Accelerating Sparse Arithmetic in the Context of Newton's Method for Small Molecules with Bond Constraints

Carl Christian Kjelgaard Mikkelsen[1], Jesús Alastruey-Benedé[2], Pablo Ibáñez-Marín[2], and Pablo García Risueño[3,4,5]

[1] Department of Computing Science and HPC2N, Umeå University spock@cs.umu.se
[2] Instituto Universitario de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza {jalastru,imarin}@unizar.es
[3] Institut für Physik, Humboldt Universität zu Berlin
[4] Fritz-Haber Institut (MPG), Berlin
[5] Instituto de Biocomputación y Física de Sistemas Complejos, Zaragoza
risueno@physik.hu-berlin.de

The final publication is available at Springer via
http://dx.doi.org/10.1007/978-3-319-32149-3_16

Abstract. Molecular dynamics is used to study the time evolution of systems of atoms. It is common to constrain bond lengths in order to increase the time step of the simulation. Here we accelerate Newton's method for solving the constraint equations for a system consisting of many identical small molecules. Starting with a modular and generic base code using a sequential data layout, we apply three different optimization techniques. The compiled code approach is used to generate subroutines equivalent to a single step of Newton's method for a user specified molecule. Differing from the generic subroutines, these specific routines contain no loops and no indirect addressing. Interleaving the data describing different molecules generates vectorizable loops. Finally, we apply task fusion. The simultaneous application of all three techniques increases the speed of the base code by a factor of 15 for single precision calculations.

Keywords: Newton's method, non-linear equations, molecular dynamics, constraints, SHAKE, RATTLE, LINCS, compiled code approach, vector level parallelism, vectorizing compiler, SIMD.

1 Introduction

Molecular dynamics (MD) of bio-molecules and organic compounds is at present an extremely important tool for bio-medical purposes and in the chemical industry [1,2]. It is central for understanding phenomena within the human body and for the design of novel drugs. For instance, MD simulations of proteins have

been instrumental in the design of HIV therapies [3]. In industry, MD simulations enable the detailed analysis of a wide range of phenomena such as catalysis and adsorption [4,5,6].

In the context of MD, it is common to *constrain* internal degrees of freedom (usually bond lengths and bond angles), i.e. to keep their values constant throughout the simulation. Constraining fast degrees of freedom allows for an increase in the time step of the MD simulation, so that larger systems and intervals of real time can be simulated [7].

The imposition of constraints requires the solution of nonlinear equations. The most widely used methods are SHAKE, RATTLE and LINCS which all converge linearly [8,9,10]. Solving the constraint equations to the limits of machine precision is normally out of the question, unless the constraint block is allowed to consume a significant fraction of the total computational time, defeating the purpose of imposing constraints in the first place. For reasons of efficiency, accuracy and stability, it is desirable to develop a constraint algorithm which can satisfy the constraints within the limits imposed by machine precision and perform the corresponding calculations in a very efficient manner. Several authors have already sought to solve the constraint equations using Newton's method, which is locally second order convergent, together with a direct or an iterative method for the linear systems. However, their proposals were generally not satisfactory due to efficiency [11] or generality [12] reasons. In this paper we apply three different optimization techniques to Newton's method together with a direct linear solver and we solve the bond constraint equations for several solvents each consisting of many identical molecules.

1. **Compiled code approach**: we construct a code generator which reads a description of a molecule and writes loop free subroutines with direct addressing, which are then compiled and used to process all molecules of the given type.
2. **Data layout transformations**: we enable vector-level parallelism for functions with irregular patterns of memory access and computation by interleaving the data describing different molecules and linear systems of the same type.
3. **Task fusion**: we fuse distinct stages of Newton's method in order to facilitate data reuse and reduce the number of memory operations.

The combined effect of our three optimization techniques is a 15-fold increase in the single precision computational speed as demonstrated by our experiments with several commonly used organic solvents.

2 Newton's Method for Molecules with Bond Constraints

In MD the most commonly constrained degrees of freedom are general bond lengths and the hydrogen bond angles. For the sake of simplicity, in this paper we only tackle bond length constraints. However, note that hydrogen bond angles

can be constrained by imposing a bond length-like constraint between two atoms that are not actually covalently bonded, so our treatment is rather general.

