Run-time thread sorting to expose data-level parallelism

Tirath Ramdas, Gregory K. Egan, David Abramson
Monash University

Kim K. Baldridge
University of Zurich

Abstract

We address the problem of data parallel processing for computational quantum chemistry (CQC). CQC is a computationally demanding tool to study the electronic structure of molecules. An important algorithmic component of these computations is the evaluation of Electron Repulsion Integrals (ERIs). A key problem with ERI evaluation is control-flow variation between different ERI evaluations, which can only be resolved at runtime. This causes the computation to be unsuitable for data parallel execution. However, it is observed that although there is variation between ERI evaluations, the variation is limited; in fact there are a limited number of ERI classes present within any given workload. Conceptually, it is possible to classify the ERIs into sizable sets, and execute these sets in a data parallel fashion. Practically, creating these sets is computationally expensive. We describe an architecture to perform this thread sorting, where high throughput is achieved with small associative and multiport memories. The performance of the prototype is evaluated with FPGA synthesis. We go on to envision other uses for thread sorting, in general-purpose manycore architectures.

1. Introduction

There is wide consensus that clock speed improvements can no longer be relied on to deliver performance improvements, and therefore parallelism is to be the new axis of interest. However, parallelism comes in many forms. Data parallel processing is a highly efficient form of parallel processing. Conceptually, data parallel (DP) processing minimizes the "control overhead" of processing systems, which allows greater resource allocation to data processing resources such as arithmetic units. It is for this reason that data parallel processing features prominently in GPUs, the Cell BE (within this multicore architecture, there are 8 SIMD cores), and it is speculated that the forthcoming Intel Larabbee architecture will prominently feature data parallel processing as well.

The burden, therefore, is to ensure that applications can leverage data parallel processing. For some codes this is a trivial process, especially those applications whose bottlenecks are linear algebra routines. However, some applications contain unique kernels. Two such applications are computational quantum chemistry (CQC) and ray-tracing.

At the heart of CQC is the Hartree-Fock method (HF). The main problem with this method is lack of uniformity in the kernel of the application, which results in a computational hotspot that is somewhat irregular. The application hotspot is the computation of $O(N^4)$ Electron Repulsion Integrals (ERIs). ERIs present two problems: 1) there is a very large number of ERIs to compute, and 2) each ERI is computationally intensive. The most critical problem in the context of the present work is the fact that there are various ERI classes needed for any given workload, and the different ERI classes vary substantially in control-flow and execution time. This variation is so great that there are different algorithms that are better suited for different ERI classes [1][4]. This seems to rule out data parallel processing.

There is a similar problem with ray-tracing, i.e. that of incoherent rays. This has prompted proposals for *ray reordering* for the purpose of improving cache utilization [6]. This problem also seems to rule out data parallel processing. This has led to a proposal for ray reordering for SIMD execution [13], although as far as we are aware the practicality of the idea is yet to be demonstrated.

Although the ideas expressed in this paper may be relevant to the incoherent rays problem, our primary focus will be on the ERI problem. The basic idea underlying the proposal is to match ERI tasks – effectively independent threads of control – which have identical control-flow, and issue these matched threads to a data parallel processor as a data parallel stream, essentially converting thread-level parallelism (TLP) to data-level parallelism (DLP). This idea is illustrated in Fig. 1. In this example, each basic scheduling block (BSB) corresponds to a contiguous chunk of instructions. The BSB-B block has a cyclic arc with an associated variable integer. The arc is a loop identifier and the associated integer specifies the loop count; e.g. in thread 2 BSB-B is executed 3 times, whereas in threads 1, 3, and 4 BSB-B is executed twice. The difference in loop count is a difference in control-flow. In this example the loop count of

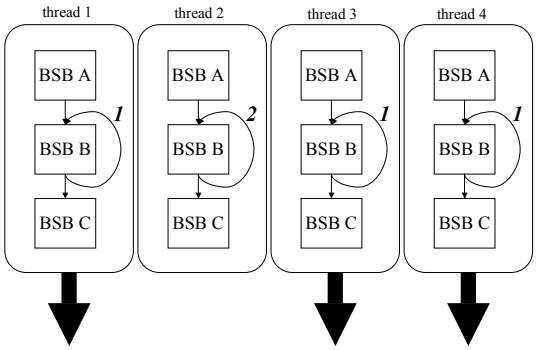

Figure 1. Matching threads for DP execution.

