Some Of My Recent Research

Andrey Asadchev

Iowa State University

May 3, 2012

Outline

- About Me
- C++
- Rys Quadrature
- Parallel Hartree-Fock
- Memory Bottleneck In Many-Body Methods
- Scalable MP2 Energy Algorithm
- Scalable CCSD(T) Energy Algorithm
- C++ DFT Implementation
- CI(SDTQ.....) C++ Prototype

About Me

According to Google: Andri Alexandr Asadchev, 18, was booked into the Lowndes County Jail on Nov. 22 on felony charges of manufacturing destructive devices, and misdemeanor charges of reckless conduct and theft by taking.
In free time (free time - get it?) do/tinker with:

- Fishing
- C++
- Algorithms
- Performance
- Python
- Domain Languages
- Arduino
- Music

C++

In my opinion, Fortran(s) is too old. C++ is a much better language for scientific computing, contrary to persistent belief among Fortran programmers that C++ is slow.
Why C++?

- C++ is not slow: consider for example games, libint
- It is a hardware language: AVR micros, GPUs, SSE
- It's actively developed and widely used: lots of libraries, large community, excellent compilers (gcc, Intel), lots of money
- Scales with large projects: frees me to concentrate on algorithms rather than fighting the language

But it is a difficult language, and oftentimes requires to learn tools to use it efficiently. So which is better, invest time up front to learn it and save time using it or go with smaller learning curve but spend more time *coding* in C/Fortran?

Features of C++

- Preprocessor: to really appreciate its code-generating utility, take a look at Boost Preprocessor
- OOP: encapsulation of data and operations
- Operator overload: give special meaning to e.g. `+,-,*,<<,...`
- Templates: compile-time Turing complete language. One of the most powerful features of C++, more on that later.
- Standard Library: containers, algorithms, etc.
- BOOST: all sorts of very useful things, often BOOST components become standardized in new C++ standard: e.g. `std::thread`, `std::array`
- Many math libraries: `boost::ublas`, `Eigen`, interfaces with BLAS, LAPACK

C++ example

Listing 1: C++ example

```
1  #define TYPES (SP)(S)(P)(D)(F)
2  enum Type { SP = -1, S, P, D, F };
3
4  template<Type A, Type B, Type C, Type D>
5  struct Braket {
6      static const int L = (A < 0 ? -A : A) + (B < 0 ? -B : B) + ...;
7      static bool equals(Type a, Type b, Type c, Type d);
8  };
9
10 template<class Braket, class enable = void> // default NULL kernel
11 struct Kernel : Eri {};
12
13 template<Type A, Type B, Type C, Type D> // enabled kernel
14 struct Kernel< Braket<A,B,C,D>,
15                typename enable_if_c<(Braket<A,B,C,D>::L < 10)>::type >
16 : Eri {
17     // implementation
18 };
19
20 Eri* eri(Type A, Type B, Type C, Type D) { // runtime arguments
21 #define ERI(r, types) \
22   if (Braket<BOOST_PP_SEQ_ENUM(types)>.equals(A,B,C,D)) \
23       return new Kernel< Braket<BOOST_PP_SEQ_ENUM(types)> >();
24 BOOST_PP_SEQ_FOR_EACH_PRODUCT(ERI, (TYPES)(TYPES)(TYPES)(TYPES))
25   return NULL; // unhandled arguments
```

Rys Quadrature

Rys quadrature is somewhat simple integral evaluation method with small memory footprint. Works best with higher angular momentum quartets, algorithmically less efficient than rotated shell or MD methods. Important quantity is the number of roots a, $N = L/2 + 1$

- Roots - computed using polynomials or Stieltjes method
- Recurrence - form $Ix(A + B, 0|C + D, 0)$ type integrals.
- Transfer - form $Ix(A, B|C, D)$ type integrals, analogous to HRR.
- $[i, j, k, l] =$
  $\sum_a Ix(a, i_x, j_x|k_x, l_x)Iy(a, i_y, j_y|k_y, l_y)Iz^*(a, i_z, j_z|k_z, l_z)$

Rys Quadrature

To optimize code I did the following:

- Pad all arrays to ensure 16-byte alignment AND tell compiler about it
- Told compiler about aliasing using `restrict` keyword
- Made all the innermost loops have compile time bounds via templates

This takes care of the first three, however to optimize the last step (most time consuming) the $i_x, j_x, \ldots$ index must be known. While possible to do so using just C++ and preprocessor, it is cumbersome and limiting. For that I went with Python cheetah generator.

