybrid algorithm we use in this paper seems to be a novel combination due to the specifics of the GPU hardware.

2.2.1 Thomas algorithm

The **Thomas** algorithm is the single-thread sequential algorithm which is described in every textbook [17]. It is simply a customisation of Gaussian elimination to the case in which the matrix is tridiagonal. If the tridiagonal system is represented as

$$a_j u_{j-1} + b_j u_j + c_j u_{j+1} = d_j, \quad j = 0, 1, \ldots, J-1$$

with $u_{-1} = u_J = 0$, then the Thomas algorithm has a forward pass in which the lower diagonal elements a_j are eliminated by adding a multiple of the row above. This is then followed by a reverse pass to compute the final solution using the modified c_j values.

The full Thomas algorithm is given in Algorithm 1. Note that this does not perform any pivoting; it is assumed the matrix is diagonally dominant, or at least sufficiently close to diagonal dominance so that the solution is well-conditioned. The computational cost per row is approximately three FMA (Fused Multiply-Add) operations, one reciprocal and two

Algorithm 1 Thomas algorithm, returning solution in d array

$d_0 := d_0/b_0$
$c_0 := c_0/b_0$
for $j = 1, \ldots, J-1$ **do**
  $r := 1\,/\,(b_j - a_j\,c_{j-1})$
  $d_j := r\,(d_j - a_j\,d_{j-1})$
  $c_j := r\,c_j$
end for
for $j = J-2, \ldots, 0$ **do**
  $d_j := d_j - c_j\,u_{j+1}$
end for

Algorithm 2 PCR algorithm, returning solution in $d^{(P)}$ array

for $p = 1, \ldots, P$ **do**
  $s := 2^{p-1}$
  for $j = 0, \ldots, N-1$ **do**
    $r := 1\,/\,(1 - a_j^{(p-1)}\,c_{j-s}^{(p-1)} - c_j^{(p-1)}a_{j+s}^{(p-1)})$
    $a_j^{(p)} := -r\,a_j^{(p-1)}\,a_{j-s}^{(p-1)}$
    $c_j^{(p)} := -r\,c_j^{(p-1)}\,c_{j+s}^{(p-1)}$
    $d_j^{(p)} := r\,(d_j^{(p-1)} - a_j^{(p-1)}\,d_{j-s}^{(p-1)} - c_j^{(p-1)}\,d_{j+s}^{(p-1)})$
  end for
end for

multiplications. If we treat the cost of the reciprocal as being equivalent to five FMAs (which is the approximate cost on a GPU for a double precision reciprocal), then the total cost is equivalent to 10 FMAs.

However, if this is executed by a single thread, it will require a lot of data transfer to/from the main graphics memory, and so the performance is likely to be to be memory-bandwidth limited. Hence, it is better to use a different algorithm which avoids the memory traffic, but possibly at the cost of an increased number of floating point operations.

2.2.2 Parallel Cyclic Reduction

PCR (Parallel Cyclic Reduction) is an inherently parallel algorithm which is ideal when using multiple threads to solve each tridiagonal system. If we start with the same tridiagonal system of equations, but normalised so that $b_j = 1$,

$$a_j u_{j-1} + u_j + c_j u_{j+1} = d_j, \quad j = 0, 1, \ldots, J-1,$$

with $u_j = 0$ for $j < 0$ and $j \geq J$. Subtracting the appropriate multiples of rows $j \pm 1$, and re-normalising, gives

$$a'_j u_{j-2} + u_j + c'_j u_{j+2} = d'_j, \quad j = 0, 1, \ldots, J-1.$$

Repeating this by subtracting the appropriate multiples of rows $j \pm 2$ gives

$$a''_j u_{j-4} + u_j + c''_j u_{j+4} = d''_j, \quad j = 0, 1, \ldots, J-1,$$

and after P such steps, where $2^p \geq J$, we obtain the final solution since $j - 2^P < 0$ and $j + 2^P \geq J$.