BSB-B is the only source of control-flow variation between different threads, therefore to identify threads with identical control-flow one merely has to check the loop count specified for BSB-B. Matching threads can then be processed with an identical instruction sequence, and may therefore be efficiently processed with a data parallel architecture.

The concept of sorting ERI threads into equivalent sets for data parallel execution has been suggested before [11][2][7], however as far as we are aware a specific approach and architecture to achieve this goal have yet to be presented. We describe a thread sorting mechanism to implement the concept, and an architecture to deliver high throughput sorting.

This paper is organized as follows. First, the salient characteristics of the application are reviewed, and motivating background material is surveyed. Sec. 4 describes the thread sorting concept, and it's application towards DLP extraction. The proposed architecture is described and evaluated in Sec. 5. Finally, avenues for further scope-expanding work are discussed, followed by concluding remarks.

2. Application Characteristics

The solution described in this paper targets the Hartree-Fock Direct Self-Consistent-Field (SCF) algorithm and other closely related algorithms such as Configuration Interaction. Problems that make heavy use of Electron Repulsion Integrals (ERIs) are the main targets.

SCF has a canonical complexity of $O(N^4)$, although it may be as low as $O(N^2)$ with certain optimizations such as distance cut-offs. It should be noted that the proposed solution is capable of exploiting such optimizations – i.e. it is not a purely "brute-force" solution – although the specific details of this will be left for a future communication. Canonically, there are $O(N^4)$ ERIs to compute for a given workload. In general, this consumes very substantial (dominant in many cases) computation time.

The workload is prescribed by the use of a basis set, which comprises N basis functions. Each basis function contains floating-point data (such as Cartesian coordinates and coefficients) and integer data (contraction specification and angular momenta terms which are related to the electron configuration over "spdf..." shells). ERIs are computed for quartets (i.e. 4-tuples) of basis functions – hence, there are $O(N^4)$ ERIs. Each ERI is independent of the others; therefore each ERI may be thought of as an independent thread of control. Seeing as there is a massive number of ERIs, and each ERI is an independent thread (with no data dependancies between threads), the application exhibits massive TLP. Table 1 lists several workloads.

Molecule	Basis functions	ERI threads
Water, H_2O	25	48×10^3
Nitrous oxide, N_2O	45	512×10^3
Butane, C_4H_{10}	110	18.3×10^6
Nicotine, $C_{10}H_{14}N_2$	250	488×10^6
Carotene, $C_{50}H_{55}$	880	75×10^9

Table 1. Workload vs. ERI count.

The integer terms in the 4 basis functions are a source of control-flow variation for ERIs, and effectively define the *class* of an ERI. This variation extends beyond loop iteration counts; the variation is also in the number of coefficients constituent in a basis function, and also in the amount of intermediary data produced within an ERI routine. Furthermore, the variation is so great that there are different ERI algorithms which are optimal for different ERI classes [1][4]. Each ERI is computationally expensive. With current generation Xeon and Opteron cores we have observed that computing ERIs consumes an average of just over 200ns per ERI for a representative workload.

Though there is control-flow variation as a consequence of various ERI classes, the number of ERI threads greatly outnumbers the number of different ERI classes; i.e. there are a very large number of ERI threads belonging to the same ERI class. This is indicated in Fig. 2, where the data points indicate the number of ERI classes for certain workloads while the curve for number of ERI threads is analytically obtained (and confirmed empirically for the associated data points). This presents opportunities for DLP extraction by sorting the entire thread population into subsets of threads with identical thread class. This is the strategy espoused in this paper.

3. Related Work and Motivation

The basic idea of matching ERI threads for concurrent data parallel processing was first reported in 1982 [11]. To date, as far as we are aware, the viability of the idea has not

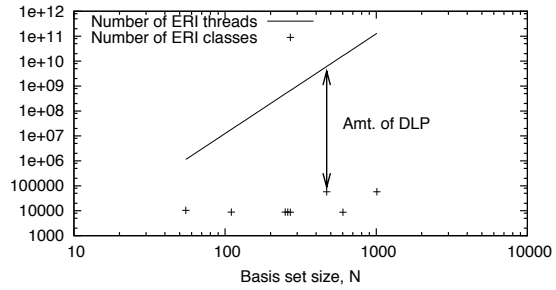

Figure 2. Threads vs thread classes.

been demonstrated. The only significant attempt to implement the proposal was reported almost a decade after the initial idea was reported; Ernenwein et al. [2] developed an implementation that relies on liberal use of buffering, memory management, and concurrent buffer management processes. This approach yields very poor performance. Almost 30% of system time is consumed setting up the data parallel ERI evaluations – a very large overhead.

