Computer Physics Communications ••• (••••) •••–•••

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

# ERI sorting for emerging processor architectures ☆

Tirath Ramdas [a,*], Gregory K. Egan [a], David Abramson [b], Kim K. Baldridge [c]

[a] Centre for Telecommunications and Information Engineering, Monash University, Melbourne, Australia
[b] Centre for Distributed Systems and Software Engineering, Monash University, Melbourne, Australia
[c] Organic Chemistry Institute, University of Zurich, Switzerland

ARTICLE INFO

*Article history:*
Received 24 January 2009
Accepted 30 January 2009
Available online xxxx

*PACS:*
02.60.Pn
07.05.Bx
31.15.Ar
89.20.Ff

*Keywords:*
Hartree–Fock Self-Consistent Field
Electron repulsion integrals
Single-instruction-multiple-data processing

ABSTRACT

Electron Repulsion Integrals (ERIs) are a common bottleneck in *ab initio* computational chemistry. It is known that sorted/reordered execution of ERIs results in efficient SIMD/vector processing. This paper shows that reconfigurable computing and heterogeneous processor architectures can also benefit from a deliberate ordering of ERI tasks. However, realizing these benefits as net speedup requires a very rapid sorting mechanism. This paper presents two such mechanisms. Included in this study are analytical, simulation-based, and experimental benchmarking approaches to consider five use cases for ERI sorting, i.e. SIMD processing, reconfigurable computing, limited address spaces, instruction cache exploitation, and data cache exploitation. Specific consideration is given to existing cache-based processors, FPGAs, and the Cell Broadband Engine processor. It is proposed that the analyses conducted in this work should be built upon to aid the development of software autotuners which will produce efficient *ab initio* computational chemistry codes for a variety of computer architectures.



## 1. Introduction

*Ab initio* computational chemistry has a long history in the computational sciences, and has produced many legacy codes. While these codes are stable and well established, it is occasionally necessary to re-engineer them to take advantage of new computer architectures. The last major re-engineering was the exploitation of shared- and distributed-memory to take advantage of coarse-grained parallelism on massively parallel supercomputers. However, at the level of individual computer processors, little change was necessary. This was due to the relatively "one-dimensional" evolution of processors, where increasing clock-rates resulted in performance gains that were readily exploited by existing codes.

Because clock-rate improvements cannot be relied upon for substantial performance improvements any longer [1], computer architects are compelled to tinker with parameters available in a wider solution space, leading to increased diversity in computer architecture [2]. Several important features have gained considerable interest in computer architecture research.

The first of these is *data-parallel processing*, which is an umbrella term encompassing architecture classes such as *Single Instruction Multiple Data* (SIMD) processing and vector processing. This trend is made apparent by increasing employment of data-parallel processing, primarily in the form of SIMD processing, in many contemporary high performance computer processor architectures, including the Cell BE [3], Graphics Processing Units (GPUs) [4], and the Intel Larrabee architecture [5]. SIMD processing is becoming increasingly attractive because of the gains provided in terms of *performance-per-transistor* [6]. However, the tradeoff is a loss of flexibility which places increased burden on the programmer to achieve high utilization.

Another major feature in computer architecture is *Reconfigurable Computing* (RC), which in practical terms is presently synonymous with a specific device: *Field-Programmable Gate Arrays* (FPGAs). FPGAs are programmable logic devices which can achieve very high performance when configured as special-purpose processing pipelines, similar to *Application Specific Integrated Circuits* (ASICs), with the advantage of re-configurability. This flexibility allows FPGAs to form arbitrary digital circuits at run-time. However, re-configuration is generally a slow process (in the order of milliseconds to seconds). Therefore, run-time reconfiguration must be employed sparingly.

A third major feature is more generic: greater on-chip *heterogeneity*. It is widely held that increasing integration will lead to increasing heterogeneity [1,2], with processors comprising not only multiple instances of similar processing cores (such as with the Intel Larrabee architecture [5]), but also multiple specialized pro-

DOI of original article: 10.1016/j.cpc.2008.01.045.
☆ This work was supported by the Monash University *Postgraduate Publications Award* (PPA).
* Corresponding author.
*E-mail addresses:* tirath@int19h.com (T. Ramdas), greg.egan@eng.monash.edu.au (G.K. Egan), david.abramson@infotech.monash.edu.au (D. Abramson), kimb@oci.unizh.ch (K.K. Baldridge).

0010-4655/$ – see front matter 
doi:10.1016/j.cpc.2009.01.029

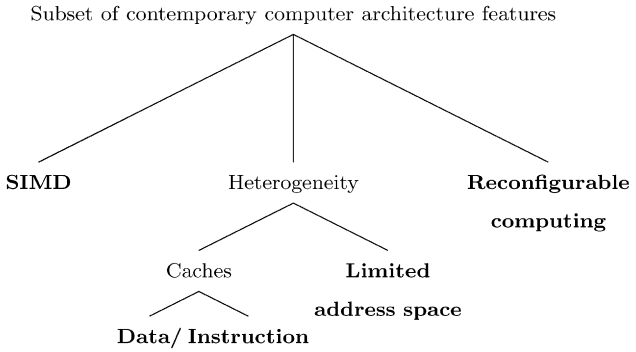



**Fig. 1.** Computer architecture features addressed by ERI sorting.

cessing cores (such as with the Cell BE architecture [3]) – leading to more asymmetric parallelism. The scope of core features that may be adjusted is very rich. Therefore, it is difficult to anticipate what general application changes would be beneficial. However, one parameter that should be watched closely is the amount of dedicated memory attached to each core, whether in the form of dedicated memory spaces or caches.

This is an incomplete cross-section of the salient changes anticipated in emerging processor architectures, and the features described – as summarized in Fig. 1 – are by no means mutually exclusive; indeed, they are all inter-related. It is very difficult to distill these features into a common unifying theme, and attempts to create "silver bullet" tool-chains and compilers to adequately utilize the broad spectrum of these architectures are at least several years away from producing anything workable. Presently, the onus is on application specialists to combine application and workload insights with an understanding of each architecture class to produce high performance processing solutions. Having said that, there will be opportunities to reuse solutions developed for one architecture class in other classes. The main difficulty is in recognizing when a technique used in one situation has relevance in other situations for which it was not originally conceived.

Such a technique was originally conceived for processing of Electron Repulsion Integrals (ERIs) with SIMD processors. The difficulty arising from ERIs is that based on run-time data, ERI tasks take different classes and require different processing programs. This would result in very poor utilization on SIMD processors. The solution to this problem is to match ERI tasks with identical class, and to issue these sets of tasks as a parallel vector to a SIMD processor. The function of matching ERI tasks with identical class requires a sorting mechanism, referred to as *ERI sorting*.

