Department of Electrical
and
Computer Systems Engineering

Technical Report
MECSE-11-2007

Exploitation of Hashing and Locality for Integral Sorting in Molecular Orbital Computations

T. Ramdas, G. Egan and D. Abramson

MONASH
UNIVERSITY

Exploitation of Hashing and Locality for Integral Sorting in Molecular Orbital Computations

Tirath Ramdas, Gregory Egan and David Abramson

***Abstract*— A software-based approach to perform run-time sorting of Electron Repulsion Integrals, with the aim of SIMD-fication of the the Hartree-Fock Self-Consistent-Field application, is presented. A hashing based approach is described. The performance of the scheme on a conventional processor is presented for various workloads. The results suggest a tradeoff between hash-efficiency and cache-efficiency. An optimal hash-table size is identified, which is suitable for various workloads. It is apparent that a hardware-based solution would be better suited for a high-throughput special-purpose computer, though some gains are possible with this generic software-based approach.**

Molecule	Basis Funcs.	ERI Threads
Water, H_2O	25	48k
Nitrous Oxide, N_2O	45	512k
Zinc Chloride, $ZnCl_2$	55	1.14M
Butane, C_4H_{10}	110	18.3M
Nicotine, $C_{10}H_{14}N_2$	250	488M
Caffeine, $C_8H_{10}N_4O_2$	260	571M

TABLE I
BASIS SET SIZE AND THREAD COUNT FOR VARIOUS MOLECULES

I. INTRODUCTION

The Hartree-Fock Self-Consistent-Field (HF-SCF) algorithm is a valuable tool in the field of computational quantum chemistry. Unfortunately, these and closely related computations suffer very poor scaling, between $O(N^2)$ to $O(N^7)$, depending on the particular level of accuracy required. In light of this massive computational requirement, and the potential gains to be made in fields such as pharmaceuticals and nanotechnology, we proposed that an application-specific computer be considered [1].

One of the most successful application-specific computers is the Protein Explorer [2], which targets large scale Molecular Dynamics (MD)[1]. At the heart of the Protein Explorer is the Gravity Pipe (GRAPE) special-purpose processor [3]. GRAPE is basically a Single-Instruction-Multiple-Data (SIMD) processor. We have observed that application-accelerators in the bioinformatics field tend to employ the systolic array architecture [4], which is a form of SIMD processing. Perhaps the most common SIMD processor today is in virtually every personal computer and gaming console – the Graphics Processing Unit (GPU), which finds widespread use outside the graphics application domain [5]. The Cell Broadband Engine also relies heavily on SIMD processing [6]. Traditionally the most prominent proponent of SIMD processing was vector processing; although vector processing has lost it's dominance in the supercomputing sphere, it is very much still alive [7].

SIMD processing is an attractive architecture because it is highly efficient for number crunching – the resources of the processor are dedicated more towards processing data and less towards control-oriented operations. However, as a result of this tradeoff, only certain applications may benefit from SIMD processing. Technically, applications that benefit the most from SIMD processing exhibit a substantial degree of Data-Level Parallelism (DLP). An application rich in DLP is an application which requires the processing of a large amount of data, however it is critical that large sets of data (i.e. vectors of data) be processed by exactly the same instruction sequence. Effectively, each operation (e.g. floating-point arithmetic) operates on a vector of data in parallel.

It would be desirable to employ SIMD processing for the HF-SCF application. Unfortunately, this is not immediately possible. The kernel of the HF-SCF algorithm, and indeed it's computational hotspot, is the evaluation of Electron Repulsion Integrals (ERIs) which are numerous and individually complex to compute. Table I lists some sample workloads and the number of ERIs required. Concurrent SIMD processing of multiple ERIs is not currently feasible because there are various classes of ERI, and different ERI classes require a different instruction sequence.

However, we have observed that for a given workload, within the entire set of ERIs there exists a significant number of ERI with identical class. Subsets of matching ERI could be constructed, and fed to a SIMD processor. The challenge is to do this sorting in a high-performance manner that does not negate the performance gains of the SIMD processor. We previously proposed that this operation be performed with a specialised memory structure [8].