Recently, the idea has resurfaced [7], citing resurgent interest in data parallel processing. This proposal included a suggestion for the use of associative memory. However, the proposal is inefficient, requiring a very large associative search memory (16k entries). It has also been shown that the loss of locality implied by this reordering does not dramatically reduce the memory access locality of the computation [8] – the application exhibits ample locality. Therefore, some locality may be traded away to obtain better load balance, e.g. data parallel balance.

Accentuating the need for a solution to this problem, excellent performance potential has been reported for data parallel evaluation of ERIs with GPUs [14][12]. However, these efforts have only indicated the potential gains that may be yielded with GPUs assuming they may be fed with sets of identical class ERIs; creating these subsets in an efficient manner remains a problem.

4. Thread Sorting

In this section we describe the thread sorting procedure, and we present results that demonstrate the viability of extracting homogenous subsets of threads from the non-homogeneous universal set of threads – essentially extracting DLP from TLP.

This scheme assumes that each ERI thread contains two pieces of information. First, a unique identifier for each thread – a *threadID*. For this application, a 56 bit word is sufficient. Second, an identifier corresponding to the class of each thread, which should be common to many threads – a *classID*. For this application, a 64 bit word is sufficient. A thread's classID indicates it's control-flow. All threads with

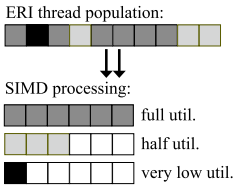

Figure 3. Threads vs. DP utilization.

the same classID may be processed with the exact same sequence of instructions – this is the basic feature of data parallel processing. Therefore, a set of threads with identical classID may be processed as a data parallel vector/stream. The task at hand is to process a set of threads with the aim of identifying subsets of threads with identical classID. The basic procedure is described in Algorithm 1:

Algorithm 1 Thread sorting

```
1: loop {for each thread x}
2:    ς = x.classID
3:    index = class_to_set_lookup(ς)
4:    set[index] ⇐ x.threadID
5:    if set[index].population = threshold then
6:       Issue all threads in set[index] for processing
7:    end if
8: end loop
```

The basic procedure is conceptually simple, but the difficulty is in developing an architecture that can perform the operation with high-throughput. The key challenges are high throughput implementation of the "class_to_set_lookup" function, and high throughput implementation of the set storage. Both issues are explicitly considered and a suitable architecture is presented in Sec. 5.

The primary measure of success is data parallel unit utilization U. Low U indicates that function units are being underutilized. Achieving the ideal score ($U = 1$) requires that, for a data parallel system with vectors of size V elements, there be V concurrent data streams – i.e. V threads with identical classID, packed into a vector of size V, issued to a DP processor with V processing elements. Partially filled vectors result in partial scores for U. This is illustrated for clarity in Fig. 3, with $V = 6$, and where the thread sorting functionality is implicit.

As one may expect based on Fig. 2, in general, larger workloads (i.e. larger N) result in larger U. This is because the increase in the number of threads is not accompanied by an increase in the number of thread classes. However, it would be impractical to perform thread sorting over the entire universal set of threads for any workload; there would be far too many threads for this to be practical (see Table 1). Therefore, thread sorting is performed over a limited window of threads. This limits opportunities for thread match-

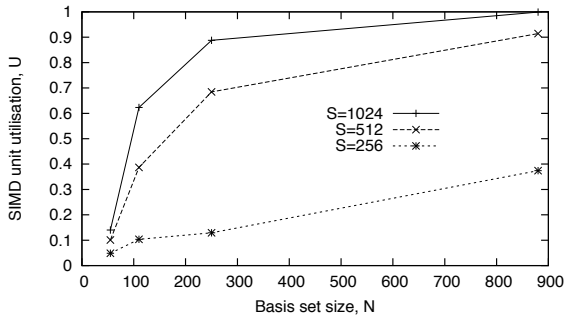

Figure 4. DP utilization.

ing since the number of threads available for matching is reduced. Consequently, the size of the thread window[1] S also has an impact on U.