The PCR algorithm is given in Algorithm 2. Note that any reference to a value with index j outside the range $0 \leq j < J$ is taken to be zero. If the computations within each step are performed simultaneously for all j, then it is possible to reuse the storage so that $a^{(p+1)}$ and $c^{(p+1)}$ are held in the same storage (e.g. the same registers) as $a^{(p)}$ and $c^{(p)}$.

The following code implements it for a single warp, with one unknown per thread, and using shuffle instructions to exchange data between threads:

```
__forceinline__ __device__
float trid1_warp(float a, float c, float d) {
  float b; uint s=1;
  for (int n=0; n<5; n++) {
    b = 1.0f / (1.0f - a*__shfl_up(c,s)
                     - c*__shfl_down(a,s));
    d = ( d  - a*__shfl_up(d,s)
             - c*__shfl_down(d,s) ) * b;
    a =      - a*__shfl_up(a,s)     * b;
    c =      - c*__shfl_down(c,s)   * b;
    s = s<<1;
  }
  return d;
}
```

The computational cost per row is approximately equivalent to 14 FMAs in each step, so the total cost is $14 \log_2 N$. This is clearly much greater than the cost of the Thomas algorithm, but there is no data transfer to/from the main memory if a, c and d can be held in registers.

2.2.3 Hybrid algorithm

The **hybrid Thomas/PCR** algorithm is a combination of the Thomas and PCR algorithms. Suppose the tridiagonal system is broken into a number of sections of size M, each of which will be handled by a separate thread. Within each of these pieces, using local indices ranging from 0 to $M-1$, a slight modification to the Thomas algorithm operating on rows 1 to $M-2$ enables one to obtain an equation of the form

$$a_j\,u_1 + u_j + c_i\,u_M = d_j, \quad i = 1, 2, \ldots, M-2 \qquad (1)$$

expressing the central values as a linear function of the two end values. Using these to eliminate the u_1 and u_{J-2} entries in the equations for rows 0 and $M-1$, leads to a reduced tridiagonal system of equations involving the first and last variables within each section. This reduced tridiagonal system can be solved using PCR, and then (1) gives the interior values.

Algorithm 3 gives the algorithm for the first part of this process to compute the coefficients a_j, c_j, d_j in (1). The cost of this, plus the back solve once the end values are known, is

Algorithm 3 First phase of hybrid algorithm

$d_1 := d_1/b_1$
$a_1 := a_1/b_1$
$c_1 := c_1/b_1$
for $j = 2, \ldots, M-2$ **do**
  $r := 1\,/\,(b_j - a_j\,c_{j-1})$
  $d_j := r\,(d_j - a_j\,d_{j-1})$
  $a_j := -r\,a_j\,a_{j-1}$
  $c_j := r\,c_j$
end for
for $j = M-3, \ldots, 1$ **do**
  $d_j := d_j - c_j\,u_{j+1}$
  $a_j := a_j - c_j\,a_{j+1}$
  $c_j := -c_j\,c_{j+1}$
end for

approximately 14 FMAs per point, about 40% more than the cost of the Thomas algorithm for the testcases considered in this paper, with $M=8$ and $N=256$.

2.3 Implicit time-marching

A standard implicit time-marching approximation leads to a tridiagonal set of equations of the form

$$a_j\, u_{j-1}^{n+1} + b_j\, u_j^{n+1} + c_j\, u_{j+1}^{n+1} = u_j^n, \quad j = 0, 1, \ldots J-1$$

with $u_{-1} = u_J = 0$.

The first implementation, `implicit1`, follows the data layout of `explicit2` with each warp handling one financial option, and each thread in the warp handling 8 grid points. The hybrid Thomas/PCR algorithm is used, with $M = 8$, and the warp-based PCR is used to solve the reduced system.

`implicit2` re-arranges the system of equations to compute the change $\Delta u_j = u_j^{n+1} - u_j^n$ using the equation

$$a_j\, \Delta u_{j-1} + b_j\, \Delta u_j + c_j\, \Delta u_{j+1} = u_j^n - a_j\, u_{j-1}^n - b_j\, u_j^n - c_j\, u_{j+1}^n.$$

The computational cost is slightly higher, but the accuracy is significantly better, with the single precision floating point errors reduced from 5e-5 to 1e-6.