Consider a molecule with m atoms and let $\mathbf{r}_i = (x_i, y_i, z_i)^T \in \mathbb{R}^3$ denote the coordinates of the ith atom. A bond length constraint is an equation of the form

$$\|\mathbf{r}_{a_k} - \mathbf{r}_{b_k}\|_2^2 - \sigma_k^2 = 0, \tag{1}$$

where $\sigma_k > 0$ is the length of the bond between atoms a_k and b_k, and $\|\cdot\|_2$ denotes the Euclidean norm. Physical bond lengths are on the order of 100 pm, where 1 pm $= 10^{-12}$ m. Let n denote the number of constrained bonds lengths and let

$$\mathbf{g} : \mathbb{R}^{3m} \rightarrow \mathbb{R}^n, \quad \mathbf{g}(\mathbf{r}) = (g_1(\mathbf{r}), g_2(\mathbf{r}), \dots, g_n(\mathbf{r}))^T \tag{2}$$

denote the constraint function where

$$g_k(\mathbf{r}) = \frac{1}{2}\left(\|\mathbf{r}_{a_k} - \mathbf{r}_{b_k}\|_2^2 - \sigma_k^2\right) \tag{3}$$

represents the bond between atoms a_k and b_k. The celebrated SHAKE and RATTLE algorithms are contingent upon the solution of nonlinear equations of the form $\mathbf{f}(\mathbf{r}) = \mathbf{g}(\phi(\mathbf{r})) = 0$, where $\phi \in \mathbb{R}^{3m} \rightarrow \mathbb{R}^{3m}$ is a function which depends on the algorithm. For the sake of clarity, this presentation is limited to the simplest case of $\phi(\mathbf{r}) = \mathbf{r}$, which corresponds to finding initial coordinates for each of the m atoms such that the n bond length constraints are satisfied. If we assume that the Jacobian $\mathbf{Dg}(\mathbf{r}) \in \mathbb{R}^{n \times 3m}$ has full row rank, then Newton's method for the constraint equation $\mathbf{g}(\mathbf{r}) = 0$ takes the form

$$\mathbf{r} := \mathbf{r} - \mathbf{Dg}(\mathbf{r})^T(\mathbf{Dg}(\mathbf{r})\mathbf{Dg}(\mathbf{r})^T)^{-1}\mathbf{g}(\mathbf{r}). \tag{4}$$

The problem of evaluating each Newton iteration (4) can be split into the following tasks:

1. Compute the n components of $\mathbf{g}(\mathbf{r})$.
2. Build a sparse representation of the symmetric matrix $\mathbf{A} = \mathbf{A}(\mathbf{r})$ given by

$$\mathbf{A}(\mathbf{r}) = \mathbf{Dg}(\mathbf{r})\mathbf{Dg}(\mathbf{r})^T. \tag{5}$$

3. Expand the representation of $\mathbf{A}$ into extended arrays which can accept any fill-in during the factorization. Simultaneously, overwrite any fill-in from previous factorizations.
4. Compute a sparse Cholesky factorization of $\mathbf{A} = \mathbf{L}\mathbf{L}^T$.
5. Solve the linear system $\mathbf{Ly} = \mathbf{g}(\mathbf{r})$ using forward substitution.
6. Solve the linear system $\mathbf{L}^T\mathbf{z} = \mathbf{y}$ using backward substitution.
7. Do the linear update $\mathbf{r} := \mathbf{r} - \mathbf{Dg}(\mathbf{r})^T\mathbf{z}$.

