ET International

SCF optimizations

Execution times of 16 atom test

Optimization	Time (s)	Speedup
Reference implementation	176.35	1
Exploit symmetry of g()	30.58	5.8
Use lookup tables in g()	19.06	9.3
BLAS and LAPACK matrix routines	18.06	9.8
Don’t recompute h() each iteration	17.95	9.8
Cache results of g()	9.45	18.7

- All tests were performed on a workstation with an Intel Xeon processor running at 2.67GHz.
- Each optimization test includes all optimizations listed above it.
- Caching g() leads to the best performance, but the size of the cache grows with N^4.

Additional work skipping test

```
double twoel(double schwmax) {
  int i, j, k, l;

  for (i = 0; i < nbfn; i++) {
  for (j = 0; j < nbfn; j++) {
    if ((g_schwarz[i][j] * schwmax) < tol2e)
      {icut1 += nbfn * nbfn; continue;}
    double  KLTest = tol2e / g_schwarz[i][j];

  for (k = 0; k < nbfn; k++) {
  for (l = 0; l < nbfn; l++) {

    if (g_schwarz[k][l] < KLTest)
      {icut2 ++; continue;}

    icut3 ++;
    double gg = g(i, j, k, l);
    g_fock[i][j] += (       gg *
     g_dens[k][l]);
    g_fock[i][k] -= (0.50 * gg *
     g_dens[j][l]);

  } } } }

  return (0.50 * contract_matrices(g_fock,
    g_dens));
}
```

- The inner-most loop always tests if `g_schwarz[k][l]` is less than `KLTest`
- While creating `g_schwarz`, we can also determine what the maximum `g_schwarz[k][l]` is for each column k and store it in `g_schwarz_max_row_value[k]`
- Then, in the k loop, we can check if `g_schwarz_max_row_value[k]` is less than `KLTest`
- If it is, then we know that all the work in the l loop will be skipped, and can skip that loop entirely
- The additional memory requirements for this are small as we only need a one-dimensional array of size nbfn

Code changes required for additional work skipping test

```
double twoel(double schwmax) {
  int i, j, k, l;

  for (i = 0; i < nbfn; i++) {
  for (j = 0; j < nbfn; j++) {
    if ((g_schwarz[i][j] * schwmax) < tol2e)
      {icut1 += nbfn * nbfn; continue;}
    double  KLTest = tol2e / g_schwarz[i][j];

  for (k = 0; k < nbfn; k++) {
    if (g_schwarx_max_row_value[k] < KLTest)
        {icut4 ++; continue;}
  for (l = 0; l < nbfn; l++) {

    if (g_schwarz[k][l] < KLTest)
      {icut2 ++; continue;}

    icut3 ++;
    double gg = g(i, j, k, l);
    g_fock[i][j] += (       gg *
    g_dens[k][l]);
    g_fock[i][k] -= (0.50 * gg *
    g_dens[j][l]);

  } } } }

  return (0.50 * contract_matrices(g_fock,
    g_dens));
}
```

```
double makesz() {
  int i, j;
  double smax = 0.0;

  for (i = 0; i < nbfn; i++) {
  double row_max = 0.0;
  for (j = 0; j < nbfn; j++) {
    double gg = sqrt( g(i, j, i, j) );
    if (gg > smax) smax = gg;
    g_schwarz[i][j] = gg;
    if (gg > row_max) row_max = gg;
  }
  g_schwarx_max_row_value[i] = row_max;
  }

  return smax;
}
```

ET International

Exploit symmetry in g()

- g(i,j,k,l) returns the same value in the following cases:
  - i and j are swapped
  - k and l are swapped
  - i,j and k,l are swapped
- We can use this to reduce the calls to g() to 1/8th of the original.

Exploit symmetry in g()

- update(a, b, c, d, gg) {
  g_fock[b][a] += (gg * g_dens[c][d]);
  g_fock[c][a] -= (0.50 * gg * g_dens[b][d]);}

- update() should be an inline function or a macro to minimize overhead.

