GPU-Enabled Steady-State Solution of Large Markov Models

Bruno R.C. Magalhães, Nicholas J. Dingle and William J. Knottenbelt
Department of Computing, Imperial College London,
South Kensington Campus, SW7 2AZ, United Kingdom.
Email: {brc08,njd200,wjk}@doc.ic.ac.uk

***Abstract*—We describe a novel parallel steady-state solver that uses NVIDIA's Compute Unified Device Architecture (CUDA) library to perform calculations on a graphics processing unit (GPU). We demonstrate speed-ups of over 8 times compared with a CPU-only solver. We also discuss a parallel implementation which runs on multiple GPUs on separate machines, and explain how we deal with allocating appropriate amounts of work to heterogeneous computing resources.**

I. Introduction

More cores, not faster clock speeds, drive performance enhancement in today's processors. Due to the recent increase in the number of cores on a computer processor and the introduction of graphics boards with their own cores and memory, nowadays most of the applications which were written for single or dual core processors do not take full advantage of the processing power of the machine, as they were written for single-threaded and sequential execution modes. Therefore, we stand in a period where graphics processing units (GPUs) and CPUs with multiple cores are left idle by most of the applications. Recent developments such as NVIDIA's Compute Unified Device Architecture (CUDA) library have, however, lowered the barrier of entry to writing programs that tap into the high floating-point performance offered by GPUs.

Within the domain of performance analysis there has been some work on harnessing the power of GPUs for computation. Such work has focused mainly on extracting qualitative measures from models, for example with model checking [1], [2], but to the best of our knowledge there has been little or no work on the computation of quantitative measures such as steady-state probabilities or passage-time distributions.

Such measures are typically computed by constructing and solving a system of linear equations. A great deal of work has been conducted on solving dense systems of equations with GPUs, but the solution of sparse systems, which more often arise from high-level performance model descriptions, has been less widely studied [3].

The aim of this paper is to present our novel work on the solution of steady-state probabilities in Continuous Time Markov Chains (CTMCs) using GPUs. After briefly describing the background theory and NVIDIA's CUDA library (Section II), we describe our GPU-enabled steady-state solver in Section III. We discuss how we deal with allocating appropriate amounts of work given the heterogeneous nature of systems composed of GPUs and CPUs, and present results which show that our GPU implementation can be over 8 times faster than a CPU-based solver. In Section IV we then consider expanding our implementation to run on multiple physical machines using the Message Passing Interface (MPI) [4] standard, and present results to show the run-time and speed-up of the resulting parallel implementation. Finally, we conclude and suggest avenues for future work.

II. Background

We briefly describe the theoretical background to the solution of CTMCs for their steady-state probability distributions, before presenting an overview of NVIDIA's CUDA library.

A. Parallel Steady State Solution

For a finite, irreducible and homogeneous CTMC, the steady-state probabilities $\{\pi_j\}$ always exist and are independent of the initial state distribution. They are uniquely given by the solution of the equations:

$$-q_{jj}\pi_j + \sum_{k\neq j} q_{kj}\pi_k = 0 \quad \text{subject to} \quad \sum_i \pi_i = 1$$

This can be expressed in terms of the vector $\boldsymbol{\pi}$ (with elements $\{\pi_1, \pi_2, \ldots, \pi_N\}$) and the matrix $\mathbf{Q}$ as:

$$\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}$$

There are a number of well-known iterative techniques for computing steady-state probabilities in Markov chains. These techniques are used for solving linear systems of the form $\mathbf{Ax} = \mathbf{b}$; in the case of CTMC analysis, $\mathbf{A} = \mathbf{Q}^T$, $\mathbf{x} = \boldsymbol{\pi}^T$ and $\mathbf{b} = \mathbf{0}$ (a vector with all elements being 0), where the superscript T denotes the transpose operator.