Rys Quadrature - Cheetah

With Cheetah, Python statements are embedded in the source code, not unlike PHP:

```
        for (int a = 0; a < int(N); ++a) {
%for ii, (fi,fj) in enumerate(fb)
%set i = ii + ifb
%set (ix,iy,iz) = fi[0:3]
%set (jx,jy,jz) = fj[0:3]
            q$(i) += Ix(a,$(ix),$(jx))*Iy(a,$(iy),$(jy))*Iz(a,$(iz),$(jz));
%end for
            /// end for functions in block
        }
```

generates

```
        for (int a = 0; a < int(N); ++a) {
            q0 += Ix(a,2,0)*Iy(a,0,0)*Iz(a,0,0);
            q1 += Ix(a,0,0)*Iy(a,2,0)*Iz(a,0,0);
            q2 += Ix(a,0,0)*Iy(a,0,0)*Iz(a,2,0);
            q3 += Ix(a,1,0)*Iy(a,1,0)*Iz(a,0,0);
            q4 += Ix(a,1,0)*Iy(a,0,0)*Iz(a,1,0);
            q5 += Ix(a,0,0)*Iy(a,1,0)*Iz(a,1,0);
        }
```

Rys Quadrature - SSE Code Generator

```
#define LOAD(m) _mm_load_pd((m))
#define D128 __m128d
...
        for (int a = 0; a < (int(N)/2)*2; a += 2) {
%for (i,j) in xset
            D128 x$(i)$j = LOAD(&Ix(a,$(i),$j));
%end for
%for (i,j) in yset
            D128 y$(i)$j = LOAD(&Iy(a,$(i),$j));
%end for
%for (i,j) in zset
            D128 z$(i)$j = LOAD(&Iz(a,$(i),$j));
%end for
%for ii, (fi,fj) in enumerate(fb)
%set i = ii + ifb
%set (x,y,z) = [''%s%i%i'' % (''xyz''[q], fi[q], fj[q]) for q in %range(3)]
            q$(i) = ADD(q$(i), MUL(MUL($(x), $(y)), $(z)));
%end for
        }
```

generates

```
        for (int a = 0; a < (int(N)/2)*2; a += 2) {
            D128 x20 = LOAD(&Ix(a,2,0));
            D128 x10 = LOAD(&Ix(a,1,0));
            ...
            q0 = ADD(q0, MUL(MUL(x20, y00), z00));
            q1 = ADD(q1, MUL(MUL(x00, y20), z00));
            ...
        }
```

Rys Quadrature - Small L

Small L integrals account for a large fraction of time. It is possible to improve Rys Quadrature performance for those by unrolling the entire expression tree to some number of leafs. To do so:

- Generate the entire expressions in terms of AST leafs.
- Pipe expressions into Mathematica via `Sage` and do CSE
- Parse optimized expressions with `sympy` and store them (Python dict dump)
- Read expressions and generate source code using Cheetah

Listing 2: Small L generated snippet

```
1        update((C[0][0])*W[a]*(Ky*Kz*Px), I+0);
2        update((C[0][0])*W[a]*(Kx*Kz*Py), I+1);
3        update((C[0][0])*W[a]*(Kx*Ky*Pz), I+2);
4        double f0 = (2*B00*Ky + Cy*(pow(Ky,2) + B01));
5        update((C[0][0])*W[a]*(Cz*f0), I+3);
6        update((C[0][0])*W[a]*(Cx*f0), I+4);
7        double f10 = (2*B00*Kx + Cx*(pow(Kx,2) + B01));
8        ...
```

Rys Quadrature on GPU

In a similar spirit, GPU implementation also differentiates between small L ($L = 0, 1, 2$) and general case.
Small L reuses CPU kernel directly, each contraction gets assigned to a thread. Thread block size is 64 due to regster limitation, gets about 0.3 occupancy on Fermi.
However, it may happen often that one quartet per thread block is not enough to saturate threads. So I made further modification where it is quartet per warp.
It should be said that with small L one really needs to go with Pople-Hehre or similar algorithm in my opinion. This implementation is used because it works ok (but not great).

Rys Quadrature on GPU

Things become complicated for general L: a wide range of quartet size and contraction combinations to account for. Moreover, the last step parallelizes easily but the preceding three not so - hence the recurrence and transfer are the major overhead.
The general algorith outline is such:

- Find out the least number of threads needed to perform recurrence and transfer. If the size is below warp size - split the computation such that blocks of power of 2 get assigned per contraction - implicit synchronization.
- Intermediate integrals are stored in shared memory.
- Each of the above blocks holds elements in registers s.t. the number of integrals per thread is $quartet/block$.
- Indexing between integral index and x, y, z indices as well as SP index is stored in global memory.