Normally, one can only monitor the components of the residual, i.e. $\mathbf{g}(\mathbf{r})$, but in our case we can estimate the relative constraint violation. Specifically, if $\|\mathbf{r}_{a_k} - \mathbf{r}_{b_k}\|_2 \approx \sigma_k$, then

$$\frac{g_k(\mathbf{r})}{\sigma_k^2} = \frac{1}{2}\frac{\|\mathbf{r}_{a_k} - \mathbf{r}_{b_k}\|_2^2 - \sigma_k^2}{\sigma_k^2} \approx \frac{\|\mathbf{r}_{a_k} - \mathbf{r}_{b_k}\|_2 - \sigma_k}{\sigma_k}, \tag{6}$$

which allows us to terminate the iteration when the constraints are satisfied to a specific tolerance. The factorization (Task 4) need not be the Cholesky factorization and there are several variants to choose from including a right-looking, a left-looking, and a multi-frontal factorization [15].

3 Compiled Code Optimization

The compiled code approach has been applied to the solution of linear systems in [13] and is rediscovered periodically in this context according to [14]. However, to the best of our knowledge, it has not been applied to all aspects of a complete Newton step.

3.1 Transforming a Numerical Routine into a Code Generator

Consider a single step of Newton's method with a sparse direct solver based on Cholesky's decomposition. As there is no need to pivot for the sake of stability, the order of the instructions depends only on the chemical structure of the molecule and not on the actual values of the spatial coordinates of the atoms.

Starting from an implementation of Newton's method, which uses a sparse direct solver and indirect addressing, we developed a generator, which writes loop-free subroutines that use direct addressing and are equivalent to a complete Newton step for a molecule of a specific type.

The process of developing the generator can be illustrated in terms of Task 5, the solution of a lower triangular linear system $\mathbf{Lx} = \mathbf{b}$. Fig. 1 contains a `C` implementation of forward substitution for matrices in compressed sparse column (CSC) format. Here `n` is the dimension of the system, `adj[k]` is the row index, `val[k]` is the value of the kth nonzero of the matrix $\mathbf{L}$, and `xadj[i]` marks the start of the ith column inside arrays `adj[]` and `val[]`.

```
void forward(int n, int *xadj, int *adj,
             float * restrict L, float * restrict b) {
  for (int i=0; i<n; i++) {
    b[i]=b[i]/L[xadj[i]];
    for (int j=xadj[i]+1; j<xadj[i+1]; j++) {
      b[adj[j]]=b[adj[j]]-L[j]*b[i]; // inner loop body
    }
  }
}
```

Fig. 1. A `C` subroutine for solving a non-singular lower triangular linear system $\mathbf{Lx} = \mathbf{b}$ in CSC format `(xadj, adj, val)` using forward substitution.

We obtained our subroutine generator by systematically replacing every computation involving floating point numbers with an instruction writing direct addressing statements equivalent to the original computation to a file. This transformation has been applied to the subroutine displayed in Fig. 1 producing the generator given in Fig. 2.

```
void generate_forward(FILE *fp, int n, int *xadj, int *adj) {
  fprintf(fp,"void forward(float *L, float *b)\n{\n");
  for (int i=0; i<n; i++) {
    fprintf(fp,"  b[%d]=b[%d]/L[%d];\n",i,i,xadj[i]);
    for (int j=xadj[i]+1; j<xadj[i+1]; j++) {
      fprintf(fp,"  b[%d]=b[%d]-L[%d]*b[%d];\n",adj[j],adj[j],j,i);
    }
  }
  fprintf(fp,"}\n\n");
}
```

Fig. 2. Given the adjacency graph of a lower triangular matrix $\mathbf{L}$, this code generates a `C` subroutine `forward` with the linear list of instructions necessary for solving $\mathbf{Lx} = \mathbf{b}$ using forward substitution.

We eliminated Task 3 (matrix expansion) by mapping the nonzero entries of $\mathbf{A}$ directly to their correct location inside an extended array, which is exactly large enough to absorb the fill-in during the numerical factorization. Moreover, our generator tracks the status of all entries during the symbolic factorization and issues instructions which treat an entry as zero until it fills in.

Since the required number of subroutine generators is low, they were all developed manually. If necessary, this task can be done automatically using a parser that transforms computation statements into `fprintf()` instructions that resolve indirect addressing.