Probably the most easily-parallelisable iterative solution technique is the Jacobi method. This is based on the observation that solving $\mathbf{Ax} = \mathbf{b}$ is equivalent to finding the solution to the n equations:

$$\sum_{j=0}^{n-1} a_{ij}x_j = b_i \quad i = 0, 1, \ldots, n-1$$

Solving the ith equation for x_i yields:

$$x_i = \frac{1}{a_{ii}}\Big(b_i - \sum_{j\neq i} a_{ij}x_j\Big)$$

which suggests the iterative method:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\Big(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\Big) \tag{1}$$

where $k \geq 0$ and $x^{(0)}$ is an initial guess at the solution vector. Iterations are performed until the normalised difference between successive iterations is less than some predefined error ε:

$$\frac{||x^{(k+1)} - x^{(k)}||_\infty}{||x^{(k+1)}||_\infty} < \varepsilon$$

Here, $||x||_\infty$ is the infinity-norm given by $||x||_\infty = \max_i |x_i|$.

In the Jacobi method the calculation of the $x_i^{(k)}$'s are independent of one another, which means vector element updates can be performed in parallel, using a data access pattern similar to sparse matrix-vector multiplication.

B. CUDA

CUDA is NVIDIA's parallel computing platform for GPUs. It was released in November 2006 and allows developers to write programs for GPUs in a very similar way to CPUs. It is essentially a C library for writing GPU-enabled programs, albeit one with several limitations (e.g. no recursive functions) on the structure of the GPU code. A CUDA program is formed of standard C/C++-code (compiled for the architecture of the CPU), and one or more "kernels" written in the CUDA subset of C that execute on the GPU when called by the functions running on the CPU.

Each CUDA kernel in a program will be executed many times in parallel by a large number of threads. Threads are grouped into blocks and are able to synchronise with other threads in the same block, but are not able to synchronise with threads in other blocks. The programmer has no control over the order in which these blocks of threads will be executed on the available GPU cores. Unlike a CPU, an NVIDIA GPU contains a large number of cores (e.g. 480 on the top-of-the-range NVIDIA GeForce GTX 295 card [5]) each of which executes one block of threads at a time, and can switch between the constituent threads of a block at low cost. This Single Instruction Multiple Thread (SIMT) execution model allows CUDA programs to scale seamlessly across different hardware with varying numbers of cores.

In addition, the GPU has a memory hierarchy which must be explicitly dealt with by the programmer [5]:

- **Private local memory** is the fastest type, but can only be accessed by a single thread and does not persist beyond the lifetime of that thread.
- **Shared memory** is shared between all threads in the same block. It is much faster to access than global memory (see below), but is limited in capacity.
- **Global memory** is the "main memory" of the graphics card and is available to all threads in all blocks. It is, however, the slowest form of memory to access.
- **Texture memory** is a read-only binding of global memory that is well-suited to storing and retrieving data with a high level of two-dimensional spatial locality. It is cached.

III. GPU Implementation

As described above, one of the central concepts of CUDA programming is that the same kernel is executed in parallel by a large number of threads. For Jacobi, the central operation is the multiplication of one row of the matrix with the current solution vector (i.e. Eq. 1) and this therefore formed the kernel of our implementation. Each thread on the GPU would then be responsible for multiplying one row of the matrix with the current solution vector. It will be recalled from the description of the Jacobi method above that the update of each $x_i^{(k)}$ value is independent of the others in the same iteration, and therefore our kernel satisfies the CUDA programming requirement that it must not matter in what order the threads are executed.

Our matrix is stored in Compressed Sparse Row format [6] as three one-dimensional vectors: a vector of non-zero weights (the q_{ij} values), a vector of column indices and a vector of offsets for the row starts. We also explicitly store the $\mathbf{x}^{(k)}$ vector and the $\mathbf{x}^{(k+1)}$ vector.

To ensure good performance of our implementation we have exploited the memory hierarchy of the GPU. The matrix must be stored in the GPU's global memory as it is too large to fit in the shared memory available to each thread block. As it is not altered by the Jacobi method, however, we can bind it into read-only texture memory and this results in a slight performance improvement even with the overhead of binding/unbinding taken into account.