Finally, `implicit3` exploits the fact that the coefficients a_j, b_j, c_j in the test application do not vary with n, and therefore some steps in the computation can be performed just once by moving code outside the main time-marching loop. The runtime is reduced significantly due to the smaller number of operations, even though the GFlop/s is poorer due to higher register use.

One important detail in the implementation concerns the computation of reciprocals. The single precision code uses the fast reciprocal provided by the SFU (Special Function Unit) which is not IEEE-compliant but provides comparable accuracy. The double precision uses a fast reciprocal in which the SFU computes an approximate reciprocal, and this is then refined to full double precision by an extension to Newton iteration in 3 FMAs.

```
static __forceinline__ double __rcp(double a)
{
  double e, y;
  asm("rcp.approx.ftz.f64 %0,%1;":"=d"(y):"d"(a));
  e = __fma_rn (-a, y, 1.0);
  e = __fma_rn ( e, e,   e);
  y = __fma_rn ( e, y,   y);
  return y;
}
```

3. THREE-FACTOR MODELS

The three-factor test application uses the Black-Scholes PDE for 3 underlying assets, each corresponding to Geometric Brownian Motion and with positive correlation between the 3 driving Brownian motions. This leads to a parabolic PDE which spatial cross-derivative terms with positive coefficients. The spatial approximation of this leads to a 13-point stencil involving offsets $\pm(1,0,0)$, $\pm(0,1,0)$, $\pm(0,0,1)$, $\pm(1,1,0)$, $\pm(0,1,1)$ and $\pm(1,0,1)$, relative to a point with 3D indices (i,j,k), as illustrated in Figure 2.

The test case uses a grid of size 256^3, with all data stored in the main graphics device memory in 1D arrays with memory location

$$\mathrm{id} = i + 256\, j + 256^2\, k,$$

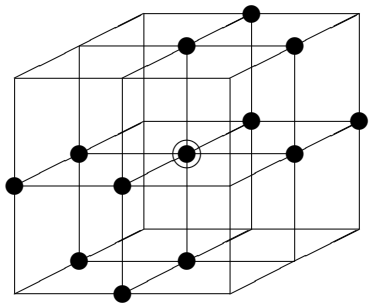

Figure 2: Finite difference stencil for 3D explicit time-marching

Table 2: Three-factor performance as reported by `nvprof`, the NVIDIA Profiler on a K40. The timings are for 500 (explicit) or 100 (implicit) timesteps, on a grid of size 256^3.

	single prec.			double prec.		
	msec	GFlop/s	GB/s	msec	GFlop/s	GB/s
explicit1	747	597	100	1200	367	127
explicit2	600	760	132	923	487	144
implicit1	447	406	146	889	243	144

so the data is contiguous in the x-direction. Note that this gives natural cache-alignment at the beginning of each row in the x-direction. If the size in the x-direction was not a multiple of 32, then array padding up to the next multiple of 32 would probably be desirable.

The computational grid in the x/y-plane is subdivided into blocks of 32×8 which are handled by separate thread blocks. Each thread corresponds to a unique index pair (i,j), and performs the computations for all values of k, working upwards from $k=0$ to $k=255$. The 32 threads in a warp process neighbouring values of i, and therefore when loading the values $u_{i,j,k}$ the warp read addresses all of an integer number of cache lines (the precise number depending on the cache line size and whether the data is SP or DP). This is known as a *coalesced* data access, and gives the maximum possible data transfer speed. The same occurs for other read/write operations except when dealing with $i \pm 1$ in which case things are slightly misaligned.

The first implementation of the explicit solver, `explicit1`, relies entirely on the cache for data reuse as k increases. The second implementation, `explicit2`, holds data in registers so that, for example, the old data for $(i,j,k+1)$ which is loaded in when computing (i,j,k) can then be re-used for the computations for $(i,j,k+1)$ and $(i,j,k+2)$.