3.2 Generated Code

Fig. 3 shows the forward substitution subroutine written by the code generator in the case of the chloroform solvent. When comparing the new subroutine with the generic subroutine in Fig. 1 we observe that:

1. The code is fully unrolled, i.e. it is loop-free.
2. There is no indirect addressing.

Replacing indirect addressing with direct addressing reduces the number of memory operations. For instance, the read access of `b[1]` in the subroutine `forward_chloroform()` is performed by a single assembly instruction and requires just one memory access to the array `b[]`. The corresponding access in the generic `forward()` (`b[adj[xadj[i]+1]]`) is a memory read with double indirection (three memory accesses) which requires many instructions.

4 Vectorization through Data Transformations

It is hard to exploit vector-level parallelism in subroutines such as those in Fig. 1 and Fig. 3 due to their irregular patterns of memory access and computation. Nevertheless, since the solvent is composed of identical molecules, we can generate vectorizable loops that process a group of molecules/linear systems simultanously by interleaving their data, instead of storing the data for each item in a single contiguous block.

```
void forward_chloroform(float * restrict L, float * restrict b) {
  b[0]=b[0]/L[0];
  b[1]=b[1]-L[1]*b[0]; // inner loop body
  b[2]=b[2]-L[2]*b[0]; // inner loop body
  b[3]=b[3]-L[3]*b[0]; // inner loop body
  b[1]=b[1]/L[4];
  b[2]=b[2]-L[5]*b[1]; // inner loop body
  b[3]=b[3]-L[6]*b[1]; // inner loop body
  b[2]=b[2]/L[7];
  b[3]=b[3]-L[8]*b[2]; // inner loop body
  b[3]=b[3]/L[9];
}
```

Fig. 3. Generated code for solving $\mathbf{Lx} = \mathbf{b}$ using forward substitution for the chloroform solvent.

Fig. 4 shows a generic subroutine for solving `p` lower triangular linear systems. The subroutine assumes that the kth nonzero entries from each of the `p` systems are stored contiguously in memory. The two loops over `t` can be vectorized by a compiler like `GCC` or `ICC` if the iteration count `p` is sufficiently large. In our experiments with `GCC` and the `AVX2` instruction set (256 bits, 8 single precison floating point numbers), the optimal value of `p` depends on the level of optimization. Generic codes peak at higher values of the loop length, compared with loop free codes, see Subsection 6.2.

```
void forward_interleaved(int n, int *xadj, int *adj,
                         float * restrict L, float * restrict *b) {
  int i, j, t;
  const int p=LOOP_LENGTH;
  for (i=0; i<n; i++) {
    for (t=0; t<p; t++) b[p*i+t]=b[p*i+t]/L[p*xadj[i]+t];
    for (j=xadj[i]+1; j<xadj[i+1]; j++) {
      #pragma GCC ivdep // b[p*adj[j]+t] and b[p*i+t] do not overlap
      for (t=0; t<p; t++) b[p*adj[j]+t]=b[p*adj[j]+t]-L[p*j+t]*b[p*i+t];
    }
  }
}
```

Fig. 4. A `C` code for solving a group of p non-singular lower triangular linear systems $\mathbf{L}_i\mathbf{x}_i = \mathbf{b}_i$ in CSC format and all with the same sparsity pattern. The parameter `LOOP_LENGTH` should be set at compile time.

The generator produces vectorizable code by generalizing a statement such as `b[1]=b[1]-L[1]*b[0]` into a simple loop with no dependencies over `p` systems:

```
for (t=0; t<p; t++) b[p*1+t]=b[p*1+t]-L[p*1+t]*b[p*0+t];
```

Vectorial performance can be improved by ensuring that the arrays are properly aligned at a suitable boundary. It can also be increased by issuing the directive

`#pragma gcc ivdep` which asserts that there is no aliasing in the loop. This avoids loop versioning and runtime checks.

5 Task Fusion

Modular programming is a key concept in software development and reflects how we think mathematically: a large problem is broken into smaller pieces which are solved separately. Modular programming facilitates code reuse, testing, debugging, maintenance, and future development, but it can increase the number of memory operations as intermediate results have to be stored and retrieved.

