



# Uniting extrinsic vectorization and shell structure for efficient SIMD evaluation of electron repulsion integrals

Tirath Ramdas<sup>a</sup>, Gregory K. Egan<sup>a</sup>, David Abramson<sup>b</sup>, Kim K. Baldridge<sup>c,\*</sup>

<sup>a</sup> Center for Telecommunications and Information Engineering, Monash University, Australia

<sup>b</sup> Center for Distributed Systems and Software Engineering, Monash University, Australia

<sup>c</sup> Institute of Organic Chemistry, University of Zürich, 190 Winterthurerstrasse, 8057 Zurich, Switzerland

Received 14 December 2007; accepted 20 February 2008

Available online 26 February 2008

## Abstract

Future computer architectures are likely to feature greater reliance on single instruction multiple data (SIMD) processing for high throughput processing of data-intensive workloads. For algorithms that rely heavily on electron repulsion integrals (ERIs), exploitation of SIMD processing requires extrinsic vectorization, i.e. the sorting of ERIs into sets with equivalent class that may be computed with an identical instruction stream. Such sorting is incongruous with the commonly exploited shell structure whereby ERI are generated over shells such that initialization/bootstrap values may be reused, yielding significant savings in ERI evaluation time. In this work, we discuss how extrinsic vectorization may be unified with shell structure through the exploitation of memory access locality.

© 2008 Elsevier B.V. All rights reserved.

**Keywords:** Electron repulsion integrals; SIMD; Extrinsic vectorization; Shell structure; Memory access locality

## 1. Introduction

The need for greater computational capacity and capability in computational quantum chemistry prompts the development of high-performance computer systems, and efficient methods to exploit them. It is virtually undisputed that massively parallel hardware and efficient parallelization of algorithms is to be a critical feature of sophisticated ab initio computations going forward [1]. An indicative example of efficient parallelization of ab initio code may be found in the parallel *multireference singles and doubles configuration interaction* (MRCISD) implementation in the COLUMBUS program system, developed by Lischka and co-workers. Seminal works of Lischka in the use of parallel multireference gas and solution phase algorithms [2–4] have had important impact in molecular investigations of, for example, photodynamical processes, effects

of solvation, and modeling of absorption processes in soils, many of which are highly cited.

Two important features of parallel computational chemistry software have been (1) message passing – a ubiquitous concept in scalable parallel processing – and (2) memory sharing between processors in a convenient configuration called “virtual shared memory”. These ideas are meant to facilitate communication between individual cooperating processors that make up a large parallel processor. In addition, efficient software also implements load balancing with a global counter that may achieve excellent utilization with fine grain task distribution. However, fine grain balance incurs significant communications cost. Therefore, one has to identify a suitable task granularity such that good balance may be achieved without incurring overly large global counter communication costs. Selecting task granularity should be made based on the characteristics of a particular computer system, in particular the communications capabilities of the system.

The salient point with the parallelization of computational chemistry software, such as the Parallel COLUMBUS program, or the present focus, GAMESS [5], is that

\* Corresponding author. Tel.: +41 44 635 42 01; fax: +41 44 635 68 88.

E-mail addresses: [tirath.ramdas@eng.monash.edu.au](mailto:tirath.ramdas@eng.monash.edu.au) (T. Ramdas), [greg.egan@eng.monash.edu.au](mailto:greg.egan@eng.monash.edu.au) (G.K. Egan), [david.abramson@infotech.monash.edu.au](mailto:david.abramson@infotech.monash.edu.au) (D. Abramson), [kimb@oci.uzh.ch](mailto:kimb@oci.uzh.ch) (K.K. Baldridge).

significant effort had to be made in order to account for the architectural weak-point in the computer systems of interest. With parallel processors, the weak-point is often the interface between processors. Therefore, in dealing with parallel processing systems, groups such as Lischka and co-workers, as well as several GAMESS groups, have put significant effort into efficient inter-process communication for efficiently carrying out molecular computations.

Computers, like many real-life systems, exhibit different behaviors at different scales. Much concern is being given to the issue of node failures in emerging “petascale” computers [6], an issue which, if left unchecked, would result in massive computer systems that achieve very low utilization due to frequent failures and failure response. This problem is mirrored at a lower level in the computational hierarchy, and is cause for concern with the emerging “many-core” computing paradigm [7]. As fabrication feature size becomes extremely fine, dynamic failures become a bigger problem, which could substantially diminish achieved processing rates [8]. It is for this reason that, just as Lischka and co-workers addressed the character of parallel processing systems as they relate to *ab initio* computations, we attempt to address the character of emerging processor cores as they relate to *ab initio* computations.

There are serious challenges on the roadmap of high-performance semiconductor fabrication. In the pursuit of greater computational performance, one may sidestep these challenges and choose instead to optimize the logical architecture of processing nodes to achieve greater efficiency for workloads of interest. This was the reasoning behind graphics processing units (GPUs) (which are presently applied to a variety of problems beyond graphics [9]), and special-purpose processors such as the gravity pipe (GRAPE) [10] and bioinformatics arrays [11]. The heterogeneous multicore cell broadband engine [12] processor is also the product of significant architectural design effort. These are all examples of systems recognized for very high potential computational throughput.

While they are very different architectures, the cell BE, GPUs, GRAPE, and bioinformatics arrays have a common architectural foundation: *single instruction multiple data* (SIMD) processing, which is also the characterizing feature of vector processing. With SIMD processing, a set of data operands are computed with the exact same sequence of computer instructions. This makes very efficient use of control-oriented resources and enables greater emphasis on processing-oriented resources (e.g., a large number of floating point units), achieving greater overall throughput. The challenge lies in the fact that an application must have certain features in order to be amenable to SIMD processing. The task of porting an application from the scalar computing paradigm to the SIMD computing paradigm is commonly referred to as *vectorization*.

Much work has been done on the vectorization of *ab initio* methods, particularly in the 1980s when vector processing was dominant in the supercomputing sphere, in the shape of Cray and CDC Cyber systems, to name just

two. While tremendous developments have been achieved, particularly for matrix operations such as diagonalization (which is the bottleneck in MRCISD [4]), a problem that was not (as far as we are aware) conclusively solved is the computation of electron repulsion integrals (ERIs). Computing multiple integrals as a vector is difficult due to the variety of ERI classes, each of which requires a different instruction sequence.

A possible solution is to sort the integrals into batches with identical class, and to execute batches of integrals concurrently in vectors – an approach advocated by Saunders and Guest over two decades ago [13] that later came to be known as “extrinsic vectorization”. However, doing so means one has to forego the substantial savings that may be achieved by computing ERI in the order of the “shell structure” described by Pople and Hehre [14].

The virtue of integrating these two methods was recognized by Hegarty over two decades ago [15]. This early work identifies many problems and pitfalls that make the task at hand difficult, but it was also based on assumptions of architectures and workloads that may no longer be true. Ultimately, Hegarty concluded that simply foregoing the benefits of shell structure may be the best course of action for vector processors. Conversely, based on contemporary workloads and architectural developments, we conclude that merging these two methods may be feasible.

