GPU-based solution of Continuous Time Markov Chains using CUSP

Dingle, Nicholas

2011

MIMS EPrint: **2011.102**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

GPU-based solution of Continuous Time Markov Chains using CUSP

Nicholas J. Dingle
School of Mathematics, University of Manchester
nicholas.dingle@manchester.ac.uk

November 29, 2011

Abstract

This technical report describes the parallelisation of the response-time analyser HYDRA using CUSP and the results of executing it on HECToR's GPGPU testbed. We achieved good speed-ups in execution time, but these were outweighed by increased setup time.

It is important to ensure that computer and communication systems will meet quality of service targets. Ideally, it should be possible to determine whether or not this will be the case at design time. This can be achieved through the modelling and analysis of the system in question. Such analysis can be conducted by capturing the behaviour of the system with a formal model; that is, identifying the possible states the system may be in and the way in which it can move between these states. The concept of time can be introduced by associating delays with the state transitions. When the choice of the next state depends only on the current state and transition delays are sampled from the negative exponential distribution, we call such a model a Continuous Time Markov Chain (CTMC).

As specifying every state and transition in the state space of a complex model of a real-life system is infeasible, high-level formalisms such as stochastic Petri nets [1], stochastic process algebras [6] and queueing networks [7] can be employed. These permit a succinct description of the model from which a CTMC can automatically be extracted and then solved for performance measures of interest.

From the steady-state probability distribution of the model's underlying CTMC, standard resource-based performance measures, such as mean buffer occupancy, system availability and throughput, and expected values of various sojourn times can be obtained. Steady-state measures allow the answering of questions such as: "What is the probability that the system will be in a failure state in the long run?" and "What is the average utilisation of this resource?". The steady-state probability distribution of a CTMC is calculated by solving a set of linear equations of the form $Ax = b$. In this case, A is a (sparse)

transposed generator matrix which has zero column sums, x is a probability vector whose elements sum to 1, and $b = 0$ (a vector of zeros).

There exist a large number of tools designed to calculate steady-state probabilities in CTMCs, but very few of them exploit recent advances in General Purpose GPU (GPGPU) computing. In this note we report on our experience of taking the existing CTMC steady-state solver HYDRA (HYpergraph-based Distributed Response Time Analyser) [4, 5] and parallelising it using the CUSP library for execution on NVIDIA GPUs. We have previously achieved good speed-ups on multicore machines by parallelising HYDRA with OpenMP [3].

1 Implementation

The workflow for the HYDRA analysis pipeline proceeds as follows: a high-level model description (specified in a TeX-like input language) is first parsed and then the resulting C++ code is compiled and executed with the state generator to produce the CTMC's generator matrix. This matrix is then read in by the steady-state solver and used to compute the model's steady-state probability vector.

As the existing HYDRA serial steady-state solver takes as input a file containing A, we were able to replace it with a solver which was written using CUSP's built-in I/O and Krylov subspace routines. We modified HYDRA's state generator to output A in Matrix Market format, as used by CUSP, rather than the proprietary format used by HYDRA; this was easy to accomplish as the Matrix Market format only requires the matrix to be specified in coordinate format. This matrix was then used as input to CUSP's built-in BiCGStab (biconjugate gradient stabilised) [8] routine to solve $Ax = b$. The entire source code of the CUSP solver (minus the timing and solution verification code) is shown in Figure 1 and closely follows the examples provided in the CUSP documentation.

2 Results

		CUSP (GPU)			CUSP (CPU)	HYDRA (CPU)		
Rows	Non-zeros	Set-up	Solve		Set-up	Setup	Solve	
			Iterations	Time			Iterations	Time
11 700	71 724	1.80	87	0.27	0.60	0.02	63	0.03
152 712	1 416 903	21.00	93	0.33	14.10	0.10	75	1.10
5 358 150	45 138 039	587.20	187	7.60	515.10	6.20	166	104.60
9 304 650	79 457 034	1 046.20	197	14.20	924.4	10.9	227	248.40
15 404 115	136 117 843	1 859.40	234	28.50	1 789.0	18.8	205	385.40

Table 1: Run-time in seconds and iteration counts for CUSP and HYDRA steady-state solution.

