Portable CPU implementation of Wilson, Brillouin and Susskind fermions in lattice QCD

Stephan Dürr [a,b]

[a] *Department of Physics, University of Wuppertal, 42119 Wuppertal, Germany*
[b] *Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich, Germany*

Abstract

A modern Fortran implementation of three Dirac operators (Wilson, Brillouin, Susskind) in lattice QCD is presented, based on OpenMP shared-memory parallelization and SIMD pragmas. The main idea is to apply a Dirac operator to N_v vectors simultaneously, to ease the memory bandwidth bottleneck. All index computations are left to the compiler and maximum weight is given to portability and flexibility. The lattice volume, $N_xN_yN_zN_t$, the number of colors, N_c, and the number of right-hand sides, N_v, are parameters defined at compile time. Several memory layout options are compared. The code performs well on modern many-core architectures (480 Gflop/s, 880 Gflop/s, and 780 Gflop/s with $N_v = 12$ for the three operators in single precision on a 72-core KNL processor, a 2×24-core Skylake node yields similar results). Explicit run-time tests with CG/BiCGstab inverters confirm that the memory layout is relevant for the KNL, but less so for the Skylake architecture. The ancillary code distribution contains all routines, including the single, double, and mixed precision Krylov space solvers, to render it self-contained and ready-to-use.

1 Introduction

An ideal lattice QCD code is short, easy to read (hence to enhance/modify), and compiles in a fully portable manner into a fast-performing executable. Such codes are hard to find, as these requirements tend to be in conflict with each other. But there are means and ways to mitigate the conflict, and this article reports on a specific effort in this direction.

Lattice QCD is the regulated theory of quarks and gluons [1–3]. While a significant amount of quantum field theory knowledge goes into its formulation, the actual computational problem is easy to describe. It is about frequently solving large linear systems[1]

$$Du = b \tag{1}$$

for u, with a given vector b and a given Dirac matrix D (which is sparse and badly conditioned). In this sense lattice QCD is a rather typical subfield of computational physics, except that D

[1] Here "frequently" means $O(10^5)$ times, "large" implies a $n \times n$ matrix with $n = 402\,653\,184$ for a Wilson fermion on a $64^3 \times 128$ lattice, and depending on the quark mass the condition number of $D^\dagger D$ is often in the range $10^6 \ldots 10^8$. The factor 10^5 reflects the production of an ensemble of 1000 gauge configurations, separated by ten $\tau = 1$ HMC trajectories, assuming that each of these requires $O(10)$ inversions.

is usually not available in a standard sparse format[2], but only implicitly, i.e. through a routine which implements the matrix-vector multiplication $v \leftarrow Du$. One is forced to use Krylov-space solvers, such as CG [4] or BiCGstab [5], whenever possible with some structure preserving preconditioning [6]. For a very nice overview of numerical issues in lattice QCD computations the reader is referred to Ref. [7]. In short, this paper is about efficiently implementing, on a CPU, the matrix-vector multiplication that is at the heart of solving Eq. (1) by means of the CG or BiCGstab algorithm, where D is a sparse matrix that encodes the properties of a "Wilson", "Brillouin" or "staggered/Susskind" fermion, i.e. $D \in \{D_{\mathrm{W}}, D_{\mathrm{B}}, D_{\mathrm{S}}\}$.

Lattice QCD production codes tend to use hybrid parallelism, i.e. there is a coarse-grained parallelism between nodes, and a fine-grained parallelism within each node. The coarse-grained parallelism is usually implemented with MPI, the fine grained parallelism with OpenMP on CPUs (alternatively: OpenACC on GPUs). The latter requires a symmetric multiprocessing (SMP) shared memory architecture within each node. Typically, a geometric domain decomposition is used on the large scale, e.g. a $48^3 \times 96$ lattice might be distributed onto a $2 \times 2 \times 2 \times 4$ grid of nodes, each of which hosts a local 24^4 lattice. Explicit MPI commands are used to organize the "halo exchange" among the 32 nodes, and OpenMP pragmas are used to organize the work sharing of the N_{thr} SMP-threads which handle a given local lattice.