In this paper, we suggest that one may take advantage of the benefits of both extrinsic vectorization and shell structure value reuse by generating ERIs according to shell structure order, and then reordering ERIs into sets of matching class within a relatively small “window” of “in-flight” ERIs. A discussion of ERIs is presented with the background context of the Hartree-Fock method, followed by a detailed discussion on shell structure (data-equivalence batching) and extrinsic vectorization (control-equivalence batching). The procedure for combining both methods is depicted algorithmically. We present access trace locality results that suggest the proposal would obtain good SIMD performance, provided adequate cache provisioning and management. Finally, we suggest avenues to extend the present work.

## 2. Electron repulsion integrals

At the heart of our considerations lies the Hartree-Fock procedure, the basic underlying approximation method for determination of stationary points and associated energetics for quantum many-bodied systems, and the starting point for many other higher order algorithms. As is well known, the differential equations are solved using an iterative numerical technique, the *self-consistent field* (SCF) method, until a set of convergence criteria is met. While convergence is typically achieved within a relatively short number of iterations (e.g.  $\approx 20$  iterations [16]), the work involved can require up to  $O(N^4)$  computation time. Although methods such as integral screening and exploitation of molecular symmetry can reduce total work load to

$O(N^2)$  or less [16], there remains a significant body of work that requires substantial computational effort. Moreover, higher order methods, such as multireference configuration interaction techniques, have even higher order scaling complexity.

In the SCF procedure, Hartree-Fock energy is generated via

$$E_{HF} = \sum_i [i|h|i] + \frac{1}{2} \sum_{ij} [ii|jj] - [ij|ji] \quad (1)$$

This equation  $E_{HF}$  involves both relatively simple *one-electron integrals*,

$$[i|h|j] = \int \chi_i^*(x_1) h(r_1) \chi_j(x_1) dx_1 \quad (2)$$

and, more computationally intensive, *two-electron integrals* which, due to their physical significance, are called *electron repulsion integrals* (ERI).

$$[pq|rs] = \int \int \chi_p^*(1) \chi_q(1) \frac{1}{r_{12}} \chi_r^*(2) \chi_s(2) dr_1 dr_2 \quad (3)$$

Higher-order post-HF electron correlation methods require significant ERI effort. Consideration of the four-index transformation, an important component of electron correlation computations, scales  $O(N^5)$  with basis size [17]. The four-index transform operation requires four terms with similar form to the following:

$$[ij|kl] = \sum_p \sum_q \sum_r \sum_s [pq|rs] C_p^i C_q^j C_r^k C_s^l \quad (4)$$

Once again the evaluation of ERI (the  $[pq|rs]$  term) is an important and very significant computational component.

The typical basis set representation of the molecular system is presented as a linear combination of atomic orbitals (LCAO), formulated in terms of a Slater determinant. Within this formulation, the integrals over basis functions, as such, must accurately represent probability clouds describing the distribution of the electrons. Various basis sets are used in practice, most of which involve the use of Gaussian functions, as described below. Whenever, discussing the performance scaling of quantum chemistry computations,  $N$  almost always refers to the number of basis functions describing the molecule. With this measure, assuming no further approximations, problems involving 1000 basis functions are typically on the high end for practical considerations, although definitely the target with current computer architectures. A problem size of  $>10$  k basis functions does require serious consideration of associated numerical challenges.

The Gaussian representation of basis functions involves the following functional form:

$$\eta = C(x - R_x)^{n_x} (y - R_y)^{n_y} (z - R_z)^{n_z} e^{-\zeta(r-R)^2} \quad (5)$$

This is called a *Gaussian-type orbital* (GTO), also often referred to as a *primitive function* or *Gaussian primitive*

(pGTO). The characterizing elements for each function  $\eta$  are

- Center coordinate –  $r: \{x, y, z\}$  (Real  $\times 3$ ).
- Exponent –  $\zeta$  (Real).
- Angular momenta –  $n_x, n_y, n_z$  (Int  $\times 3$ ).
- Coefficient –  $C$  (Real).

The angular momenta are an important component of the GTO in the sense that the computation time of individual ERI depend on the angular momenta of the four input GTOs. The physical significance of the angular momenta is related to the electron configuration over “spdf...” shells.

Typically, a set of GTOs are combined linearly, or *contracted*, to form a *single contracted-GTO* (CGTO). The level of contraction,  $K$ , specifies the number of primitive functions collected into the single contracted function – higher  $K$  generally leads to higher accuracy. The CGTOs  $\chi$  are the basis functions. Each CGTO consists of a collection of GTOs, however all the GTOs share a common center coordinate and common angular momenta. A CGTO may be thought of as a container with the following elements:

- Center coordinate –  $r: \{x, y, z\}$ .
- Angular momenta –  $n_x, n_y, n_z$ .
- $K$  partial GTOs (pGTO), which only consist of a coefficient and an exponent.

The ERI computation involves a quartet (4-tuple) of CGTO, where each CGTO may have a different level of contraction  $K$ . A typical range for  $K$  is, e.g.  $1 \rightarrow 6$  for 6-31G (d,p) or related basis, and much higher (e.g.  $K = 15$ ) for more sophisticated basis such as Dunning basis sets. To summarize, an ERI evaluation requires a set of information

$$E_{ijkl} = \{(K, a_x, a_y, a_z, r, (\zeta_1, \zeta_2, \dots, \zeta_K), (C_1, C_2, \dots, C_K))_{i,j,k,l}\} \quad (6)$$

as illustrated in Fig. 1.

Integration over Gaussian functions warrants the application of several mathematical rules that simplify the computation, such as the Gaussian product rule [16]. Mathematical derivations are outside the scope of this paper; interested readers are referred to the reference lists of the papers cited within this section. Rather, in this section, we describe the broad characteristics common to many ERI algorithms, and then elaborate one specific ERI algorithm in detail.

The computation time for each ERI is determined by the level of contraction and the angular momenta associated with each CGTO, with higher values implying higher necessary computational effort. Most ERI algorithms consists of two broad steps

- Initial “bootstrap” step.
- Recurrence step.

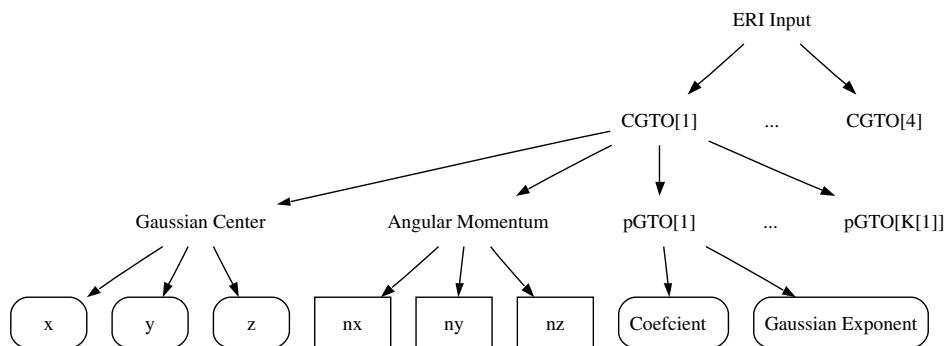


Fig. 1. ERI input. Typical range for  $K$  is  $1 \rightarrow 15$ , depending on basis. All pGTO/CGTO components are real values except the angular momentum  $\{nx, ny, nz\}$  triplet, which are unsigned integers.

