

Converting Massive TLP to DLP: a Special-Purpose Processor for Molecular Orbital Computations

Tirath Ramdas
tirath.ramdas@eng.monash.edu.au
Center for Telecommunications
and Information Engineering
Monash University

David Abramson
david.abramson@infotech.monash.edu.au
Center for Distributed Systems and
Software Engineering
Monash University

Gregory K. Egan
greg.egan@eng.monash.edu.au
Center for Telecommunications
and Information Engineering
Monash University

Kim Baldridge
kimb@oci.unizh.ch
Organic Chemistry Institute
University of Zurich

ABSTRACT

We propose an application specific processor for computational quantum chemistry. The kernel of interest is the computation of electron repulsion integrals (ERIs), which vary in control flow with different input data. This lack of uniformity limits the level of data-level parallelism (DLP) inherent in the application, thus apparently rendering a SIMD architecture unfeasible. All ERIs may be computed in parallel, therefore there is much thread-level parallelism (TLP). We observe that it is possible to match threads with certain characteristics in a manner that reveals significant DLP across multiple threads. Our thread matching and scheduling scheme effectively converts TLP to DLP, allowing SIMD processing which was previously unfeasible. We envision that this approach may expose DLP in other applications traditionally considered to be poor candidates for SIMD computation.

Categories and Subject Descriptors: C.1.2 [PROCESSOR ARCHITECTURES]: Multiple Data Stream Architectures (Multiprocessors)—*Single-instruction-stream, multiple-data-stream processors (SIMD), Associative processors*; C.3 [SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS]; J.2 [PHYSICAL SCIENCES AND ENGINEERING]: Chemistry, Physics)

General Terms: Algorithms, Design, Performance

Keywords: Thread-level parallelism, data-level parallelism, content-addressable memory, vector processing, address generation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CF'07, May 7–9, 2007, Ischia, Italy.

Copyright 2007 ACM 978-1-59593-683-7/07/0005 ...\$5.00.

1. INTRODUCTION

Modern computer architectures must exploit parallelism to be competitive, and this is especially the case for scientific numerical computation. Scientific applications are extremely demanding computationally and are generally rich in parallelism. For instance, it is well known that massive parallelism is required to break new ground in computational chemistry [13].

In the context of computing, parallelism comes in many flavours, the main major forms being data level parallelism (DLP), instruction level parallelism (ILP) and thread level parallelism (TLP). Recently, there has been a great deal of emphasis on exploiting TLP. At *ACM Computing Frontiers 2006* [3] an entire track was devoted to multithreaded systems such as *chip multiprocessor* (CMP) architectures, which are designed in particular to exploit TLP efficiently and to scale easily with incremental technology improvements. This reflects the reality that many of today's computing challenges exhibit a high degree of TLP.

In fact, large TLP is a characteristic of our target application: *Molecular Orbital* (MO) computations, using the Hartree-Fock Self-Consistent Field algorithm¹, which essentially aims to solve Schrödinger's equation for a given molecular configuration. This application yields important information on the chemical behaviour of a molecule, such as details of the electronic structure, mechanism of reaction, and vibrational phenomenon, the results of which fill in important gaps in many domains such as drug discovery, nanotechnology, and materials science.

In general, the algorithm suffers $O(N^4)$ scaling. Specifically, this scaling applies to the number of *Electron Repulsion Integral* (ERI) procedure calls required. In fact, ERIs present a “double-whammy”: 1) the massive number of ERIs to compute, and 2) each ERI is relatively computationally demanding, with control flow that varies with the nature of the data. Fortunately, each ERI is independent of the others, i.e. one may compute all ERI concurrently. The salient

¹Note that our application is based on quantum mechanics, which is not the same as quantum chromodynamics, which was the problem targeted by the QCDOC system [9].

points are that there is a massive number of ERI to compute, and all ERIs may be computed concurrently – in other words, there is massive TLP.

On the other hand, the amount of DLP within each thread is limited. Vector processing – the traditional means of exploiting DLP – of individual ERI has provided limited gains [25][4][19] (as far as we are aware, the best achieved speedup on modern systems is only around a factor of 5 [19]). Immediate vectorization over all ERIs is not possible because each ERI has different control-flow. From our analysis and profiling we know there is only some limited ILP within each thread, and certainly does not constitute “massive” parallelism. With this in mind, one feels compelled to immediately consider a homogeneous CMP architecture for the problem. Computing individual ERI threads in parallel processing elements (PEs) is a natural path to consider, however one additional issue to consider is the impact of varying execution time of each ERI. This variation results in imbalance of load-distribution across PEs, which can degrade overall performance [14]. By converting TLP to DLP we effectively eliminate fine-grained imbalances which could have a highly beneficial impact on overall parallel system performance, though load-balancing implications are outside the scope of this paper.

As we will show, exploiting some of the traits of the application, it is possible to design an architecture capable of elegantly exposing greater DLP in the application, and thus allowing a *single instruction multiple data* (SIMD) architecture. Vector processing, one form of SIMD architecture, is a highly efficient form of parallel computation, so much so that vector cores are an important part of the burgeoning *Cell Broadband Engine* platform, which is a CMP architecture [12].

The lynchpin of our system is *thread matching and scheduling*, which is in some respects related to *dynamic thread scheduling* on heterogeneous multiprocessors as described by Becchi and Crowley [6], who envision a scheme whereby threads are dynamically migrated between heterogeneous cores in a CMP system. The migration policy would have an awareness of core capability and availability.