In this report we investigate the feasibility of implementing the sorting facility on a conventional 64-bit RISC processor. We find that the performance does not approach the theoretical performance of the hardware approach previously proposed. Some insight into the balance of hash table size, vis-a-vis hashing efficiency vs. cache performance, is also presented.

T. Ramdas {tirath@int19h.com} is with the Center for Telecommunications and Information Engineering, Monash University.

G. Egan {greg.egan@eng.monash.edu.au} is with the Center for Telecommunications and Information Engineering, Monash University.

D. Abramson {david.abramson@infotech.monash.edu.au} is with the Center for Distributed Systems and Software Engineering, Monash University.

[1]Note that (MD) is roughly in the same scope as HF-SCF, however MD merely approximates electronic quantum behaviour, whereas HF-SCF accounts for these rigourously – and therefore is much more computationally expensive, but also much more accurate.

II. Design

The strategy employed with our software based ERI sorting solution is to perform early/eager subset creation by placing ERI quartets into ERI class subsets as soon as they are issued by the generating nested loop. Each subset is a container which will contain threads of identical class that may be computed with an identical instruction sequence. The space for each subset is pre-allocated for V threads – this may result in space wastage, however it avoids the relatively massive expense of dynamic memory allocation. In addition to each subset being pre-allocated, the number of subsets is also set a priori and pre-allocated, with no runtime dynamic memory allocation being used. Let the number of subsets be S. If one has S subsets and each subset has capacity for V quartets, then the system has the capacity to hold a maximum of $S \times V$ quartets. In order to mitigate the effects of the memory wall, specifically to increase the degree of spatial locality in the program, all S subset containers are allocated in contiguous chunks of memory (and each subset container is in turn allocated as a contiguous chunk of memory as well). We have chosen $V = 64$.

To achieve good SIMD unit utilisation one has to maximise the number of threads present in the system, since a larger population of threads increases the likelihood that there will be a sufficiently large number of threads with identical class. This requires a large S – we have found 1024 to be a good value. Unfortunately, with a conventional processor, this imposes a worst-case $O(1024)$ search operation when trying to match a new thread with it's corresponding subset. This problem can be mitigated through exploitation of temporal locality exhibited by the subsetting process, as well as use of *hashing*.

We have observed that subset mapping tends to exhibit a significant degree of temporal locality when the canonical ERI generation nested loop scheme [1] is employed. Basically, when a subset has been used, it is likely to be used again within a short period of time. To exploit this behaviour the history of subset access is maintained in the form of a queuing structure, such that whenever a new quartet is to be placed in a subset, the subset lookup/matching is performed in *most recently used* (MRU) sequence which would typically result in most lookups being completed in $\ll O(S)$ time.

The next optimisation is incorporation of a hashing function, which further reduces the lookup time and is particularly effective at "mopping up" the remaining subset lookups which do not benefit from exploitation of temporal locality. Hashing dramatically limits the number of subsets that need to be checked exhaustively. We propose that the MRU queue data structure be incorporated with this hashing approach. We use a CRC hash function. Our proposed system is illustrated in Fig. 1.

The design is implemented in C++ utilising the standard template library (STL); specifically, the MRU queues were implemented with vector containers. Each vector is reserved with *(S/HASH_TABLE_SIZE)+1*, though experimentation even with the most pessimistic reservation – i.e. *S* – did not affect performance significantly. Performance with deque containers instead of vector containers was noticably slower.

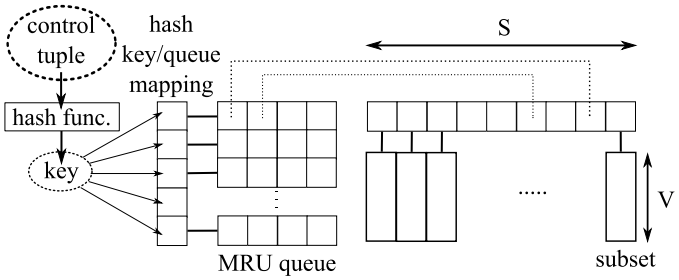

Fig. 1. ERI Sorting with conventional systems. The control tuple is a token comprising all the data that expresses the control-flow of the thread – a 64 bit control tuple is sufficiently large for this application.