Table 1 compares the runtimes of the CUSP and HYDRA solvers. For the CUSP version, the setup and solution times were measured as the execution of the corresponding

```
#include <cusp/krylov/bicgstab.h>
#include <cusp/hyb_matrix.h>
#include <cusp/io/matrix_market.h>

// where to perform the computation
typedef cusp::device_memory DevMemorySpace;
typedef cusp::host_memory HostMemorySpace;

// which floating point type to use
typedef float ValueType;

int main(){

    // begin setup
    cusp::hyb_matrix<int, ValueType, DevMemorySpace> A;

    cusp::io::read_matrix_market_file(A, "./matrix.mtx");

    cusp::array1d<ValueType, DevMemorySpace> x(A.num_rows,
                                          1.0/(double)A.num_rows);

    cusp::array1d<ValueType, DevMemorySpace> b(A.num_rows, 0);

    cusp::verbose_monitor<ValueType> monitor(b, 1600, 1e-5, 1e-5);
    // end setup

    // begin solution
    cusp::krylov::bicgstab(A, x, b, monitor);
    // end solution

    return 0;
}
```

Figure 1: Source code for the CUSP solver.

code portions identified in the listing above; the times for HYDRA were measured for the analogous portions of that code. The input matrices used were generated from the well-known Courier [9] and FMS [2] models.

The CUSP results were produced on the HECToR GPGPU testbed, which comprises a quad-core Intel Xeon 2.4GHz CPU with 32GB RAM and 4 NVIDIA Fermi C2050 GPUs each with 3GB RAM. The HYDRA results were produced on an Intel Core2 3.0GHz quad-core CPU workstation with 8GB RAM. HYDRA is a single-threaded implementation that was compiled using `gcc 4.4.3` with the `-O3` flag. All entries in the table are the averages of 5 runs. Note that HYDRA uses double precision whereas CUSP uses single due to compilation problems attempting to use doubles with CUSP BiCGStab.

We generally observe that the solution time is faster on the GPU than the CPU, but that the setup overhead (reading the matrix file from disk and copying the data to the device) is higher for the GPU code. For all the matrices tested, the extra setup overhead on the GPU outweighed the improvement in solution time relative to the original CPU version.

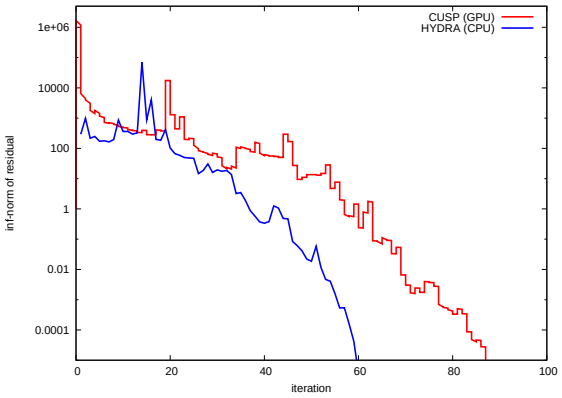
In an attempt to gauge the GPU data transfer overhead, we also carried out the calculations using CUSP but on the CPU not the GPU (achieved by putting the A, x and b in `HostMemorySpace` rather than `DevMemorySpace`). These results are shown in Table 1 under the "CUSP (CPU)" heading. The overhead in this case should therefore be only that of reading the matrix file from disk and populating the appropriate data structure in main memory. This gives us some indication of how much time could be saved by skipping the I/O phase (for example by maintaining A in memory when it is generated rather than writing it to and reading it from disk). The current high CUSP setup overheads ("CUSP(CPU)" v. "HYDRA(CPU)") are probably because CUSP reads the matrix from a text file, while HYDRA uses a proprietary binary storage format.

From Table 1 it is also noticeable that the convergence behaviour of the two implementations is different, with HYDRA generally requiring fewer iterations to converge than CUSP for the same problem size. Figure 2 provides more detail of this by plotting the ∞-norm of the residual at each iteration for both implementations for all problem sizes.

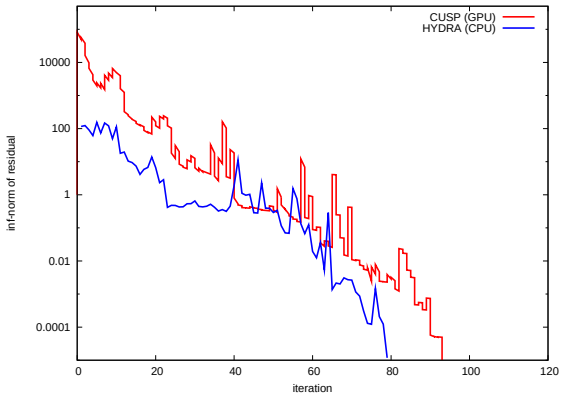
Rows	CUSP speed-up wrt HYDRA	CUSP (GPU) GFLOPS	CUSP (GPU) GFLOPS/s	HYDRA (CPU) GFLOPS	HYDRA (CPU) GFLOPS/s
11 700	0.11	0.03	0.13	0.02	0.82
152 712	3.33	0.56	1.71	0.45	0.41
5 358 150	13.65	37.97	5.00	33.71	0.33
9 304 650	17.49	69.89	4.92	80.52	0.32
15 404 115	13.52	139.55	4.90	122.27	0.32

Table 2: Performance of CUSP and HYDRA solvers.