In addition to SIMD processing, there are five specific usage cases for variations of the ERI sorting concept which are explored in this paper:

(i) *SIMD processing*: This is the main application for ERI sorting. By sorting ERI tasks into vectors of control-equivalent tasks, high SIMD utilization may be achieved. (ERI sorting for SIMD processing was explored in a previous communication [7], but this work is reviewed here for completeness and context.)
(ii) *FPGA processing*: ERI sorting may be used in tandem with run-time reconfiguration to produce a "virtual" ASIC corresponding to the various ERI classes.
(iii) *Limited program size*: ERI sorting may be used to overcome the limited memory attached to processors and coprocessors, such as with the coprocessors within the Cell BE architecture.
(iv) *Enhanced instruction cache utilization*: For architectures that are sensitive to instruction cache limits, ERI sorting can provide some performance gains.
(v) *Heterogeneous data cache limits*: It has been shown that data cache capacity dictates the practicality of bootstrapping for various integral classes [8]. ERI sorting can deliberately map ERI tasks to cores within a system by matching data cache requirements to data cache capacities.

This paper is organized as follows. Section 2 reviews the salient characteristics of ERIs as is relevant to the issue of ERI sorting. Section 3 introduces two high-performance methods for ERI sorting that are appropriate for different situations. Section 4 describes precisely how ERI sorting may be applied to five key situations, enabling high performance processing with existing and emerging computer architectures. Finally, concluding remarks are made, along with recommendations for further work.

## 2. Electron Repulsion Integrals

Electron Repulsion Integrals (ERIs), also known as *two-electron integrals*, are an important component in the Hartree–Fock Self-Consistent Field (SCF) method. A broad overview of the SCF method and a detailed exploration of ERI processing has been previously published [7]. The most important characteristics will be reviewed here.

With the SCF method, there are (canonically) $N^4/8$ ERIs required for each iteration,[1] given a workload consisting of $N$ basis functions, though the actual number of ERIs computed is typically much less through the use of cutoffs and symmetry exploitation. Nevertheless, the number of ERIs remains high, and ERI processing is a significant consumer of processing cycles for SCF computation.

The difficulty with ERI evaluation is that there are a number of different classes of ERIs, arising from run-time data-dependencies, specifically the angular momenta and level of contraction of the four basis functions constituent in any given ERI task. Each ERI class requires a different evaluation routine. Some algorithms, such as the Rys quadrature [10,11] are capable of computing ERIs of any practical class; however, branches within the algorithm essentially result in different processing sequences contained within a single subroutine. The end result is that different ERI classes require different ERI processing instruction sequences.

Furthermore, the Rys quadrature is known to be efficient only for lowly contracted high angular momenta basis function quartets [12]. For this reason, some code-bases such as the *General Atomic Molecular Electronic Structure System* (GAMESS) employ multiple ERI processing routines, driven by a routine switching code that assigns (at run-time) the "best" routine to any given ERI quartet. The end result is that only a handful of ERI classes are handled by any given ERI evaluation routine, and a small number of routines are required to process the entire range of possible ERI classes.

Thus, the problem with ERIs, as is relevant to the present work, is variation of execution. This variation may be conceptualized between two levels: (1) variation at the fine-grained execution level (e.g., individual processor instructions), and (2) variation at the coarse-grained subroutine level (i.e. different evaluation algorithms suitable for different sets of classes). Fig. 2 illustrates the mapping between the level of granularity to the computer architecture features of interest. This mapping will be explained in the following paragraphs.

Variation at the fine-grained execution level is highly pertinent to SIMD processing, since efficient SIMD processing requires all PEs in the processor to execute in "lock-step", with all PEs executing an identical sequence of instructions. Therefore, efficient SIMD processing requires that all PEs process ERIs of identical class. A similar situation exists with the use of FPGAs. In order to achieve maximum efficiency, FPGA designs are generally required to be highly specialized and deeply pipelined. Therefore, an FPGA-

[1] There would be roughly 20 iterations for typical workloads [9].

Please cite this article in press as: T. Ramdas et al., ERI sorting for emerging processor architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.01.029

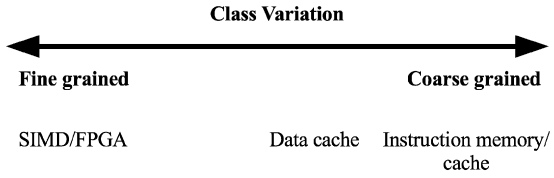



**Fig. 2.** ERI class variation granularity, and the relevance of the computer architecture features of interest as per Fig. 1.

based ERI processing pipeline should be specialized for the exact computational sequence associated with individual ERI classes, and sequences of ERI tasks of identical class are required to keep the pipeline fully utilized between infrequent reconfigurations.

Variation at the coarse-grained execution level, i.e. the use of different ERI processing algorithms, is relevant to processor cores with limited address space and cache size. This level of granularity is to do with the size of the ERI evaluation program, and the impact of this factor on the microarchitecture of the executing processor. In the case of processor cores with limited address space, this is a "hard" problem: a 256 kB address space generally cannot accommodate 400 kB of instruction memory.[2] In the case of cache limitations, in particular instruction cache limitations, large instruction memory segments would result in a performance loss.

The level of granularity may be considered as a continuum rather than two discrete extremes. In fact, considering granularity as a continuum is relevant to the degree of heterogeneity of a manycore processor architecture. For example, in an architecture consisting two types of processor core with different specifications, there would be two levels of interest; this is very coarse grained. A processor with 5 different core classes could be considered medium-grained.

If the variation in the processor core types were limited to processor throughput, then the issue is simple load-balancing. Existing applications such as GAMESS achieve good dynamic load-balancing through shared counters [13]. However, there are microarchitectural specifications that should drive the functional behavior of ERI processing in order to make more efficient use of each processor core. One specific example is the practice of *bootstrapping* to pre-compute a number of values which are used repeatedly in the processing of a single ERI task. An alternative to the "pre-compute and re-use" approach is to re-compute the values whenever they are required. Reusing bootstrap values results in substantial processor time savings [8]. However, some ERI classes may require as much as 4 MB of bootstrap data [14]; reusing these values is only practical if a processor core has sufficient cache capacity for such a large dataset. In the case of heterogeneous architectures containing a mix of small coprocessors with small caches and large main processors with large caches, it would make sense to map ERI tasks with large cache requirements to the large cache processors.

ERI sorting can provide these functions. Furthermore, as will be seen in the next section, ERI sorting can be performed very efficiently.

## 3. ERI sorting mechanism

Each ERI task is a concurrent thread of execution, and thus a set of ERI tasks may be computed in arbitrary order. This research proposes that there are several benefits to executing ERI tasks in an order that blocks together tasks that have similar class – a technique called ERI sorting. As described in the previous section, these benefits arise from the performance implications of various features in computer architecture. However, these benefits are only realized as net speedup if ERI sorting may be performed in a manner that introduces minimal overhead.

```
1:  if (i.K × j.K × k.K × l.K) == 1 then
    {"i.K denotes level of contraction of basis function i"}
2:  Uncontracted = true {"else false"}
3:  end if
4:  if (i.a <= p) ∧ (j.a <= p) ∧ (k.a <= p) ∧ (l.a <= p) ∧ 2 then
    {"i.a denotes shell type of basis function i"}
5:  sp = true {"else false"}
6:  end if
7:  if (i.a >= l) ∧ (j.a >= l) ∧ (k.a >= l) ∧ (l.a >= l) ∧ 2 then
8:    l = true {"else false"}
9:  end if
10: if (l ∨ (Uncontracted ∧ (¬sp))) then
11:   Rys(i, j, k, l)
12: else
13:   ERIC(i, j, k, l)
14: end if
```

**Algorithm 1.** ERI routine selection function.

In principle, it is possible to implement the Hartree–Fock method such that integrals are implicitly generated in the desired order [15]. However, this requires the relevant routines and data structures to be designed with such ordering in mind. In general, and especially for legacy codes, such design cannot always be assumed. For legacy designs, a dynamic sorting mechanism may be incorporated to explicitly produce the desired ordering. In this section, two highly efficient sorting mechanisms will be proposed.

