Final Report: The Super Instruction Architecture

Victor Lotrich Nakul Jindal Erik Deumens Rodney Bartlett
Beverly A Sanders

1 Abstract

Important classes of problems in computational chemistry, notably coupled cluster methods, consist of solutions to complicated expressions defined in terms of tensors. Tensors are represented by multidimensional arrays that are typically extremely large, thus requiring distribution or backing on disk. A parallel programming environment, the Super Instruction Architecture (SIA) comprising a domain specific programming language Super Instruction Assembly Language (SIAL) and its runtime system Super Instruction Processor (SIP) has been developed which is specialized for this class of problems. An important feature of SIAL is that algorithms are expressed in terms of blocks (or tiles) of multidimensional arrays rather than individual floating point numbers. The computational chemistry package ACES III, has been developed using the SIA platform, and has successfully ported to Blue Waters and used to study processes involving biological enzymes and organic explosives. ACES III has also been extended to allow it to effectively utilize GPUs. Speedups on Blue Waters from utilizing GPUs relative to CPUs in the range of 2.0-2.2 for a CCSD calculation and from 3.4-3.7 for CCSD(T) have been obtained.

2 Introduction

Domain experts developing new computational chemistry methods in the context of the SIA [SBD+10] and ACES III[LFP+08, ACE] express their algorithms in SIAL, a simple parallel DSL providing a parallel loop construct and intrinsic support for distributed and disk backed arrays. Memory management, communication, and I/O are provided by the runtime system.

An important feature of SIAL is that algorithms are expressed in terms of blocks (or tiles) of multidimensional arrays rather than individual floating point numbers. Programming with blocks enhances programmer productivity by eliminating the need for tedious and error prone index arithmetic. Frequently used super instructions such as tensor contractions are intrinsic and supported by the language syntax; additional super instructions can be implemented by domain programmers.

Although blocking arrays is a well-known technique in parallel programming, it is rarely supported at the programming language level. Expressing algorithms in terms of blocks is very natural in the domain and has several significant consequences:

- Data is handled at a granularity that can be efficiently moved between nodes.
- Computation steps will be time consuming enough for the runtime system to be able to effectively and automatically overlap communication and computation.

For the purposes of conveniently exploiting GPUs, programming with blocks provides the following additional benefits:

- The computation is already partitioned into tasks that map conveniently onto CUDA kernels
- Most super instructions lend themselves to straightforward data parallel implementations.

3 Overview

In this section, we give an brief overview of the SIA, first describing the language SIAL, then the runtime system, SIP. A more complete description can be found in [SBD+10].

3.1 SIAL

The most important features of SIAL are intrinsic distributed and disk backed arrays, explicit parallelism with a **pardo** statement, and support for expressing algorithms in terms of blocks of multi-dimensional arrays.

3.1.1 Arrays and indices

SIAL exposes the following qualitative differences in the size of arrays: small enough to fit in the memory of a single process, distributed, and disk-backed. This is done by offering several array types: **static**, **local**, **temp**, **distributed**, and **served**. Static arrays are small and replicated in all processes. Distributed arrays are partitioned into blocks and distributed. Served arrays, also partitioned into blocks, are stored on disk. Local and temp arrays are local to a process and are used for holding intermediate results. In the SIA extension, blocks of local and temp arrays may be allocated in GPU memory.

Arrays are declared with segment indices that refer to segments rather than individual elements. The shape of an array is defined in its declaration by specifying index variables for each dimension. Index variables themselves are declared with a range, which may be defined using either a constant value, or a symbolic constant that is determined during program initialization. As a result, the size of an array and the size of its segments are known and fixed during the program execution, but need not be known when the program is written. There are three types of indices: segment indices[1] count segments and enable programming in blocks; simple indices, most commonly used to count iterations; and subindices. A subindex is related to a specific segment index and allows access to subblocks.

One-sided access to blocks of distributed arrays occurs via **get** and **put** commands. Analogous commands for served arrays are **request** and **prepare**. Both **put** and **prepare** have variations that atomically accumulate results into the block on the home node or IO server.

Local and temp arrays are used within a process to hold intermediate results. Local arrays are explicitly allocated and deallocated and are typically fully formed in at least one dimension. Temp arrays are automatically allocated and deallocated from CPU memory by the runtime system. Blocks of local and temp arrays may be allocated and, if necessary, initialized in GPU memory. This will be explained in more detail in the context of the example in Sect. 4.1.