Fock on GPU

The quadrature kernels have a template parameter Transform, an object which transforms the integrals, e.g. into Fock tiles.
The Fock algorithm looks like this:

- Each warp does one of the six Fock submatrixes.
- The Fock elements are computed one per thread
- Once elements are computed, the corresponding Fock tile in global memory is locked using `atomicCAS`. There is a lock for each Fock tile.
- To minimize lock contention, the integral list is traversed in steps of > 1.

Parallel Fock

The entire thing looks like this:

- The entire matrix is partitioned submatrices of contiguous uniform blocks.
- The matrix is shared
- OpenMP region spawns threads, each thread gets dynamically assigned an entire submatrix.
- Update to matrix done by locking individual submatrices
- Some threads may use GPU primarily - the maximum quartet GPU can handle now is $L < 10$.
- In case GPU cannot handle quartet, it uses CPU code.

So the program can handle any quartet and accommodate varying number of devices/threads.

Performance

Table: C++ CPU performance

Input	GAMESS	C++	Improvement (%)
Cocaine 6-31G	21.3	37.2	-74.6/29.0 %
Cocaine 6-31G(d)	65.0	75.2	-15.7/33.4 %
Cocaine 6-31G++(d,p)	402.7	405.1	-0.60/31.6 %
Cocaine 6-311G++(2df,2p)	3424.4	2356.3	31.2/36.1 %
Taxol 6-31G	310.2	474.1	-52.8/31.4 %
Taxol 6-31G(d)	1104.2	1040.0	5.8/39.8 %
Taxol 6-31G++(d,p)	11225.9.5	10288.0	8.4/33.1 %
Valinomycin 6-31G	853.6	1104.4	-29.3/35.3 %
Valinomycin 6-31G(d)	2285.0	2104.8	7.9/38.9 %

Performance

Table: Taxol/6-31G(d) GPU performance

quartet size	CPU % by time	speed-up
1	1.7	28.5
4	6.5	20.9
6	1.9	18.8
16	12.3	13.1
24	6.6	10.6
36	1.1	11.7
64	13.9	13.7
96	16.8	15.8
144	5.9	19.5
216	0.6	15.4
256	12.4	23.5
384	11.5	20.9
576	7.0	20.2
864	1.7	21.3
1296	0.2	16.6
overall	1031.94 s	16.6

Memory Bottleneck In Many-Body Methods

Some numbers first:
Let $i, j, k, l, \ldots$ refer to O indices
Let $a, b, c, d, \ldots$ refer to V indices
Let $p, q, r, s, \ldots$ be atomic indices N
O is on the order of a few hundred
V is on the order of a few thousand
$V >> O$
MP2 Energy requires $v(iajb)$ integral, if symmetry is used the algorithm needs $O^2V^2/2$ words. Lets say computation has 200 occupied and 4000 virtual orbitals - we need 2.56 TB of storage plus whatever local memory.
Memory requirements must be able to scale with computational improvements, otherwise method is limited.

Memory Bottleneck In Many-Body Methods

I adopted the following philosophy:

- Local memory requirements must not exceed VO^2
- Large data sets must reside in either disk or distributed memory
- There should be facility to perform collective I/O.
- The algorithm should not care where the data is, but should work well with either
- Reads and writes should be contiguous on the order of hundreds of megabytes.
- V^3O and higher datasets should be avoided whenever possible.
- I/O should overlap with computations
- Reads are slower than writes
- Dynamic loop blocking should be used to optimize I/O

MP2 Implementation

Naive approach: data stored as (O, O, N, N), first transform O indices, then V - requires very poor read pattern.
A better approach is to tricked into having contiguous $B * N^2$ read where B is a blocking parameter. Main features:

- All reads are contiguous, writes are noncontiguous but the overlap with computation
- Transformations are performed in order OO, VV
- The dataset is either in Global Arrays or in HDF5, depending on runtime argument
- Dataset blocking adapts to amount of memory in the system.