The initial “bootstrap” step is so-called because the terms generated by this step enable the evaluation of ERIs through the use of efficient recurrence relations. As far as possible, complex arithmetic operations (such as exponential and square root evaluations) are limited to just the bootstrap step, which often executes in constant time. Conversely, the recurrence step – which may require up to  $O(N'^2)$  time, where  $N'$  is related to the angular momentum of the input GTOs – is limited to multiplication and addition operations as far as possible. Several salient points regarding the computational requirements of ERIs, common to all ERI algorithms, are

- All ERI are independent of each other, therefore, all ERIs may be computed in parallel.
- The computation time for individual ERIs may vary substantially.

The selected ERI algorithm in this paper is the *Rys quadrature* [18–20], which is an efficient yet generally applicable ERI algorithm. The implementation of the algorithm is also compact and relatively regular, which are desirable traits for a dedicated hardware implementation (and a possibility that we pursue [21,22]). In fact, a dedicated hardware implementation of this algorithm was suggested as far back as 1990 by Augspurger et al. [20], who specifically cite the concise/simple nature of the algorithm (and especially their particular implementation) as making it a good candidate for specialized hardware. Like many other ERI algorithms, Rys quadrature consists of a *bootstrap* stage and a *recurrence* stage. In addition, there is also a *roots* and *weights generation* stage.

Rys quadrature is a form of Gaussian quadrature, involving a  $N'$  point numerical quadrature using Rys polynomials – specifically the roots  $t$  and weights  $W$  of a polynomial of order  $N'$ , where  $N'$  is related to the total angular momentum of the four GTOs. The standard polynomial generation techniques typically employed for Gaussian quadratures may be employed, and the *discretized Stieltjes* [23] method is known to be especially good (in terms of computational efficiency and accuracy) for Rys quadrature. However, because the angular momentum is low in

practice (we target  $N'$  up to 13, which is sufficient for most practical computations), it is possible to partially tabulate values to generate the required roots and weights with less computational effort. Therefore, we will not elaborate on polynomial generation methods here but will cover this in a future communication.

The *Gaussian primitive* ERI problem is expressed as

$$[\eta_\mu \eta_\nu | \eta_\lambda \eta_\sigma] \equiv 2(\rho/\pi)^{1/2} \sum_{\alpha=1}^{N'} I_x(t_\alpha) I_y(t_\alpha) I_z(t_\alpha) W_\alpha \quad (7)$$

where  $t_\alpha$  represents the root of the polynomial corresponding to the weight  $W_\alpha$ . Beginning with the bootstrapping, one evaluates

$$I_x(0, 0, 0, 0, u) = \frac{\pi}{\sqrt{AB}} \exp \left( -\frac{\zeta_i \zeta_j}{\zeta_i + \zeta_j} (x_i - x_j)^2 - \frac{\zeta_k \zeta_l}{\zeta_k + \zeta_l} (x_k - x_l)^2 \right) \quad (8)$$

Recall that the  $\zeta$  terms are the exponents associated with the GTOs. The various coefficients are defined as follows:

$$A = \zeta_i + \zeta_j \quad (9)$$

$$B = \zeta_k + \zeta_l \quad (10)$$

$$\rho = \frac{AB}{A+B} \quad (11)$$

$$B_{00} = \frac{t_\alpha^2}{2(A+B)} \quad (12)$$

$$B_{10} = \frac{1}{2A} - \frac{B t_\alpha^2}{2A(A+B)} \quad (13)$$

$$B'_{10} = \frac{1}{2B} - \frac{A t_\alpha^2}{2B(A+B)} \quad (14)$$

$$x_A = \frac{\zeta_i x_i + \zeta_j x_j}{\zeta_i + \zeta_j} \quad (15)$$

$$x_B = \frac{\zeta_k x_k + \zeta_l x_l}{\zeta_k + \zeta_l} \quad (16)$$

$$C_{00} = (x_A - x_i) + \frac{B(x_B - x_A)t_\alpha^2}{A+B} \quad (17)$$

$$C'_{00} = (x_B - x_k) + \frac{A(x_A - x_B)t_\alpha^2}{A+B} \quad (18)$$

The roots  $t$  are not required in subsequent stages of computation, so the  $t$  dependence can be omitted in expressions of the  $I$ -factors. The following set of recurrence relations are then computed

$$\begin{aligned} I_x(n+1, 0, m, 0) &= nB_{10}I_x(n-1, 0, m, 0) \\ &+ mB_{00}I_x(n, 0, m-1, 0) + C_{00}I_x(n, 0, m, 0) \end{aligned} \quad (19)$$

$$\begin{aligned} I_x(n, 0, m+1, 0) &= mB'_{10}I_x(n, 0, m-1, 0) \\ &+ nB_{00}I_x(n-1, 0, m, 0) + C'_{00}I_x(n, 0, m, 0) \end{aligned} \quad (20)$$

$$\begin{aligned} I_x(a_x, b_x, m, 0) &= I_x(a_x+1, b_x-1, m, 0) \\ &+ (x_b - x_a)I_x(a_x, b_x-1, m, 0) \end{aligned} \quad (21)$$

$$\begin{aligned} I_x(a_x, b_x, c_x, d_x) &= I_x(a_x, b_x, c_x+1, d_x-1) \\ &+ (x_d - x_c)I_x(a_x, b_x, c_x, d_x-1) \end{aligned} \quad (22)$$

Eqs. (19) and (20) together populate a discrete two-dimensional plane, the width and breadth of which are a function of total angular momentum; we refer to this stage as the *2-index recurrence* (2iR). Next, a set of reduction operations is performed to effectively “shift” the dependence from two indices to all four indices. Eq. (21) essentially reduces values along partial strips of the 2iR plane; this stage is referred to as the *3-index shift* (3iS). Finally, Eq. (22) performs a reduction to similarly “shift” the dependence to all 4 indices, and is termed the *4-index shift* (4iS). Fig. 2 illustrates the computation of the recurrence relations. Observe the workload dependent nature of the 3iS and 4iS stages. The “footprint” of the 3iS reduction tree set is input-dependent in terms of dimensions and is always aligned to the lower right corner of the 2iR plane, but the dimensions of the 2iR plane itself are also workload-dependent. The width of the 4iS reduction tree depends on the width of the 3iS “footprint”. This is repeated for the  $y$  and  $z$  axes.

Thus far, only the Gaussian primitives (GTOs) have been dealt with. The contracted functions (CGTOs) must also be considered. To do this, one may simply iterate over all the pGTO contained within a CGTO. Alternatively, one may exploit other recurrence relations [24] to minimize the cost of highly contracted functions. For generality, we will ignore the potential vectorization benefits of contraction unrolling or any other potential vectorization benefits inherent within specific ERI evaluation routines.