The novelty of our approach lies in our proposal to invoke an awareness of *thread requirements/behaviour* with the aim of *merging several threads* together in a manner that *exposes high DLP*, implicitly constructing uniform sub-streams from the overall non-uniform data stream. A vector of threads with identical control-flow are scheduled for execution in a single vector core. This relies on the target application being throughput-oriented, and comprised of threads with *limited* variation in control-flow.

Our application fits the bill. We propose to re-order/schedule ERI threads into sets of vectors, where all threads in a vector have common control-flow. The thread matching is highly application specific and makes use of specific knowledge of the variables that affect control flow as well as permutational symmetries arising from the mathematical specifics of the algorithm. However, the conceptual ideas discussed could just as easily be applied to other applications.

In this paper, we first briefly describe the salient characteristics of the target application. We then launch into our discussion of the proposed architectural features in section 3, specifically on how we intend to achieve thread scheduling with a content-addressable associative architecture. We consider the impact of exploiting application specific per-

mutational symmetry, as well as the impact of limiting the scheduling operation to a constrained window of threads. In section 5 we assess the practical performance of our proposed system. Finally, we conclude and propose ideas on further work.

2. APPLICATION CHARACTERISTICS

As we previously mentioned, our application suffers $O(N^4)$ scaling, where N is the number of *basis functions* used to describe the molecules. Each basis function approximates the “distribution cloud” for each electron in the molecule. The HF-SCF algorithm is basically as follows:

1. Generate the molecular data (basically 3D coordinates of atomic nuclei), specify choice of basis set database and other specific algorithmic details.
2. Generate initial guess molecular orbitals.
3. **Compute electron repulsion integrals (ERIs) – $O(N^4)$ expense.**
4. Generate G-matrix by reducing integrals with density matrix.
5. Fock matrix formation: one electron integral matrix + G-matrix.
6. Diagonalize Fock matrix to obtain improved density matrix – $O(N^3)$ expense.
7. If density matrix convergence criteria not met, goto 3 (using newly obtained density matrix).
8. Calculate properties based on wavefunction, then END.

Step 3 is our kernel of interest in this paper. We know from analysis, performance profiling as well as literature [10] that this is the “hot-spot” of the computation.

The wavefunction form is represented in terms of molecular orbitals, which are written as linear combinations of basis orbitals that resemble the orbitals of atoms (so called, Atomic Orbitals, or AO’s). The basis functions in this linear combination of atomic orbital-molecular orbital method (LCAO-MO) are atomic orbitals, which in practical calculations mostly are atom-centered functions that resemble orbitals as they can be found for isolated atoms. The radial part of such orbitals is an exponentially decaying function. Basis orbitals of this type are called Slater-type orbitals (STO). For practical calculations these functions are time-consuming, and therefore these orbitals are approximated by a linear combination of Gaussian basis functions (GTOs). Each GTO is a record consisting five real numbers and three short integers. The three short integers describe the *angular momentum* on the xyz axes. The physical significance of the angular momenta is related to the electron configuration over “spdf...” shells. Angular momenta of the four GTOs going into an ERI dictate the control-flow of the ERI².

²Readers familiar with the application will notice that we neglect basis function *contraction* – in fact our ERI algorithm of choice is the Rys quadrature [11][21][5] which delays contraction until the end of the computation. This allows us to tackle contraction by computing each level of contraction in parallel (therefore improving our key metric, vector unit utilization), or by scheduling the contractions sequentially. This issue will be left for further work.

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

Table 1: Basis set size and thread count for various molecules

Table 1 lists some example molecules and the number of basis functions that would typically be used to describe the molecule, as well as a rough estimate of the number of ERI threads available – note that all these molecules are in fact *very small*, and that molecules of interest in drug design such as proteins can consist of as much as 10k basis functions. All threads are concurrent, therefore there is massive TLP for significant workloads (our goal is to eventually target workloads of $N \gg 10^3$).

Each ERI operation requires a 4-tuple of basis functions as input, and returns a scalar value. The control-flow of the algorithm varies with the nature of the input data. Through analysis and profiling of the algorithm, we have identified the specific input variables that affect the control-flow, and we have developed a set of rules to identify ERI threads that have identical control-flow. As one might expect, this application is very floating-point intensive, however the variables that dictate control-flow are short integers.

Let the maximum angular momentum value be 7 (in other words, just 3 bits wide) – this is sufficient for most large-scale HF-SCF jobs of interest. There are 3 angular momentum values (corresponding to the xyz axes) associated with each basis function. Since each ERI takes a 4-tuple of orbitals, there are $12 \times 3b$ values characterizing the control-flow of each thread.

As will be shown later, an important part of our strategy is to exploit the permutational symmetry inherent in the computation of ERIs. Each ERI is characterised by a 4-tuple of input functions, i.e. $\langle i, j | k, l \rangle$. Due to the mathematical foundation underlying the formulation of ERI algorithms, the following set of equalities hold true:

$$\begin{aligned}
 \langle i, j | k, l \rangle &= \langle j, i | k, l \rangle = \\
 &\langle i, j | l, k \rangle = \langle j, i | l, k \rangle = \\
 &\langle k, l | i, j \rangle = \langle l, k | i, j \rangle = \\
 &\langle k, l | j, i \rangle = \langle l, k | j, i \rangle \quad (1)
 \end{aligned}$$

Each of the input functions has an angular momentum term, and the combination of the four angular momentum terms dictates the control flow of the ERI routine. Therefore, we hypothesize that by reordering the input functions for each ERI, subject to the constraints described in equation 1) there could exist a larger number of threads with matching control-flow, thus improving the viability of vector processing. This scenario is illustrated in figure 4.