To minimize execution time, the single-node CPU performance must be maximized through OpenMP parallelization and vectorization pragmas, while overlapping communication and computation must be organized through clever MPI calls. These issues are logically separated, and this is why it makes sense[3] to first upgrade a serial code into a fast OpenMP code (with vectorization pragmas), and to address MPI parallelization in the second step. This paper reports on a dedicated effort to efficiently handle the first step, and establishes a ranking among various memory layout options. With such results in hand, it will be a more straightforward endeavor to address the second step (adding suitable MPI calls) in a forthcoming publication.

Two properties of the code presented in this article are essential for good balance between CPU portability and acceptable performance figures (see below). The first feature is the use of multiple right-hand sides (RHS) in Eq. (1), i.e. b and u are matrices of size $n \times N_v$, where N_v is the number of RHS. This is crucial on systems limited by memory bandwidth, since the Dirac matrix D depends on a gauge field U which is loaded from memory (besides u and v, of course) in each call $v \leftarrow Du$. The second feature is the decision to use the RHS-index as loop index for the single-instruction-multiple-data (SIMD) pipeline. In QCD terminology one would say that – besides the N_c color and 4 spinor degrees of freedom – an additional "internal degree of freedom" is introduced in the storage scheme. The main difference is that its value is a matter of convenience, i.e. it has has no physics implication. This avoids the reshuffle operation from a generic read-in format (where the site indices on b or v are slower than any internal degree of freedom) to a dedicated SIMD format that is needed if, say, a subset of the space-time coordinates of the local lattice is used as SIMD index.

The philosophy behind the code presented in this article is to leave all index computations and all optimization work to the compiler. Only a set of SIMD pragmas is used to tell the compiler that it should SIMD-ize loops over the RHS-index (which runs from 1 to N_v) with

[2]Such as "compressed sparse row" (CSR) or "compressed sparse column" (CSC) format; see e.g. the MATLAB documentation on `sparse` for a quick introduction and a reference.

[3]Unfortunately this is somewhat orthogonal to the way how CPU resources at large computational infrastructure centers are allocated. In the technical review the focus is on the scaling behavior versus the number of nodes. The worse the single-node performance, the easier it is to make the node-scaling graph look nice.

explicit unrolling of color and spinor operations inside the loop. This turns out to be sufficient for reaching reasonable performance figures on many integrated core (MIC) architectures, such as the "knights landing" (KNL) chip by Intel which has up to 68 physical cores. A key feature of this processor (and more modern successors) is the ability to use the new intrinsic instruction set AVX-512. Accordingly, excellent code performance hinges on the ability to organize fused multiply-add (FMA) operations on 512 bit long[4] data sets. Hence, the primary goal of this article is to explore whether this challenging task can be left to the compiler, if all relevant information (e.g. the values N_c, N_v and N_x, N_y, N_z, N_t) is given at compile time.

The code presented in this article handles three choices of the fermion discretization in lattice QCD. The Wilson definition $D_{\rm W}$ [1], and the staggered definition $D_{\rm S}$ [2,3] are well known, with publicly available codes (see e.g. Refs. [8–15]). The Brillouin definition $D_{\rm B}$ [16–18] is less popular – in part since there is no publicly available implementation with full[5] documentation. In this article each of these formulations is implemented for several vector layouts (see below for a detailed specification), and the CG and BiCGstab inverter routines (which are part of the ancillary code distribution) are written in a completely generic manner. The author hopes that this will enable PhD students and young postdocs to write their own QCD code (e.g. for hadron spectroscopy) with manageable effort, and give them a handle to study their field of interest with minimal human constraints and/or dependencies.

This brief exposition of the subject cannot do justice to the effort spent by other authors to maximize performance on a specific architecture for a given Dirac operator D. Recent review talks on the interplay between algorithms and machines in lattice QCD include [19–23]. In addition, there is a number of HPC projects in lattice QCD with similar objectives on several architectures [24–35]. Preliminary accounts[6] of this work were given in [36,37]. All performance measurements were done on three machines at Jülich Supercomputing Centre. The KNL figures were obtained[7] on DEEP-knl (booted in flat mode) and the JURECA-booster (booted in cache mode), but the performance difference was marginal. The results for the Skylake architecture were obtained on DEEP-dam and on JUWELS.