### 3.1. Basic sorting

In the case of coarse-grained variation, ERI sorting is especially simple. Consider a situation where two integral routines are employed:

(i) *Rys quadrature* [10,11]: Let this routine be used for uncontracted ERIs of all angular momenta types excluding S and P, as well as ERIs of type L or higher regardless of level of contraction.
(ii) *Electron Repulsion Integral Calculator* (ERIC) code [13]: Let this routine be used for contracted S, P, D, F and G type ERIs not computed with Rys quadrature.

In this situation there are only two ERI classes, and the total set of ERIs would be divided into two subsets.

The number of subsets (specified by parameter $W_c$, which stands for "class window" due to it's previous usage [7], as will be seen in Section 3.2) is arbitrary in principle, provided there is an objective function to uniquely map an ERI task to a subset. In this example, each ERI task is associated with a single ERI processing routine, and the set selection function is driven by very simple logic, similar to the algorithm selection driver in GAMESS.

This logic is easily modified to incorporate ERI sorting; calls to the ERI evaluation routine (e.g., *Rys*(...) and *ERIC*(...)) are replaced with calls to routines that insert the ERI quartet into a buffer corresponding to the particular routine (e.g., *InsertRys*(...) and *InsertERIC*(...)). Once the population of a buffer exceeds some threshold $V$ ("vector width" due to it's previous usage [7]), the entire set of ERI tasks contained in the buffer are processed concurrently, while other sets wait. Provided the number of sets $W_c$ is kept at a practically low value, the overhead of this sorting approach would be negligible.

This approach may be easily augmented to suit other purposes; e.g. for sorting based on bootstrap value dataset size, one could create two (or more) buffers, and use a selection function that choses the buffer based on level of contraction, similar to line 1

[2] In practice there would be methods to overcome this hard limit, such as the use of instruction memory overlays as practiced with the Cell BE architecture [3], however such practices would incur a performance penalty.

Please cite this article in press as: T. Ramdas et al., ERI sorting for emerging processor architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.01.029

```
1:  τ_ijkl = classID(i, j, k, l)
    {"Extract classID from ERI task, E_ijkl"}
2:  s = set.Lookup(τ_ijkl)
    {"Map the ERI class to a set/buffer contained in the current window"}
3:  if s = no match found then
4:    if ∀ sets C_1 → C_Wc there is a free set then
5:      s = free set identifier
6:    else
7:      s = random value (1 ⩽ s ⩽ W_c)
8:      Issue all tasks in C_s for data-parallel processing
        {"Set C_s is issued for processing prematurely, suffering a performance
        loss, but this is a necessary step because a new set is needed to
        accommodate the incoming ERI task"}
9:    end if
10: end if
11: C_s ⇐ (i, j, k, l) {"Insert the ERI task into set C_s"}
12: if C_s.population = V then
13:   Issue all tasks in C_s for data-parallel processing
      {"Perfect utilization achieved for this set"}
14: end if
```

**Algorithm 2.** ERI sorting with windowing.

of Algorithm 1. Wherever the number of sets is reliably small,[3] this simple approach to ERI sorting may be employed.

### 3.2. Hash-based sorting

Fine grained sorting is made difficult by an explosion in the possible number of classes for a given workload [14], and thus an explosion in the number of set buffers required. Employing the approach in Algorithm 1 would not be practical because there would be far too many subsets to consider concurrently. To address this problem, one has to employ a "window" of subsets. The window greatly improves the scalability of ERI sorting by only considering a partial number of sets ($W_c$) at any given moment, thus allowing computations with a very large number of ERI tasks. In the event that an ERI task belongs to a class that is not currently represented by a subset contained within the window, then some existing subset is evicted prematurely (and the ERI tasks contained within the set are issued for processing) and is replaced with the new set, which is associated with a specific class of ERIs. This windowing functionality is incorporated into the ERI sorting algorithm listed in Algorithm 2. While this approach does occasionally require subsets to be evicted prematurely, thus incurring a performance penalty, in practice this scheme works well because premature eviction occurs rarely, and is a cheap operation anyway [7].

The next problem is "hidden" in line 2 of Algorithm 2; while it is functionally compact to specify the class-to-subset lookup/mapping function, the function may be computationally intensive. The approach taken with basic sorting is infeasible because of the $O(W_c)$ expense, since $W_c$ for fine grained sorting can be as large as 16 k [7]. Therefore, the *set.Lookup* function should employ a hash function to substantially reduce the cost of searching.

The general approach is illustrated in Fig. 3. The information relevant to identify the class of an ERI task (generally the level of contraction and shell type of each of the four basis functions forming the ERI quartet) are collected into a record called the *classID*. This record is run through a hash function; for reasons that will be explained in the next paragraph, *CRC32* is found to be a good function for this application. The resulting hash key is used to identify a partial list of the subsets presently in the system, i.e. the *ClassIDMap* data structure, which uniquely maps each *classID* to a set which contains task descriptors (i.e. the basis function quartets) for ERI tasks which all have the same class. With this approach, even though an exhaustive search of each *ClassIDMap* would still be required for a definitive mapping, the size of each *ClassIDMap* is very small, $\ll W_c$. In fact if the number of hash chains (i.e. the number of *ClassIDMap* structures) is made slightly larger than $W_c$, then on average the population of each *ClassIDMap* is close to 1, which means the expense of class-to-subset lookup is $O(1)$, and practically close to the cost of a single *CRC32* hashing of the *classID* data, which can be made as small as a single 64b word with simple concatenation of the various data fields. This sorting approach introduces roughly 100 ns overhead on a single core of an Opteron processor, where each ERI evaluation requires, on average, roughly 25 microseconds, which is an overhead of less than 1%.

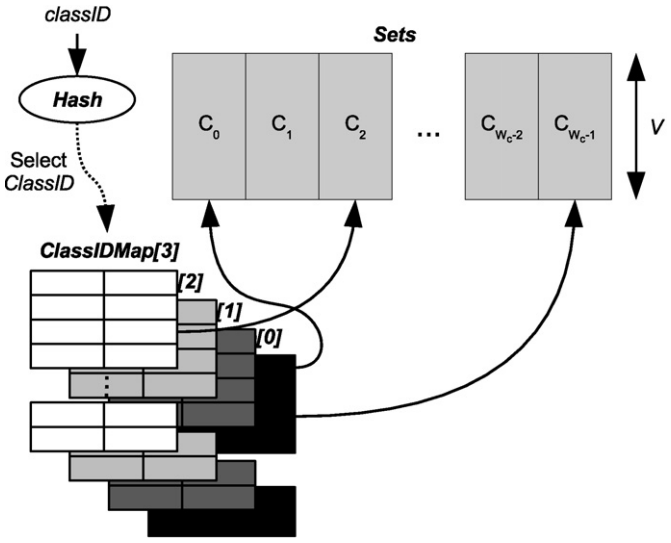



**Fig. 3.** Hash-based class-to-set lookup.

There are many hash functions that may be applied to this problem, and the performance of hash functions for this application may be evaluated based on two metrics: (1) the cost (in terms of required processing time) of evaluating the hash function, and (2) the uniformity of the hash chain coverage over a range of $W_c$. Our experimentation has been limited to three hash functions: a simple modulo hash, a shifted-XOR hash, and *cyclic redundancy check* (CRC). With this limited set of functions, CRC had the most consistent performance. Although the modulo and the XOR functions occasionally outperformed CRC, there were many performance pitfalls where hashing conflicts resulted in a very non-uniform distribution of hash chain selection – these conflicts tend to occur especially when $W_c$ is a power of 2. CRC suffers no such conflicts, and although the evaluation of the function was more costly than either the modulo or XOR functions, the cost was still very low.[4]