We achieved a much larger performance improvement by storing the $\mathbf{x}^{(k+1)}$ vector in shared memory. As the ith element of this is only accessed by the thread responsible for multiplying row i with the current solution vector, there was no requirement for threads outside the block containing the thread responsible for row i to access $\mathbf{x}_i^{(k+1)}$ – had there been then shared memory would not have been appropriate and global memory would have had to have been used. We recorded an approximately 20% reduction in solution times by storing the $\mathbf{x}^{(k+1)}$ vector in shared (as opposed to global) memory.

A. Dealing with large matrices

A major limitation on the use of standard GPUs for computation is the limited amount of memory on the graphics card. For example, the NVIDIA GeForce 8600GTS cards used to generate the results presented in this paper have 256MB of main memory. It should be noted also that this is the total amount of memory available, and that other applications (especially graphical ones) may consume a portion of this, further reducing the amount available to use for steady-state solution. Furthermore, the largest amount of memory available to be allocated with a single `malloc()` call will often be appreciably smaller; in our experiments, of the 256MB of total memory on the graphics card, approximately 180MB was consistently available for our steady-solver but no more than 20MB of contiguous memory at a time could be allocated.

For small matrices, this was not a problem. For those which we are interested in analysing, which may have many millions of states, this presented a major problem. The solution was to split the matrix into smaller portions and then perform the computation on each portion in turn, by first copying the portion from system memory to the GPU, perform one iteration of Jacobi on it and then copying the updated vector elements back to system memory.

The major drawback of this scheme was that main memory to GPU copies are extremely time-consuming, and as a result the performance of our steady-solver was unacceptable. To overcome this, we have implemented a scheme inspired by the use of asynchronous iterations in distributed memory steady-state solution [7], [8], [9]. Rather than perform only one iteration on a block of the matrix per copy, we instead perform multiple iterations with the copied data before proceeding to the next matrix portion. This reduces the amount of copy operations at the expense of requiring more iterations to converge, but as copying data to and from the GPU is far more expensive than performing computation with the data, the result is an overall improvement in the run-time of the steady-state solver – see Section III-C for more details.

B. Resource discovery

Our intention in this work was to produce a steady-state solver capable of making maximum use of all available computing resources (CPU and GPU). As these resources will have differing capabilities it is not sufficient to delegate equal amounts of work to each. We have therefore implemented a resource discovery mechanism that tests all available processing elements to determine their relative computing power and assign portions of the matrix to each accordingly.

From `/proc/cpuinfo` we are able to determine the number of CPU cores in the machine on which we are running, while the CUDA library contains built-in functions to enable us to extract the characteristics of the installed GPUs. As the raw clock rates of CPUs and GPUs are not comparable, we instead determine their relative processing power empirically by conducting a number of sample Jacobi iterations on each processing element. We take the average duration per iteration of each core, and calculate the amount of matrix rows per second that each core is able to process. We then partition the matrix according to the percentage of processing power each core has when compared to the total processing power (the sum of the rows per second of all cores).

C. GPU vs. CPU Results

Table I shows run-time and speedup results for our steady-state solver running on an Intel Core2 Duo 2.13GHz with 2GB of memory and fitted with an NVIDIA GeForce 8600GTS graphics card with 256MB of memory. As this is a dual-core machine, we compare our solver's performance running on a single core of the CPU, both cores of the CPU, the GPU alone and finally on all the cores available in the machine (i.e. both CPU cores and also the GPU). Results are presented for three vector exchange frequencies (1, 10 and 20 iterations) – note that convergence checks are conducted at the same range of frequencies, which accounts for the three entries for the single CPU core case that otherwise features no vector exchanges.

We observe that the speed-up achieved using only the GPU (compared with using only a single CPU core) exceeds that achieved by using both CPU cores for all vector exchange frequencies for all CTMC sizes, although the effect is most marked for vector exchange frequencies of 10 and 20 iterations. This illustrates the size of the overhead imposed by main-memory-to-GPU copies.