It is possible to fuse the construction of the right hand side (Task 1) and the matrix (Task 2) with the factorization (Task 4) and the forward sweep (Task 5). This hinges on the fact that pivoting is not necessary for systems which are symmetric positive definite, but a left looking, rather than a right looking factorization is required. In our case we merely fused the factorization of the matrix with the forward sweep. It is not necessary to compute all components of z (Task 6) before initiating the linear update (Task 7). Specifically, since

$$\mathbf{r} := \mathbf{r} - \mathbf{Dg}(\mathbf{r})^T \mathbf{z} = \mathbf{r} - \sum_{k=1}^{n} \mathbf{v}_k z_k \tag{7}$$

where $\mathbf{v}_k$ is the kth column of $\mathbf{Dg}(\mathbf{r})^T$, we can define $\mathbf{r}^{(n+1)} = \mathbf{r}$ and compute $\mathbf{r}^{(k)} := \mathbf{r}^{(k+1)} - \mathbf{v}_k z_k$ as soon as the backward substitution sweep has produced z_k. The vector $\mathbf{r}^{(1)}$ will then contain the result of the linear update (7). In this manner we fused the backward sweep and the linear update.

6 Numerical Experiments

In this paper we do not discuss how to integrate Newton's method into existing libraries for molecular dynamics. This allows us to concentrate on the effect of the three different optimization techniques and we avoid the discussion of a number of questions which are application or even library specific.

6.1 Methodology

In order to demonstrate the effect of the three different optimization techniques we wrote $8 = 2^3$ different implementations with the generic name `newtonXYZ`. The coding is as follows:

1. Sequential data layout (X=0) versus interleaved data layout (X=1).
2. Generic (Y=0) versus subroutines for molecules of a specific type (Y=1).
3. Complete task separation (Z=0) versus partial task fusion (Z=1).

	structural information				flops						
molecule	atoms	bonds	nnz	fill-in	$\sqrt{a}$	a/b	$a \cdot b$	$a+b$	$a-b$	$\lvert a \rvert$	total
acetone	10	9	24	0	9	42	252	93	255	9	660
acetonitrile	6	5	12	0	5	22	126	49	129	5	336
butanol	15	14	39	0	14	67	417	148	420	14	1080
chloroform	5	4	10	0	4	18	102	40	104	4	272
ethanol	9	8	19	0	8	37	223	82	226	8	584
methanol	6	5	12	0	5	22	126	49	129	5	336
THF	13	13	38	2	13	66	400	141	401	13	1034

Table 1. An alphabetical list of the molecules used in our experiments. The dimension of the matrix $\mathbf{A}(\mathbf{r}) = \mathbf{Dg}(\mathbf{r})\mathbf{Dg}(\mathbf{r})^T$ is equal to the number of bonds. The number of structural nonzeros on or below the main diagonal of $\mathbf{A}(\mathbf{r})$ is given in the column labeled "nnz". The flop count for one complete Newton step as implemented in `newton000` is given for each molecule. Tetrahydrofuran is abbreviated as "THF".

We implemented a solver based on a right-looking Cholesky factorization. We interleaved all the information representing groups of molecules and enforced suitable memory alignment. For all versions, we assisted the compiler with pragmas in order to vectorize specific loops. Moreover, we used the `restrict` qualifier to inform the compiler that different pointers do not alias. This allows for better code generation as there is no need to generate both scalar and vectorial versions, and runtime checks for aliasing are avoided. We explored partial task fusion as described in Section 5. Experiments were carried out on a workstation with an Intel i7-4770 processor (Haswell microarchitecture, 3.4GHz, 8MB L3 cache) and 8GB of RAM running Mageia 5 Linux (3.19.6-desktop-2.mga5 kernel).

We used the `GCC` compiler (version 4.9.2) with the following flags: `-O3 -std=c11 -march=native -fno-math-errno` which generate AVX2 instructions.

We selected 7 organic solvents for our numerical experiments: acetone, acetonitrile, butanol, chloroform, ethanol, methanol, and tetrahydrofuran, see Table 1. They are all produced and used on an industrial scale[6]. We simulated solvents composed of 1,600,000 molecules. For each solvent and variation of Newton's method we applied 10 iterations of Newton's method. For each execution we measured the wall-clock time. We made several repetitions of each experiment in order to ensure the reliability of the measurements.