While very efficient, the overhead of the hash-based approach is still higher than that of the basic sorting approach. Furthermore, the hash-based approach introduces greater complexity into the system, and has higher memory requirements. Therefore, wherever possible, the basic approach described in Section 3.1 should be employed. Where the range of possible *classID*s – and therefore the number of required ERI task buffers – is large, then the hash-based approach should be employed. The middle ground between fine grained and coarse grained, corresponding to the middle ground

[3] This is, of course, relative to the architecture implementing the sorting. To be precise, the expense of this approach is $O(W_c)$, where $W_c$ is the number of classes (and therefore the number of buffers) considered of relevance, and in this example $W_c = 2$. For coarse-grained applications as previously described, it is expected that $W_c$ would be in the low single digits, and therefore this approach would be sufficient.

[4] This was the case for a look-up-table-based CRC32 implementation run on a conventional 64-bit processor core. A CRC64 implementation was also tested, and was found to be more costly than CRC32.

Please cite this article in press as: T. Ramdas et al., ERI sorting for emerging processor architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.01.029

between a very wide range of *classID*s and a limited number of possible *classID*s, is uncertain. Fortunately, as considered in the present work, the five applications of ERI Sorting either require a very large number of subsets, or very few.

## 4. ERI sorting applications

There are five applications of ERI sorting considered in the present work. SIMD processing and FPGA processing require a very large number of subsets, and hence employ the hash-based sorting method described in Section 3.2. ERI sorting for systems with limited instruction memories, improved instruction cache performance, and heterogeneous systems with different data cache capacity require ERI sorting with a very small number of subsets, and hence employ the basic sorting approach, as described in Section 3.1.

### 4.1. SIMD processing

ERI sorting was initially proposed for ERI processing on vector processors [16]. The issue of ERI sorting for SIMD processing was explored in a prior communication [7], which covered issues such as data access locality and sensitivity of performance to basis sets. Some salient points of that work will be reviewed here. SIMD processors distribute a workload over multiple processing elements (PEs), with the requirement that all PEs execute an identical sequence of instructions. If a set of $V$ ERI tasks are distributed to a set of $V$ PEs, maximum efficiency is only achieved if all ERI tasks are of identical class. Therefore, ERI sorting is employed to match ERI tasks which have identical class, and to issue these class-equivalent sets to a SIMD processor.

An important characteristic of SCF is that the number of ERI tasks rapidly increases with workload size, but this increase in the number of ERI tasks is not accompanied by an increase in the number of ERI classes. Therefore, one would expect that as workload size increases, there would be a larger number of available items for matching with a limited number of classes, resulting in increased utilization. This hypothesis is supported empirically by the results illustrated in Fig. 4. The $ZnCl_2$ workload has $> 20\times$ the number of tasks of the $H_2O$ workload, and it can be seen that utilization improves as workload size increases. Fig. 4 also shows that, as one would expect, a wider SIMD processor would be more difficult to fully utilize. It has been shown that, in general, workloads of significance are capable of saturating wide SIMD processors [7].

A counter-argument to the need for ERI sorting with SIMD processors is the work of Ufimtsev and Martinez on ERI processing with GPUs [17], which are a form of SIMD processor. This work did not employ ERI sorting. While high absolute performance was achieved (relative to the performance of conventional RISC processors), it is worth noting that the performance achieved was only a portion of the peak theoretical performance of the GPU device. Furthermore, the performance reported for a variety of workloads indicates a very large degree of workload sensitivity, where performance deteriorates for complex basis sets with a large variety of classes.

### 4.2. FPGAs

Special-purpose hardware conventionally includes two solution spaces: *Application Specific Integrated Circuits* (ASICs) and reconfigurable computing, which is currently synonymous (for practical intents and purposes) with *Field Programmable Gate Arrays* (FPGAs). ASICs are static processing pipelines with fixed functionality. One example of an ASIC solution to a computational sciences problem is the Gravity Pipe (GRAPE) [18] project, which has produced variants suitable for applications such as molecular mechanics [19],

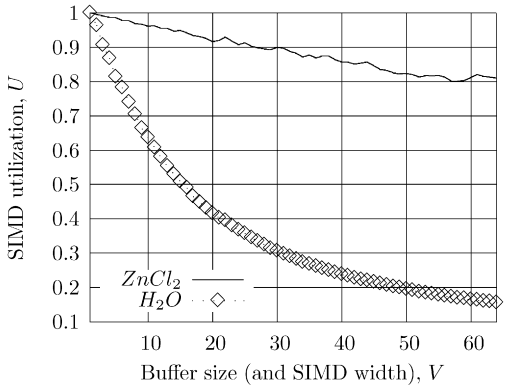



**Fig. 4.** SIMD utilization: effect of workload size. Tasks generated by the canonical SCF routine, with $ZnCl_2$ 6-31G(d, p) and $H_2O$ 6-31G(d, p). An infinitely large sorting window $S$ is assumed here.

whereby a special processing pipeline is employed to compute long-range charge interactions. Such a solution is problematic with ERI evaluation because of the various ERI classes, which require different processing instructions. A static pipeline is still possible, i.e. it is possible to implement a number of fixed datapaths for all ERI classes, however such an architecture would require the use of predication to nullify invalid execution branches, thus wasting processing cycles/logic area.

In contrast, because FPGAs may be reconfigured at run-time, it is possible to change the ERI execution datapath according to the requirements of individual ERIs. While there have been proposals for high-speed parallel context switching on RC devices [20], presently, as far as the authors are aware, FPGAs generally require relatively slow sequential reconfiguration in the order of milliseconds. It is therefore not practical to reconfigure for each individual ERI task.

However, employing ERI sorting, it is possible to construct sets of ERI tasks (each of which is an independent process thread) that will require the same FPGA configuration, and prior to streaming these tasks to the FPGA for processing, the FPGA would be configured with the required dataflow graph. The result of this is that the cost of FPGA reconfiguration is ameliorated over a large number of ERI evaluations.