For now we will let $V = 64$, as this corresponds to the vector width of the classic Cray vector processors, though other values of V may be more appropriate (e.g. for many GPUs, $V = 32$, whereas for the Cray C90, $V = 128$). Larger V will require larger matching effort, and therefore larger S. This may be considered in a later communication. In the current discussion, since V is fixed, U will vary with S for a given workload. The results for N and S vs. U, with $V = 64$, are presented in Fig. 4, with workloads from Table 1. These results indicate that utilization improves as 1) workload size increases, and 2) as the thread window size increases. From these results it is apparent that, for workloads of interest (i.e. large molecules), $S = 512$ will be sufficient to achieve high utilization for a data parallel architecture with $V = 64$.

5. System Architecture

Recall that in Sec. 4, two key requirements were identified: 1) high throughput lookup function, and 2) high throughput implementation of the set storage. Expanding the scope of discussion, we will first describe the overall architecture of the application processor, to see where thread sorting fits in. Next, we will describe the thread sorting architecture in depth (in particular, how it resolves the second issue described above). We will then describe two alternative solutions for the high throughput lookup function. Finally, performance based on an FPGA prototype is reported.

5.1. Overall architecture

The proposed architecture (Fig. 5) has 3 main blocks: a *Thread Producer* (TP), a *Thread Sorter* (TS), and a *Thread Consumer* (TC). The TP generates threads, and performs application-specific pre-processing. The TS performs thread sorting as described in Sec. 4. The TC is a data parallel processor (e.g. GPU) for evaluation of sets of ERIs. In addition, there is a host system to load the relevant data into the memories of the ERI evaluation system.

Although the Thread Producer (TP) functions might be served by the host system, the functionality is highly amenable to pipelining, and therefore suitable for efficient FPGA implementation. Furthermore, the TP must transmit the threadID (56 bits) and classID (64 bits) to the Thread Sorter (TS), so communications bandwidth (120 bits) is a factor to consider. In contrast, the TS only transmits the thread ID (56 bits) to the Thread Consumer (TC). Therefore, tight TP-TS coupling is desirable. The TP is a custom 5-stage pipeline that loads and combines basis functions iteratively to generate threads. In addition to this basic TP functionality, our prototype also performs pre-processesing on each thread, specifically permutational symmetrization [7], which increases the DP utilization for a given workload by symmetrizing threads, which makes a larger portion of threads class-equivalent. For very large workloads the TP is able to sustain performance of almost 1 cycle per thread[2].

As threads are produced, they are sent to the TS. As sufficiently large sets of control-equivalent threads are accumulated, entire sets of threads are transmitted to the TC. As previously mentioned, the TC can be any form of data parallel processor, such as GPUs, DP arrays, or vector processors. Because threads are matched rigorously, it is not necessary to rely on non-universal flexibility in data parallel processors such as vector lane threading [9].

5.2. Thread Sorter architecture

Our Thread Sorter (TS) is a 3-stage pipeline. FIFO request queues are implemented in several critical parts of the TS, e.g. the Stream Controller. Hazard resolution logic is required to deal with data hazards and structural hazards. The Class Store implements the lookup function, which was one of the problems alluded to in Sec. 4. The required Class Store capacity is S. The Thread Store maintains sets of matched threads, and the required capacity is $S \times V$.

As threads are produced, they are sent to the TS. The classID is sent through the Class Store block, which implements the lookup function and will be discussed in Sec. 5.3. A set index is produced by the Class Store and sent to the Control Unit, which then writes the threadID into the appropriate set contained in the Thread Store. When a set is full, the entire set is streamed to the TC.

[1] It is more precise to say that the thread window is between S and $S \times V$; S indicates the number of sets in the window, and each set may contain between 1 to V threads.

[2] The limitation here is that the basis set memory can only provide 1 access per cycle. A thread is constructed from four basis functions, and as iteration boundaries are reached it becomes necessary to load multiple basis functions to generate a thread, thus requiring multiple cycles.