### 2.1. Data-equivalence batching

We define a “shell” of basis functions as do Pople and Hehre [14]: a set of basis functions that have the same center coordinate  $r: \{x, y, z\}$  and exponents  $\zeta_K$  (where there are  $K$  different  $\zeta$  values within a CGTO). These are the only terms required to compute the “bootstrap” values for an ERI evaluation such as Rys quadrature: specifically, these are the only terms required to compute Eq. (8), as well as

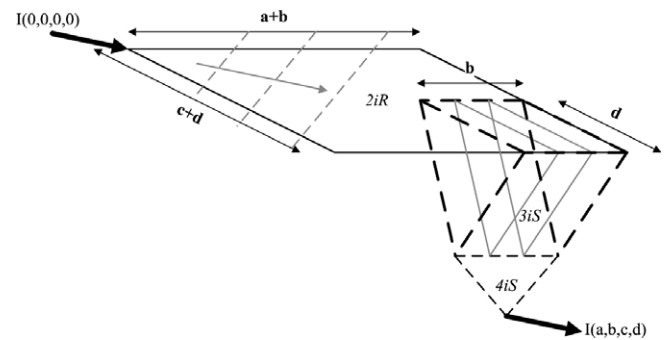


Fig. 2. Rys quadrature recurrence relations. This figure depicts the manner in which the three stages – *2-index recurrence* (2iR), *3-index shift* (3iS) and *4-index shift* (4iS) – are related. Note that  $a$ ,  $b$ ,  $c$ , and  $d$  are related to the angular momentum of the four input GTOs, and a typical maximum value for any of the four non-negative integers is 8.

the  $A$ ,  $B$ ,  $B_{00}$ ,  $B_{10}$ ,  $B'_{10}$ ,  $C_{00}$  and  $C'_{00}$  values. We define a bootstrap tuple  $\varpi$  to contain these values

$$\varpi = \{A, B, B_{00}, B_{10}, B'_{10}, C_{00}, C'_{00}, I_x(0), I_y(0), I_z(0)\}_{1 \dots (K_i \times K_j \times K_k \times K_l)} \quad (23)$$

Let us define a shell-tuple

$$\varsigma_{ijkl} = \{(r, K, (\zeta_1, \zeta_2, \dots, \zeta_K))_{i,j,k,l}\} \quad (24)$$

If the entire set of ERIs in a workload were contained in the universal set

$$U = \{E_{ijkl} | i, j, k, l = 1 \dots N\} \quad (25)$$

then the set  $S_n^\varsigma$  contains all ERI quartets with shell-tuple  $\varsigma_n$

$$S_n^\varsigma = \{x \in U : \varsigma(x) = \varsigma_n\} \quad (26)$$

(Practically, the container implementing  $S_n^\varsigma$  would also contain the common bootstrap tuple  $\varpi_n$ , however we omit this from Eq. (26) so that the set only contains elements of the same type, i.e. ERI quartets.)

The shell structure is easy to exploit, and is a feature of many codes such as GAMESS [5] and GAUSSIAN [14]. Before ERI quartet generation proceeds, basis functions are organized into shells. One then iterates over these shells when generating ERI quartets; this effectively enforces  $\varsigma$ -equivalent batching, since the “outer-loop” iteration over shells would hold a fixed  $\varsigma$  for a number of quartets. All these quartets would share bootstrap values, and their evaluation is therefore substantially less costly.

The number of ERI quartets and the number of shell quartets for several workloads are plotted on Fig. 3. A bigger gap between the number of shell quartets and the number of ERI quartets indicates that there would be more reuse of bootstrap values. We have also observed that the Dunning (cc-pVnZ) basis sets in particular enjoy a larger degree of bootstrap value reuse.

### 2.2. Control-equivalence batching

The computational procedure for each ERI depends on the contraction and angular momenta terms from each of

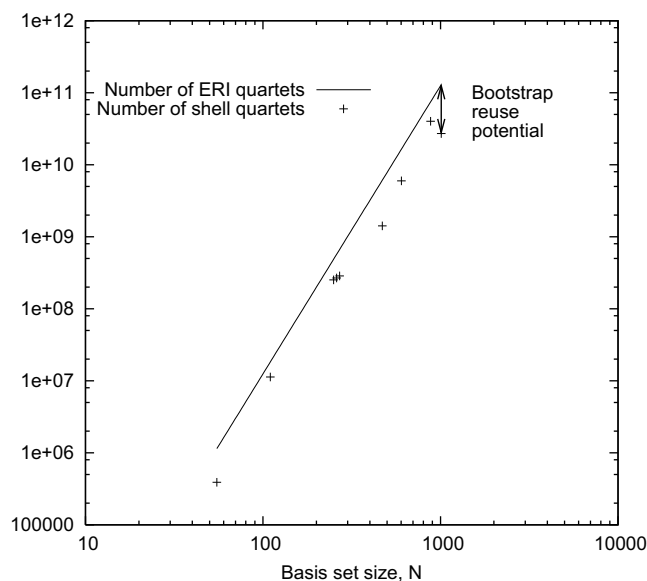


Fig. 3. Number of ERIs vs number of ERI shell quartets for various workloads.

the constituent basis functions. Hegarty suggests [15] that extrinsic vectorization should disregard level of contraction as a source of control-flow variation because including contraction places “too stringent” a requirement on ERI matching and would result in poor vector utilization. Our more recent findings, with larger workloads, suggests that the burden of ERI matching is generally relaxed as workload size increases [22]. Therefore we suggest that basis function contractions should also be matched – in doing so one does not need special irregular contraction vector instructions [15], and with contemporary workloads of interest achieving high vector utilization would not be difficult.

We may therefore specify the control tuple for any given ERI quartet

$$\tau_{ijkl} = \{(K, n_x, n_y, n_z)_{i,j,k,l}\} \quad (27)$$

The control tuple  $\tau_{ijkl}$  may uniquely and completely specify the exact computational procedure of an ERI evaluation  $[\chi_i \chi_j | \chi_k \chi_l]$ , down to the level of individual computer instructions. One may say that  $\tau$  represents the class of a given ERI. The set  $S_n^c$  contains all ERI quartets with control-tuple  $\tau_n$

$$S_n^c = \{x \in U : \tau(x) = \tau_n\} \quad (28)$$

All members of  $S_n^c$  may be computed with the exact same computer instructions, but with different data operands (i.e. arbitrary coefficients  $C$ , gaussian exponents  $\zeta$ , and center coordinates  $r: \{x, y, z\}$ ) – in other words, SIMD processing is a viable option. The extent to which SIMD processing is attractive is limited by the availability of a relatively large number of ERI tasks and a relatively low number of ERI classes. The former varies with workload size  $N$  (canonically one expects  $N^4/8$  ERI tasks for a workload of  $N$  basis functions), however the latter depends on