Table 2 shows that this leads to slightly higher performance, at the cost of slightly more complex programming. Unlike the table for the one-factor results, Table 2 includes device memory bandwidths as measured by the NVIDIA Profiler. It can be seen that we are obtaining 35-50% of the peak memory bandwidth, and 15-30% of the peak GFlop/s performance. Hence, the application is relatively balanced between computation and communication, but is slightly

more bandwidth-limited overall.

The times shown in the table are again just for the execution of the kernel performing the calculation. In this case the data transfer cost is more significant, approximately 30 msec in single precision and 50 msec in double precision. This is because the initial values are set on the CPU and then transferred to the GPU, and at the end the entire solution is brought back to the CPU for comparison with separate CPU results to validate the GPU computations. In a real application code, the initialisation would be performed by a separate kernel on the GPU, and only a few at-the-money values would be transferred back to the CPU, and so the transfer cost would again become negligible.

Table 2 also shows the performance for the implicit solver `implicit1`, but the development and optimisation of this was a much harder task than the two explicit solvers.

Mathematically, an ADI (alternating-direction-implicit) approximate factorisation is used, which leads to a system of implicit equations of the form:

$$A_x\, A_y\, A_z \Delta U = D^n$$

where ΔU represents the vector of increment $\Delta U_{i,j,k} \equiv U^{n+1}_{i,j,k} - U^n_{i,j,k}$. The matrix A_x corresponds to a set of tridiagonal equations along each grid line in the x-direction, and A_y and A_z are similar in the other two directions.

Computationally, each timestep is performed by 4 kernels, one to compute the r.h.s. D^n, and then one kernel for the tridiagonal solutions in each coordinate direction.

$$\begin{aligned} A_x\ V^{(1)} &= D^n, \\ A_y\ V^{(2)} &= V^{(1)}, \\ A_z\ \Delta U &= V^{(2)}. \end{aligned}$$

Each of the tridiagonal solutions uses the same hybrid solver approach as the 1D implicit solver, with each thread responsible for 8 grid elements. However, the details are different because of the data layout, and these details are crucial to achieving the performance shown in Table 2.

The solution in the x-direction is the simplest of the three. Each warp is responsible for one line, and as usual each thread is responsible for 8 data elements. The problem is that the data is stored contiguously, so a natural coalesced read will lead to thread 0 having elements $0, 32, 64, \ldots, 224$, but mathematically it needs elements $0, 1, 2, \ldots, 8$. The solution to this problem is a data transposition, following the approach explained in the Appendix, which is a generalisation of a technique described by Harris [6].

The implementation in the y-direction is trickier. Here we have a thread block of 256 threads, consisting of 8 warps of 32 threads, and collectively the thread block will perform the tridiagonal solutions for 8 lines in the y-direction, with the 8 lines being neighbours in the x-direction. The tricky bit is that threads work on different lines at different times.

In the first phase, in the first read by the 32 threads in warp 0:

- threads 0-7 load element 0 of the 8 lines
- threads 8-15 load element 8 of the 8 lines
- threads 16-23 load element 16 of the 8 lines
- threads 24-31 load element 24 of the 8 lines

Each quarter of the warp is loading 8 consecutive data elements, since the 8 lines are neighbours in the x-direction. This is not perfectly coalesced when working in single precision, but it does achieve a high percentage of cache line re-use. Subsequent reads shift in the y-direction by one position, so that thread 0 ends up with elements $0 - 7$ of the first line, and can then carry out the calculation for the first stage of the hybrid solution algorithm.

In this way, the first warp is performing the calculations for the first 32 elements in each of the 8 lines, the second warp is doing the next 32 elements, and so on. The problem is when we come to the PCR step in the hybrid solution. Now, the data for the first line is spread across all of the warps, and so the PCR calculation cannot be carried out using warp shuffles.