3.1.2 Expressing Coarse Parallelism

Coarse parallelism, where tasks are mapped onto MPI processes, is explicitly expressed using a **pardo** command that is given a list of index variables and an optional list of **where** clauses, each with a boolean expression. The SIP executes iterations, in parallel in MPI processes, over all possible combinations of the values in the range of the given indices that also satisfy the **where** clauses. The **where** clause is most frequently used to eliminate redundant computation when arrays are symmetric. Scheduling of pardo iterations and mapping onto processors is done by the SIP.

3.1.3 Other control structures

Other control structures include procedure calls[2], **if** and **if-else** commands and a **do** loop. The latter is given a single index variable and conducts a sequential iteration over the range of the index variable. Typically, a computation will require looping over blocks of two or four

[1] There are actually several segment index "types" corresponding to domain specific concepts. For example, aoindex and moindex represent atomic orbital and molecular orbital. This allows the type system to perform useful checks on the consistent use of index variables.

[2] Procedures in SIAL are somewhat nonstandard and are almost like macros except that the **return** instruction allows an early return.

dimensional arrays. The combination of the pardo loop with the sequential do loop provides a convenient and straightforward way for the SIAL programmer to structure computations.

3.2 SIP

SIAL programs are compiled into SIA bytecode, which is interpreted by the SIP. The SIP is a parallel virtual machine written C, Fortran, and MPI that manages the complexities of dealing with parallel hardware, including communication and I/O.

The SIP is organized as a master, a set of workers, and a set of I/O servers, each implemented (in the current release) using a sequential MPI process. When execution of a SIAL program is initiated, the master performs the management functions required to set up the calculation. The focus of the SIP design effort was to produce a well-engineered system that can efficiently execute SIAL programs and be easily ported to and tuned for different systems. A design principle of the SIP is to maximize asynchrony; all message passing is asynchronous and all barriers are explicit.

The SIAL **get**, **put**, **prepare** and **request** statements may require transferring blocks between nodes, either another worker node for a distributed array or an IO Server for a served array. The SIP first determines whether the indicated block is available at the current node. It may be available because it was assigned to be stored there, or because it is still available in the block cache from a recent use. If not, non-blocking communication is initiated to acquire or send the indicated block using information in the block's data descriptor. As much as possible, instructions are executed asynchronously: Those involving communication are started and then control returns to the SIP task so that more computations or different communications can be performed. When an instruction that needs a block executes, it will transparently wait if the communication to acquire the block is still in progress.

3.2.1 Super Instructions

A SIAL programmer has a rich collection of super instructions at his or her disposal. Super instructions are provided for a variety of operations including I/O and utility functions. Computational super instructions perform computationally intensive operations on blocks; they simply take blocks as input and generate new blocks as output and do not involve communication. Those that will be executed on a CPU are implemented in Fortran or another general purpose programming language and thus can take advantage of high quality optimizing compilers. CUDA implementations have been provided for the most important super instructions, enabling them to be executed on the GPU.

4 Extensions for GPU Utilization

The Super Instruction Architecture provides a straightforward approach to enabling applications to utilize GPUs by mapping super instructions to kernels. The super instructions, in most cases, lend themselves to straightforward data parallel implementations. Two approaches were tried to manage the necessary data transfer.

The first attempt required no change to SIAL programs. The GPU-enabled super instructions were self-contained computational units that would be executed on a GPU whenever one was available. Each GPU-enabled super instruction would test for the presence of a GPU and if available would transfer required data blocks to the GPU, perform the operation, and transfer the results back to the host. If a GPU was not available, the work would be done on the CPU. The advantages of this approach are that it requires no changes to SIAL programs and enabled an incremental approach to providing the necessary CUDA implementations of super instructions. However, although the super instruction implementations in isolation exhibited significant speedups on the GPU, the performance improvement of the overall computation was much less impressive due to the inherent synchrony and frequent unnecessary data transfer. Also, whether it is worthwhile to perform an operation on the GPU depends on the amount of data used in the operation. For example, the time to perform a contraction operation involving only two dimensional arrays (where for a typical (but problem and system dependent) segment size of around 35 element, the size of the block would be 35^2), including data transfer time

Directive	Description
gpu_begin	Start of a region SIAL code whose super instructions will be implemented on the GPU if one is available.
gpu_end	End of the region of SIAL code.
gpu_allocate ⟨block⟩	Allocate memory to hold local or temp block ⟨block⟩ on GPU (initialize to 0)
gpu_free ⟨block⟩	Free memory associated with ⟨block⟩ on GPU
gpu_put ⟨block⟩	Copy data from local or temp block ⟨block⟩ from the CPU to the GPU. If necessary, allocate memory on the GPU.
gpu_get ⟨block⟩	Copy contents of ⟨block⟩ from GPU to CPU

Figure 1: Directives for GPU use

might be longer on the GPU than on the CPU. A contraction involving four dimensional arrays, would have blocks big enough to make execution on the GPU worthwhile. The bottom line is that getting the best results from this approach would require building logic into every super instruction significantly more complex than checking a flag to see whether a GPU exists.