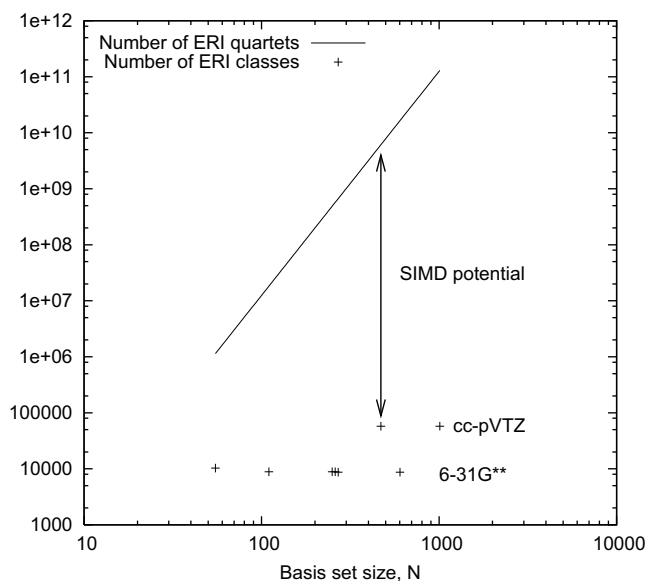


Fig. 4. Number of ERIs vs number of ERI classes for various workloads.

workload type, i.e. the nature of the basis set used for a particular computation. As illustrated in Fig. 4, 6-31G (d,p) produces fewer ERI class types than does a more sophisticated basis like Dunning cc-pVTZ [25]. More importantly, the number of ERI classes does not scale with workload size  $N$ . In fact, for a given basis with upper bound contraction  $K_{\max}$  and upper bound angular momentum term  $a_{\max}$ , the maximum number of ERI classes is given by the combinatorics expression  $K^4 \binom{a_{\max}+3}{3}^4$  (see Appendix A for a derivation). This presents tremendous opportunity for SIMD processing: batches of  $\tau$ -equivalent ERIs may be constructed from the universal set of ERIs in a given workload, and each batch may be computed with the exact same instruction sequence.

### 3. Exploiting control-equivalence and data-equivalence

Recall that the primary motivations for the techniques of interest are as follows:

*Extrinsic vectorization:* Efficient SIMD processing.

*Shell structure:* Bootstrap value reuse.

Comparing Fig. 3 with Fig. 4, it is apparent that  $\tau$  batches have much larger populations than  $\varsigma$  batches. In spite of this, shell structure is so ubiquitous and extrinsic vectorization is far less so. This is a reflection of the dominance of scalar processing nodes in mainstream scientific computing over the past two decades, relegating extrinsic vectorization to *niche* status. The benefits of shell structure to scalar processing systems are almost “free”, costing very little in terms of computational requirements, and providing substantial reduction in floating-point operations. However, as we shall soon see, shell structure ordering is not congruent with extrinsic vectorization ordering, and the latter is required to obtain good SIMD processor

utilization. For SIMD processing one may be tempted to simply abandon the advantages of shell structure, however in addition to substantially prolonging average per-ERI evaluation time, it would also make it more difficult to exploit several other optimizations that have been built on top of the shell structure, such as screening entire shell quartets and efficient exploitation of molecular symmetry [16,14]. This provides the motivation to combine  $\tau$ -equivalence batching (extrinsic vectorization) and  $\zeta$ -equivalence batching (shell structure).

If the entire set of ERIs in a workload were contained in the universal set  $U$  (Eq. (25)) one may seek to exploit shell structure to reuse bootstrap values by creating subsets of integrals with identical  $\zeta$  – i.e.  $S_n^\zeta$  (Eq. (26)) – or, similarly, one may seek to perform extrinsic vectorization by creating subsets of integrals with identical  $\tau$  – i.e.  $S_n^\tau$  (Eq. (28)). To combine  $\tau$ -equivalence batching and  $\zeta$ -equivalence batching one might require the intersection between  $S_n^\zeta$  and  $S_n^\tau$ .

However, observe that an ERI (Eq. (6)) is almost completely characterized by it is  $\tau$ -tuple (Eq. (27)) and  $\zeta$ -tuple (Eq. (24)), i.e.:

$$E_{ijkl} = \tau_{ijkl} \cup \zeta_{ijkl} \cup C_{ijkl} \quad (29)$$

The only information not contained by  $\tau_{ijkl} \cup \zeta_{ijkl}$  are the contraction coefficients, which are applied at the end of an ERI evaluation. Normally,  $S_n^\zeta \cap S_n^\tau$  would yield a very small set, typically with a single member. This leaves one with the benefits of neither extrinsic vectorization nor shell structure.

Thus far we have assumed that the sets  $S_n^\zeta$  and  $S_n^\tau$  exist within the universal set  $U$  which contains exactly one instance of each ERI quartet in a workload. If we relax this requirement, such that there are essentially two copies of the universal set  $U_1$  and  $U_2$ , and with  $S_n^\zeta$  sets existing only in  $U_1$  and  $S_n^\tau$  sets existing only in  $U_2$ , then we need not impose the requirement of  $S_n^\zeta \cap S_n^\tau$ . However, we require a practical means of uniting the two universes.

Algorithm 1 depicts the procedure we propose to obtain high SIMD utilization while simultaneously enjoying the benefits of bootstrap value reuse. Shells are prepared and bootstrap values are computed and stored locally a priori. One then proceeds with ERI quartet generation over shells of functions, which is common practice in many codes. As quartets are generated they are placed into  $\tau$ -matched batches, which are batches of quartets that require the exact same computational procedure (and computation time). When a batch reaches a particular (architecture specific) threshold, the entire  $\tau$ -batch of integrals is evaluated with a SIMD processor. The SIMD processor would load the required bootstrap values and proceed to evaluate all ERIs in the  $\tau$ -batch. Obviously this approach requires scatter–gather mode accesses; such a requirement would be placed on the chosen SIMD processor.

The algorithm proposed may easily be modified by moving the bootstrap tuple  $\varpi$  generation into the shell iteration loop; in doing so bootstrap values would only be computed and stored shortly before they are required, and subsequently bootstrap values may be removed from storage once

all the ERI quartets within the shell quartet have been computed (one may implement this by attaching a usage counter to each  $\varpi$  tuple; when the counter reaches zero, the tuple may be deleted from memory). Here we have presented the algorithm such that all bootstrap values are pre-computed before ERI quartet generation begins, for the sake of clarity.