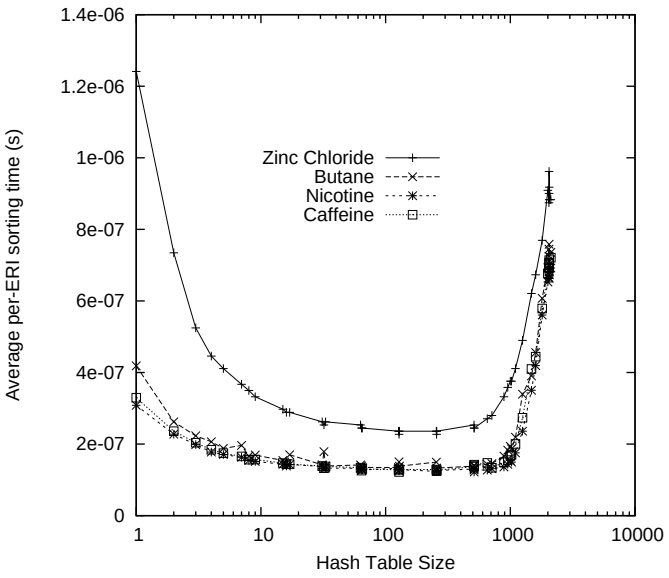

Fig. 2. Average per-thread sorting time vs hash table size.

III. Results

Fig. 2 depicts the average per-thread sorting time for various workloads on a 1.8GHz AMD Opteron based HP DL145 G3 system. Only a single core was used. The code was compiled with gcc, with the following switches: *-pipe -O3 -ffast-math -funroll-all-loops -fpeel-loops -ftracer -funswitch-loops -funit-at-a-time*. The best performance is around 120ns per integral – this is not a good result because 200ns is approximately the average ERI computation time on conventional cores. Therefore, a 120ns sorting time would not enable substantial speedup.

Performance tends to be better with larger workloads, up to a lower-bound. The more interesting observation is that the best sorting time is achieved with a hash table of 256 entries. This inflection point appears to be a consequence of a trade-off between hash efficiency and cache efficiency.

Perfect hashing would mean that a match/search operation could be performed in $O(1)$ time. It would also require a hash table size $\geq S$, and in this case $S = 1024$. Corresponding to our design, a large hash table implies shorter individual MRU queues, and a small hash table implies longer individual MRU queues. A very large hash table would result in greater hash efficiency – i.e. resolving collisions would only require a very

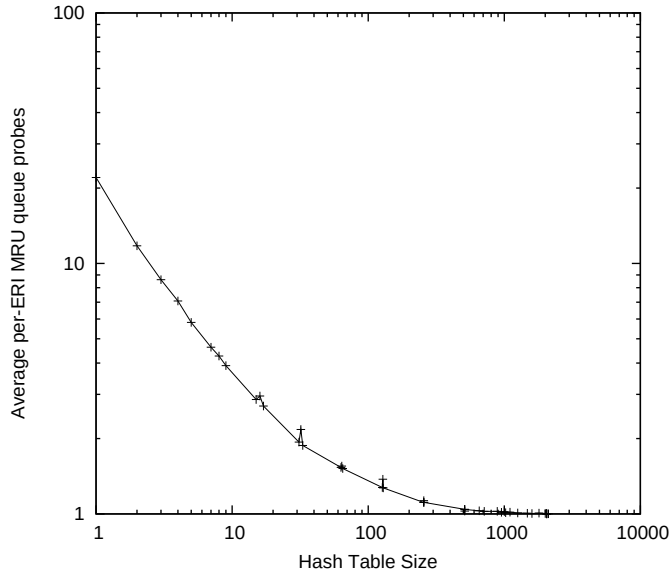

Fig. 3. Average per-thread MRU queue probes. Only the Nicotine workload is presented.