3. ARCHITECTURE REQUIREMENTS

Our proposed system is depicted in figure 1. Conceptually, there are two main blocks in our proposed design: 1) the *thread match and issue* block, and 2) the *vector processing*

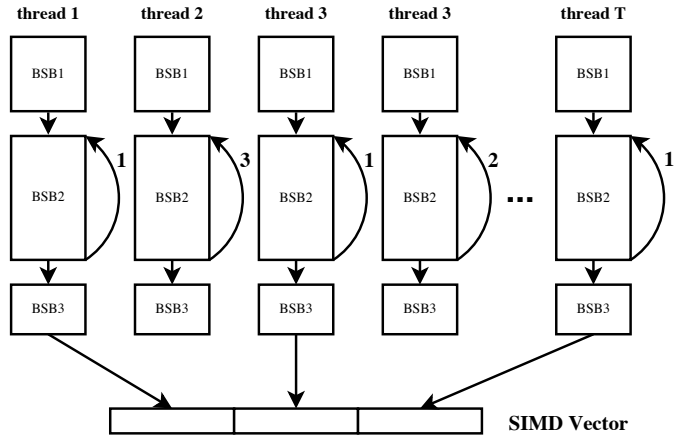


Figure 2: Converting TLP to DLP.

core. In this paper we focus most of our attention on thread matching and issue.

3.1 Thread Matching and Issue

The basic idea behind our proposal is to group together threads with identical control-flow, effectively converting TLP to DLP, and therefore allowing highly efficient and high performance vector computation.

The concept is illustrated with a simple example workload in figure 2. There are several threads ($1 \rightarrow T$) that consist of a common initial stage, an iterative mid-section, and a common final stage. Each thread has a different iteration constant, thus exhibiting poor DLP and frustrating “wholesale” vector processing. However, several threads have an identical iteration constant, therefore selectively constructing vectors of threads with identical iteration constant would, in effect, expose greater DLP and allow efficient vector processing. Clearly this would be most practical in situations where the scope of control variability is limited and known *a priori* (i.e. at the start of thread execution). This is the case for our application.

When considering the viability of this approach for our application, one is effectively asking this question: *can we achieve high vector functional unit utilization?* As one might expect, the answer is *it depends*. First of all, the approach relies on not just large TLP, but *massive* TLP to increase the likelihood that a sufficient number of threads will have matching control-flow. This point is made in figure 3 where we examine the vector utilization³ of two different workloads. Recall the TLP numbers for these two workloads, provided in table 1. As our goal is to target large workloads, massive TLP is a safe bet, however one must keep

³A note on the methodology used to produce our vector utilization plots, i.e plots 3, 5, and 6: we used an existing quantum chemistry prototype program as a testbench framework to generate batches of threads (though in the case of figures 3 and 5 there was only 1 batch). Subsequently, the threads within a batch were matched and a vector-packing operation was simulated; a full vector obtained a utilization score of 1, a half-full vector obtained a utilization score of 0.5, etc. This was repeated for all batches. The utilization scores are then summed up and divided by the total number of vector packing operations performed, to obtain the final utilization metric.

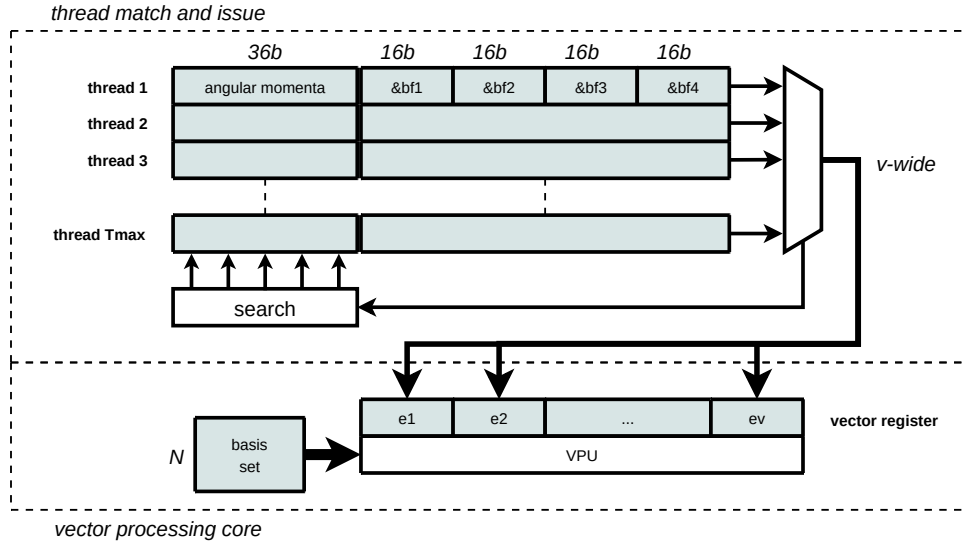


Figure 1: System overview

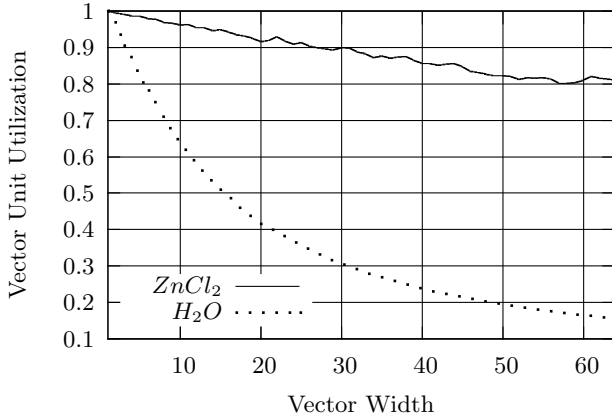


Figure 3: Vector unit utilization: effect of workload.

in mind that it would be impractical for the system to be aware of the *complete* TLP available as this would require performing a search over all threads – surely a daunting requirement, considering the $O(N^4)$ thread-count scaling. It is more likely that any given search would only operate on a limited subset of the total number of threads.