The current version provides directives with which to annotate SIAL programs which

- Indicate which parts of a SIAL program should be executed on a GPU (if available)
- Explicitly manage memory allocation on the GPU and data transfer between the host and device.

This requires some work from the programmer, but has yielded much better performance. Directives are provided to indicate regions of SIAL programs that should be executed on a GPU if one is available. Directives are also provided to allocate and free blocks in device memory, and transfer data between the host and device memories. As a result, a sequence of super instructions can reuse data on the GPU. These instructions are shown in Fig. 1.

Future work will explore compiler analysis to reduce the annotation burden on the programmer.

Since a hardware platform may have fewer GPUs than compute cores, SIAL programs with GPU directives should remain correct when no GPU is available. This must be supported by the SIP since the same compiled SIAL program must work in either case.

4.1 Example

In this section, we show a fragment of a CCSD calculation that has been annotated for GPU execution.[3] This will serve to illustrate both the SIAL language and directives. Declarations of index variables and arrays are not shown; TA0_ab and T2A0_ab are 4 dimensional served (disk-backed arrays). LTAO_ab, LT2AO_ab1, LT2AO_ab2, are local arrays, and Yab and Y1ab are temp arrays.

The **PARDO** lambda, sigma statement in line 1 sets up the parallel computation. The index space, formed by the ranges of segment indices lambda and sigma, is partitioned among the worker processes and the instances of the body are performed in parallel. Exactly how this is done is determined by the chosen load balancing mechanism. The next few statements allocate blocks of local array LTAO_ab and fill them with data obtained from served array TAO_ab. The **DO** command, first seen in line 10, indicates a serial loop over the range of the given index variable. The first GPU directive, **gpu_begin** appears in line 17. If the node has a GPU, it will be used in the subsequent super instructions. If not, the GPU related directives will have no result, and the entire computation will be performed on the CPU. The command **gpu_put** allocates memory on the GPU and initializes it with data copied from the indicated block on the host. **gpu_allocate** allocates a temp block on the GPU. Lines 35-38 perform calculations on the

[3] The syntax has been slightly simplified.

GPU. These intrinsic super instructions are implemented as CUDA kernels. The contraction[4] in line 35 is one of the more computationally intensive super instructions. Note that it is not necessarily the case that each block of an array is the same size, so the temp blocks allocated on the GPU need to be freed and reallocated rather than being reused in the next iteration.

Lines 44-54 copy results stored in blocks of the arrays LT2AO_ab1 and LT2AO_ab2 back to the GPU and free GPU memory. The remainder of the code accumulates the computed results into blocks of the served array T2AO_ab, lines 59- 60 and frees CPU memory.

5 Results

In this section, we provide results from experiments on Blue Waters. To eliminate, as much as possible, extraneous effects that might affect the timings, the GPU and CPU computations presented were run immediately after one another on exactly the same set of machines.

Figure 3 shows the total time and idle time (when nodes are waiting for data to arrive from another node) of a relatively small calculation ranging from 4 to 32 processors, each with a GPU attached. This was a CCSD calculation for Ar_4 in cc-pvQZ basis with 236 basis functions and 36 correlated electronic orbitals. The segment sizes were 42 for arrays representing atomic and virtual orbitals and 36 for occupied orbitals, thus a block of an N-dimensional array would contain between 36^N and 42^N elements. For each processor count, the first bar is the total time using GPUs, the second bar is the total time without GPUs. Speedups ranged from 2.0-2.2. The third and fourth bars show the total idle time (i.e. the time spent waiting for data to arrive from a different node) in the computation. As would be expected from the structure of the computation[5], these values are nearly the same with and without GPUs.

Figure 4 shows the performance of the time required for three of the most time consuming procedures in the CCSD calculation. For each processor count, the first two bars show the time required with and without GPUs respectively for the LADDER procedure which scales as N^4O^2. The third and fourth bars show the time for the WAEBF procedure, which scales as V^2O^4. The fifth and sixth bars show the time for the WMEBJ procedure which scales as V^3O^3 with a relatively large prefactor.