Sorting for a higher value of $V$ sets up longer class-equivalent task sequences, thus reducing the required number of FPGA reconfigurations for a given workload, and correspondingly increasing the number of tasks processed per configuration, as illustrated in Fig. 5. Note that the *threads per reconfiguration* metric is similar to SIMD utilization, but its implications and behavior are different. While SIMD utilization is a measure of vector packing, reconfiguration minimization is purely a measure of class-equivalent task sequence length. For example, for $V = 4$, a workload with a set of 4 tasks of class $\tau_1$ and 2 tasks of class $\tau_2$ would have better SIMD utilization than a workload with 5 tasks of class $\tau_1$ and 1 task of class $\tau_2$, since the former would require 3 vectors and the latter case would require 2 vectors, while both would compute 6 tasks. On the other hand, both workloads would require exactly the same number of FPGA configurations, i.e. 2. The consequence of this is that with reconfiguration minimization, increasing $V$ will always either decrease or maintain the required number of reconfigurations, thus increasing or maintaining the number of tasks per reconfiguration. In contrast, increasing $V$ over a short-scale may decrease SIMD utilization, as illustrated in Fig. 4.

### 4.3. Limited instruction memories

Conventional architectures have provided very large memory spaces to accommodate very large programs (for example, the

Please cite this article in press as: T. Ramdas et al., ERI sorting for emerging processor architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.01.029

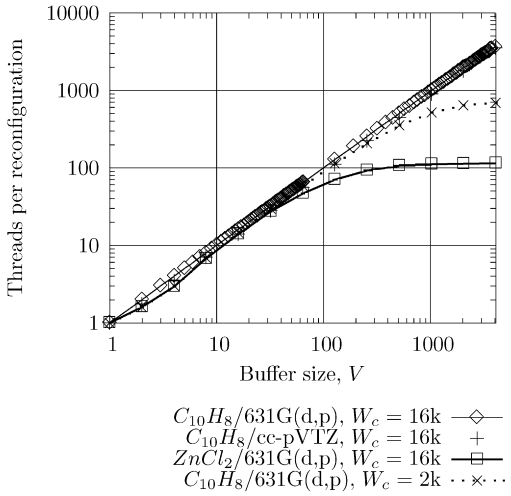



**Fig. 5.** FPGA ERI pipeline reconfigurations vs. $V$.

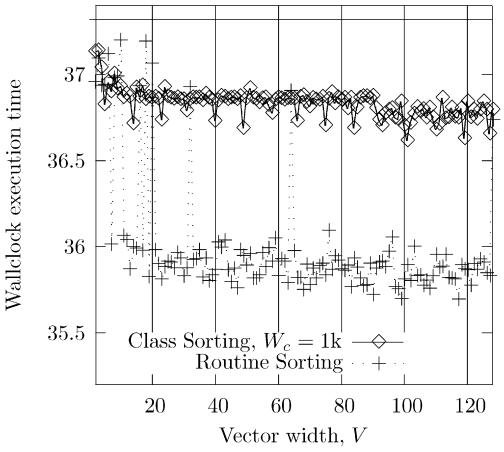



**Fig. 6.** Execution time with sorting vs. $V$, with the $C_{10}H_8$/6-31G(d,p) workload. Baseline performance (i.e. without any sorting) is 37.3183 s. These results were obtained with a version of GAMESS, modified to incorporate ERI sorting.

GAMESS executable is 16 MB large, and the object files for ERI processing alone require over 3.8 MB). It cannot be assumed that emerging architectures will continue to support such large instruction memories. This constraint is exemplified in the Cell BE architecture.

The Cell BE [3,21] contains 8 SIMD processing cores called SPUs. This subsection does not seek to address the SIMD processing problem; the issue here is that each SPU only has a 256 kB local store that must be used for data and instruction. If multiple ERI routines were to be used, this local store will very likely prove to be insufficient. Frequent code changes may be required to load different ERI routines into the SPUs for different ERI classes. This is undesirable because the code reload penalty will not only effectively "idle" the SPUs for a number of cycles, but the code transfer will consume valuable internal bandwidth that can otherwise be used for data transfer.

An alternative solution would be to fix the ERI evaluation routine on any given SPU, and to distribute these based on an a priori estimate of the likely distribution of ERI processing requirements based on workload. Therefore, one can "gear" the distribution of ERI evaluation algorithms across the coprocessors; for example, 3 of the 8 SPUs may employ Algorithm A, while the remaining 5 may employ Algorithm B. ERI sorting would then be responsible to match ERI tasks to either of the two classes (and a load-balancer would be responsible for distributing the load across each of the coprocessors implementing the same algorithm). When a subset reaches a particular size, it can be block-transferred to the target processor. Such transfer modes are highly desirable in architectures like the Cell BE, where initial transfer startup costs are high, but transfer of large blocks is, on aggregate, very efficient.

Clearly, this level of ERI sorting is substantially simplified because there are fewer matching classes, although the exact number of classes depends on the size of the ERI evaluation routine relative to the size of the instruction memory on the processor core in question. Processors with very limited instruction memories may have to resort to ERI evaluation programs constrained to a single class. In such cases, the ERI sorting and evaluation configuration is similar to that of FPGA-based solutions, where programs will have to be periodically reloaded onto the coprocessors. This incurs a performance hit analogous to an FPGA reconfiguration, and ERI sorting may be employed to minimize reprogramming frequency.

### 4.4. Instruction caching effects

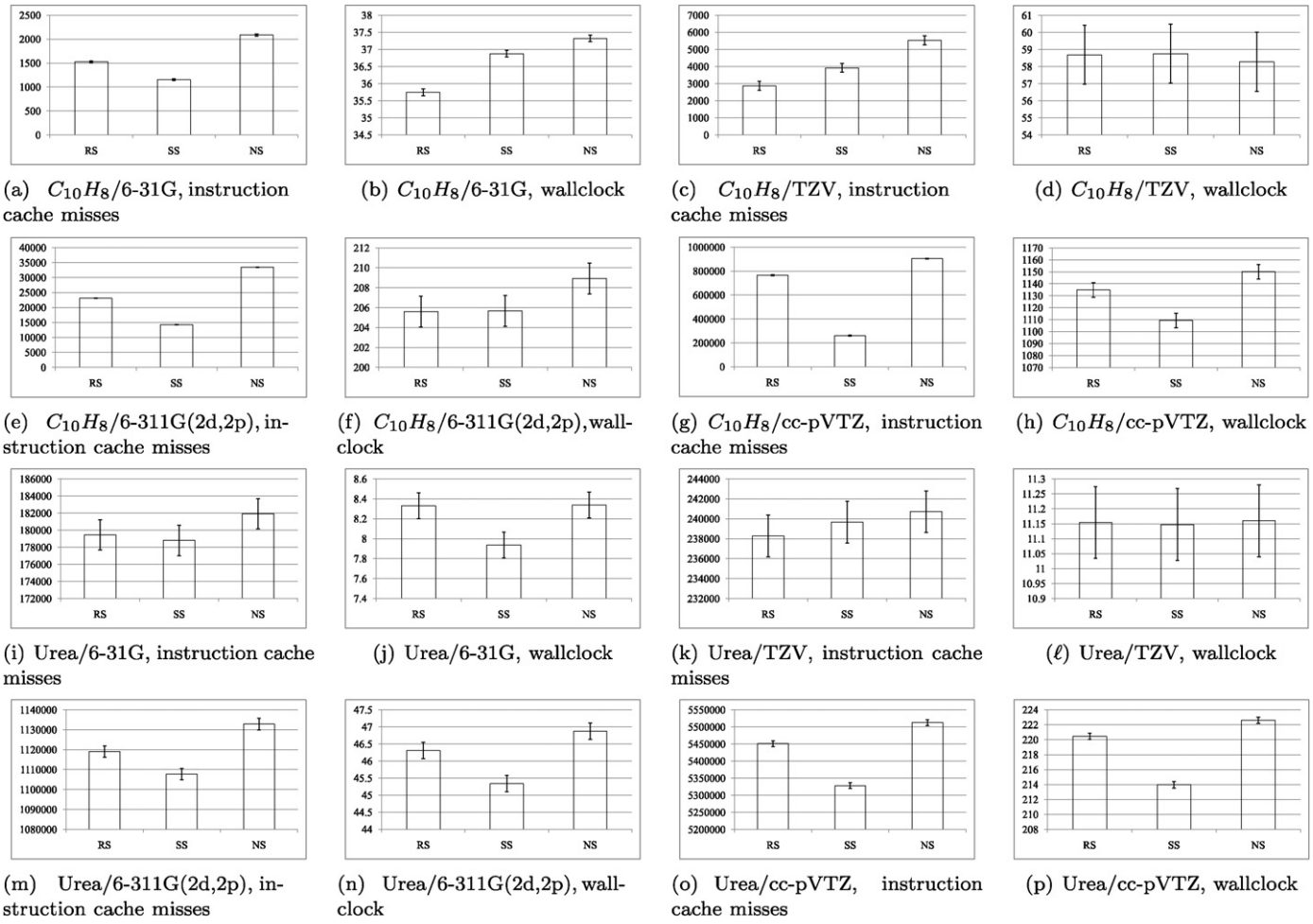
Although it is well known that caching can have a very significant impact on performance, consideration is most often only given to data caching, and instruction caching is usually neglected. It is usually unnecessary to consider instruction caching when designing/implementing scientific codes, because the size of hot-spot kernels is usually small. However, ERI evaluation kernels are sufficiently large that instruction caching can have a measurable impact on overall performance.