Fortunately, there is a way to significantly enhance the thread-matching hit rate. Handsome gains can be obtained by reordering the four basis functions within the ERI tuple (subject to the permutational symmetries arising from the mathematical foundations of the algorithm), which increase the likelihood of discovering matching threads. This scenario is illustrated in figure 4. The impact of this symmetry exploitation on vector utilization is plotted in figure 5.

The solution we are proposing consists of a SIMD processing back-end and a thread-matching “pre-processing” front-end. The role of the front-end is to group together threads with identical control-flow and issue these threads

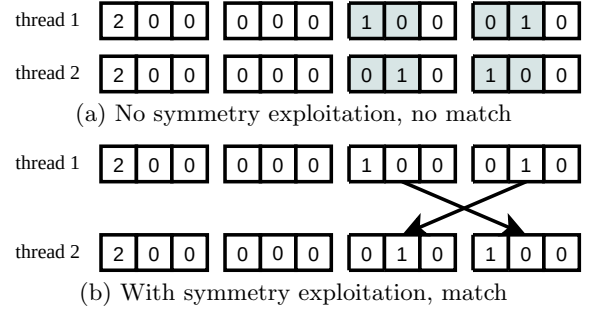


Figure 4: Thread matching with symmetry.

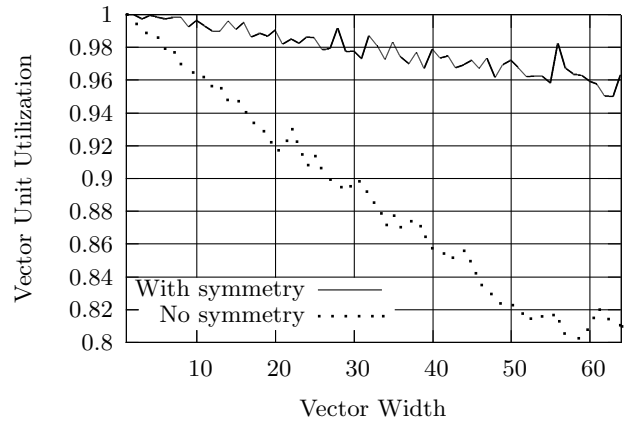


Figure 5: Vector unit utilization: effect of matching with permutational symmetry.

as a vector to the SIMD back-end. The job of the front-end is complicated by the fact that threads are created with essentially random control-flow. In other words, we have to *randomly* pick out threads with particular characteristics from a set of threads. The task of detecting matching threads is, fortunately, reasonably simple: all one has to do is identify threads with a limited number of identical variables. The challenge is carrying out the search with high throughput.

The back-end vector processing promises compelling performance gains, however the thread matching front-end “pre-processing” would come at a cost. To minimise this cost, we propose an *associative processor architecture* to perform thread matching and vector construction operations. Such an architecture would make a good choice in this case because:

- The operands to match are short integers, thus suitable for content-addressing.
- Such a highly parallel architecture would provide high performance.

3.1.1 CAM – Thread Matching

For our application, the task of matching threads with identical control flow is equivalent to the task of matching ERI tuples that consist of identical angular momenta values (subject to permutational symmetry). A working set of threads would be stored in a memory, and searching this memory associatively in parallel would provide the bandwidth required to effectively utilize a backend vector processing solution. Therefore, a *content-addressable memory* (CAM) [20] is an option worth considering.

ERI threads with identical control-flow are ERI threads with the same 36b combined angular momenta word, and only a simple exact comparison operation would be required. This is actually on the low-end of current generation CAM capabilities; for example, Xilinx’s FPGA-based CAM supports a word size as wide as 512b [1], and Pagiamtzis and Sheikholeslami observe that in practice word sizes around 144b are quite common [20].

Thread matching does not require *ternary* logic (i.e. “don’t cares”, or “wildcards”). Therefore, only binary CAM cells are required. Avoiding the need for ternary cells reduces SRAM storage requirements in the CAM [20].

Loading the search-word will be handled by control logic. The operation of this logic is also quite application specific. We know that there are certain angular momenta patterns that are more common than others, therefore a good strategy would be to keep firing these common-case search-words until the hits drop below a certain threshold. Once this threshold has been reached, rather than perform a blind-search, one possibility is to extract a search-word from one of the non-hit threads. As this involves the decoder logic, we will touch on this in the next section.

3.1.2 CAM Decoder Unit – Vector Issue

Once the searches are completed, *matchlines* for each thread in the CAM will indicate to the decoder unit whether or not the corresponding thread matches the requested search-word. At this point, a policy decision has to be made: *in the case that the number of hits exceeds the back-end vector width, which threads will be selected?* As for the case that the number of hits is less than the vector-width, one could

just issue a partial vector – this case should occur rarely, and the slight loss of utilization is tolerable. Alternatively, one might include a scalar core (i.e. let the system be a heterogeneous CMP) to handle these “unique” threads. Such a proposal, as well as specific policy decisions, are outside the scope of this paper, and will be treated as further work.