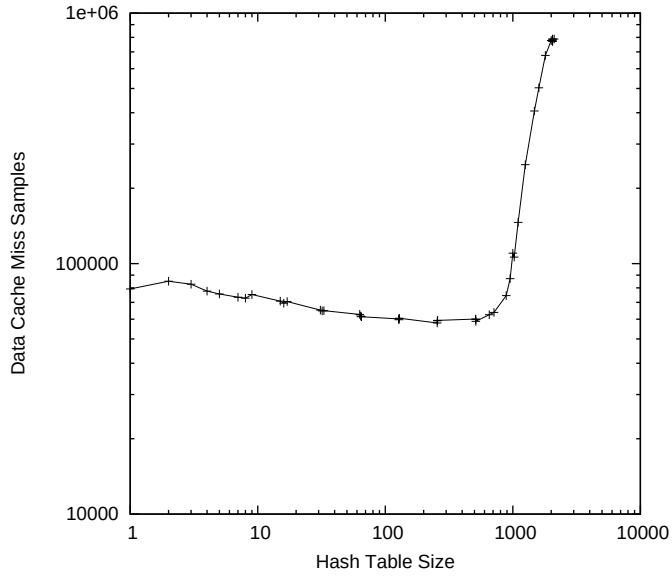

Fig. 4. Data cache misses (interval sampling). Only the Nicotine workload is presented.

short exhaustive search along the corresponding MRU queue. This is reflected in Fig. 3.

However, a very large hash table may result in inefficient use of caches in contemporary processors. Traversing the MRU queue is a spatially-local operation, and would be very amenable to wide cacheline pre-fetching policies. However, accesses to a hash table may be random. Furthermore a larger hash table involves a larger memory footprint, which could potentially lead to cache capacity misses. Therefore, although a large hash table would result in fewer memory access operations, the average cost of each operation increases. This is reflected in Fig. 4. Note that these results were obtained through performance counter sampling, and should not be taken as absolute values.

IV. Conclusion

The results have indicated that with high-throughput rapid sorting, there has to be a balance between hash-efficiency and cache-efficiency. For this application, with this architecture, and with CRC hashing – a hash table size of 256 is a good value. Further work should include repetition of these experiments on other architectures, and with other hashing functions.

For the purpose of development of a special-purpose computer system, it appears that a special hardware structure based on an associative search memory [8] may indeed be warranted.

Note

Execution time and MRU queue probing count numbers are averaged for all threads, cache misses are raw sample count. All results presented in this report were obtained on *marklar* with sources tagged "SOFTSORT-MARKLAR-TR-NOV-07" (qindy_cell) on 5-6 November 2007.

References

[1] T. Ramdas, G. K. Egan, D. Abramson, and K. Baldridge, "Towards a special-purpose computer for Hartree-Fock computations," Theoretical Chemistry Accounts [online first], http://dx.doi.org/10.1007/s00214-007-0306-6, 2007.

[2] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, and A. Konagaya, "Protein Explorer: A Petaflops Special-Purpose Computer System for Molecular Dynamics Simulation," in *Proceedings of Supercomputing 2003*, 2003.

[3] J. Makino, "The GRAPE Project," *Computing in Science and Engineering*, vol. 8, no. 1, pp. 30–40, 2006.

[4] T. Ramdas and G. Egan, "A Survey of FPGAs for Acceleration of High Performance Computing and their Application to Computational Molecular Biology," in *Proceedings of IEEE TENCON*, 2005.

[5] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Eurographics 2005, State of the Art Reports*, Aug. 2005, pp. 21–51.

[6] M. Gschwind, "Chip multiprocessing and the cell broadband engine," in *CF'06: Proceedings of ACM Computing Frontiers 2006*, 2006, pp. 1–7.

[7] P. A. Agarwal, R. A. Alexander, E. Apra, S. Balay, A. S. Bland, J. Colgan, E. F. D'Azevedo, J. J. Dongarra, T. H. D. Jr., M. R. Fahey, R. A. Fahey, A. Geist, M. Gordon, R. J. Harrison, D. Kaushik, M. Krishnakumar, P. Luszczek, A. Mezzacappa, J. A. Nichols, J. Nieplocha, L. Oliker, T. Packwood, M. S. Pindzola, T. C. Schulthess, J. S. Vetter, J. B. W. III, T. L. Windus, P. H. Worley, and T. Zacharia, "ORNL/TM-2004/13: Cray X1 Evaluation Status Report," Oak Ridge National Laboratory, Tech. Rep., 2004.

[8] T. Ramdas, G. K. Egan, D. Abramson, and K. Baldridge, "Converting Massive TLP to DLP: A Special-Purpose Processor for Molecular Orbital Computations," in *CF '07: Proceedings of the 4th Conference on Computing Frontiers*. New York, NY, USA: ACM Press, 2007.