---

#### Algorithm 1 Generating ERI quartets with exploitation of $\tau$ -equivalence and $\zeta$ -equivalence

---

```

1: for  $i = 1$  to  $i = N$  do {Prepare shells...}
2:   Put  $CGTO_i$  into appropriate  $S^\zeta$ 
3: end for
4: for  $i = 1$  to  $i = N_\zeta$  do {Prepare bootstrap constants...}
5:    $\varpi_i \leftarrow EvalBootstrap(S_i^\zeta)$ 
6: end for
7: for  $i', j', k', l' = 1$  to  $i', j', k', l' = N_\zeta$  do {Iterate over shells...}
8:   Generate  $E_{ijkl}$  quartets; pre-screen, etc.
9:    $S_{ijkl}^\tau.insert(E_{ijkl})$ 
10:  if  $S_{ijkl}^\tau$  sufficiently full then
11:    for All quartets  $q$  in  $S_{ijkl}^\tau$  in parallel do
12:       $\zeta_n \leftarrow q.\zeta$ 
13:       $\varpi_n \leftarrow BootstrapStorage[\zeta_n]$ 
14:      Result[q.index]  $\leftarrow EvalERI(\varpi_n, \tau_{ijkl})$ 
15:    end for
16:  end if
17: end for

```

---

There are several issues that warrant attention with this proposal. The first issue is to do with the cost of  $\tau$ -batching. Unlike  $\zeta$  batches (which are created a priori), the  $\tau$  batches are created at “runtime”: i.e. it is performed while ERI generation is proceeding. Furthermore, there are canonically  $O(N^4)$  ERI quartets to be  $\tau$ -batched. It is therefore imperative that the batching operation be very rapid. To meet this requirement we have suggested the use of a specialized memory structure [22], similar to the specialized memory systems used to perform high-speed address mapping in Internet routers. We have a preliminary *field-programmable gate array* (FPGA) prototype that suggests that high SIMD unit utilization and high throughput (enabling a speedup factor of 20 over existing processor cores) may be achieved with modest hardware requirements, and that an FPGA implementation is viable. Details of such an implementation are beyond the scope of this paper, and work continues on this component. Alternatively, we have also investigated the potential of sorting the integrals at run-time with a conventional processing core through an efficient hashing mechanism [26]. Such a solution would eliminate the need for special dedicated hardware or an FPGA coprocessor, however thus far we have found the sorting performance of conventional cores to be somewhat limited. However, further optimization of the code may yield a more useful outcome.

Another issue is the loss of regularity implied by runtime reordering of ERI evaluations, which randomizes basis function address patterns. We have observed that the severity of this is reduced by the high temporal locality exhibited by basis function accesses [27]. Therefore, high-speed caches will be very well utilized by the application.

A potential problem with the proposed approach is the heavy reliance on the *BootstrapStorage* element in Algorithm 1. We hypothesize that the proposed scheme will exhibit sufficient temporal locality such that implementing the *BootstrapStorage* element as a local high-speed cache with pre-caching would yield good results. However, there will be (hopefully rare) instances where a quartet contained in a  $\tau$  batch will be issued for execution, but the corresponding bootstrap values would not be available in the cache (e.g. the values may have been evicted from the cache due to capacity limits). In this case, although the SIMD computation proceeds, the ERI evaluation does not proceed for that particular quartet (in a practical implementation the evaluation may proceed, but it's invalid result is simply nullified). Clearly there will be a loss in SIMD processing efficiency. More importantly, there needs to be a mechanism for the unevaluated ERI quartet to be passed back into the system for scheduling. Employing our present prototype this is a relatively trivial process; the ERI quartet is simply passed back into the *ERI Sorter* block [22].

An important question was alluded to in the previous passage: *how much cache capacity is needed to yield good performance?* This question is considered in Section 4.

#### 4. Bootstrap access locality

A potential performance killer in computer systems is the *memory wall* phenomenon, whereby the processor idles for a very large number of cycles waiting for data to arrive from memory. To alleviate this problem, two features may be exploited

*Spatial locality:* One may *pre-fetch* data, i.e. initiate a data load from memory before the data is required, so that it reaches the processor by the time it is required. The challenge with this approach lies in knowing what data to pre-fetch. The simplest and most widely used approach is to assume that if a certain memory location is referenced, neighboring locations will soon also be referenced; if location  $i$  is read at cycle  $t$ , it is likely that location  $i + 1$  will be read at cycle  $t + 1$ . Therefore, when data  $i$  is loaded, one may also initiate loads for all data that are “close-by”, or *spatially local* (e.g.  $i \pm 1$ ). If the shell structure ordering were preserved (i.e. if extrinsic vectorization were not performed) then one would expect the indexing of  $\varpi$  tuples to be perfectly sequential, and therefore the application would be said to exhibit complete spatial locality.

*Temporal locality:* It has been observed that for many applications, a given piece of data may be referenced many times within a relatively short time period. This presents opportunities to employ hierarchical memory systems, with a small high-speed cache positioned very close to the processor core that may be rapidly accessed. If location  $i$  is read at cycle  $t$ , it is likely that the same data would also be read at cycle  $t + 1$ . Therefore when data  $i$  is loaded it is kept in cache on the assumption that

it will be needed again within a short time. Applications that exploit caches well are therefore considered to exhibit significant *temporal locality*.

In this section we estimate – through simulation – the degree of locality exhibited by  $\varpi$  tuple loading in Algorithm 1.

Algorithm 1 was implemented with an assumed vector width of 64, hence 64 ERI quartets of the same class should be issued concurrently to a SIMD processor in order to achieve maximum SIMD utilization. Typically, utilization rates of over 90% may be achieved.

As ERI quartets are issued for execution, bootstrap tuples ( $\varpi$ ) are loaded. The indices corresponding to different  $\varpi$  tuples are traced. These traces are then fed into a locality measurement kernel with the goal of identifying the level of spatial locality and temporal locality inherent within the traces. The methodology employed is based on metrics proposed by Weinberg et al. [28]. The simulations are performed for a variety of cache capacities: e.g. a cache capacity of 64 indicates that  $64 \times \varpi$  tuples may reside in the cache. One obtains a final ratio of accesses that were spatially local and temporally local. These two ratios may be summed to obtain a measure of the overall access locality exhibited by the computation: e.g. 95% combined locality suggests that 95% of accesses incur relatively low cost, and the remaining 5% may incur the very high expense of a complete DRAM access (having said that, even within the remaining 5% there may be pre-fetching techniques that work well for the application – this could be the subject of further work).

The results for various workloads are presented in Fig. 5. The results indicate that even with minimal caching high locality is achieved (>94%) due to the very good spatial locality exhibited by the computation. This is as expected. Although extrinsic vectorization executes ERI quartets in an order that is not exactly the same as the shell structure order, the reordering is only apparent within a relatively small window of ERI threads. On a coarse granularity, the general sequence of ERI quartet evaluation is still largely according to the shell structure ordering.

It appears that, roughly speaking, larger workloads exhibit greater locality – e.g. see the butane, nicotine, and caffeine 6-31G (d,p) locality curves in Fig. 5. Basis set type is also a factor: Dunning basis sets appear to yield especially high locality – e.g. note that although nicotine/6-31G (d,p) ( $N = 602$ ) is a larger workload than butane/cc-pVTZ ( $N = 470$ ), the latter exhibits greater locality.

#### 5. Further work

The results presented in the previous section were that of a locality measurement, with the intention of identifying locality patterns exhibited by the computation in terms of the sequence with which  $\varpi$  tuples are indexed. These results have indicated that there is substantial locality that may be exploited, and with this, one is motivated to explore the design and configuration of a memory system.