Another task handled by the decoder unit is to extract the angular momenta data-word of a single non-hit thread; this word will be used to set up a new search data word, once the search control unit exhausts its dictionary of common search terms.

3.1.3 CAM Populating

One function not represented in figure 1 is the task of populating the CAM with threads. The omission was for clarity. This functionality is actually important because:

- The CAM should be saturated with threads at all times to maximise the possibility of matching a sufficient number of threads to achieve good vector unit utilization.
- As previously alluded to, for practical reasons the CAM will only be populated with a subset of threads at any one time. Clearly the number of threads present in the CAM – or the size of the *thread window* – will impact vector utilization.
- The CAM populating (i.e. thread construction) must be done in a way that automatically exposes permutational symmetry, by re-organizing/sorting the individual basis functions within the 4-tuple.

In the context of our proposed system, the CAM needs to be constantly reloaded with new threads, as individual threads are fired into the vector processor and therefore removed from the CAM. CAMs tend to be optimized for reading. The apparent need for high performance CAM *writing* is somewhat unconventional for CAMs⁴. However, repopulating the CAM may *not* require massive parallelism and bandwidth. This is because the CAM is fired only when the vector processor needs to be fed. While the vector processor is busy the CAM is idle, and in that time it may be repopulated sequentially. We will revisit this issue in section 5.

Practical constraints limit the number of threads that can constitute a matchable population. The results in figure 6 are encouraging: a 256k thread window is sufficient to yield vector utilization of almost 90%, and we can expect this to improve with larger workloads.

Rather than place the burden of matching across permutational symmetry upon the CAM matching circuitry (which would significantly increase the complexity of the CAM) we can actually *re-order* the basis functions within the tuples, subject to permutational symmetry constraints, in a manner that would effectively “equalize” all threads. Quite simply, before threads are admitted into the CAM, the basis functions are sorted in such a way that the largest angular momentum terms would be shifted to the left. In this way, the constituent basis functions are re-ordered before they are

⁴Typical CAM applications, such as network routing table lookup, require parallel reading of the same values repeatedly – writing to the CAM is only required very rarely, i.e. when the routing table is updated due to a change in network configuration.

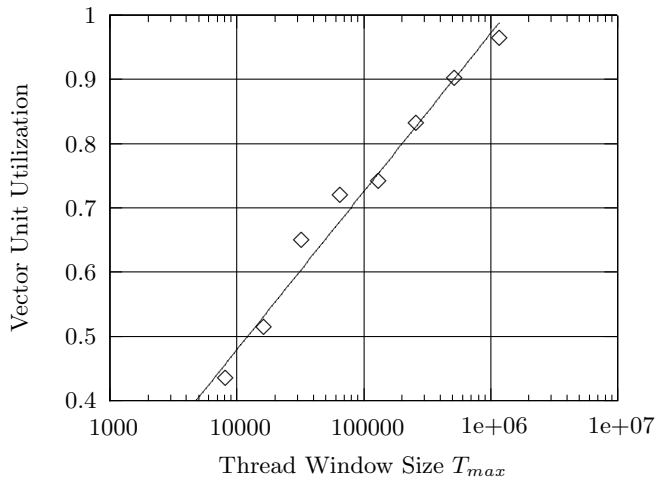


Figure 6: Vector unit utilization: effect of number of threads in CAM, vector width = 50.

admitted in the CAM, and subsequently the CAM simply performs identical word matching, but the final effect is as though the matching does exploit permutational symmetry.

3.2 Vector Processing

With general-purpose systems like the Cell BE, the vector width of SIMD units are kept quite small (128b in the Cell BE) in order to allow good hardware utilization for a very wide variety of workloads[12]. With our particular application, we have the luxury of adopting very wide vectors, which allow a very efficient number crunching, i.e. roughly $O(1)$ control-overhead with $O(v)$ parallelism, where v is the vector width. In this paper we have arbitrarily set the upper-bound on vector width to be $v = 50$, however we may revise this number upwards once we have conducted analysis with larger workloads.

The vector processing architecture design-space is in itself very rich [23]. We shall consider aspects of the design-space, armed with workload characteristics of our application, in a future communication.

We do envision, however, that the vector processor will be attached to the basis set memory through a high-throughput pipelined controller capable of efficiently serving concurrent memory requests to identical memory locations. A similar solution has been applied to network processing [22], for example.

4. COMPARISONS TO RELATED WORK

There are two aspects of our proposal that are comparable to other works. First we discuss our solution from the perspective of multithreading. Then we mention a previous attempt at the development of a special-purpose ERI processor.

4.1 Multithreading

Comparisons may be drawn between our approach and *multithreading* (MT) architectures [24]. In a sense, our architecture may be characterised as *interleaved multithreading* (IMT) since within the vector processor a new thread

commences execution on every cycle. In fact an IMT architecture has been applied to this particular problem, specifically the *Tera MTA* processor [8], with some success. The most obvious difference between our proposal and general IMT is *single instruction stream vs. multiple instruction streams*. In fact, according to Ungerer et al. [24] pipeline support for multiple instruction streams (e.g. multiple program counters) are a defining characteristic of MT architectures, which disqualifies our proposal, though there is a clear similarity in that our approach, like MT, exploits coarse-grained thread-level parallelism.