CCSD(T) calculations are more accurate than CCSD, but the accuracy comes at a significant computational price. Figure 5 shows timings results from the (T) contribution for the same molecule, Ar_4, and same basis set as in Figures 3 and 4. However, for this calculation, the segment sizes were reduced in order for the calculation to fit in the available CPU memory.

The (T) contribution is comprised of two parts, aaa, and aab consisting of 9 and 8 permutation steps, respectively. Each permutation step involves and initial permutation of the matrices followed by a contraction, followed by a permutation; the different steps perform different permutations, but are other wise the same. Figure 6 shows the CPU time per permutation. The varying results for the CPU only computations reflect the different memory access patterns and interaction with caches on the CPU. When the permutations and contractions are performed on the GPUs, the timing results are much more uniform.

The Ar_4 molecule used in the previous results, being relatively small and admitting well-behaved calculation, is useful for studying the performance of GPU-enabled ACESIII. However, it is also desirable to show results for larger scale calculations of genuine scientific interest In Figure 7, we show timing results for RDX, an organic explosive that requires significantly more computational power than Ar_4. These calculations used 534 basis functions (cc-pvTZ basis) with 84 correlated electrons and segments sizes of 21, 34, and 42 for atomic, virtual, and occupied orbitals, respectively. Note that the vertical axis is now hours rather than seconds and the number of processors ranges from 500 to 1000. The speedup achieved by exploiting the GPUs ranges from 3.4 for 1000 processors to 3.7 for 500.

[4]Tensor contraction operations occur frequently in the domain and are defined as follows: Let α, β, γ be mutually disjoint, possibly empty lists of indices of multidimensional arrays representing the tensors. Then the contraction of $A[\alpha, \beta]$ with $B[\beta, \gamma]$ yields $C[\alpha, \gamma] = \sum_{\beta} A[\alpha, \beta] * B[\beta, \gamma]$. Typically, contractions are implemented by (possibly) permuting one of the arrays and then applying a DGEMM.

[5]As can be seen from the SIAL code fragment in Figure 2 , data transfer between nodes do not overlap with GPU instructions.

```
1  PARDO lambda, sigma
2  #allocate and initialize CPU memory, compute integral block on CPU
3     allocate LTAO_ab(lambda,*,sigma,*)
4    DO i
5    DO j
6       request  TAO_ab(lambda,i,sigma,j) #get block from IO server
7       LTAO_ab(lambda,i,sigma,j) = TAO_ab(lambda,i,sigma,j)
8    ENDDO j
9    ENDDO i
10   DO mu
11   DO nu
12      WHERE mu < nu
13      allocate LT2AO_ab1(mu,*,nu,*)
14      allocate LT2AO_ab2(nu,*,mu,*)
15      compute_integrals aoint(lambda,mu,sigma,nu)
16 #start of GPU region
17      gpu_begin
18 #allocate and initialize blocks on GPU
19      gpu_put aoint(lambda,mu,sigma,nu)    #allocate and copy data from CPU
20      DO i1
21      DO j1
22         gpu_put LT2AO_ab1(mu,i1,nu,j1)
23         gpu_put LT2AO_ab2(nu,j1,mu,i1)
24         gpu_put LTAO_ab(lambda,i1,sigma,j1)
25      ENDDO j1
26      ENDDO i1
27      gpu_begin
28      DO i
29      DO j
30 #perform computations on GPU
31         Yab(mu,i,nu,j)  = 0.0
32         Y1ab(nu,j,mu,i) = 0.0
33         gpu_allocate Yab(mu,i,nu,j)       #allocate temp blocks on GPU
34         gpu_allocate Y1ab(nu,j,mu,i)
35         Yab(mu,i,nu,j) = aoint(lambda,mu,sigma,nu)*LTAO_ab(lambda,i,sigma,j) #contraction
36         Y1ab(nu,j,mu,i) = Yab(mu,i,nu,j)    #permutation
37         LT2AO_ab1(mu,i,nu,j) += Yab(mu,i,nu,j)    #element-wise sums
38         LT2AO_ab2(nu,j,mu,i) += Y1ab(nu,j,mu,i)   #element-wise sums
39         gpu_free Yab(mu,i,nu,j)    #free temp blocks on GPU
40         gpu_free Y1ab(nu,j,mu,i)
41      ENDDO j
42      ENDDO i
43 #copy results to CPU , free blocks on GPU
44      DO i1
45      DO j1
46         gpu_get LT2AO_ab1(mu,i1,nu,j1)
47         gpu_get LT2AO_ab2(nu,j1,mu,i1)
48         gpu_free LT2AO_ab1(mu,i1,nu,j1)
49         gpu_free LT2AO_ab2(nu,j1,mu,i1)
50         gpu_free LTAO_ab(lambda,i1,sigma,j1)
51      ENDDO j1
52      ENDDO i1
53      gpu_free aoint(lambda,mu,sigma,nu)
54      gpu_end
55 #end of GPU region
56      DO i
57      DO j
58        #send blocks containing results to IO servers
59         prepare T2AO_ab(mu,i,nu,j) += LT2AO_ab1(mu,i,nu,j)
60         prepare T2AO_ab(nu,j,mu,i) += LT2AO_ab2(nu,j,mu,i)
61      ENDDO j
62      ENDDO i
63         deallocate LT2AO_ab1(mu,*,nu,*)      #free local blocks on CPU
64         deallocate LT2AO_ab2(nu,*,mu,*)
65      ENDDO nu
66      ENDDO mu
67    deallocate LTAO_ab(lambda,*,sigma,*)
68 ENDPARDO lambda, sigma
```