The remainder of this article is organized as follows. The coding guidelines, and the options for the internal degrees of freedom in the vectors b, u, v are discussed in Sec. 2. A comparison of the vector layouts for the task of computing vector norms, dot-products, and vectorial multiply-adds (all employing SIMD pragmas) is found in Sec. 3. The coding of the clover term (which is used in conjunction with $D_{\rm W}$ and $D_{\rm B}$) is explained Sec. 4. The implementations of the Wilson Laplacian $\triangle^{\rm std}$ and Dirac operator $D_{\rm W}$ are specified in Sec. 5. The details for the Brillouin Laplacian $\triangle^{\rm bri}$ and Dirac operator $D_{\rm B}$ are arranged in Sec. 6. Analogous reasonings and timings for the staggered Dirac operator $D_{\rm S}$ are found in Sec. 7. In addition, it is interesting to study the performance as a function of the compile-time parameters N_c, N_v, and $N_x N_y N_z N_t$; such results are assembled in Sec. 8. The CG and BiCGstab inverters for all five operators (working with any ordering of the internal degrees of freedom) are presented Sec. 9. Finally, in Sec. 10 a summary is attempted. All technical issues are relegated to appendices A-E, a guide to the ancillary code distribution is given in the Supplementary Material.

[4] The 64 bytes amount to SIMD pipelines handling 16 real_sp numbers (equivalently 8 real_dp, 8 complex_sp, or 4 complex_dp numbers) simultaneously.

[5] At https://github.com/g-koutsou/qpb there is an undocumented C++ implementation by Giannis Koutsou.

[6] The attentive reader will notice that performance figures increased quite a bit since these early accounts.

[7] In flat mode `numactl --preferred 1 ./testknl_main` is used, in cache mode `./testknl_main` suffices.

2 Coding guidelines and vector layouts

The code to be presented is written in Fortran 2008, which is an excellent choice for scientific problems with static data structures. Lattice QCD is in this category, and after declaring the number of colors and the box size via the compile time parameters

```
integer,parameter :: Nc= 3, Nv=Nc*4             !!! note: number of colors and rhs
integer,parameter :: Nx=34, Ny=Nx,Nz=Ny,Nt=2*Nz !!! note: box size for T=0 physics
```

an object like the gauge field $U_\mu(n)$, with $\mu \in \{1,\ldots,4\}$ and $n = (x, y, z, t)$ the position in discrete four-dimensional space-time, is conveniently defined as a rank-seven array

```
complex(kind=sp),dimension(Nc,Nc,4,Nx,Ny,Nz,Nt) :: U
```

with the intrinsic complex data type. Objects defined in "single precision" (sp) use four bytes per real component, those in "double precision" (dp) use eight bytes per real component.

It is important to keep in mind that Fortran uses the "column major" convention for matrices and arrays; in `U` the first color index (from $1 \ldots N_c$) is the fastest moving index, while the fourth space-time coordinate (from $1 \ldots N_t$) is the slowest moving index. In a nested set of loops

```
do t=1,Nt
do z=1,Nz
do y=1,Ny
do x=1,Nx
   do mu=1,4
      U(:,:,mu,x,y,z,t)=float(mu)*eye(:,:)
   end do
end do
end do
end do
end do
```

the t-coordinate must thus be in the outermost loop, followed by the z, y, x coordinates, the direction index μ, the column and row indices in color space, to ensure that the elements of U are addressed in the order in which they lie in memory. In this example we use the stride notation to access a contiguous patch of memory through two implied do loops (with correct ordering built-in); the N_c^2 complex unit long patch for the matrix $U_\mu(n)$ is overwritten by μI where I is the $N_c \times N_c$ identity matrix in color space, if `eye(:,:)` was defined accordingly.

Furthermore, this example illustrates our decision to avoid explicit site-index computations in the code. One might have used `dimension(Nc,Nc,4,Nx*Ny*Nz*Nt)` in the definition of `U`, and an extra line `n=(((t-1)*Nz+(z-1))*Ny+(y-1))*Nx+x` ahead of the loop over `mu`, along with `U(:,:,mu,n)=float(mu)*eye(:,:)`. But this is potentially error prone (especially in view of a future MPI-spreading of up to four space-time dimensions over several nodes), and it seems more elegant to leave all index computations to the compiler.