6.2 Results

All results in this section are related to calculations which were performed using a single core and single precision floating point numbers. Since solvents typically consist of many copies of the same small molecule, parallelization across a mul-

[6] Ethanol is frequently consumed by humans during festive occasions such as conference dinners.

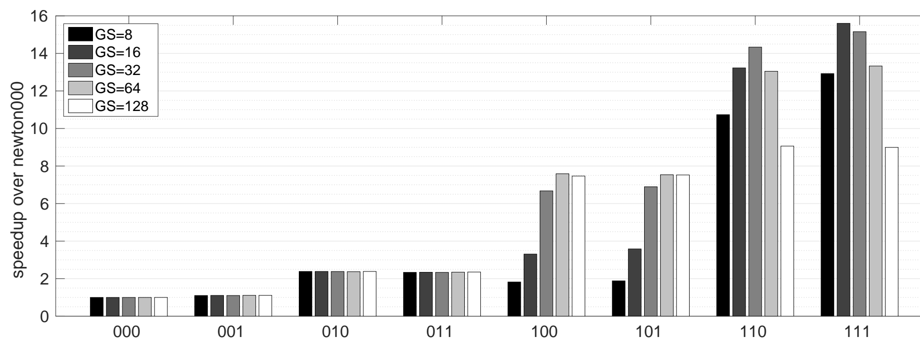

Fig. 5. This figure shows the speedup over the base code averaged over all molecules and computed for each implementation of Newton's method and group size (GS) separately.

ticore machine is straight forward and maximizing the single core performance is always a necessary first step.

We interleaved the molecules in groups of size 8, 16, 32, 64, and 128. With 8 codes, 7 molecules and 5 different values of the group size, there were 280 benchmarks to evaluate. In each case we did 20 repetitions. We measured the wall-clock time and computed the median run-time which is less sensitive to the effect of outliers. In the vast majority (265 of 280) cases the coefficient of variation was less than 5% and it was less than 10% in all cases. The speedup S_{XYZ} of `newtonXYZ` over the base code `newton000` is computed as $S_{\text{XYZ}} = \frac{m(T_{000})}{m(T_{XYZ})}$, where $m(T_{\text{XYZ}})$ is the median runtime of `newtonXYZ`. Speedups S_{XYZ} corresponding to group size 16 are given in Figure 2.

For each version of Newton's method and each group size we computed the average of the speedup for all molecules with respect to the base code `newton000`. These results are displayed in Figure 5.

We also examined our codes using Intel's SDE (Software Development Environment). Deterministic counts for different types of instructions were determined for a benchmark involving 16,000 tetrahydrofuran (THF) atoms and 10 Newton iterations per molecule. Table 3 shows, for each version of Newton's method, the total number of executed instructions, the total number of floating-point operations (FLOPs), the fraction of FLOPs computed by scalar instructions, the fraction of FLOPs computed by vector instructions, and the FLOPs per instruction ratio.

The highest speedups are achieved through the application of all three optimization techniques. The code `newton111` achieves speedups in the interval from 14.97 to 16.14, when the group size is 16, see Table 2. As the optimizations are applied the total number of instructions required to execute the THF benchmark is reduced by more than 97%, from 1169.2 million instructions (MI) to a mere 33.6 MI, see Table 3.

Generic subroutines (Y=0) versus specific subroutines (Y=1). Specific subroutines achieve speedups between 2.1 and 2.4 with sequential data layouts

molecule	sequential layout				interleaved layout			
	000	001	010	011	100	101	110	111
acetone	1.00	1.13	2.32	2.28	3.27	3.53	13.04	15.59
acetonitrile	1.00	1.11	2.28	2.25	3.31	3.63	13.58	15.80
butanol	1.00	1.14	2.29	2.44	3.41	3.66	13.27	16.04
chloroform	1.00	1.09	2.30	2.14	3.26	3.59	13.22	15.41
ethanol	1.00	1.13	2.42	2.35	3.46	3.71	13.56	16.14
methanol	1.00	1.10	2.22	2.19	3.32	3.65	13.03	15.26
THF	1.00	1.05	2.86	2.74	3.14	3.32	12.89	14.97