- We can do gg = g(i,j,k,l) once then perform the following updates:
  - update(i, j, k, l, gg)
  - update(i, j, l, k, gg)
  - update(j, i, k, l, gg)
  - update(j, i, l, k, gg)
  - update(k, l, i, j, gg)
  - update(l, k, i, j, gg)
  - update(k, l, j, i, gg)
  - update(l, k, j, i, gg)

ET International

ET International

Exploit symmetry in g()

- There are special cases to consider:
  - We must be sure to only iterate over i,j,k,l values which we haven't already processed due to the symmetry.
  - If i=j, we can't perform updates which swap i and j. Doing so would lead to the wrong values being accumulated into various g_fock entries.
  - In addition to i=j, we must also take care when k=l or i,j=k,l as well as any combinations of them.

ET International

Exploit symmetry in g()

- We end up with 6 separate variants of the twoel loop:
  - i, j, k, l can be freely swapped
  - i = j
  - k = l
  - i,j = k,l
  - i = j and k = l
  - i = j = k = l

Exploit symmetry in g()

- Loop structure when i, j, k, l can be freely swapped:

```
for (i = 0; i < nbfn; i++) {
  for (j = i + 1; j < nbfn; j++) {
    for (k = i; k < nbfn; k++) {
      if (k == i) l_start = 1 + j;
      else l_start = 1 + k;
      for (l = l_start; l < nbfn; l++) {
        //calculate g(i,j,k,l);
        //perform all eight updates
      }}}}
```

- The starting index for the `l` loop is a special case which depends on whether `k` is equal to `i`
- The other loops can be easily created by removing the loop and index for the appropriate symmetry
  - The number of updates to perform will be fewer since one or more of the swaps will not alter the arguments
  - The special case for the initial value of `l` is only required in the fully symmetric case. In all other cases, the `l` loop is either removed, or the initial value reduces to `1 + k`
  - All other variants have `l` set equal to another index:
    - k = l
    - i,j = k,l
    - i = j and k = l
    - i = j = k = l
- All loops could potentially be combined into one, but doing so may hurt performance.
  - Conditional updates in the inner loop would create a lot of overhead.
  - Other methods might cause difficulties when trying to parallelize the code or could make automatic compiler optimizations impossible.

ET International

ET International

Precomputing lookup table for g()

- Many calculations in g() only rely on i,j or k,l.
  - Intermediate values are calculated for both ij and kl inputs, then later used together.
  - All calculations which only rely on one of the pairs can easily be computed during initialization and stored in an N^2 sized array.

Removing exp() function from g()

Calculation of exijkl

```
double dxij = x[i] - x[j];
double dyij = y[i] - y[j];
double dzij = z[i] - z[j];
double dxkl = x[k] - x[l];
double dykl = y[k] - y[l];
double dzkl = z[k] - z[l];

double rab2 = dxij * dxij + dyij * dyij + dzij * dzij;
double rcd2 = dxkl * dxkl + dykl * dykl + dzkl * dzkl;

double expntIJ   = expnt[i] + expnt[j];
double expntKL   = expnt[k] + expnt[l];

double facij = expnt[i] * expnt[j] / expntIJ;
double fackl = expnt[k] * expnt[l] / expntKL;
double exijkl = exprjh(-facij * rab2 - fackl * rcd2);
```

Equivalent

Simplifying exijkl calculation

$$r_{ij}{}^2 = (x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2$$
$$expnt_{ij} = expnt_i + expnt_j$$
$$fac_{ij} = \frac{expnt_i * expnt_j}{expnt_{ij}}$$
$$ex_{ijkl} = e^{(-fac_{ij} * r_{ij}{}^2) + (-fac_{kl} * r_{kl}{}^2)}$$

ex_{ijkl} can be substituted with:

$$ex_{ijkl} = e^{(-fac_{ij} * r_{ij}{}^2)} * e^{(-fac_{kl} * r_{kl}{}^2)}$$

Then, we can define ex_{ij} as:

$$ex_{ij} = e^{(-fac_{ij} * r_{ij}{}^2)}$$

Now we can express ex_{ijkl} as the product of two terms, each of which only depends on ij or kl.