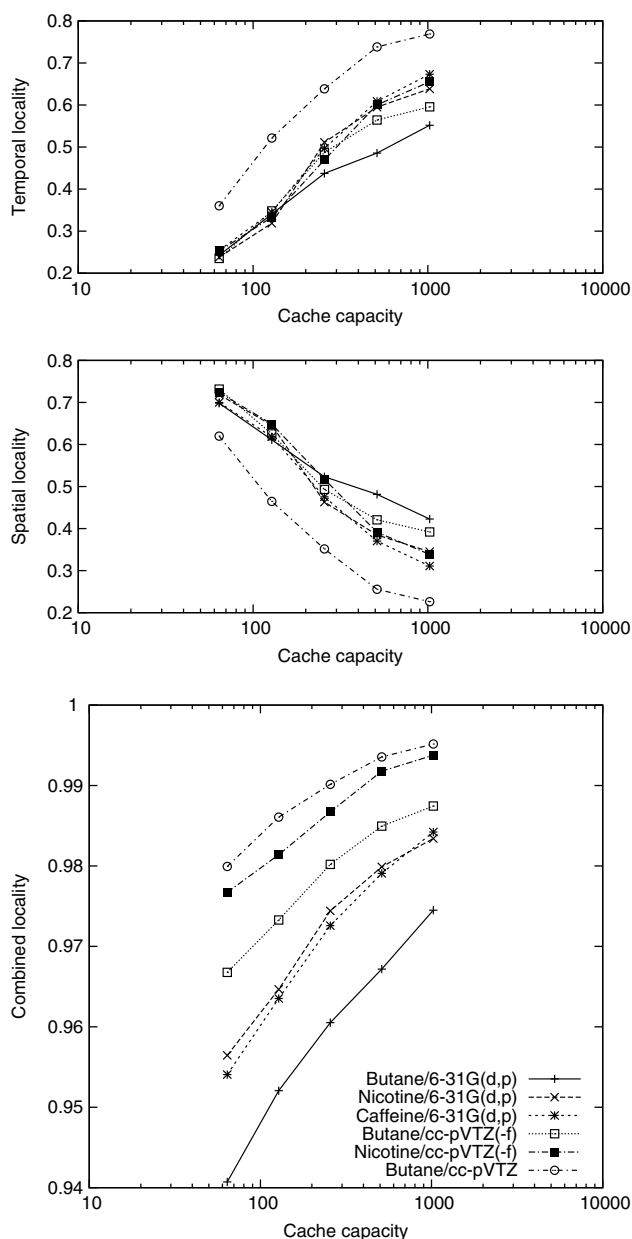


Fig. 5. BootstrapStorage access locality.

There are several aspects to the design of contemporary caches, such as set-associativity, cache word length, cache sharing (in the context of multi/many-core architectures) etc. However, the variable size of  $\varpi$  tuples introduces additional complications.

We assume that a  $\varpi$  tuple should be completely cached in order for it to be useful; if only half the elements of a  $\varpi$  tuple are present in the cache, then the tuple is assumed to be of no use. The size of a  $\varpi$  tuple scales  $O(\bar{K})^4$  – where  $\bar{K}$  is the mean level of contraction of the constituent shells – therefore there would be very large variation in the size of  $\varpi$  tuples. This suggests that caching  $\varpi$  tuples would likely require some “intelligent” cache management. Firstly, the placement of a  $\varpi$  tuple has to be done in a manner that minimizes cache memory fragmentation. Secondly,

when evicting a  $\varpi$  tuple from cache in order to allocate the space to a new tuple (probably utilizing the common “least recently used” eviction policy), one may have to evict multiple tuples from the cache if the new tuple is large. Alternatively, one may select tuples for eviction based on size rather than usage history, so that more  $\varpi$  tuples may reside in cache for the benefit of more ERI quartets. The variable size of  $\varpi$  tuples introduces application-specific complexity into the design of a cache system that may ultimately only be resolved through application-specific explicit cache management in place of generally applied cache policies. While not ubiquitous, such explicit cache management is in fact well entrenched in the embedded computing domain, and is a requirement for efficient use of the Cell BE architecture as well [29].

It may prove beneficial to simply introduce an upper bound on  $\varpi$  tuple size, such that only tuples of less than a certain size are cached. This would be a necessary step in certain cases. Note that the size of a  $\varpi$  tuple is  $\bar{K}^4 \times 80$  bytes. With  $\bar{K} = 15$ , one has to generate and store a 4MB  $\varpi$  tuple, which is clearly a serious problem. In these cases the most practical course of action may be to simply recompute the required bootstrap values at ERI quartet evaluation time, rather than preparing them a priori. Although basis sets do include functions with very high contraction, most common workload would still be dominated by lowly-contracted or uncontracted functions, therefore in most cases the vast majority of  $\varpi$  tuples would be reasonably small and convenient to cache.

## 6. Conclusion

It has been stated that for a significant period of time, the evolution of early Cray vector supercomputers was characterized not by improving vector performance, but rather by improving scalar performance [30]. Although it is premature to categorically declare that the opposite reaction is occurring today, it is difficult to ignore the fact that some of the most significant realized improvements in computer performance have recently come from improving vector performance, as evidenced by GPUs, the Cell BE, and potentially even the upcoming Intel Larrabee architecture [31].

Although great progress was made in the vectorization of ab initio methods for early vector supercomputers, it is necessary to re-evaluate contemporary architectural characteristics and workload characteristics to identify new pitfalls and new opportunities. One example of a new opportunity is the unification of extrinsic vectorization (for the sake of high SIMD efficiency) and shell structure (for the sake of bootstrap value reuse, as well as shell-based symmetry and screening).

By generating ERI quartets according to the shell structure and reordering these within a relatively small window, one may exploit pre-fetching and caching to make repeated use of bootstrap values in a manner that is sequential on a coarse-grain but out-of-order on a fine-grain. Access locality results indicate that, overall, the access patterns are

highly predictable and amenable to pre-fetching and caching. The development of SIMD ERI evaluation architectures and codes may therefore proceed, with the anticipation of seamless integration with existing shell based ERI quartet generation schemes in stable production-level codes.

## Appendix A. Maximum number of ERI classes for a given basis

The class of an ERI evaluation task  $[\chi_i\chi_j|\chi_k\chi_l]$  is provided by its control-tuple  $\tau$ , which as specified by Eq. (27), consists of the level of contraction  $K$  and angular momenta terms  $(n_x, n_y, n_z)$  for each basis function  $\chi_i, \chi_j, \chi_k$  and  $\chi_l$ . Determining the maximum possible number of ERI classes, corresponding to the maximum possible number of unique  $\tau$  terms, based on the maximum angular momenta and maximum level of contraction for a given basis is a permutational and combinatorics problem.

Let  $N_\tau$  be the number of ERI tasks. We first deal with the permutational possibilities introduced by contraction. Let the maximum level of contraction of a given basis be specified by  $K_{\max}$ : e.g. for 6-31G (d,p)  $K_{\max} = 6$ , whereas for cc-pVTZ  $K$  goes as high as 15. Due to the quartet structure of the ERI, where each basis function has an associated  $K$ , where  $1 \leq K \leq K_{\max}$ :

$$N_\tau \propto K_{\max}^4 \quad (\text{A1})$$

The angular momenta dependence is a combinatorics problem. We define the maximum angular momenta  $n_{\max}$  as:  $n_x + n_y + n_z \leq n_{\max}$ . In addition, we define a term  $n_l$ , where  $n_x + n_y + n_z = n_l$ , and  $n_l \leq n_{\max}$ . The term  $n_{\max}$  is the set of all  $n_l$ , where  $0 \leq n_l \leq n_{\max}$ . Therefore, for  $n_{\max} = 2$ , the set of possible  $n_l$  is (0, 1, 2) and one can expect to see the following  $(n_x, n_y, n_z)$  tuples in a basis set: (000), (001), (010), (100), (002), (020), (200), (110), (101) and (011).

