Daniel Egloff

GPUs in Financial Computing Part II: Massively Parallel PDE Solvers on GPUs

Continuing to speed you on your way ...

In our previous article in Wilmott (Egloff, 2010), we started to look at one-dimensional partial differential equation (PDE) solvers on graphics processing units (GPUs) targeting industry standards like local volatility models, single-factor convertible bond models, or common one-factor interest rate models. We introduced the finite difference discretization schemes for one-dimensional PDEs and a financially motivated coordinate transformation to reduce the problem in case the underlying asset pays discrete dividends.

Implicit and mixed time-stepping schemes of the Crank–Nicholson type require a linear system to be solved in every time step. For the commonly used finite difference schemes, the system is of a tridiagonal structure. Solving tridiagonal systems efficiently on parallel computers is challenging because of the inherent dependency between the rows of the system and the low computation-to-communication ratio. Therefore, we focused in our previous article (Egloff, 2010) on the algorithmic details of parallel tridiagonal solvers. We introduced the parallel cyclic reduction algorithm and showed how to implement it on a GPU by fully exploiting fine-grained parallelism and specific close to the ALU memory such as shared memory on an NVIDIA GPU.

As explained previously (Egloff, 2010), pricing a single instrument in a one-factor model context with a finite difference-based PDE solver does not lead to enough parallel work to keep a modern GPU busy. Fortunately, in realistic financial applications, it is sensible to do several pricing calculations at the same time. If an underlying changes, usually a whole set of options with different strikes, maturities, payoff profiles, and further characteristics needs to be repriced. Another example is risk management, where whole books of derivatives have to be priced under multiple scenarios. Computing a large number of option prices with finite difference schemes in parallel leads to enough data-parallel work to fully load a modern GPU.

We propose a massively parallel PDE solver which simultaneously prices a large collection of similar or related derivatives with finite difference schemes. The performance analysis shows a remarkable speedup by a factor of 25 on a single GPU and up to a factor of 40 on a dual-GPU configuration against an optimized CPU version.

Massively parallel GPU PDE solver

Our target application is to price a large number of similar or related one-factor option pricing problems with a PDE solver based on finite difference discretization schemes. We process them with one or multiple GPUs. The overall collection of pricing problems is split into subsets according to some load-balancing strategy. Each subset is scheduled for execution to one of the available GPUs. For every GPU, a dedicated CPU thread is responsible for the scheduling and execution of subsets of pricing problems. This thread also performs the data transfer to and from the GPU and launches the data preparation and pricing kernels.

A single PDE pricing problem of a subset scheduled for execution on a specific GPU is solved with a block of threads, where each thread is handling a discretization node of the finite difference scheme. This thread organization optimally utilizes the hierarchical hardware structure of the GPU. It allows us to use shared memory for the data exchange between threads and to synchronize threads working on the same PDE. Furthermore, we can apply the fine-grained parallel tridiagonal solver described previously (Egloff, 2010) for implicit or Crank–Nicholson time-stepping schemes. Because thread blocks are executed on one of the available streaming multiprocessors of the GPU, we can process several PDE pricing problems in parallel on a single GPU. For instance, as a Tesla C1060 GPU has 30 multiprocessors, we can process as many as 30 pricing problems in parallel. Note that if we assign more pricing problems to a GPU than numbers of multiprocessors on the GPU, the hardware utilization can be further optimized because the hardware thread manager can switch between different PDE pricing problems, thereby hiding memory latency more efficiently.

Design and implementation considerations

Some design and implementation considerations are worth mentioning. We will see in the performance study that a good overall GPU utilization can only be achieved with a large problem set. Therefore, in order to maximize the number of derivatives which can be bundled for parallel execution, the PDE solver is designed to handle all kinds of payoff features, including single and double barriers, time window barriers, as well as early exercise of the Bermudan or American type.

On the implementation side, we must pay attention to scalability within a GPU and across

multiple GPUs, clever data management to reuse common data, and efficient data transfer.

For a multi-GPU configuration, the task-splitting strategy is very important to achieve scalability in the number of GPUs. A fairly simple and still efficient strategy is to first group pricing problems of the same underlying and then sort them according to the computational cost, measured in terms of the number of time steps times the number of discretization nodes. The idea is to build subsets of pricing problems of roughly the same computational cost and with as much common input data as possible.