$$ex_{ijkl} = ex_{ij} * ex_{kl}$$

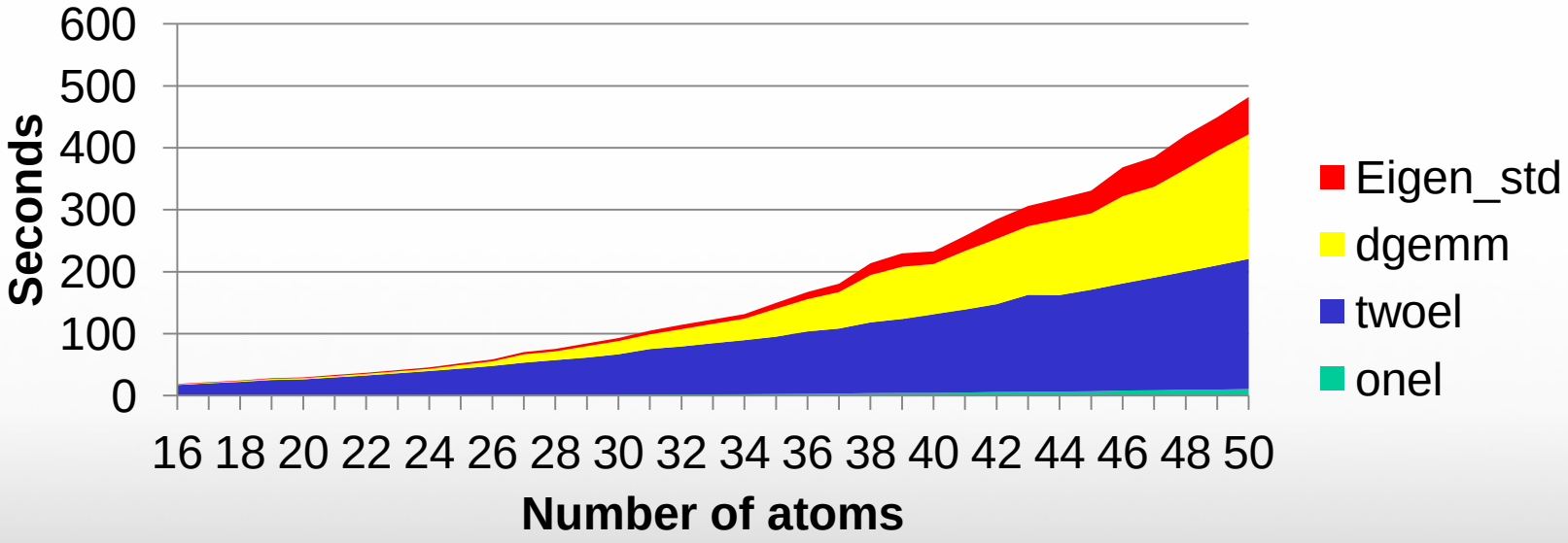
So by storing a table of all possible values for ex_{ij}, we can replaced 16 memory lookups, 10 multiplications, 13 additions/subtractions, 2 divisions and an e^x calculation with two[1] memory lookups and a single multiplication.

[1] There will be two additional memory lookups required to get $expnt_{ij}$ and $expnt_{kl}$ for other calculations in g()

Precomputing lookup table for g()

- Six values can be precomputed:
  - ex_{ij}
  - $expnt_{ij}$
  - x_{ij} (Replaces xp and xq)
  - y_{ij} (Replaces yp and yq)
  - z_{ij} (Replaces zp and zq)
  - $rnorm_{ij}$
- The same lookup tables are use for both i,j and k,l indicies
- Requires additional $6*N^2$ of memory
- Cache performance can be improved by storing all precomputed values as an array of structs rather than in separate arrays.