The solution is to use data transposition again, so that warp 0 ends up the data for the first line, wrap 1 has the data for the second line, and so on. Once the PCR step is completed, the solution data is transposed back, and the hybrid algorithm is completed.

This is undoubtedly a complex implementation, but it is required to achieve the best performance. Those interested should read the code to fully understand the details [5].

The solution in the z-direction is essentially the same as in the y-direction, with the 8 lines in the z-direction being neighbours in the x-direction.

4. INTEL CPU IMPLEMENTATIONS

Tables 3 and 4 report the performance of the CPU implementations of the same one-factor and three-factor applications on two systems:

- a server with two 8-core 2.9 GHz Intel Xeon E5-2690 (Sandy Bridge) CPUs, each with 32kB L1 cache and 256kB L2 cache per core, a shared 20MB L3 cache and up to 51.2GB/s memory bandwidth (but 66GB/s total bandwidth over the two sockets is the best achieved on our system by the STREAM benchmark [7]);
- a workstation with one 3.6GHz 4-core Intel i7-4770K (Haswell) CPU, with 32kB L1 cache and 256kB L2 cache per core and a 8MB L3 cache and up to 25.6GB/s memory bandwidth (23GB/s bandwidth is achieved on our system by the STREAM benchmark).

With full vectorisation, each of the Xeon E5-2690 CPUs is capable of delivering 318 GFlop/s (SP) or 171 GFlop/s (DP) in the standard S/DGEMM matrix-matrix multiply benchmark [2]. Despite its smaller number of cores, the i7-4770 is capable of 343 GFlop/s (SP) or 184 GFlop/s (DP) because of its new AVX2 instruction set which includes the ability to perform vector FMA (Fused Multiply-Add) operations.

The performance metrics reported in Tables 3 and 4 are based on the theoretical number of floating point operations that have to be carried out, divided by execution time. We have also used Likwid [15] to verify these numbers; this showed a close match (within 10-15%), however, several counters are unsupported or unreliable on our platforms, according to the documentation, and therefore these are not reported here.

OpenMP multithreading is used in combination with AVX vectorisation. For the one-factor application, `explicit1` uses compiler auto-vectorisation applied to the loop over the 256 grid points. `explicit2` uses the same technique as the

Table 3: One-factor performance on a pair of Intel Xeon E5-2690 (Sandy Bridge) CPUs and an Intel i7-4770K (Haswell) CPU. The timings are for 50000 (explicit) or 2500 (implicit) timesteps, for 2048 options each on a grid of size 256.

Dual-socket Intel Xeon E5-2690				
	single prec.		double prec.	
	msec	GFlop/s	msec	GFlop/s
explicit1	563	279	1188	132
explicit2	398	394	781	201
explicit3	806	194	1570	100
implicit1	187	139	470	48
implicit2	157	166	473	47

Intel i7-4770				
	single prec.		double prec.	
	msec	GFlop/s	msec	GFlop/s
explicit1	1069	147	1845	85
explicit2	749	209	1531	102
explicit3	1443	109	2890	54
implicit1	505	51	844	26
implicit2	256	102	775	29

CUDA implementation, with threads corresponding to vector lanes and a warp mapping to a full AVX vector, therefore each lane processes either 32 or 64 grid points in single or double precision respectively. `explicit2` is implemented using the low-level AVX vector intrinsics, and the permute intrinsic is used in place of CUDA's shuffle. `explicit3` uses compiler auto-vectorisation over different options. The performance results clearly show the advantage of vectorisation over just a single option – this is because all the data can be contained in the L1 cache whereas it spills to L2 when vectorising over multiple options. Furthermore, the manual vectorisation over just a single option (`explicit2`) is significantly faster than the compiler auto-vectorisation, achieving 62% of peak compute performance in single and 59% in double precision on the Sandy Bridge server chip, and 61% and 55% on the Haswell chip.