Conceptually, one may think of this as a partitioning problem. We have to make three discrete batches out of a global set of identical objects, corresponding to the three angular momenta terms  $(n_x, n_y, n_z)$ . To make three batches from a universal set, one has to insert two partitions. To allow for the possibility of two out of three batches to consist of zero “items” (i.e. an angular momenta term  $n_{\{x|y|z\}} = 0$ ) we allocate a global set of  $n_l + 2$ . We then have to choose the position of two partitions, which corresponds to “ $(n_l + 2)$  choose 2”, or  ${}^{n_l+2}C_2$ , and recall that:

$${}^nC_r = \frac{n!}{(n-r)!r!} \quad (\text{A2})$$

Extending this reasoning to  $n_{\max}$  one may conceive of a “fourth batch” which allows  $n_l$  to iterate from  $0 \rightarrow n_{\max}$ . For example let  $n_{\max} = 4$ ; then with  $n_l = 4$  the fourth set is empty, with  $n_l = 3$  the fourth set contains 1, etc. The combinatorics expression is therefore  ${}^{n_{\max}+3}C_3$ . Incorporating quartet permutation, one obtains

$$N_\tau \propto ({}^{n_{\max}+3}C_3)^4 \quad (\text{A3})$$

and combining (A1) and (A3) one obtains:

$$N_\tau = K^4 ({}^{n_{\max}+3}C_3)^4 \quad (\text{A4})$$

Note that this derivation neglects any form of symmetry. Also, the number of ERI classes specified by  $N_\tau$  is an upper-bound which may not be practically reached for common workloads; e.g. even if a given basis has a very high  $n_{\max}$  the highest observed  $n_l$  for common workloads (e.g. excluding transition metals) may be low in practice.

## References

- [1] M.F. Guest, P. Sherwood, J.A. Nichols, *Massive Parallelism: The Hardware for Computational Chemistry* (1998), URL <<http://www.cfs.dl.ac.uk/parallel/MPP/mpp.shtml>>.
- [2] R. Ahlrichs, H.-J. Bohm, C. Ehrhardt, P. Scharf, H. Schiffer, H. Lischka, M. Schindler, *J. Comput. Chem.* 6 (1984) 200.
- [3] R. Shepard, I. Shavitt, H. Lischka, *J. Comput. Chem.* 23 (2001) 1121.
- [4] H. Dachsel, H. Lischka, R. Shepard, J. Nieplocha, R.J. Harrison, *J. Comput. Chem.* 18 (1996) 430.
- [5] M.W. Schmidt, K.K. Baldridge, J.A. Boatz, S.T. Elbert, M.S. Gordon, J.H. Jensen, S. Koseki, N. Matsunaga, K.A. Nguyen, S. Su, et al., *J. Comput. Chem.* 14 (1993) 1347.
- [6] B. Schroeder, G.A. Gibson, *J. Phys.: Conf. Ser.* 78 (2007).
- [7] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al., Tech. Rep. Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006, URL <<http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>>.
- [8] M.J. Irwin, J.P. Shen, Tech. Rep., Computing Research Association, 2005, URL <<http://www.cra.org/Activities/grand.challenges/architecture/home.html>>.
- [9] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T.J. Purcell, in: *Eurographics 2005, State of the Art Reports*, 2005, p. 21.
- [10] J. Makino, *Comput. Sci. Eng.* 8 (2006) 30.
- [11] T. Ramdas, G. Egan, in: *Proceedings of IEEE TENCON 2005*, 2005.
- [12] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, in: *CF'06: Proceedings of ACM Computing Frontiers 2006*, 2006, p. 9.
- [13] V.R. Saunders, M.F. Guest, *Comput. Phys. Commun.* 26 (1982) 389.
- [14] J.A. Pople, W.J. Hehre, *J. Comput. Phys.* 27 (1978) 161.
- [15] D. Hegarty, *Proceedings of the NATO Advanced Research Workshop on Vectorization of Advanced Methods for Molecular Electronic Structure*, Kluwer Academic Publishers, Hingham, MA, USA, 1983, p. 39.
- [16] J. Cioslowski, *Ab initio calculations on large molecules: methodology and applications* *Reviews in Computational Chemistry*, vol. 4, VCH Publishers Inc., 1994, p. 1.
- [17] S. Wilson, *Four-index transformations* *Methods in Computational Chemistry*, vol. 1, Plenum Press, 1987, p. 251 (Chapter 3).
- [18] M. Dupuis, J. Rys, H.F. King, *J. Chem. Phys.* 65 (1976) 111.
- [19] J. Rys, M. Dupuis, H.F. King, *J. Comput. Chem.* 4 (1983) 154.
- [20] J.D. Augspurger, D.E. Bernholdt, C.E. Dykstra, *J. Comput. Chem.* 11 (1990) 972.
- [21] T. Ramdas, G.K. Egan, D. Abramson, K. Baldridge, *Theoretical Chemistry Accounts* (2007), URL <<http://dx.doi.org/10.1007/s00214-007-0306-6>>.
- [22] T. Ramdas, G.K. Egan, D. Abramson, K. Baldridge, in: *CF'07: Proceedings of ACM Computing Frontiers 2007*, 2007, p. 267.
- [23] R.P. Sagar, J. Vedene, H. Smith, *Int. J. Quantum Chem.* 42 (1992) 827.
- [24] M. Dupuis, A. Marquez, *J. Chem. Phys.* 114 (2001) 2067.
- [25] T.H. Dunning, *J. Chem. Phys.* 90 (1989) 1007.
- [26] T. Ramdas, G. Egan, D. Abramson, Tech. Rep., Center for Telecommunications and Information Engineering, Monash

- University, 2007), URL <<http://www.ds.eng.monash.edu.au/techrep/reports/2007/MECSE-11-2007.pdf>>.
- [27] T. Ramdas, G.K. Egan, D. Abramson, K.K. Baldrige, *Comput. Phys. Commun.*, 2008, URL <<http://dx.doi.org/10.1016/j.cpc.2008.01.045>>.
- [28] J. Weinberg, M.O. McCracken, E. Strohmaier, A. Snavely, in: SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, 2005, p. 50.
- [29] M. Gschwind, in: CF'06: Proceedings of ACM Computing Frontiers 2006, 2006, p. 1.
- [30] M.F. Guest, S. Wilson, *Supercomputers in Chemistry*, American Chemical Society, 1981, p. 1 (Chapter 1).
- [31] J. Stokes, Clearing up the confusion over Intel's Larrabee, 2007, URL <<http://arstechnica.com/articles/paedia/hardware/clearing-up-the-confusion-over-intels-larrabee.ars/1>>.