Figure 2: SIAL CCSD fragment

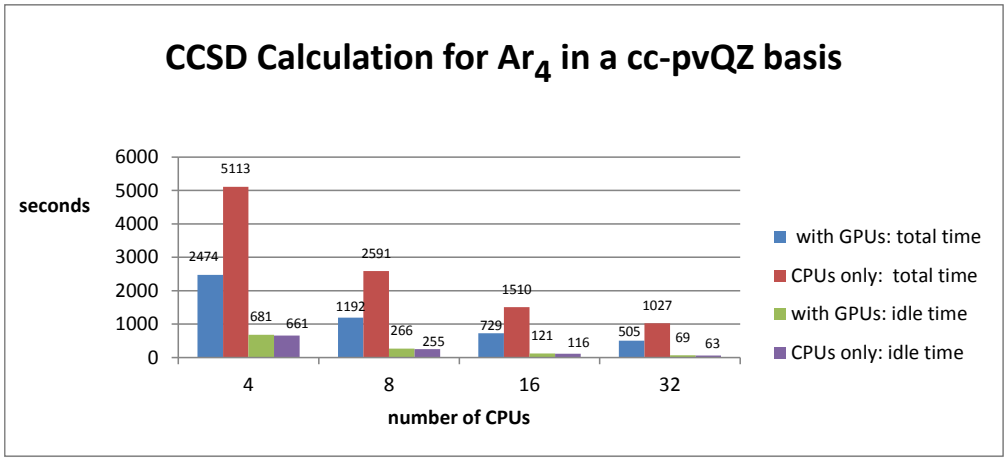

Figure 3: Total and idle time for the Ar_4 CCSD calculation

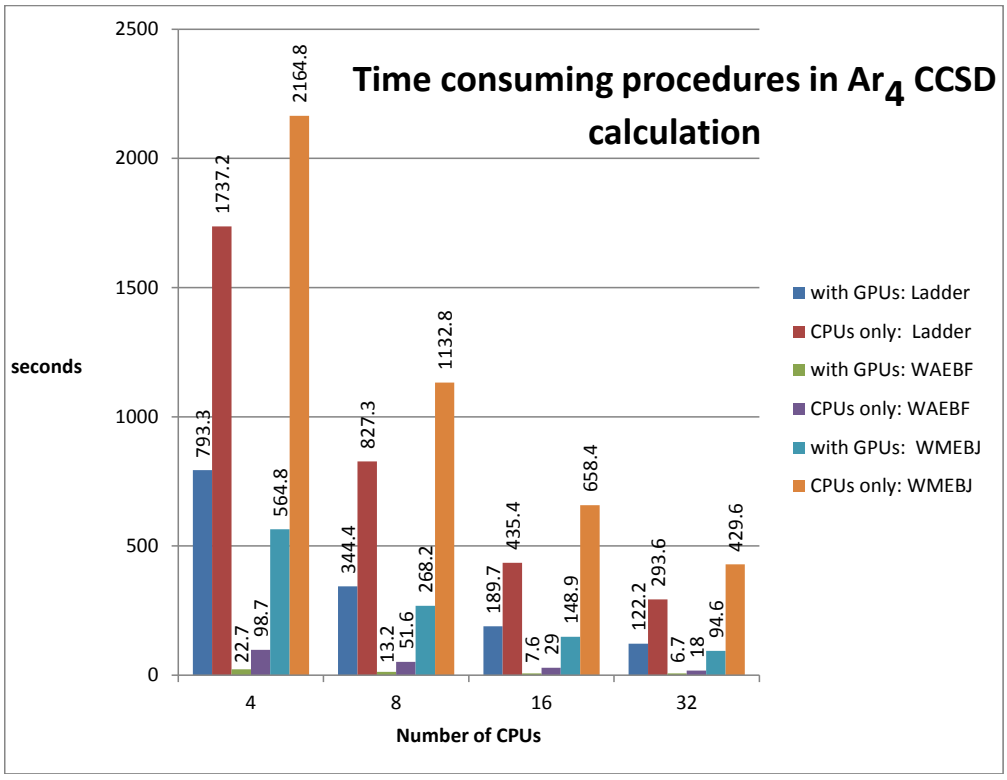

Figure 4: Time Consuming Procedures in the Ar_4 CCSD calculation. Ladder scales as N^4O^2, WAEBF scales as V^2O^4, and WMEBJ scales as V^3O^3 with a relatively large prefactor.

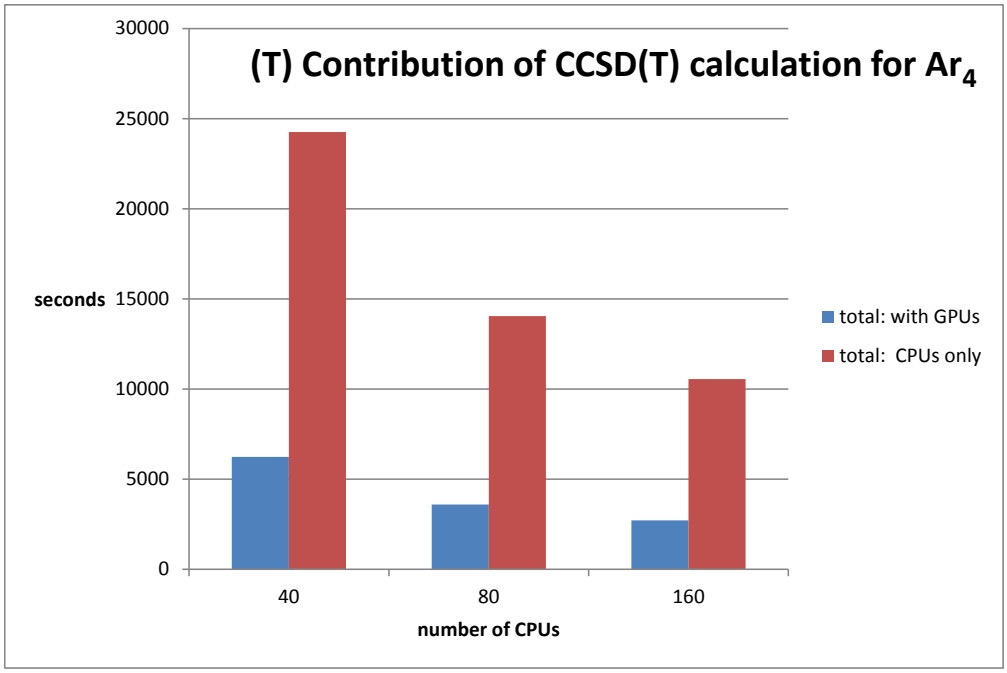

Figure 5: Triples contribution of Ar4 in CCSD(T) calculation

6 Scientific Calculations with ACES III on Blue Waters

ACES III and Blue Waters have already been successfully used for significant scientific calculations in two areas: understanding how biological enzymes operate, and the design of new organic explosives. What unites both of these problems is a need for high-accuracy methods. Our simulations calculate the simultaneous repulsions and attractions of all charged particles in these molecules to unprecedented accuracy. These systems have been the subject of numerous previous studies, but all were inconclusive due to the accuracy challenges of these systems.