Beyond definitions, there are more tangible differences between our approach and conventional MT. With the proposed approach to thread-awareness, threads as distinct control units cease to exist when they leave the CAM; as such, any multithreaded-related control overheads are eliminated in the vector processor. This means the nature of the computation within the vector processor is very much dominated by number crunching, with very little control-flow, which effectively eases the development process of a high performance specialised ERI pipeline. This approach also raises the possibility of interesting implementation alternatives, such as the option of implementing the CAM with an FPGA variant that is rich in reconfigurable “gates” and the vector processor with an FPGA variant that more DSP oriented and therefore rich in arithmetic units – correspondingly the LX and SX range of Xilinx Virtex4 FPGAs [2]. This is possible because thread-awareness has been “lumped” at the front-end of the vector processor.

In addition, it has been quantitatively argued recently that vector processing can provide very good energy efficiency [16], which is an extremely desirable trait for massively parallel systems. The CAM would certainly increase the level of power consumption of the overall system, however this is a balancing act and would be the subject of further work.

4.2 ERIC special-purpose processor

As far as we are aware, there has only been one prior attempt at a special-purpose processor specifically targeting ERIs and the HF-SCF algorithm. The strategy underlying the ERI Computer (ERIC) [15][17][18] was to exploit fine-grained instruction-level parallelism (ILP) inherent in the computations. To achieve this ends, a VLIW core was proposed, with some novel components such as specialised arithmetic units. The system was implemented using VLSI technology, and the performance of the completed system was found to be disappointing, exhibiting a very low IPC score, with the primary culprit being a bandwidth bottleneck [18].

5. PERFORMANCE/PRACTICALITIES

There are two perspectives with which we may consider the performance of the CAM in our system:

1. The interaction of the CAM with other subsystems.
2. The internal design of the CAM itself.

We will consider both in this section. However, the material presented within this section (especially in subsection 5.1) is highly application specific. Readers seeking only a general overview of our approach may skip this section. On the other hand, this section may also be treated as a commentary on



Figure 7: System-level block diagram.

the way we tackled the specifics of our target application, and certainly the thought process presented may prove applicable to other computations.

5.1 System-level Bandwidth

Figure 7 depicts the block diagram of our system. Two details of interest are the bandwidth between the thread generator and the CAM, and between the CAM and the vector processor. The CAM output bandwidth to the vector processor is necessarily large; such is the principle requirement of vector processing. However, the bandwidth at the input of the CAM may not be necessarily large.

The CAM input bandwidth must only be sufficiently large such that the CAM is full whenever a vector issue is performed. Once a vector issue has been performed, the vector processor commences its crunching and cannot receive new ERI input vectors. During this time the CAM may be repopulated. It is therefore desirable to balance the re-population time with the vector number crunching time, in order to ensure CAM saturation before the vector processor becomes idle, while at the same time not over-provisioning expensive bandwidth to the CAM input.

The longer the vector processor is busy with a batch of threads, the more time is available for CAM re-population. Therefore, in order to ascertain worst-case bounds, let us assume the minimum vector processing time. We estimate that the vector processor can complete crunching a set of threads in as little as 200 clock cycles, which implies that in the worst-case the CAM must be re-populated with v threads (we have thus far considered v up to 50) within 200 clock cycles. If we let $v = 50$, then we know that in order to maintain CAM saturation, the thread generator must be able to re-populate the CAM at a rate of 1 thread every 4 clock cycles. This is good news because it allows us to provision as little as $100b/4=25b$ TG/CAM bandwidth (BW1 in figure 7). In the possible case that we adopt a wider vector width, we can simply increase the TG/CAM transfer bandwidth by a factor of up to 4 (i.e. vectorize 200 threads) without the need for parallel thread generation units.

Within 5 clock cycles, the thread generator must perform the following tasks:

1. Generate a 4-tuple address.
2. Retrieve the angular momenta of the 4 basis functions.
3. Perform tuple sorting/re-ordering to expose permutational symmetry.
4. Load the 4-tuple addresses and angular momenta into the CAM.

In order to achieve high-performance thread generation, one could exploit application specific insight to provision a specialized memory infrastructure. A memory access sub-system could be optimized for a particular application by employing:

- Specialized address generation.

Vector width, v	50	200
TG/CAM cycles per thread	5	1
TG/CAM bandwidth, $BW1$	25b	100b

Table 2: Required system-level bandwidth.

- Specialized data packing and unpacking.
- Static memory access scheduling.

Generating the 4-tuple addresses is quite simple, requiring increment operations and occasional masking to eliminate redundancies arising from permutational symmetry (i.e. we eliminate 7/8 of the possible N^4 ERI threads, which are redundant).

However, the 4-tuple must subsequently be re-ordered according to the angular momenta of the constituent basis functions to increase the thread-matching hit-rate exploiting the permutational symmetry as previously discussed. Performing this task requires access to the angular momenta of the constituent basis functions. Fortunately, most of the time only one index is incremented each time a new tuple is constructed, therefore only one basis function must be read. Furthermore, the access patterns are predictable and exhibit high spatial locality. Therefore, the required data may be pre-loaded. This does necessitate a cache attached to the thread generator, however the capacity would only be $O(v)$, since most of the time each of the four basis functions constituting a thread would only differ by one basis function. Furthermore, assuming a fully associative cache, each cacheline only needs to store a 16b address word identifying the basis function and a 36b data word to store the angular momenta

The tuple re-ordering stage is actually a reasonably simple data comparison and packing operation which can take as little as two cycles to complete for each thread.

Finally, once the thread is generated, it is loaded into the CAM. The only data transferred from the thread generation unit to the CAM is the 100b thread characterization word (i.e. angular momenta and basis function identification). The address of the CAM-line will be generated by control-logic close to the CAM that can detect empty lines.