Combining both CPU cores and the GPU results in performance which usually matches or, in some cases, exceeds that of using only the GPU. The occasions where this scheme under-performs compared with the GPU-only scheme may be attributable to our resource-discovery mechanism: although we partition the matrix between the cores according to their observed performance, we have no way of enforcing that the thread of our solver that has the larger portion of the matrix is assigned to the same core that ran the test iterations faster. This means that more work end up being given to a more heavily loaded core, and hence that overall performance may suffer. Rectifying this is a matter for future work. Copying data from the CPU to the GPU is also a significant bottleneck as it requires the whole execution (on both CPUs cores and on the GPU) to pause.

IV. GPU and MPI Implementation

So far, we have described a scheme centred on the use of a single machine containing one or more CPU cores and a GPU. Using MPI we are able to launch multiple instances of our GPU-enabled solver on a number of machines, which can then work together to solve CTMCs with state spaces too large to be held within the memory of a single machine. Data is exchanged between machines at the same frequency that it is copied between main memory and the GPU on each machine – as described above, this is not necessarily after the completion of each iteration. We assign blocks of contiguous matrix rows to participating machines, and then determine how much of the portion assigned to each machine should be given to the computational cores within that machine using our resource discovery method described above.

Table II shows run-time and speedup results for our steady-state solver running on two Intel Core2 Duo 2.13GHz with 2GB of memory and both fitted with an NVIDIA GeForce 8600GTS graphics card with 256MB of memory. We note that the speed-ups here are much lower than those achieved for the solver running on a single machine (see Section III-C), and we believe that this is due to the overheads incurred in exchanging vector elements between the participating machines after each iteration. Work is currently on-going to investigate the use of hypergraph partitioning [10], [11] to reduce the amount of CPU-to-GPU and inter-machine communication required.

V. Conclusion

We have described the implementation of a steady-state solver for CTMCs that uses the GPU to compute results up to 8 times faster than a CPU-only solver. We have also described a parallel extension of this solver that uses MPI to distribute the work across a number of machines, each of which runs our

No. of States	Configuration	Run-time (seconds)			Speed-up		
		1	10	20	1	10	20
501 501	1x CPU core	6.82	7.29	6.55	-	-	-
	2x CPU cores	4.59	3.40	3.66	1.49	2.14	1.93
	1x GPU	3.15	1.02	0.96	2.17	7.15	6.82
	2x CPU cores + 1x GPU	2.45	1.29	1.27	2.78	5.65	5.16
2 003 001	1x CPU core	26.15	29.60	26.37	-	-	-
	2x CPU cores	14.58	14.18	14.40	1.79	2.09	1.83
	1x GPU	11.57	5.91	5.67	2.26	5.01	4.65
	2x CPU cores + 1x GPU	8.53	4.51	3.90	3.07	7.59	8.59
8 006 001	1x CPU core	117.42	114.88	107.41	-	-	-
	2x CPU cores	58.17	57.36	55.23	2.02	2.00	1.94
	1x GPU	48.83	17.39	15.54	2.40	6.61	6.91
	2x CPU cores + 1x GPU	33.20	17.82	16.75	3.54	6.45	6.41

TABLE I
RUN-TIME AND CORRESPONDING SPEED-UPS ON ONE MACHINE, FOR THREE DIFFERENT VECTOR EXCHANGE FREQUENCIES.