One problem is the catalytic cycle of cytochrome P 450. As this enzyme governs oxygen absorption in the body there are few more important processes. The enzyme cytochrome p450 achieves what is commonly called the holy grail of organic chemistry: making unreactive compounds reactive. This enzyme has the ability to take worthless chemicals and make them very valuable, synthetically. What nature does casually, we still lack the ability to do with all modern technology. In simulation of the biological process, we are closer to understanding how this process occurs in cells. Additionally, knowledge of how this process works in cells (including human cells) allows us to improve medical resistance to toxins as well as exploit microbial vulnerability to toxins. All previous theoretical work has been limited to density functional theory (DFT). Yet the critical steps in this cycle depend upon changes in the oxidation state of the iron center and the related multiplicity of the intermediates, and DFT cannot provide spin states, nor any reasonable multiples for tradition metal atoms. To the contrary the acknowledged most accurate, predictive methods available are those from coupled-cluster theory [[CCSD(T)] and it's equation-of-motion extensions (EOM- CC). Previously, it has not been possible to use these tools for a problem of this complexity. With the development of ACESIII and Blue Waters it is now possible to make this newsworthy study. [MLBb]

Existing explosives, including RDX, are known empirically, but the specifics of how they work are not understood. We have succeeded in establishing the chemical play-by-play of the atoms to know how the explosives move from stable organic molecules through detonation. In calculating the various repulsions/attractions, we found the most likely energetic pathway. We now have insights as to how we can adjust the explosive yield as well as the shock-sensitivity to detonation.[MLBa]

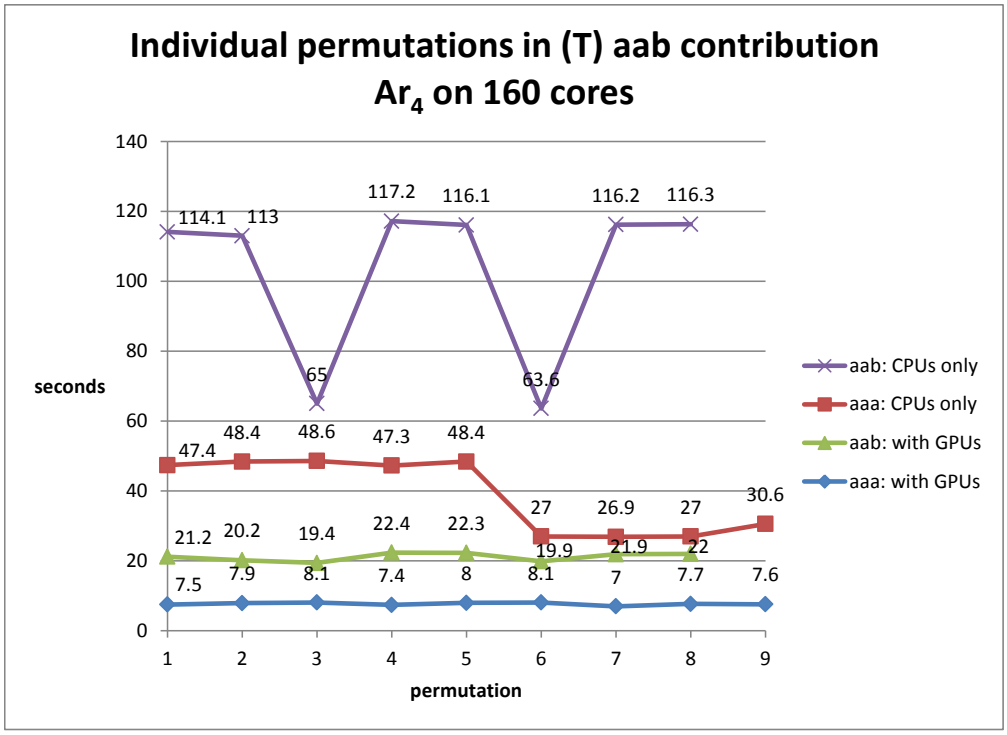

Figure 6: Time per permutation in (T) contribution

7 Conclusion and future work

We have described an enhancement to ACES III to allow GPUs to be exploited. The results provide confidence in the block-oriented approach of the Super Instruction Architecture. The changes to the SIA were implementation of a set of super instruction as CUDA kernels and and fairly minor changes to the runtime system. The organization of the SIA allowed the effort to enable GPUs to proceed incrementally as more CUDA implementations were provided. At this point, we have GPU-enabled implementations for all of the intrinsic super instructions and those required by CCSD and CCSD(T) calculations. Additional implementations will be provided as needed. Most admit straightforward data parallel implementations. Changes to SIAL programs involved identifying the computationally intensive parts of the code and inserting directives indicating which super instructions should be executed on the GPU, and when memory for a block should be allocated on the GPU and when the data belonging to a block should be moved between the GPU and host, expressed in the abstractions supported by SIAL, the SIA's DSL. This is in contrast to other efforts to exploit GPUs in computational chemistry that typically required major one-shot reworking of complete parts of the code. The benefits for the end user of ACES III will be substantial; for example, using GPUs can reduce the time required for CCSD(T) calculations on the RDX molecule on 1000 cores from nearly ten hours to three.