Because the transfer of data from CPU host memory to GPU device memory and back can easily cost a significant amount of the overall processing time, a careful design is indispensable for high performance and low latency. A rule of thumb is to:

1. Keep the data on the GPU for as long as possible and reuse it for multiple computations;

2. Avoid lots of small data transfers; instead, pack data into blocks before sending it to the GPU device memory;

3. Optimize the layout of the packed data by introducing padding, memory alignment, or interleaving, such that the data structure, once copied to device memory, allows for efficient memory access patterns from multiple blocks of threads.

A further optimization is achieved by asynchronous data transfer, such that memory transfer and computations can overlap.

Table 1: CPU solver timings in milliseconds

	Single precision			Double precision		
log (n)	Forsythe–Moler	Serial CR	MKL sgtvs	Forsythe–Moler	Serial CR	MKL dgtvs
6	0.0047	0.0048	0.0033	0.0027	0.0031	0.0027
8	0.0161	0.0156	0.0097	0.0083	0.0111	0.0082
10	0.0611	0.0702	0.0356	0.0271	0.0435	0.0274
12	0.2354	0.2887	0.1373	0.1176	0.1386	0.1043
14	0.6541	0.8122	0.3847	0.4372	0.7128	0.4493
18	9.4445	13.6012	6.5998	9.8787	20.4651	8.7308

Table 2: CPU solver timings relative performance compared to MKL sgtvs and dgtvs

	Single precision		Double precision	
log (n)	Forsythe–Moler	Serial CR	Forsythe–Moler	Serial CR
6	1.43	1.46	1.00	1.15
8	1.65	1.60	1.01	1.36
10	1.72	1.97	0.99	1.59
12	1.71	2.10	1.13	1.33
14	1.70	2.11	0.97	1.59
18	1.43	2.06	1.13	2.34

Table 3: MKL runtime versus PCR kernel execution time in milliseconds for 12000 systems

Dim	MKL	PCR GPU	Speedup
32	13.360	0.974	13.714
64	23.739	1.731	13.712
128	47.028	3.634	12.941
256	89.076	7.927	11.237
512	177.304	19.069	9.298

Single precision versus double precision

The current GPU generation features substantially more throughput in single precision than in double precision. Our numerical experiments show that, for practical applications, the accuracy of single precision is usually sufficient. The stability and accuracy of the solver benefits from the following implementation decisions.

1. The proper parallel algorithm for solving tridiagonal systems must be selected. Our implementation of the parallel cyclic reduction does not provide pivoting, so it can exhibit numerical instabilities for general tridiagonal systems. However, it is stable for diagonally dominant matrices, which occur from finite difference discretization of diffusion-dominated PDEs. Zhang et al. (2010) found that the recursive doubling algorithm does not achieve a good accuracy and may even suffer from overflow.

2. The time grid and discretization nodes of the finite difference scheme must have proper concentration and alignment of grid points to avoid the propagation of oscillation effects in the solution.

3. Accumulation of discrete dividends in small time intervals lead to many jump discontinuities, which adversely affect the accuracy of the result. In such a case, the PDE is better solved in the transformed coordinates, as introduced in Egloff (2010), which remove the jump discontinuities completely.

For a multi-GPU configuration, the task-splitting strategy is very important to achieve scalability in the number of GPUs

Table 4: MKL runtime versus PCR kernel execution time in milliseconds for dimension 256

Systems	MKL	PCR GPU	Speedup
300	2.287	0.405	5.646
600	4.585	0.618	7.414
1200	9.022	0.987	9.136
6000	44.675	4.076	10.959
9000	67.835	5.992	11.321
12000	89.076	7.927	11.237

Table 5: MKL runtime versus PCR kernel execution time in milliseconds for dimension 256. All data stored in global memory

Systems	MKL	PCR GPU	Speedup
300	2.287	7.391	0.309
600	4.585	8.773	0.523
1200	9.022	11.123	0.811
6000	44.675	30.973	1.442
9000	67.835	43.134	1.573
12000	89.076	56.203	1.585