Table 2. Speedups S_{XYZ} for 8 different implementation of Newton's method `newtonXYZ` over the base code `newton000`. The codes are identified by their three digit binary extension `XYZ`. The molecules were interleaved in groups of size 16 for `newton1YZ`.

version	instr.	flops			
		total	$\frac{\text{scalar}}{\text{total}}$	$\frac{\text{vector}}{\text{total}}$	$\frac{\text{flops}}{\text{instr.}}$
000	1169.2	165.5	1	0	0.14
001	1130.4	165.5	1	0	0.15
010	226.2	134.7	1	0	0.60
011	226.1	134.7	1	0	0.60
100	282.9	165.5	0.152	0.848	0.58
101	267.7	165.5	0.139	0.861	0.62
110	52.2	135.4	0	1	2.59
111	33.6	135.4	0	1	4.03

Table 3. Instruction and flop counts in millions for a benchmark consisting of 16,000 THF molecules and 10 Newton iterations per molecule. The group size was 16.

(`newton00Z` versus `newton01Z`) and between 1.2 and 6.9 for the interleaved data layouts (`newton10Z` versus `newton11Z`) depending on the group size, see Figure 5. Replacing generic codes `newtonX0Z` with specific codes `newtonX1Z` removes more than 80% of the instructions, see Table 3. The deleted instructions include counter increments, comparisons, and jump instructions required for loops, as well as memory operations associated with indirect addressing, all of which are no longer necessary. The instruction count is reduced even further as the matrix expansion (Task 3) is avoided. The flop count is reduced as dummy operations involving zeros are avoided during the sparse factorizations.

Sequential data layout (X=0) versus interleaved data layout (X=1). For the sequential data layout versions `newton0YZ`, the compiler is not able to generate any vectorial code: all floating point operations are performed using scalar instructions. On the other hand, when interleaving the data layout `newton1YZ`, the compiler generates codes with high percentages of their FLOPs performed by vector instructions (around 85% and 100% for generic and specific

codes, respectively). This reduces the number of executed instructions by more than 75% in all cases, see Table 3. The performance of the vectorized codes peaks at a specific value of the group size, see Figure 5. This happens due to a trade-off between vectorization profitability and the temporal reuse of vector registers and cache. For small group sizes, some loops are not vectorized because their iteration counts are too small. The generic code is less amenable to vectorization and require group sizes above 32 to show speedups close to the AVX SP vector length. On the other hand, when the group size is increased it is impossible to retain in vector registers all the values which could be reused, and the compiler has to generate spill code which increases L1 cache traffic. Eventually, the L1 data cache is exhausted and we experience a substantial drop in performance. A simple calculation can be offered in support of this second part of the argument. The THF molecule involves 13 atoms and 13 bonds. The amount of memory required to store the data necessary to formulate and solve the constraint equation can be computed as follows: 39 floating-point (FP) numbers for the spatial coordinates of the atoms, 13 FP numbers for the right hand side, and 40 FP numbers for the matrix, a total of 92 FP numbers or 368 bytes in single precision. If 128 molecules are interleaved, 47104 bytes are required and we exhaust the 32kB L1 data cache capacity of our i7-4770 CPU. When we interleave 64 THF molecules, we only require 23552 bytes, i.e. less than the L1 data cache capacity.

Complete task separation (Z=0) versus partial task fusion (Z=1). Task fusion has a significant effect on specific vectorized code (`newton110`) and small groups. For instance, when the group size is 16, it causes speedups in the interval from 1.16 to 1.21 depending on the molecule, see Table 2.

7 Conclusions

If you are solving a large number of identical sparse problems, then you should consider the simultaneous application of three distinct optimization techniques: the compiled code approach, partial task fusion, and interleaving the data describing different problems, as compiler technology has advanced to the point were a 15-fold increase in computational speed using `AVX2` instructions may be possible.