Execution time breakdown as problem size increases



- As the problem size increases, the time spent diagonalizing the matrix (Eigen_std and dgemm in above chart) becomes a larger portion of the overall execution time
- Fortunately, Eigen_std and dgemm can be replaced with standard matrix operations from BLAS and LAPACK

ET International

Replacement Matrix Routines

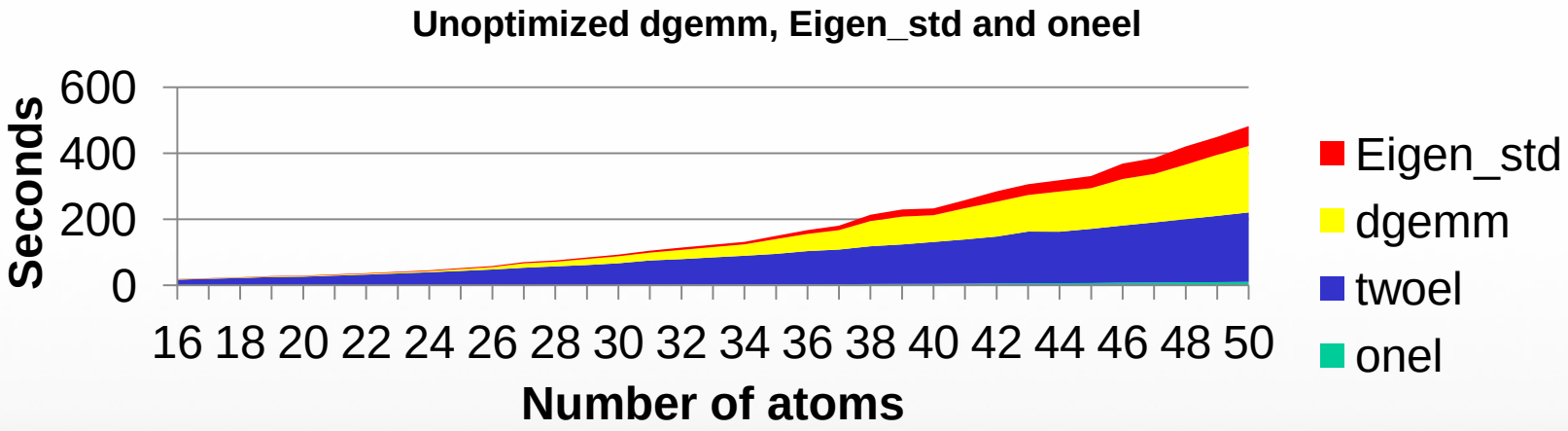
- BLAS can provide replacements for the following operations:
  - dgemm() calls in diagon()
  - ddot() can replace contract_matricies()
  - dscal() and daxpy() can replace damp()
    - `cblas_dscal(nbfn*nbfn, fac, g_dens, 1);`
      - `// g_dens = fac * g_dens`
    - `cblas_daxpy(nbfn*nbfn, 1.0 - fac, g_work, 1, g_dens, 1);`
      - `// g_dens = ((1.0 - fac) * g_work) + g_dens`
    - The resulting calculation is:
      - `g_dens = fac * g_dens + (1.0 - fac) * g_work`
      - Which is identical to the original operation in damp()
    - This may not increase performance if the compiler was already able to vectorize damp() since using BLAS routines requires two iterations over the array instead of one.
- LAPACK routines can replace the functions in diagonalize.c:
  - rsg() → dsygv()
  - rs() → dsyev()