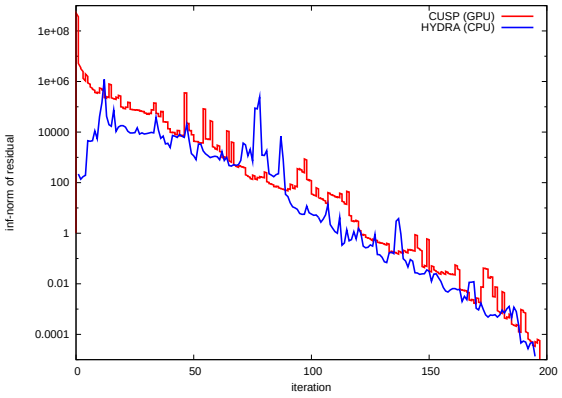
Based on the number of non-zeros and the details of the BiCGStab algorithm, we estimated the number of flops required to solve each of the five problem sizes. These results



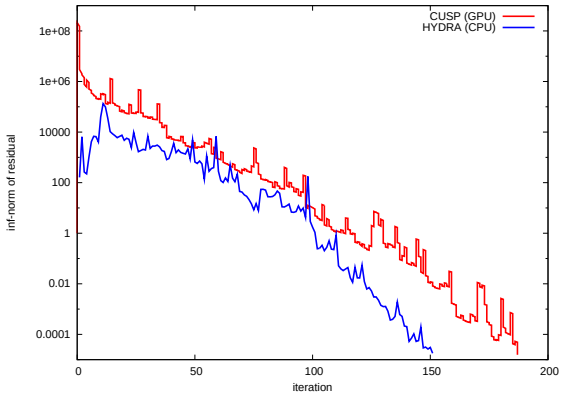
(a) 11 700 rows



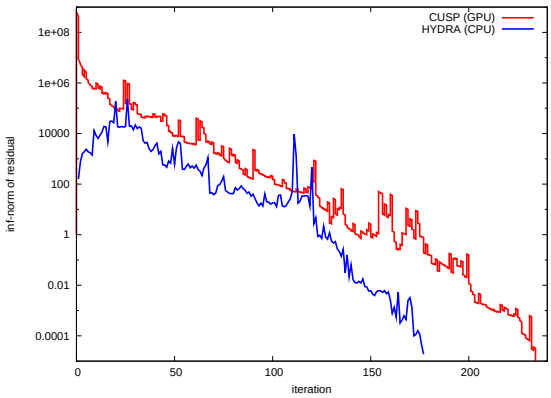
(b) 152 712 rows



(c) 9 304 650 rows



(d) 5 358 150 rows



(e) 15 404 115 rows

Figure 2: Convergence behaviour for each of the five problem sizes.

are shown in Table 2, along with the solution-time speedup of CUSP versus HYDRA. Our estimates give a performance of 5 and 0.3 GFLOPS/sec for CUSP on the GPU and HYDRA on the CPU respectively, for the 3 largest problems. These figures are far below the peak performance of either device.

3 Conclusion

We have implemented a GPU-based steady-state solver using the CUSP library, and have compared this with an existing serial CPU solver. We observed that the CUSP version solves the system of equations faster than the CPU version, but that the overhead involved in copying data to the GPU outweighs any runtime savings. There are still areas for improvement; for example getting closer to peak performance (perhaps via improved memory accessing), better CUSP error reporting and optimisation of CUSP's I/O routines.

References

[1] F. Bause and P.S. Kritzinger. *Stochastic Petri Nets – An Introduction to the Theory*. Verlag Vieweg, Wiesbaden, Germany, 1995.

[2] G. Ciardo and K.S. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[3] N.J. Dingle. HydraMP: Exploiting shared memory parallelism in HYDRA with OpenMP. In *Proceedings of the 27th UK Performance Engineering Workshop (UKPEW 2011)*, pages 203–214, Bradford, UK, July 2011.

[4] N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. HYDRA: HYpergraph-based Distributed Response-time Analyser. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, pages 215–219, Las Vegas NV, USA, June 23rd–26th 2003.

[5] N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models. *Journal of Parallel and Distributed Computing*, 64(8):908–920, August 2004.

[6] J. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, University of Edinburgh, 1994.

[7] Ng Chee Hock. *Queueing Modelling Fundamentals*. John Wiley and Sons, 1996.

[8] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 13(2):631–644, 1992.

[9] C.M. Woodside and Y. Li. Performance Petri net analysis of communication protocol software by delay-equivalent aggregation. In *Proceedings of the 4th International Workshop on Petri nets and Performance Models (PNPM'91)*, pages 64–73, Melbourne, Australia, December 1991.