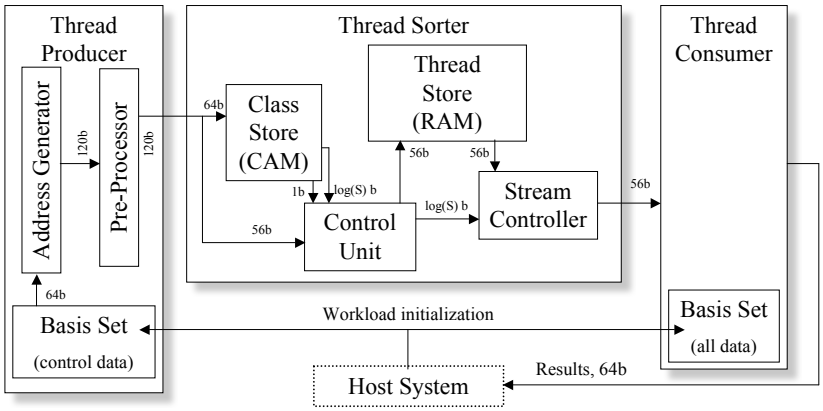

Figure 5. System diagram.

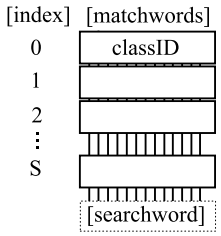

Figure 6. Logical view of a CAM.

In order to sustain high throughput, it is necessary that a new threadID can be written into the Thread Store while another is being streamed out. To enable this, a dual-ported RAM [10] is used (1 read port, 1 write port). This is implemented in our prototype with the FPGA's internal block RAM. If multiport RAM is unavailable a similar result may be obtained by using multiple RAM blocks.

5.3. Lookup function

We consider two approaches to implement the Class Store lookup functionality. One approach is to employ associative searching with a *content addressable memory* (CAM) (illustrated in Fig. 6). While conventional memory accepts an index and returns data, a CAM accepts a data word and returns the index at which the data resides, within a single clock cycle. CAMs are widely used in IP routers for address lookup [5].

One argument against the use of CAMs is that they are expensive in terms of logic and power consumption. However, because the required CAM size S is practically low ($S = 512$), this criticism is diminished (unlike an earlier proposal which requires a 16k entry CAM which implements the functionality of the Thread Store [7]). Furthermore, the required word size (64 bits) is also practically low – word sizes of up to 144 bits are quite common for CAMs [5]. Finally, with our application, ternary matching is not required. The current prototype utilizes a CAM for Class Store lookup functionality.

However, there may be an alternative approach that is cheaper in terms of logic area and power consumption, as previously described [8]. This approach involves a CRC hashing of each thread's classID, which is then mapped to a hash table. To handle hashing conflicts, each table entry may be an array of pointers (i.e. hash table probing) to the actual Thread Store index. For this application, we have observed that it is highly beneficial to implement this array with a most-recently-used (MRU) queue; it is apparent that there is some temporal locality inherent in the sequence of classID's generated.

Although this approach has potential for better efficiency, presently it is inferior to the CAM-based approach. In spite of the MRU queue enhancement, resolving hashing-conflicts through probing is a significant expense. In order to minimize hashing-conflicts, a large hash table ($\gg S$ entries) is required and even so, conflict resolution is still necessary, introducing significant area/time cost and complexity. If a better hash function can be identified for this application (i.e. one that results in less clustering) then it may be worth re-investigating this option.

5.4. Throughput

The prototype (implemented with a Xilinx Virtex-4 LX200 FPGA) achieves near 1-thread-per-cycle throughput. This is thanks in large part due to the use of a dual-port memory for set management, which almost eliminates structural hazard pipeline stalls entirely. However, with the present design, there is significant propagation delay through the Thread Sorter, which reduces the clock rate to 66 MHz. Additional pipelining would be highly beneficial. Nevertheless, in a configuration where dual prototype blocks are replicated and multiplexed through a bus controller operating at 133 MHz, and with a commodity GPU as a TC, it is possible to sustain thread throughput such that a speedup factor of 20 $\times$ over a scalar processor may be achieved. This projection is based on the assumption that the ERI evaluation time can be hidden behind the thread sorting time – this assumption is known to be valid, based on reports of ERI computation times with GPUs [14][12]. In terms of logic resources, the prototype consumes about 34 % of the FPGA's LUTs. Consumption of other resources (e.g. block RAM) is even less critical. With additional pipelining and design effort, the speedup factor would in-

crease; the bottleneck is currently the TS throughput.

6. Further Work

In addition to the microarchitectural enhancements mentioned throughout the paper, there are several ways in which the work may be continued, and in particular the application scope may be broadened. The thread sorting concept is actually fairly broad. It is possible to have multiple and heterogeneous TPs and TCs, and varying matching criteria. In this work, classIDs reflect the control-flow of the thread; this does not have to be the case. The classID merely characterizes the thread in some meaningful way.