For the implicit time-marching, `implicit1` and `implicit2` use the Thomas algorithm and vectorise over options, so that each lane corresponds to a different financial option. `implicit1` uses `icc` auto-vectorisation over options, whereas `implicit2` uses vector intrinsics to explicitly assign options to different lanes. Furthermore, the single precision implementation uses fast reciprocals and a subsequent correction step, which is much faster than the generic division instruction. The amount of memory these approaches require leads to data being spilled to the L2 cache, but the penalty from this is much lower than the spills to graphics device memory which would have resulted from using the Thomas algorithm for the GPU implementation.

Note that the permute instructions used to communicate between lanes (the equivalent of CUDA's shuffles) do not support arbitrary shuffles on Sandy Bridge, but they do on Haswell as part of the new AVX2 instruction set, and therefore the relative performance of `implicit2` is much better on the Haswell CPU.

Overall, the one-factor results show a performance difference of approximately 2× between the two Xeons and the single i7, in line with their computational capabilities.

Table 4: Three-factor performance on a pair of Intel Xeon E5-2690 (Sandy Bridge) CPUs, and an Intel i7-4770K (Haswell) CPU. The timings are for 500 (explicit) or 100 (implicit) timesteps, for a single option on a grid of size 256^3.

Dual-socket Intel Xeon E5-2690						
	single prec.			double prec.		
	msec	GFlop/s	GB/s	msec	GFlop/s	GB/s
explicit1	1903	233	34	3911	114	33
implicit1	2561	82	23	4966	42	23

Intel i7-4770						
	single prec.			double prec.		
	msec	GFlop/s	GB/s	msec	GFlop/s	GB/s
explicit1	4538	99	14	11140	40	11
implicit1	6883	30	8	11816	18	9

Table 5: Execution times for best explicit/implicit one-factor (1F) and three-factor (3F) implementations on K40 GPU and Intel E5-2690 Xeon CPUs.

	K40 GPU		2 Xeon CPUs	
	SP	DP	SP	DP
1F explicit	52	107	398	781
1F implicit	19	57	157	473
3F explicit	600	923	1903	3911
3F implicit	447	889	2561	4966

The three-factor explicit time-marching implementation uses auto-vectorisation in the x-direction in which data is stored contiguously. For the ADI implicit time-marching, vectorisation in the x-direction is used for the solution in the y- and z-directions, but no vectorisation is used for the solution in the x-direction because of the unfavourable data layout in that direction.

The GFlop/s and GB/s performance documented in Table 4 are estimates based on the execution time, the number of floating point operations required, and the amount of data which was be transferred at least once between the main memory and the CPU(s). Because performance is much more limited by bandwidth to off-chip memory, the difference between the two Xeons and the single i7 is $2.5-3\times$, corresponding to the larger difference in available bandwidth.

The poorer performance of the implicit implementations, measured as either GFlop/s or GB/s, indicates that there may be scope for further improvement. More detailed timing shows that around half the execution time is for the x-direction tridiagonal solution which is currently not vectorised. Vectorising over multiple financial options is certainly possible, in principle, but it would require careful hand-coding to perform vector register data transposition, analogous to that used in the GPU implementation.

5. CONCLUSIONS

The primary conclusion is that it is possible to achieve excellent performance with both explicit and implicit computational finance PDE approximations on GPUs. On an absolute basis, the performance is approximately 50-90% of the peak capability of the NVIDIA K40 GPU, based on either its computational or bandwidth capabilities. Furthermore, Table 5 shows that on a relative basis it is a factor $3-8\times$ better than using a pair of Intel E5-2690 Xeon CPUs which together have a similar cost and power consumption. The performance ratio is $7-8\times$ for the one-factor application which is compute-limited, and $3-5.5\times$ for the three-factor application which is primarily bandwidth-limited.

With one-factor models, the key is to use one warp per financial option, and compute all timesteps within one CUDA kernel avoiding all intermediate data transfer to/from the graphics memory. For explicit methods, the implementation is fairly straightforward, using warp shuffles instructions to move data between threads. For implicit methods, the implementation is significantly harder, using a hybrid solution method for the tridiagonal equations, combining the classic Thomas algorithm with the highly-parallel PCR algorithm.