Fig. 6 presents the wallclock execution time[5] for a single-point energy RHF evaluation for a particular molecular basis set. The plot compares both ERI sorting methods against the same computation without ERI sorting. A significant improvement may be observed for both sorting techniques over the same computation without sorting. Class Sorting (a form of hash-based sorting as described in Section 3.2) achieves a speedup of 1.1% over the same execution without ERI sorting, while Routine Sorting (a form of basic sorting as described in Section 3.1) achieves a speedup of 4.4% over the same execution without ERI sorting. It should be noted that ERI sorting only affects ERI evaluation; therefore, it would be more accurate to determine the speedup of the ERI processing component of the computation in isolation. This may be done by profiling the application to determine the relative runtimes of the various subroutines within the program. For this particular workload, ERI processing accounts for approximately half the wallclock runtime. Therefore, it can be estimated that Routine Sorting results in a 9% speedup for ERI processing for the $C_{10}H_8$/6-31G workload.

The relationship between performance and $V$ in Fig. 6 indicates that $V$ should be sufficiently large so that sufficient code reuse can be achieved, thus realizing instruction cache performance gains. Setting $V$ as a power of two can result in very poor performance, possibly due to data cache conflicts, as the ERI sorter causes data cache pollution as it traverses buffers. Choosing $V$ as an odd or prime number is a reasonable strategy because this would minimize cache-line conflicts, and the results in Fig. 6 show that such values are favorable in terms of overall wallclock performance. At the upper bound, $V$ is constrained by the size of the data cache. If $V$ is too large, ERI task buffering will spill over the data cache boundary and into main memory, where ERI sorting will suffer long access latencies. Fortunately, the results in Fig. 6 indicate that it is not necessary to set $V$ to such a large value where there is a risk of data cache capacity misses; as a preliminary recommendation, a reasonable range for $V$ would be $21 \leqslant V \leqslant 127$ for both Class Sorting and Routine Sorting. For Class Sorting, the parameter $W_c$ must also be set. A low value results in poor sorting, thus not obtaining improved instruction cache utilization. A high

[5] These results were obtained from a single core of a 1.8 GHz Opteron processor.

Please cite this article in press as: T. Ramdas et al., ERI sorting for emerging processor architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.01.029




(a) $C_{10}H_8$/6-31G, instruction cache misses
(b) $C_{10}H_8$/6-31G, wallclock
(c) $C_{10}H_8$/TZV, instruction cache misses
(d) $C_{10}H_8$/TZV, wallclock
(e) $C_{10}H_8$/6-311G(2d,2p), instruction cache misses
(f) $C_{10}H_8$/6-311G(2d,2p),wallclock
(g) $C_{10}H_8$/cc-pVTZ, instruction cache misses
(h) $C_{10}H_8$/cc-pVTZ, wallclock
(i) Urea/6-31G, instruction cache misses
(j) Urea/6-31G, wallclock
(k) Urea/TZV, instruction cache misses
($\ell$) Urea/TZV, wallclock
(m) Urea/6-311G(2d,2p), instruction cache misses
(n) Urea/6-311G(2d,2p), wallclock
(o) Urea/cc-pVTZ, instruction cache misses
(p) Urea/cc-pVTZ, wallclock

**Fig. 7.** Instruction cache and wallclock runtime performance, canonical unsorted (NS – No Sorting) vs. Class Sorting (SS – equivalent to SIMD Sorting) with $W_c = 1$ vs. Routine Sorting (RS), $V = 99$. These results were obtained with a version of GAMESS, modified to incorporate ERI sorting.

value results in higher sorting overhead, thus negating the instruction cache gains. $W_c = 1$ k was found to be a good value for the system under test. The optima for all these parameters – Class Sorting vs. Routine Sorting, $V$, and $W_c$ for Class Sorting – will be architecture-specific. Furthermore, as indicated in Fig. 7, there is also a significant workload dependence, which will be discussed at the end of this section.

Fig. 6 indicates that Routine Sorting produces better performance. However, the results of performance counter sampling in Fig. 7 indicates that Class Sorting usually results in better instruction cache utilization. Class Sorting achieves lower instruction cache miss rates because each ERI routine (in this case Rys quadrature and ERIC) actually consists of a number of alternate code paths wrapped within a single routine. Sorting ERI tasks based on class (i.e. Class Sorting) exercises the same paths repeatedly. Sorting ERI tasks based on the more coarse grained routine level (i.e. Routine Sorting) results in better instruction cache reuse than the canonical unsorted approach, however, the divergence in execution path within each routine leads to instruction cache pollution between calls to the same subroutine. Nevertheless, the added overheads of Class Sorting – hash function evaluation, management of a large number of buffers, etc. – negate the wallclock gains that may have been achieved through better instruction cache performance, and ultimately Routine Sorting is highly competitive, outperforming Class Sorting in most of the workloads tested in Fig. 7.

It is indicated in Fig. 7 that ERI sorting performance appears to be sensitive to workload characteristics. For example, based on these results, comparing two different molecules for four different basis set types, one may make the following observations:

- For TZV basis, Routine Sorting (RS) achieves better instruction cache utilization than Class Sorting (SS). For all other basis types, SS achieves better instruction cache utilization than RS.
- Nevertheless, for TZV basis, ERI sorting appears to be of no benefit; wallclock performance of RS and SS is virtually identical to wallclock performance with no ERI sorting.
- For the correlated cc-pVTZ basis, SS achieves much better instruction cache utilization than RS, to the extent that SS outperforms RS in terms of overall wallclock.