4. Use higher-order finite difference schemes to calculate Greeks and sensitivities on the grid of discretization nodes.

Performance evaluation

Tridiagonal solver

First, we compare the Forsythe-Moler version of Gaussian elimination described by Forsythe and Moler (1967) and the serial version of the cyclic reduction algorithm against the SSE-optimized Intel MKL solver sgtsv and dgtsv in single and double precision. The following benchmark (Table 1) is executed on an Intel Core 2 Duo CPU T9600 at 2.8 GHz. All tests have been built with the Microsoft 32-bit C++ compiler with full optimization. It is interesting to note that the Forsythe–Moler algorithm in double precision has the same performance as dgtsv. Cyclic reduction algorithms are interesting candidates for parallel solvers; already the serial version performs remarkably well (see Table 2).

For all subsequent tests and benchmarks, we have used an Intel Core 2 Quad CPU Q6602 system, running at 2.4 GHz, with two NVIDIA Tesla C1060 GPUs. All test have been built with the Microsoft 32-bit C++ compiler and CUDA version 2.3.

Table 6: European option, dimension 128. Timings are measured in milliseconds

Problem size	Timing CPU	Timing 1 GPU	Speedup 1 GPU	Timing 2 GPUs	Speedup 2 GPUs	GPU Scaling
300	582	33	17.6	28	20.8	1.2
600	1149	53	21.7	45	25.5	1.2
900	1707	76	22.5	61	28.0	1.2
1200	2277	94	24.2	72	31.6	1.3
1800	3437	141	24.4	101	34.0	1.4

Table 7: European option, dimension 256. Timings are measured in milliseconds

Problem size	Timing CPU	Timing 1 GPU	Speedup 1 GPU	Timing 2 GPUs	Speedup 2 GPUs	GPU Scaling
300	1091	52	21.0	41	26.6	1.3
600	2204	93	23.7	72	30.6	1.3
900	3254	132	24.9	91	35.8	1.4
1200	4405	174	25.1	120	36.7	1.5

Table 8: European option, dimension 512. Timings are measured in milliseconds

Problem size	Timing CPU	Timing 1 GPU	Speedup 1 GPU	Timing 2 GPUs	Speedup 2 GPUs	GPU Scaling
300	2192	106	20.7	73	30.0	1.5
600	4400	189	23.4	119	37.0	1.6
900	6603	274	24.1	172	38.4	1.6

The performance is quite impressive, even though GPUs are generally not optimized for solving tridiagonal systems

We compare the SSE optimized MKL solver sgtsv and the kernel execution time of our fully optimized parallel cyclic reduction solver from Egloff (2010). We do not include memory transfer in the comparison because in realistic application settings, the tridiagonal solver is usually part of a larger algorithm, which also sets up the tridiagonal systems to be solved directly on the GPU. Table 3 displays timings for a large problem set of 12,000 equations of dimensions 32 up to 512. The restriction of the dimension to less than 512 allows the tridiagonal solver to fully exploit shared memory and to run a maximum number of threads per block. The performance is quite impressive, even though GPUs are generally not optimized for solving tridiagonal systems. It is interesting to see that small systems are solved visibly more efficiently.

The size of the problem set is crucial as well, which can be seen in Table 4. If there is not enough workload on every individual multiprocessor, then the memory access latency cannot be hidden properly.

The timings in Tables 3 and 4 are very much in line with those obtained in Zhang et al. (2010), where a speedup of a factor of 12 over an optimized tridiagonal solver on the CPU is reported. Table 5 documents the importance of shared memory. We execute the kernel in such a way that the three vectors of the tridiagonal system and the right-hand side are placed in global memory. The timings show that a proper usage of shared memory is mandatory to achieve superior performance.

PDE solver

We benchmark the efficiency of our parallel GPU-based PDE solver on a set of European put and call options of different maturities and strikes. The finite difference scheme uses 50 time steps per year. We measure overall execution time, including the time required to transfer data to and from

GPU device memory and the overhead for thread management in case of multiple GPUs. Tables 6 to 8 display the timings based on the fully optimized tridiagonal solver kernel, where all the vectors of the tridiagonal system fit into shared memory.