With three-factor models the data has to be held in the main graphics memory, and therefore separate kernels have to be called for each timestep. The mathematical approach is the same as for the one-factor problem, but because the performance is bandwidth-limited great care has to be taken with data transposition in the implicit solver implementation.

The paper has not considered two-factor and four-factor models. Assuming that a four-factor model fits within the 12GB graphics memory in a K40, the implementation would be very similar to the three-factor model. With two-factor models, it is possible that a single two-factor model would fit inside a single SMX unit, in which case the implementation would be similar to the one-factor implementation, but with just a single two-factor model per thread block. However, it is more likely that a two-factor model will have too much data for a single SMX, and therefore the implementation will need to be more like the three-factor implementation, but with multiple financial options being solved simultaneously to increase the degree of natural parallelism.

6. ACKNOWLEDGMENTS

The research at the Oxford e-Research Centre has been partially supported by the ASEArch project on Algorithms and Software for Emerging Architectures, and partly by an Impact Acceleration Award, both funded by the UK Engineering and Physical Sciences Research Council.

7. REFERENCES

[1] S. Bondeli. Divide and conquer: A parallel algorithm for the solution of a tridiagonal linear system of equations. *Parallel Computing*, 17(4):419–434, 1991.

[2] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[3] J. du Toit, J. Lotz, and U. Naumann. Adjoint algorithmic differentiation of a GPU-accelerated application. `http://www.nag.co.uk/Market/articles/adjoint-algorithmic-differentiation-of-gpu-accelerated-app.pdf`, 2013.

[4] M. Fatica and E. Phillips. Pricing American options with least squares Monte Carlo on GPUs. In *Proceedings of the 6th Workshop on High Performance Computational Finance*, page 5. ACM, 2013.

[5] M.B. Giles. 1D/3D Black-Scholes code for GPUs. `http://people.maths.ox.ac.uk/gilesm/codes/BS_1D/` `http://people.maths.ox.ac.uk/gilesm/codes/BS_3D/`, 2014.

[6] M. Harris. An efficient matrix transpose in CUDA C/C++. `http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/`, 2013.

[7] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[8] NAG. Numerical Routines for GPUs. `http://www.nag.co.uk/numeric/GPUs/`, 2014.

[9] NVIDIA. CUDA Programming Guide 6.0. `http://docs.nvidia.com/cuda/`, 2014.

[10] NVIDIA. CURAND Random Number Generation Library. `http://docs.nvidia.com/cuda/curand/`, 2014.

[11] NVIDIA. NVIDIA Tesla Datasheet. `http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf`, 2014.

[12] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2014.

[13] E. Phillips and M. Fatica. Implementing the Himeno benchmark with CUDA on GPU clusters. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[14] H.S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 1(4):289–307, 1975.

[15] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[16] H.A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Computing*, 5(1):45–54, 1987.

[17] W.T. Vetterling, S.A. Teulokshy, W.H. Press, and B.P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1985.

[18] H.H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software (TOMS)*, 7(2):170–183, 1981.

[19] Y. Zhang, J. Cohen, and J.D. Owens. Fast tridiagonal solvers on the GPU. *ACM Sigplan Notices*, 45(5):127–136, 2010.