There could be applications of thread sorting in the context of general-purpose manycore processing. Scheduling batches of threads at run-time offers several advantages, especially adaptability to run-time conditions (e.g. core powerdown to save energy/reduce heat dissipation) which would necessitate the rapid transfer of threads to some component in the system. Clearly for general applicability there are issues that need to be resolved. Dependencies between threads need to be considered. It may be possible to introduce dependency tagging as a part of the classID.[3] Potentially diminished memory access locality inherent to thread reordering should also be considered.[4] It may also be worthwhile to consider the application of thread sorting to ray-tracing [13], as mentioned earlier.

7. Conclusion

The problem of applying data parallel processing to ERI evaluation is a long-standing one. The basic idea of ERI sorting has been a goal since at least 1982 [11], in response to initial findings by Guest and Wilson [3], who candidly described the vector performance of CQC codes as "depressing" – and note that this was in the heyday of vector processing supercomputing dominance. Some gains have been made for the linear algebra components of the computation, but the problem of data parallel processing of ERIs remains.

Given resurgent interest in data parallel processing, ranging from the evolution of GPUs to accomodate general-purpose applications, to the DP grunt of the Cell BE, we believe it is necessary to revisit this issue. We have demonstrated a high-throughput method to perform run-time dynamic thread sorting for the purpose of transforming the workload into a stream amenable to data parallel processing. The key to high throughput is the use of dual-port RAM for thread handling, and a CAM structure for rapid lookup. For the most important performance metric – i.e. utilization – very good performance was obtained while keeping the design practically small, consuming about 1/3 of a contemporary FPGA device. With additional pipelining and design effort, an FPGA implementation will yield a speedup factor well over $20\times$ when coupled with a GPU. Furthermore, a fixed-logic implementation would not only yield significantly higher clock rate, but, as a component of a manycore architecture, may also enable novel and efficient manycore solutions.

[3] This is not a problem with ERI since all threads are independent.

[4] This is not a problem with ERI, thanks to high temporal locality [8].

References

[1] M. Dupuis and A. Marquez. The Rys quadrature revisited: A novel formulation for the efficient computation of electron repulsion integrals over Gaussian functions. *J. Chem. Phys.*, 114(5):2067–2078, 2001.

[2] R. Ernenwein et al. A program for ab initio MO calculations on vector and parallel processing machines I: Evaluation of integrals. *Comp. Phys. Commun.*, 58:305–328, 1990.

[3] M. F. Guest and S. Wilson. *Supercomputers in Chemistry*, chapter 1, pages 1–37. American Chemical Society, 1981.

[4] M. Head-Gordon and J. A. Pople. A method for two-electron Gaussian integral and integral derivative evaluation using recurrence relations. *J. Chem. Phys*, 89(9):5777–5786, 1988.

[5] K. Pagiamtzis et al. Content-addressable memory circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006.

[6] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, New York, NY, USA, 1997. ACM Press.

[7] T. Ramdas et al. Converting Massive TLP to DLP: A Special-Purpose Processor for Molecular Orbital Computations. In *CF'07: Proceedings of ACM Computing Frontiers 2007*, pages 267–276, 2007.

[8] T. Ramdas, G. K. Egan, D. Abramson, and K. K. Baldridge. On ERI Sorting for SIMD Execution of Large-Scale Hartree-Fock SCF. *Comp. Phys. Commun.*, 2008.

[9] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector lane threading. In *ICPP'06: Proceedings of the 2006 International Conference on Parallel Processing*, 2006.

[10] S. Rixner et al. Register organization for media processing. In *HPCA-6: Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 375–386, 2000.

[11] V. R. Saunders and M. F. Guest. Applications of the Cray-1 for Quantum Chemistry Calculations. *Comp. Phys. Commun.*, 26:389–395, 1982.

[12] I. S. Ufimtsev et al. Quantum Chemistry on Graphical Processing Units. 1. Strategies for Two-Electron Integral Evaluation. *J. Chem. Theory Comput.*, 4:222–231, 2008.

[13] I. Wald, et al. UUSCI-2007-012: SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering. Technical report, Scientific Computing and Imaging Institute, University of Utah, 2007.

[14] K. Yasuda. Two-Electron Integral Evaluation on the Graphics Processor Unit. *J. Comp. Chem.*, 29:335–342, 2007.