To gain further insight into the overall efficiency, it is interesting to analyze the runtime cost for the data transfer. Because we minimize the number of memory copy operations by packing data into large blocks, we find that only 2–4 percent of the overall execution time is required for data transfer, which also contains the conversion from double to single precision.

Our tridiagonal solver is highly optimized for systems of dimension 512 or less. Larger systems require a different solution strategy. We implemented an alternative solver which subsequently places data into global memory if shared memory is running out. It is based on a dynamic kernel-dispatching methodology, which dispatches to the optimal kernel based on the tridiagonal system dimension, the amount of shared memory, the register count, and the desired number of threads per block. For example, increasing the dimension to 768 requires two vectors out of five to be placed in shared memory, and the performance increase drops from 23 to a factor of 12. This shows that the dynamic kernel-dispatching method performs well for a wide range of problem sizes but is by far not optimal.

So far, we have only considered the pricing of European options. The benchmark results for barrier options are very much similar. The pricing of American options with a parallel operator splitting method, which resolves the nonlinear early exercise constraints row by row and fully data-parallel, shows even better performance figures.

Conclusions

The peculiarities of the GPUs and their physical hardware limitations make the development of high-performance general purpose solvers for one-dimensional finite difference schemes a challenging task. Our benchmark results prove that, with the proper algorithm and a well-designed GPU resource management, it is possible to implement solvers which perform exceptionally well in the range of discretization nodes from 128 to 512. Fortunately, most realistic problems can be handled within that range. For more than 512 nodes, our dynamic kernel-dispatching method has a performance degradation of about 60 percent. The performance figures clearly document the importance of a large problem set of at least 300 or more pricing problems.

It is possible to implement solvers which perform exceptionally well in the range of discretization nodes from 128 to 512

Further applications

Our tridiagonal solver can also be applied to implement alternating direction methods for higher-dimensional PDEs. Recently developed finite difference schemes are capable of treating the cross-derivative terms properly (see Craig and Sneyd, 1988; in 't Hout and Welfert, 2007, 2009).

Local volatility calibration, as developed for instance by Andersen and Brotherton-Ratcliffe (1997), also boils down to a series of tridiagonal systems. Because the tridiagonal systems are decoupled over time, we find additional parallelism, which we can exploit with a different thread mapping.

An interesting approach is to combine the local volatility calibration with the finite difference solver for pricing. The first advantage is that the input data are reduced significantly because instead of transferring a possibly large local volatility matrix, only a few implied volatility slices have to be copied to the GPU. Caching the local volatility surface directly on the GPU and reusing it for multiple calculation further enhances the overall performance. The advantage is even more pronounced if the Greeks are calculated by taking volatility smile dynamics into account, which would require a recalibration of the local volatilities after shifting the spot value.

Yet another application comes from spline interpolation and spline smoothing, which are often applied to preprocess implied volatility smiles before passing them to local volatility calibration.

Acknowledgments

The author would like to thank Gernot Ziegler and Richard Rossmanith for valuable feedback on an early draft of this article.

REFERENCES

Andersen, L.B.G. and Brotherton-Ratcliffe, R. 1997. The equity option volatility smile: An implicit finite-difference approach. *Journal of Computational Finance* 1(2), 5–37.

Craig, I.J.D. and Sneyd, A.D. 1988. An alternating-direction implicit scheme for parabolic equations with mixed derivative terms. *Computers & Mathematics with Applications* 16, 341–350.

Egloff, D. 2010. Part I: High-Performance Tridiagonal Solvers on GPUs. *Wilmott* September, 32–40.

Forsythe, G.E. and Moler, C.B. 1967. *Computer Solution of Linear Algebraic Systems*, Prentice-Hall.

in't Hout, K.J. and Welfert, B.D. 2007, Stability of ADI schemes applied to convection diffusion equations with mixed derivative terms. *Applied Numerical Mathematics* 57, 19–35.

K.J. in't Hout, and Welfert, B.D. 2009. Unconditional stability of second order ADI schemes applied to multi-dimensional diffusion equations with mixed derivative terms. *Applied Numerical Mathematics* 59, 677–692.

Zhang, Y., Cohen, J. and Owens, J.D. 2010. Fast tridiagonal solvers on the GPU. Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010), p. 10.