Future work will involve enhancing the SIAL compiler to help automate placement of the directives. The first step will automatically determine appropriate memory allocation and data movement directives given programmer-inserted **gpu_begin** and **gpu_end** statements. Further efforts will explore using performance models, such as SIPMaP [JLDS13] to provide further automation.

Acknowledgments Shawn McDowell provided the CUDA implementation of the contraction operator. In addition to the NEIS-P2 grant that primarily supported the GPU work reported here, development of the SIA and ACES III has been supported by the US Department of Defense's High Performance Computing Modernization Program (HPCMP) under the two pro-

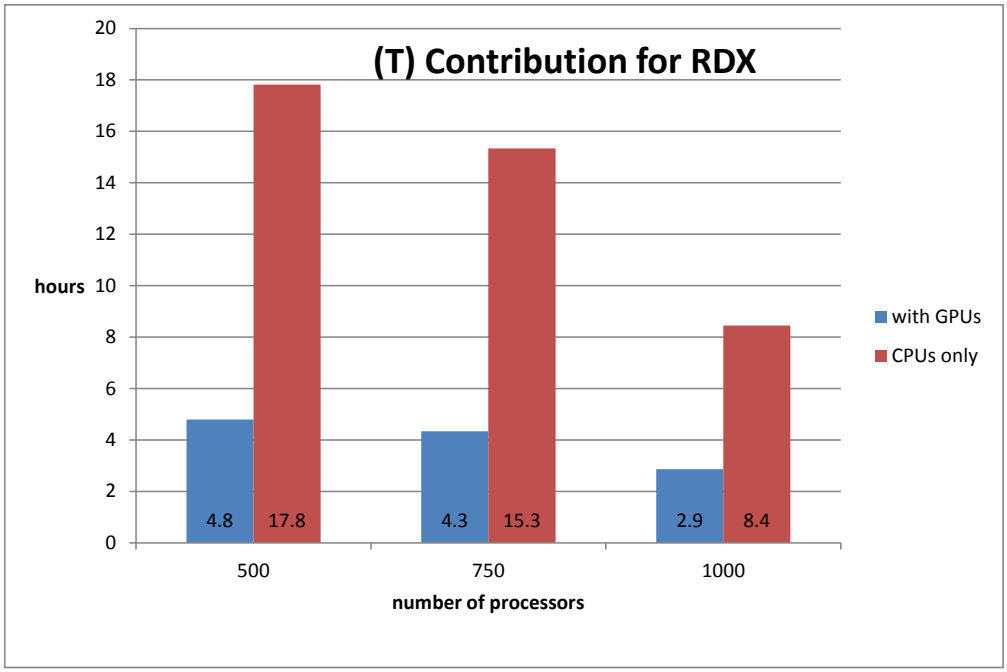

Figure 7: CCSD(T) on RDX

grams, Common High Performance Computing Software Initiative (CHSSI), project CBD-03, and User Productivity Enhancement and Technology Transfer (PET) and the US Department of Energy ASCR SciDAC program. We also thank the University of Florida High Performance Computing Center for use of its facilities.

References

[ACE] Aces III. `http://www.qtp.ufl.edu/ACES/`.

[JLDS13] Nakul Jindal, Victor Lotrich, Erik Deumens, and Beverly A. Sanders. SIPMaP: A tool for modeling irregular parallel computations in the super instruction architecture. In *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2013)*, 2013.

[LFP+08] V. Lotrich, N. Flocke, M. Ponton, A. D. Yau, A. Perera, E. Deumens, and R. J. Bartlett. Parallel implementation of electronic structure energy, gradient and Hessian calculations. *J. Chem. Phys.*, 128:194104 (15 pages), 2008.

[MLBa] Robert W Molt, Victor Lotrich, and Rodney J. Bartlett. The mechanism of rdx decomposition using ccsd(t). in preparation.

[MLBb] Robert W Molt, Victor Lotrich, and Rodney J. Bartlett. Relative energies of cytochrome p450 intermediates using coupled cluster many-body methodologies. in preparation.

[SBD+10] Beverly A. Sanders, Rod Bartlett, Erik Deumens, Victor Lotrich, and Mark Ponton. A block-oriented language and runtime system for tensor algebra with very large arrays. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.