All four steps in thread generation may be aggressively pipelined – even the tuple re-ordering step, which requires at least two cycles due to dependencies, may operate upon two threads concurrently. Therefore a “cycles per thread” rate of 1 is achievable. Our projections are summarised in table 2.

Should the required cycles per thread drop to below 1 for some reason (for instance if we increase v to > 200) then the solution would be to implement multiple thread generators, and multiple CAM *write-lines* (one set for each thread generator), with *thread loading logic* managing the parallel writes. These elements are elaborated on in section 5.2.

5.2 CAM Internal Design

Turning our attention to the internals of the CAM, recall that each thread is represented in the CAM by the angular momenta and addresses of four basis functions – 100b total storage, of which 36b (the angular momenta) are content-addressable. A thread window of 256k threads would there-

fore require 25Mb storage, of which 9Mb would be content addressable, and this would result in good vector utilization – almost 90% with a relatively small workload, and we would expect utilization to increase with larger workloads. A 9Mb CAM is certainly practical; 18Mb CAMs have been commercially available for quite some time [20].

Besides storing threads and flagging matchlines upon search hits, the CAM system has several other responsibilities. We discuss these functions and some possible design scenarios.

The decoder unit is to fire a subset of matches into the vector unit - deciding which threads to launch could employ a policy as simple as “top matches first” or as complicated as a policy that attempts to group threads which have the maximum number of identical basis functions, so that pressure upon the vector processor’s memory interface may be reduced through common loads.

Loading the angular momenta search-word requires a small database of common search words as well as a mechanism to generate search words on the fly (since storing all possible search-words would be infeasible – doing so would require $O(N^4)$ storage!). This mechanism could be as simple (and slow) as a random search. A more interesting approach may be to extract a *miss* from the decoder unit and load the angular momenta of that thread into the search-word – this would greatly increase the likelihood that for all CAM searches there would be at least one hit.

Re-populating the CAM requires a knowledge of which lines are currently empty. An availability bit will indicate which lines are available for re-population. This bit will be set by the decoder logic. Additional thread loading logic is required to switch threads incoming from the Thread Generation unit to available lines. The loading of each line would be line serial but bit parallel – bit shifting is not a possible option given the stringent time requirements.

Thus, there are a four data lines going through the CAM:

- Searchline – the line that carries the search word across all the lines for matching. There is one wire for each bit in the search-word, therefore in our context there are 36 common searchlines going to all threads. This line is common to all CAMs.
- Matchline – the line that indicates when a dataword matches the searchword. There is one matchline for each entry. This line is common to all CAMs.
- Write-data-line – this line writes a fresh thread into the CAM. As we require bit-parallel writing into the CAM, we require the full thread bitwidth – i.e. 100 bits. Fortunately, we may perform line-serial writing, therefore similar to the searchline, we only require one set of 100 write-data-lines for the all datawords.
- Write-enable-line – as it’s namesake implies, this line indicates which dataword is to be written with the data on the write-data-line. One line is required for each dataword, but only one line may be active in any given cycle.

The physical performance of the solution (e.g. in terms of clock speed and logic area) would be highly sensitive to the choice of underlying technology (on one extreme full-custom ASIC, and on the other commercial FPGA). However, although in some ways our CAM does have requirements that exceed those of mainstream CAMs, on the other hand the

fact that we do not require ternary logic does ease a lot of the burden. Therefore, we believe the solution is viable, and certainly worthy of further investigation.

6. CONCLUSION

We have presented an approach to convert TLP to DLP in order to allow SIMD vector processing. Our work has specifically targeted the computation of electron repulsion integrals (ERIs) in the Hartree-Fock Self-Consistent-Field (HF-SCF) algorithm, however the general concept of identifying the variables that impact control-flow and consequently matching threads with identical control-flow is something that may be adapted to other applications. It is apparent that searching variables that dominate control-flow can expose DLP to an extent whereby vector processing becomes viable for applications traditionally considered to be poor candidates for vector processing, and at the same time the time-cost of searching and matching threads is minimized through the use of associative processing, specifically utilizing content addressable memories.

The approach works especially well for large workloads, which are in urgent need of higher performance. Furthermore, the exploitation of permutational symmetry (which is a feature common to many classes of applications, such a N-body computations) can further increase the match hit-rate between threads, thus increasing the utilization of SIMD vector functional units.

Dwindling gains in clock-speed necessitate an architectural approach to pursue greater gains in computing performance. While clock speed gains are diminishing (constrained by the laws of physics) logic densities still exhibit quite a bit of growth. Growing logic density makes possible the use of large content-addressable memories, especially in the context of heterogeneous system-on-a-chip (SoC) based solutions. In our proposal, a CAM is one sub-system in the SoC, the other major component being a vector processor.

Once the limits of synchronous circuitry constrain vector width (or CAM capacity), the ideas presented in this paper can be merged with other highly attractive architectural ideas, in particular Globally Asynchronous Locally Synchronous (GALS) architectures such as network-on-chip (NoC) [7]. Whatever the case may be, the design of optimal cores will continue to have a profound impact on the performance of applications. SIMD architectures provide a highly efficient means for providing high performance, provided the functional units’ wide vectors can be highly utilized. We have presented an idea for achieving high vector processing utilization, in particular for the computation of ERIs in computational quantum chemistry, which will enable breakthroughs in computing capability and capacity in fields such as drug discovery and nanotechnology.