We demonstrated speedups of this magnitude by solving the constraint equations for a solvent consisting of a large number of identical molecules of a specific type using Newton's method. We wrote 8 different implementations of Newton's method using a direct solver based on a right looking Cholesky factorization algorithm. We tested our codes on 7 different organic solvents which are produced on an industrial scale. The combined effect of all three optimization techniques is a single precision speedup between 14.97 and 16.14 for each of the different solvents.

8 Acknowledgments

The work is supported by eSSENCE, a collaborative e-Science programme funded by the Swedish Research Council within the framework of the strategic research areas designated by the Swedish Government. It is also supported in part by grants TIN2013-46957-C2-1-P and Consolider NoE TIN2014-52608-REDC (Spanish Gov.), gaZ: T48 research group (Aragón Gov. and European ESF), and HiPEAC-3 NoE (European FET FP7/ICT 287759). P.G. Risueño is funded by MPG. We would like to thank our reviewers as their comments made it possible to improve the clarity of our manuscript. We are grateful to our editor Roman Wyrzykowski who encouraged us to continue improving our code and allowed us to exceed the page limitation.

References

1. Adcock, S.A., McCammon, J.A.: Molecular Dynamics: Survey of Methods for Simulating the Activity of Proteins. Chem. Rev. 5, 1589–1615 (2006)
2. Frenkel, D., Smit, B.: Understanding molecular simulations: From algorithms to applications. 2nd Edition, Academic Press (2002)
3. Moraitakis, G., Purkiss, A. G., Goodfellow J. M.: Simulated dynamics and biological macromolecules. Reports on Progress in Physics 66, 383 (2003)
4. Liu, H., Sale, K. L., Holmes, B. M., Simmons, B. A. and Singh, S.: Understanding the Interactions of Cellulose with Ionic Liquids: A Molecular Dynamics Study. J. Phys. Chem. B 114-12, 4293–4301 (2010)
5. Li, C., Tan, T., Zhang, H., Feng, W.: Analysis of the Conformational Stability and Activity of Candida antarctica Lipase B in Organic Solvents: insights from MD and QM simulations. J. Bio. Chem. 285, 28434–28441 (2010)
6. Skoulidas, Anastasios I., Sholl, David S.: Self-Diffusion and Transport Diffusion of Light Gases in Metal-Organic Framework Materials Assessed Using Molecular Dynamics Simulations. J. Phys. Chem. B. 33, 15760–15768 (2005)
7. García-Risueño, P., Echenique, P., Alonso, J. L.:, Exact and efficient calculation of Lagrange multipliers in biological polymers with constrained bond lengths and bond angles: Proteins and nucleic acids as example cases. J. Comp. Chem. 32, 3039-3046 (2011)
8. Ryckaert, J. P., Ciccotti, G., Berendsen, H. J. C.: Numerical integration of the Cartesian equations of motion of a system with constraints: Molecular dynamics of n-alkanes. J. Comp. Phys. 23, 327–341 (1977)
9. Andersen, H. C.: Rattle: A “velocity” version of the Shake algorithm for molecular dynamics calculations. J. Comp. Phys. 52, 24–34 (1983)
10. Hess, B., Bekker, H. Berendsen, H. J. C. Fraaije, J. G. E. M.: LINCS: A Linear constraint solver for molecular simulations. J. Comp. Chem. 18, 1463–1472 (1997)
11. Barth, E., Kuczera, K., Leimkuhler, B., Skeel, R.: Algorithms for constrained molecular dynamics, J. Comp. Chem. 16,10, 11921209 (1995)
12. Bailey, A.G., Lowe, C.P.: MILCH SHAKE: An efficient method for constraint dynamics applied to alkanes, J. Comp. Chem. 30,15, 2485–2493 (2009)
13. Gustavson, F. G., Liniger, W., Willooughby R.: Symbolic generation of an optimal Crout algorithm for sparse systems of linear equations. J. Assoc. Comput. Mach. 17, 87–100 (1970)

14. Duff, I. S.: The impact of high-performance computing in the solution of linear systems: trends and problems. J. Comput. Appl. Math. 123, 515–530 (2000)
15. Davis, T. A.: Direct methods for sparse linear systems. SIAM (2006)