No. of States	Configuration	Run-time (seconds)			Speed-up		
		1	10	20	1	10	20
4 504 501	2x1 CPU	15.87	12.13	12.53	-	-	-
	2x2 CPU	12.93	8.21	8.03	1.22	1.48	1.56
	2x1 GPU	15.37	15.442	14.02	1.03	0.79	0.89
	2x2 CPU + 2x1 GPU	13.90	15.06	19.83	1.14	0.81	0.63
12 507 501	2x1 CPU	39.17	33.46	34.85	-	-	-
	2x2 CPU	27.16	18.86	19.44	1.44	1.77	1.79
	2x1 GPU	43.79	19.43	18.76	0.89	1.72	1.86
	2x2 CPU + 2x1 GPU	32.84	14.11	15.51	1.19	2.37	2.25
24 510 501	2x2 CPU	50.21	41.41	45.58	-	-	-
	2x1 GPU	87.02	39.85	38.61	0.58	1.04	1.18
	2x2 CPU + 2x1 GPU	56.52	31.07	24.80	0.89	1.33	1.84

TABLE II
RUN-TIME AND CORRESPONDING SPEED-UPS USING TWO MACHINES, FOR THREE DIFFERENT VECTOR EXCHANGE FREQUENCIES.

GPU-enabled solver. As these machines may have differing computational power, we have described a resource discovery mechanism that assigns an appropriate amount of work to each based on their measured performance.

We are aware that there are still many opportunities for further work that will improve the performance of our implementation. As described above, we will investigate the scheduling of threads on the CPU to devise a method for ensuring that the correct amount of work is assigned to each CPU core. We will also investigate the use of hypergraph partitioning and a better CUDA-to-MPI interface to more efficiently parallelise our calculations across multiple machines. In addition, we will investigate more advanced sparse matrix layouts (e.g. those described in [3], [12]). Our ultimate intention is to use GPUs for the analysis of more complex measures such as response time distributions, which can also be computed with iterative sparse matrix-vector multiplication methods [13], [14].

REFERENCES

[1] J. Barnat, L. Brim, and M. Češka, "DiVinE-CUDA - a tool for GPU accelerated LTL model checking," in *Proceedings of the 8th International Workshop on Parallel and Distributed Methods in Verification (PDMC'09)*, Eindhoven, November 2009, pp. 107–111.

[2] D. Bosnacki, S. Edelkamp, and D. Sulewski, "Efficient probabilistic model checking on general purpose graphics processors," in *Proceedings of the 16th International SPIN Workshop on Model Checking of Software (SPIN'09)*, Grenoble, June 2009, pp. 32–49.

[3] L. Buatois, G. Caumon, and B. Lévy, "Concurrent Number Cruncher: An efficient sparse linear solver on the GPU," in *Proceedings of the 3rd International Conference on High Performance Computing and Communications (HPCC'07)*, Houston, September 2007, pp. 358–371.

[4] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd ed. Cambridge, Massachussetts: MIT Press, 1999.

[5] "NVIDIA CUDA programming guide," February 2010, version 3.0.

[6] I. Duff, A. Erisman, and J. Reid, *Direct Methods for Sparse Matrices*. Oxford: Clarendon Press, 1986.

[7] K. Blathras, D. Szyld, and Y. Shi, "Timing models and local stopping criteria for asynchronous iterative algorithms," *Journal of Parallel and Distributed Computing*, vol. 58, no. 3, pp. 446–465, September 1999.

[8] N. Dingle and W. Knottenbelt, "Distributed solution of large Markov models using asynchronous iterations and graph partitioning," in *Proceedings of the 18th UK Performance Engineering Workshop (UKPEW'02)*, Glasgow, July 2002, pp. 27–34.

[9] A. Frommer and D. Szyld, "On asynchronous iterations," *Journal of Computational and Applied Mathematics*, vol. 123, pp. 201–216, 2000.

[10] C. Berge, *Graphs and Hypergraphs*. North-Holland, Amsterdam, 1973.

[11] U. Catalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, July 1999.

[12] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *Proceedings of the 5th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC'10)*, Pisa, January 2010, pp. 111–125.

[13] N. Dingle, P. Harrison, and W. Knottenbelt, "Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, pp. 908–920, August 2004.

[14] J. Bradley, N. Dingle, W. Knottenbelt, and H. Wilson, "Hypergraph-based parallel computation of passage time densities in large semi-Markov models," *Linear Algebra and its Applications*, vol. 386, pp. 311–334, 2004.