These findings are not conclusive; more sampling should be done across a wider population of workloads in order to establish statistical significance. This point is made clear by the inconsistency in the results for 6-311G(2d, 2p) and 6-31G basis sets. These observations have been presented to demonstrate the possibilities inherent to such performance analyses. Furthermore, these findings should be qualitatively explained by incorporating a detailed understanding of workload characteristics.

For example, consider the results for cc-pVTZ basis workloads, i.e. Figs. 7(g), 7(h), 7(o), and 7(p). These results clearly favor Class Sorting over Routine Sorting to an extent beyond other basis sets. This is because the cc-pVTZ basis type produces basis sets with a very large population of uncontracted basis functions. With the existing GAMESS ERI routine selection driver, uncontracted quartets are processed with the Rys quadrature routine regardless of

Please cite this article in press as: T. Ramdas et al., ERI sorting for emerging processor architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.01.029

angular momenta. Therefore, the impact of Routine Sorting is diminished because the workload tends to run Rys quadrature very frequently. However, within the Rys quadrature code, there are alternate code paths that are dependent on angular momenta. Class Sorting optimizes the ERI task sequence to exercise these code paths consecutively, thus resulting in substantially improved instruction cache performance. Similar analysis should be conducted for the other basis set types, based on basis function population statistics and other information.

This workload-specific performance analysis reveals an opportunity for further work in the area of *software autotuning* [2] for ERI codes, where an input specification may take the form of:

- Salient characteristics of the processor system (e.g., details on the memory hierarchy, including information such as instruction cache size); and
- Information on the workload, e.g., basis set statistics such as the proportion of uncontracted basis functions.

This information can drive an expert system that will configure the computation, e.g., by setting appropriate pre-processor definitions and then compiling and linking the required software. Possible parameters include whether sorting should be enabled, and if so whether to employ Class Sorting or Routine Sorting, what value to set for $V$, and if Class Sorting is employed, what value would be appropriate for $W_c$.

### 4.5. Data cache/memory limits

The final use case for ERI sorting considered in this paper is to map ERI tasks (i.e. independent threads) to cores based on the thread's cache requirements and the core's cache capacity – such an approach may be employed to map ERI threads to processing cores based on bootstrap value dataset size and cache size. ERI task bootstrap storage requirement versus the target processor's cache capacity has been recognized as an important consideration in efficient ERI processing [8]. Considering the Rys quadrature specifically, the size of the data set required for storage of bootstrap values is estimated to be $\bar{K}^4 \times 80$ bytes [14], where $\bar{K}$ is the mean level of contraction of the four basis functions forming the ERI quartet.

In addition to being treated as a data cache issue, this is a more general data store issue. Consider the example of the Cell BE once again, where each coprocessor has a 256 kB address space which must hold instruction, data, and scratch space. Also consider a workload which contains basis functions with level of contraction as indicated in Fig. 8. With the data presented here, the largest bootstrap value dataset is requires for a quartet with basis functions contracted to a depth of $K = 8$, for which 320 kB of bootstrap data is required. Similarly, $\bar{K} = 6$ quartets require more than 100 kB of bootstrap data, which is likely to be too much when added on top of ERI processing program and other data such as the basis set. Such quartets are unsuitable for execution on the Cell BE coprocessors, and should be processed on the "host" PowerPC core instead. For this application ERI sorting would be performed on the basis of basis function level of contraction, similar to Algorithm 1. At least two sets would be maintained:

(i) A set for ERI tasks which should be processed on the PowerPC core; and
(ii) A set for ERI tasks which should be processed on the coprocessors.

In addition to this, if multiple ERI programs are employed on different coprocessors, Set 2 can be further partitioned into subsets based on which ERI program is required.

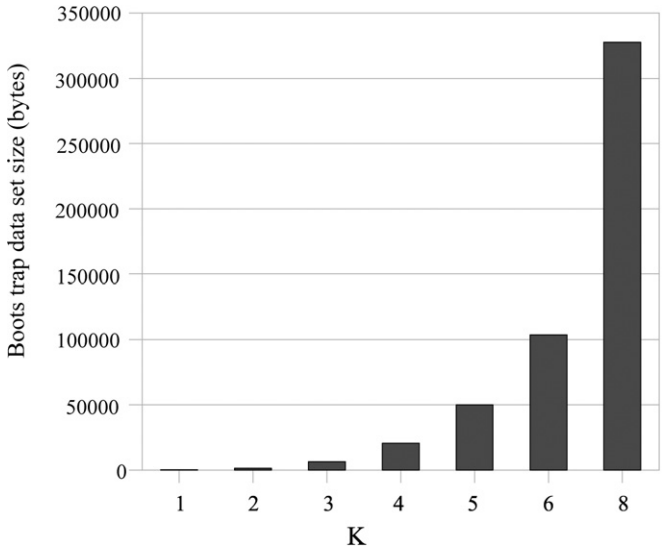



**Fig. 8.** Bootstrap data set size vs. $\bar{K}$.

For architectures with large address spaces but small data caches, this problem becomes an issue of performance. While it is possible to disable bootstrap value reuse entirely, this is not ideal because ERI processing time would increase substantially. If, however, there are a number of processor cores with different data cache allocations within a single processor system, ERI sorting may be employed to map ERI tasks (based on level of contraction) to individual processor cores, thus making efficient use of data caches.

The size of each set, i.e. the $V$ parameter, must also be considered carefully. Assuming the set of tasks will be block-transferred to the destination core, two issues arise:

(i) *Block transfer overheads*: Transferring blocks of data between processors in a multiprocessor system such as manycore architecture would suffer overheads and latencies which, in the example of the Cell BE, can be substantial [3]. To ameliorate the overhead, blocks should be made as large as possible.
(ii) *Storage limits*: Ideally, blocks would be streamed to coprocessors such that the rate at which ERI tasks arrive at coprocessors is matched by the rate at which they are processed by the coprocessor. In practice, the coprocessor would require sufficient memory to store complete blocks.

Therefore, $V$ should be maximized, constrained by the storage capacity of the target coprocessor. Considering the example of Cell BE coprocessors, with 256 kB storage capacity, the following rough assumptions may be made:

- ERI instruction program: 100 kB. This corresponds to a compact Rys quadrature formulation based on the formulation of Augspurger et al. [22].
- Set a limit on the basis functions which may be processed with this core: $K \leqslant 5$, which would require at most 50 kB of bootstrap data.
- With $K \leqslant 5$, a basis set with 200 basis functions would require 22 kB of storage, assuming a simple basis function data structure and a segmented basis set [23].