MP2 Scalability

On 4 Cray XE nodes (128 cores)

```
active: 164, virtual: 2348, atomic: 2935
Array::HDF5: MP2.v(qsij) 936194 MB parallel
OpenMP threads: 32
memory (per thread):1539.78 MB
    eri: 00:26:21.419691
    trans 1: 00:23:41.455598
    trans 2: 00:22:41.947598
    I/O: 00:09:50.369464, 10.2001 MB/s
    time: 01:40:59.232784
    ---------------------------------
    Ran out of time
```

MP2 Scalability

On 6 Intel 6-core nodes (36 cores, 144GB)

```
active: 76, virtual: 1653, atomic: 1995
Array::GA: MP2.v(qsij) { 63680400, 183 }, 93228.1 MB
6 threads
    eri: 00:42:45.194878
    t1: 00:31:28.224942
    t2: 00:05:25.490181
    I/O: 00:00:22.969283, 119.64 MB/s
    ------------------------------
    trans 3/4: 00:06:26.058619
    I/O: 00:02:36.191121, 104.501 MB/s
```

Coupled Cluster

Coupled cluster is a difficult beast to handle, due to difficult I/O patterns and the number of diagrams. But with some tricks I could get rather scalable and elegant algorithm.

- To make algorithm scalable, memory must not exceed VO^2. For (T) this implies outmost loops over V indices
- To make a disk implementation viable, all I/O operations must be either load or store, never accumulate.
- By using partially transformed amplitudes, there is a number of tricks available :-)
- Have capability to mix transparently disk and distributed memory, with less frequently used data sets in the latter

Coupled Cluster VVVV term

VVVV term is too large to store, evaluated on the fly.
Many ways to do so, but one is more interesting:
$V_{cd}^{ab} t_{ij}^{cd} = (V_{qs}^{ab} C_c^q C_d^s) t_{ij}^{cd} = V_{qs}^{ab} (C_c^q C_d^s t_{ij}^{cd}) =$
$V_{qs}^{ab} t_{ij}^{qs} = (C_a^p C_b^r) V_{qs}^{pr} t_{ij}^{qs} =$
$(C_a^p C_b^r) U_{ij}^{pr}$
Why?

- U_{ij}^{pr} symmetry:
  $U_{ij}^{pr} == U_{ji}^{rp}$ for any quartets p, r. 2x fewer computations.
- Other VT terms:
  Most of work is in first two transformations. Third and fourth transformations are cheap. Since last indices are in AO basis, any $VT2_{ij}$ can be formed for free.
- Small memory footprint

Coupled Cluster OVVV terms

With $VT2_{ij}$, $OVVV$ storage can be nearly dropped. Two more diagrams are needed:

- $V^{ij}_{qs} t^q_a t^s_b$ - no problem
- $V^{ie}_{qs} t^j_e$ - no problem
- $V^{ie}_{sq} t^j_e$ - Houston ...
  we have exchange integral. The simplest solution and the best is to reevaluate integrals
- VT1 terms are tricky - to still use symmetry, need to compute $U(i,j,qs) += C(i,p) * t(j,r) * V(p,q,s,r)$ and $U(j,i,qs) += t(i,p) * C(j,r) * V(p,q,s,r)$

Now CC can be implemented much simpler with the cost of secondary storage $O(N^2O^2)$ and per process memory $O(NO^2)$
MP2 gradient expressions have similar OVVV terms

Coupled Cluster Performance

```
CCSD N=556, O=57, V=436
2 Cray XE nodes, 64 cores - 00:28:45.158273
4 Cray XE nodes, 12 cores - 00:15:37.674860
Exalted 4 X5650@2.67GHz, 24 cores - 00:27:01.038026
```

C++ DFT prototype

I reimplemented DFT in C++ in such way that time consuming operations are in terms of matrix multiplies, functional kernels can be translated from GAMESS, and composed together easily

```
    functional* functional::new_(const std::string &name) {
#define FUNCTIONAL(NAME, F)                                    \
        if (name == NAME)                                      \
            return new_(boost::fusion::make_vector F);
        FUNCTIONAL("b3lyp", (slater(0.08), b88(0.72), lyp(0.81), vwn5(0.19)));
        CCHEM_ERROR("invalid functional \"%s\"", name);
    }
```

One idea that I had was to implement matrix multiplies in such way as to take into account partial sparsity while still using blocking and vector instructions through Eigen library

C++ CI prototype

I begin CI in C++ with arbitrary truncation, distributed storage, and playing tricks with bitsets to represent determinant strings
Generating determinants:

```
std::vector<bool> bits(no-ne, 0);
bits.resize(no, 1);
std::vector<String> strings;
do {
  // here you can skip unwanted permutations
  String::value_type bs(bits);
  strings.push_back(String(bs));
} while (std::next_permutation(bits.begin(), bits.end()));
```

Manipulating determinant:

```
int A, B; // determinant strings
~A; // space to excite to
A ^ B; // excitation between strings
__builtin_popcount(A ^ B); // excitation level
```

The idea is to have easy to tinker with CI code which can scale beyond (32,32) space.

Thank You

Thanks to Alexey for inviting me
Thank you for your attention and time
Questions, comments, criticism?