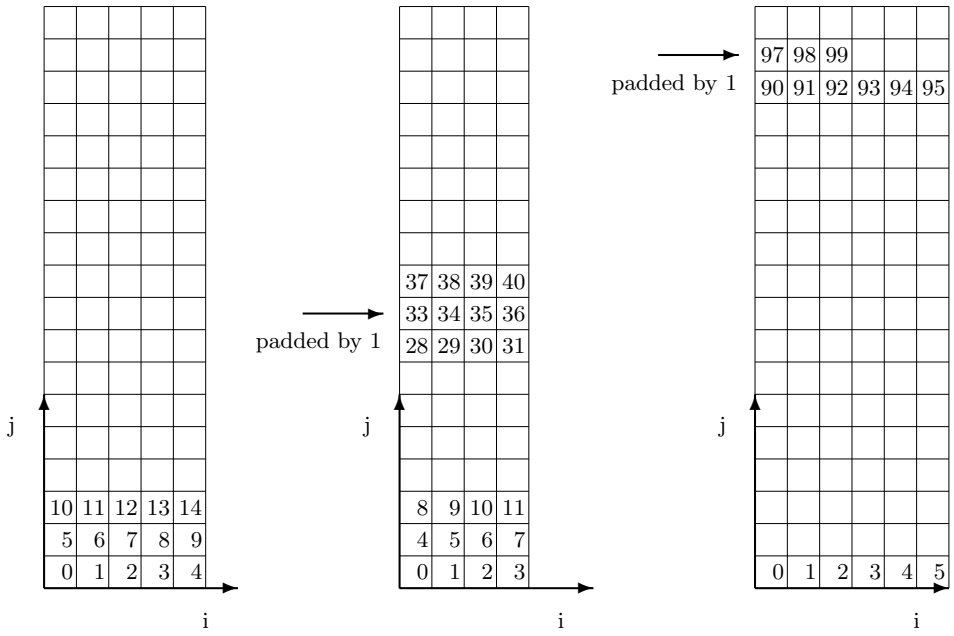

Figure 3: Illustration of shared memory array to be written into, and read from, for different widths I

APPENDIX

A. ARRAY TRANSPOSITION

As illustrated in Figure 3, we want to work with a shared memory array which is mathematically of width I, and height 32 (equal to the warp size).

We want to effectively transpose some data by writing into it with the threads in the thread block filling it row-wise, working across the first row, then the second, and so on in ascending order of $i + jI$, and then (after a synchronisation) reading data out of it column-wise, working up the first column, then the second and so in in ascending order of $j + 32i$. Or, vice versa, we might want to fill it by columns, and then read it out by rows.

The challenge is to come up with a mapping $(i, j) \to k$ to an index k in the linear shared memory array so that there are no memory bank conflicts when accessing the data in either direction.

When I is odd we can define

$$k = i + j\,I.$$

This naturally gives no bank conflicts when reading row-wise, since each warp gets 32 contiguous addresses and current NVIDIA GPUs have 32 shared memory banks.

Furthermore, there are no bank conflicts in each column, because $j = 32$ is the smallest strictly positive integer such that

$$j\,I \bmod 32 = 0$$

which would lead to a bank conflict with the element $j = 0$.

When I is a power of 2, then the definition

$$k = i + j\,I$$

would lead to bank conflicts along each column. The first bank conflict is when $i = 0, j = 8, k = 32$. This suggests the idea of padding by 1 after every 32 elements, giving the mapping

$$k = i + j\,I\, +\, (i + jI)/32$$

where the division is interpreted in the integer sense (i.e. discarding the remainder).

Having handled the two extreme cases, we now consider the general case in which $I = P\,F$, where P is a power of 2, and F is odd. In this case

$$k = i + j\,I$$

leads to the first bank conflict when $i = 0$ and

$$j\,I \bmod 32 = 0.$$

If $P < 32$ then this implies

$$j\,F \bmod (32/P) = 0,$$

which happens first when $j = 32/P$, and hence $jI = 32F$. Thus, the padded definition to avoid conflicts is

$$k = i + j\,I + (i + j\,I)/(32F).$$

Note: when $I = F$, then $(i + j\,I)/(32F) = 0$ which correctly gives us back the unpadded version.

If $P \geq 32$ then the first bank conflict occurs when $j = 1$, and an appropriate padding is

$$k = i + j\,I + j.$$

As well as being useful for the application in this paper, this data transposition is also very useful for application in which data is stored as an Array-of-Structs, each of size I. Threads can load in contiguous vectors from device memory and fill the shared memory array by rows, and then, after synchronisation, read it back into registers from columns so that each thread gets its required struct data. The process would be reversed for writing back to device memory.