7. REFERENCES

- [1] Xilinx Datasheet DS253: Content Addressable Memory v5.1, June 2004.
- [2] Virtex-4 family overview, June 2005.
- [3] *CF ’06: Proceedings of the 3rd conference on Computing Frontiers*. ACM Press, New York, NY, USA, 2006.
- [4] P. A. Agarwal, R. A. Alexander, E. Apra, S. Balay, A. S. Bland, J. Colgan, E. F. D’Azevedo, J. J. Dongarra, T. H. D. Jr., M. R. Fahey, R. A. Fahey,

- A. Geist, M. Gordon, R. J. Harrison, D. Kaushik, M. Krishnakumar, P. Luszczek, A. Mezzacappa, J. A. Nichols, J. Nieplocha, L. Oliker, T. Packwood, M. S. Pindzola, T. C. Schulthess, J. S. Vetter, J. B. W. III, T. L. Windus, P. H. Worley, and T. Zacharia. ORNL/TM-2004/13: Cray X1 Evaluation Status Report. Technical report, Oak Ridge National Laboratory, 2004.
- [5] J. D. Auspurger, D. E. Bernholdt, and C. E. Dykstra. Concise, Open-Ended Implementation of Rys Polynomial Evaluation of Two-Electron Integrals. *The Journal of Computational Chemistry*, 11(8):972–977, 1990.
- [6] M. Becchi and P. Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF'06: Proceedings of ACM Computing Frontiers 2006*, pages 29–39, 2006.
- [7] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1):1–51, 2006.
- [8] B. Bolding and K. Baldridge. Multithreaded shared memory parallel implementation of the electronic structure code GAMESS. *Computer Physics Communications*, 128:55–66, 2000.
- [9] P. Boyle, D. Chen, N. Christ, M. C. anbd S.D. Cohen, C. Cristian, Z. Dong, A. Gara, B. Joo, C. Jung, C. Kim, L. Levkova, X. Liao, G. Liu, R. Mawhinney, S. Ohta, K. Petrov, T. Wettig, and A. Yamaguchi. Overview of the QCDSP and QCDOC computers. *IBM J. Res. and Dev.*, 49(2):351–365, 2005.
- [10] J. Cioslowski. *Ab Initio Calculations on Large Molecules: Methodology and Applications*, volume 4 of *Reviews in Computational Chemistry*, pages 1–33. VCH Publishers, Inc., 1993.
- [11] M. Dupuis, J. Rys, and H. F. King. Evaluation of molecular integrals over gaussian basis functions. *The Journal of Chemical Physics*, 65(1):111–116, 1976.
- [12] M. Gschwind. Chip multiprocessing and the cell broadband engine. In *CF'06: Proceedings of ACM Computing Frontiers 2006*, pages 1–7, 2006.
- [13] M. F. Guest, P. Sherwood, and J. A. Nichols. Massive parallelism: The hardware for computational chemistry. <http://www.cfs.dl.ac.uk/parallel/MPP/mpp.shtml>, 1998.
- [14] R. J. Harrison and R. Shepard. Ab Initio Molecular Electronic Structure on Parallel Computers. *Annu. Rev. Phys. Chem.*, 45:623–658, 1994.
- [15] K. Hashimoto, H. Tomita, K. Inoue, K. Metsugi, K. Murakami, S. Inabata, S. Yamada, N. Miyakawa, H. Takashima, K. Kitamura, S. Obara, T. Amisaki, K. Tanabe, and U. Nagashima. MOE: a special-purpose parallel computer for high-speed, large-scale molecular orbital calculation. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 58, New York, NY, USA, 1999. ACM Press.
- [16] C. Lemuett, J. Sampson, J.-F. Collard, and N. Jouppi. The potential energy efficiency of vector acceleration. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 77, New York, NY, USA, 2006. ACM Press.
- [17] K. Nakamura, H. Hatae, M. Harada, Y. Kuwayama, M. Uehara, H. Sato, S. Obara, H. Honda, U. Nagashima, Y. Inadomi, and K. Murakami. Eric: A Special-Purpose Processor for ERI Calculations in Quantum Chemistry Applications. In *HPC-Asia*, 2002.
- [18] K. Nakamura, H. Honda, K. Inoue, H. Sato, M. Uehara, H. Komatsu, H. Umeda, Y. Inadomi, K. Araki, T. Sasaki, S. Obara, U. Nagashima, and K. Murakami. A High-Performance, Low-Power Chip Multiprocessor for Large Scale Molecular Orbital Calculation. In *Proceedings of the Workshop on Unique Chips and Systems (UCAS)*, pages 87–94, 2005.
- [19] S. Obara and A. Saika. Efficient recursive computation of molecular integrals over Cartesian Gaussian functions. *The Journal of Chemical Physics*, 84(7):3963–3974, 1986.
- [20] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3):712–727, March 2006.
- [21] J. Rys, M. Dupuis, and H. F. King. Computation of Electron Repulsion Integrals using the Rys Quadrature Method. *Journal of Computational Chemistry*, 4(2):154–157, 1983.
- [22] T. Sherwood, G. Varghese, and B. Calder. A pipelined memory architecture for high throughput network processors. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 288–299, 2003.
- [23] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures: A Design Space Approach*. Addison-Wesley, 1997.
- [24] T. Ungerer, B. Robic, and J. Silc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.
- [25] S. Yahiro and Y. Gondo. Vector processing algorithm for electron repulsion integrals in ab initio HF calculation based upon the PK supermatrix. *J. Comp. Chem.*, 13(10):1246–1254, 1992.