These approximations are best-case in the sense that they neglect alignment requirements. This also assumes no density matrix data is stored on the coprocessor; i.e. it is assumed that the coprocessor only evaluates ERIs, without performing on-the-fly Fock matrix reduction. The remaining 80 kB has to be shared between the list of ERI tasks and the resulting output. Each task, a record of four ba-

Please cite this article in press as: T. Ramdas et al., ERI sorting for emerging processor architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.01.029

sis function identifiers, may be represented by a single 64b word.[6] Each ERI task evaluates to a single real number, assumed to be a double-precision 64b value. Thus, halving the 80 kB between the input ERI task list and the output buffer, allows room for 10 k ERI tasks – thus $V \leqslant 10$ k. This is an extremely large value, and has been derived to show that an approach based on ERI sorting is fundamentally feasible. In practice, the 256 kB local store will be impossible to utilize so efficiently, and there will be other factors that limit $V$.

Also note that in this specific example ERI sorting for SIMD processing may also be relevant since the Cell BE coprocessors are SIMD processors. However, the SIMD width is narrow: i.e. $V = 2$ for double precision. Therefore, there may be sufficient data-parallelism within each ERI task, such as in recurrence relations and contraction unrolling [23], such that vectorizing over multiple ERI tasks with ERI sorting may be unnecessary.

## 5. Conclusions and further work

*Ab initio* computational chemistry is a highly complex application. Compounding this complexity, a challenge for the future is increasing complexity in computer architectures, a theme which formed a core motivation for this work. In this paper, we have demonstrated how a single technique – i.e. ERI sorting – can beneficially aid a single component of *ab initio* computations – i.e. ERI evaluation – for a number of different architectural configurations.

Two different mechanisms for ERI sorting have been described. A basic sorting method was proposed for applications that require coarse-grained sorting. A more sophisticated hash-based method was proposed for applications that require fine-grained sorting.

The benefits of ERI sorting for SIMD processing have been acknowledged for over two decades [24], and so this paper only provided a brief overview of ERI Sorting for SIMD (a more comprehensive treatment of the subject has been previously published [7]). The application of ERI sorting to reconfigurable computing (specifically with FPGAs) is conceptually similar to the case of SIMD processing. This is because the underlying assumption on the usage of FPGAs is that achieving high-performance requires the configuration of fixed-function data-parallel pipelines, with run-time reconfiguration employed to accommodate various ERI classes. This operation is conceptually the same as with SIMD architectures.

As processor architectures continue to diversify, it is difficult to predict what processing techniques will be appropriate in different situations. What is clear, however, is that the eventual likelihood of reductions in single threaded performance will force application designers to employ a finer grain of parallelism. One complication likely to arise from this is differentiation of memory/cache size of coprocessors within processor architectures. It has been shown that ERI sorting can be applied to dynamically manage workloads efficiently across a number of processors. It has also been shown that ERI sorting can yield overall speedup by enabling more efficient use of instruction caches, the effects of which can presently be realized on existing architectures.

It should also be noted that ERI sorting is a fairly “noninvasive” component that may be included into legacy codes with little effort. This was the experience of the authors with the GAMESS code.

It is difficult to reliably predict what form future computer architectures will take, especially at the fine level of detail that this paper has addressed. Therefore, whenever possible, it is beneficial to generalize existing techniques relevant to a specific architectures. ERI sorting has had limited importance in recent years, given the fact that computer architecture (especially in parallel supercomputing) has been dominated by coarse-grained parallel computers employing scalar processors. However, with computer architectures evolving in the manner described in this paper, ERI sorting may be posed to play a highly expanded role, enabling high performance electronic structure prediction over a wide variety of systems.

An important avenue for further research is to study the impacts of workload characteristics on the performance of techniques such as ERI sorting. Some comments were made in Section 4.4 alluding to the use of *software autotuning* technology, which is anticipated to be an important component in the future of parallel computing [2]. Our initial workload analysis for ERI sorting performance in the context of instruction cache utilization (Section 4.4) is only a small step towards the vision of autotuning for computational chemistry codes – much more work is required in this area to facilitate automatic tuning of computations for specific computer architectures and specific workloads. Autotuning technologies require an expert understanding of the complex interactions between computational workloads, computer architectures, and software design. Therefore, such research is best undertaken in collaboration with computer architects and computational scientists. Such efforts are necessary to enable efficient *ab initio* computations across a variety of computer architectures.

## References

[1] M.J. Irwin, J.P. Shen, Revitalizing computer architecture research, Technical report, Computing Research Association, 2005.
[2] K. Asanovic et al., The landscape of parallel computing research: A view from Berkeley, UCB/EECS-20060183, Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.
[3] J.A. Kahle, et al., IBM J. Res. Dev. 49 (2005) 589.
[4] E. Lindholm, J. Nockolls, S. Oberman, J. Montrym, IEEE Micro 28 (2008) 39.
[5] L. Seiler, et al., ACM Transactions on Graphics 27 (2008).
[6] B. Flachs, et al., Solid-state circuits, IEEE Journal of Solid-State Circuits 41 (2006) 63.
[7] T. Ramdas, G.K. Egan, D. Abramson, K.K. Baldridge, Comput. Phys. Comm. 178 (2008) 817.
[8] J. Antony, M.J. Frisch, A.P. Rendell, Modelling the performance of the Gaussian chemistry code on x86 architectures, in: Modeling, Simulation and Optimization of Complex Processes, Springer, Berlin–Heidelberg, 2008, pp. 49–58.
[9] J. Cioslowski, Ab initio calculations on large molecules: Methodology and applications, in: Reviews in Computational Chemistry, vol. 4, VCH Publishers, Inc., 1993, pp. 1–33.
[10] M. Dupuis, J. Rys, H.F. King, J. Chem. Phys. 65 (1976) 111.
[11] J. Rys, M. Dupuis, H.F. King, J. Comp. Chem. 4 (1983) 154.
[12] P.M.W. Gill, J.A. Pople, International Journal of Quantum Chemistry 40 (1991) 753.
[13] M.W. Schmidt, et al., J. Comp. Chem. 14 (1993) 1347.
[14] T. Ramdas, G.K. Egan, D. Abramson, K.K. Baldridge, Chemical Physics 349 (2008) 147.
[15] P.M.W. Gill, Adv. Quantum Chem. 25 (1994) 141.
[16] P.M.W. Gill, M. Head-Gordon, J.A. Pople, J. Phys. Chem. 94 (1990) 5564.
[17] I.S. Ufimtsev, T.J. Martinez, J. Chem. Theory Comput. 4 (2008) 222.
[18] J. Makino, Computing in Science and Engineering 8 (2006) 30.
[19] R. Susukita, et al., Comput. Phys. Comm. 155 (2003) 115.
[20] K. Puttegowda, Context switching strategies in a run-time reconfigurable system, Master's thesis, Virginia Polytechnic Institute and State University, Blacksburn, Virginia, USA, 2002.
[21] M. Gschwind, Chip multiprocessing and the cell broadband engine, in: CF'06: Proceedings of ACM Computing Frontiers 2006, 2006, pp. 1–7.
[22] J.D. Augspurger, D.E. Bernholdt, C.E. Dykstra, J. Comp. Chem. 11 (1990) 972.
[23] T. Ramdas, G.K. Egan, D. Abramson, K.K. Baldridge, Theor. Chem. Acc. 120 (2008) 133.
[24] V.R. Saunders, M.F. Guest, Comput. Phys. Comm. 26 (1982) 389.

[6] This allows 8b per basis function identifier, which allows for up to 256 basis functions.

Please cite this article in press as: T. Ramdas et al., ERI sorting for emerging processor architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.01.029