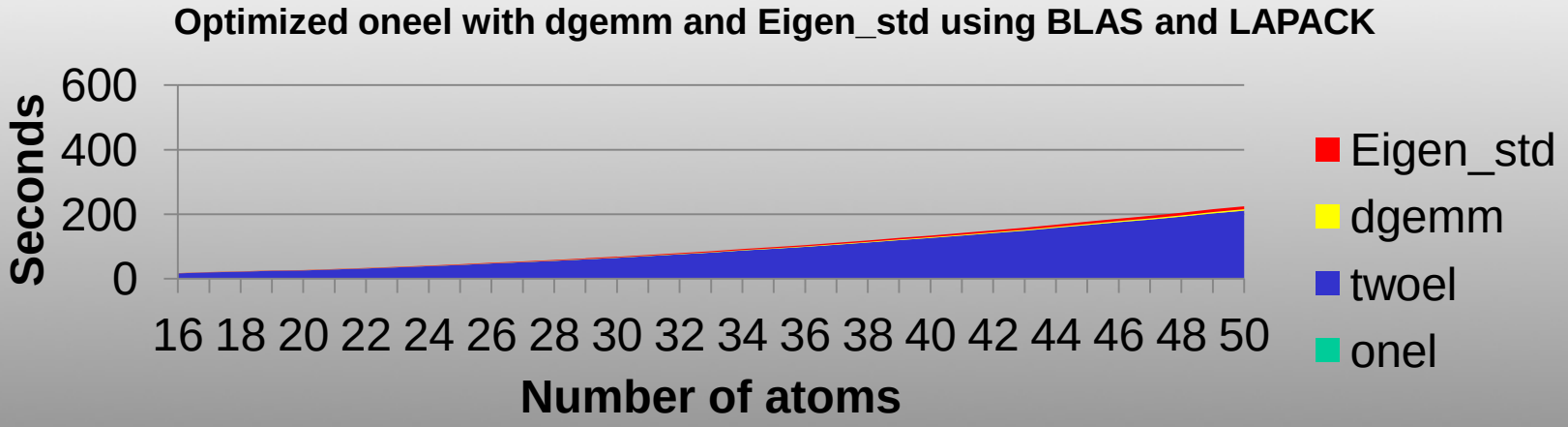
ET International

Removing oneel()

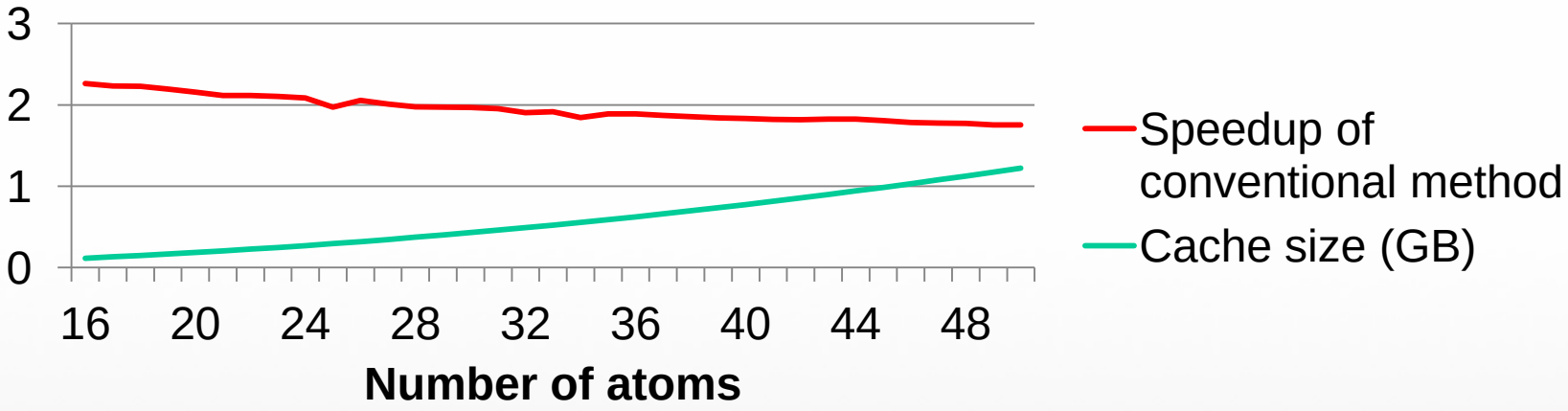
- The oneel() function doesn't take long to run, but still performs unnecessary calculations.
- g_fock is initialized with the same values each iteration.
- Instead, we can save a copy of the initial g_fock values and copy it back each iteration.
- We still have to call contract_matricies() to determine the one-electron energy contribution.

Execution time comparison





Conventional method vs. direct method



- Both conventional and direct methods included all optimizations.
- The conventional method is faster than the direct method in all tests performed.
- However, the usefulness of the conventional method is limited due to the amount of memory required, the additional power requirements, and the decreasing performance as the problem size increases.

ET International

Summary

- g() has an 8-way symmetry which can be exploited to reduce required calculations.
- Many intermediate calculations in g() can be precomputed and stored in a lookup table.
  - Allows removing a slow e^x calculation.
  - Reduces number of memory operations required in g().
- Matrix and vector routines can be trivially replaced with optimized libraries.
- Conventional method can speedup g() even further.
  - Very high memory requirements.
  - Incremental speedup with other optimizations only ~2x.