Given our goal of combining N_v column vectors into one object [for D being the staggered operator $D_{\rm S}$ the left-hand side of Eq. (1) represents an $n \times n$ matrix which acts on a $n \times N_v$ matrix with $n \equiv N_c N_x N_y N_z N_t$], it seems natural to define a staggered multi-RHS vector through

```
complex(kind=sp),dimension(Nc,Nx,Ny,Nz,Nt,Nv) :: suv_sp
```

but this would imply a detrimental loop-ordering in the staggered Dirac routine (cf. Sec. 7). For good performance it is crucial to promote the RHS-index to an *internal degree of freedom* (like color) which is *ahead* of all space-time indices. This leaves us with the two options

```
complex(kind=sp),dimension(Nc,Nv,Nx,Ny,Nz,Nt) :: suv_sp !!! [Nc,Nv]=layout1
complex(kind=sp),dimension(Nv,Nc,Nx,Ny,Nz,Nt) :: suv_sp !!! [Nv,Nc]=layout2
```

for a "staggered utility vector", and we shall implement both of them and compare the respective timings. For $D_{\rm W}, D_{\rm B}$ the vectors have yet another internal degree of freedom (called "spinor", ranging from 1 to 4), and this implies that we should consider the six options

```
complex(kind=sp),dimension(Nc,04,Nv,Nx,Ny,Nz,Nt) :: vec_sp !!! [Nc,Ns,Nv]=layout1
complex(kind=sp),dimension(04,Nc,Nv,Nx,Ny,Nz,Nt) :: vec_sp !!! [Ns,Nc,Nv]=layout2
complex(kind=sp),dimension(Nc,Nv,04,Nx,Ny,Nz,Nt) :: vec_sp !!! [Nc,Nv,Ns]=layout3
complex(kind=sp),dimension(Nv,Nc,04,Nx,Ny,Nz,Nt) :: vec_sp !!! [Nv,Nc,Ns]=layout4
complex(kind=sp),dimension(04,Nv,Nc,Nx,Ny,Nz,Nt) :: vec_sp !!! [Ns,Nv,Nc]=layout5
complex(kind=sp),dimension(Nv,04,Nc,Nx,Ny,Nz,Nt) :: vec_sp !!! [Nv,Ns,Nc]=layout6
```

for vectors to be used in conjunction with Wilson-type Dirac matrices (for $D_{\rm W}$ see Sec. 5, for $D_{\rm B}$ see Sec. 6), and compare the respective timings.

In Sec. 9 it will be demonstrated that with u, b in the class `suv_sp` or `vec_sp` a tolerance $\epsilon = 10^{-12}$, where $\epsilon \equiv ||Dx - b||/||b||$, cannot be reached. Such relative residual norms tend to stagnate at $\epsilon \sim 10^{-6}$, as is typical for attempts to solve Eq. (1) in sp. In practice, this means that, in order to reach a relative residual norm $\epsilon \ll 10^{-6}$, in addition to $v_{\rm sp} \leftarrow D_{\rm sp} u_{\rm sp}$ also the operation $v_{\rm dp} \leftarrow D_{\rm dp} u_{\rm dp}$ must be implemented, and a dp-solver must be used. In other words, the code needs the ability to allocate objects of the classes `suv_dp` or `vec_dp`, with the same two or six options for the layout of the internal indices, and to feed them to routines which code the left-multiplication with $D_{\rm S}$, $D_{\rm W}$, or $D_{\rm B}$ in dp.

A peculiar feature of this lattice QCD code is that the gauge field U is always in sp, in order to be conservative on using disk space and memory bandwidth. Hence, even in the dp-version of the matrix-vector multiplication routine, the Dirac operator depends on a gauge field which is defined in sp, i.e. $v_{\rm dp} \leftarrow D_{\rm dp}[U_{\rm sp}] u_{\rm dp}$. In Sec. 9 it will be shown that this is perfectly sufficient[8] to reach relative residual norms as small as $\epsilon = 10^{-12}$.

Another feature is that the Dirac operators need not necessarily use the original ("thin link") gauge field $U_\mu(n)$, but may use a smeared ("fat link") gauge background which is referred to as $V_\mu(n)$ in this article. Specifically a stout-smearing routine [38] is included[9] in the module `testknl_util.f90`, with parameters $\rho_{\rm stout} = 0.12, n_{\rm stout} = 3$ as default[10].

In summary, the goal of this article is to find out whether one can use a high-level language such as Fortran 2008 to write a lattice QCD code which performs well on modern many-core architectures. By construction such a code is fully portable, but the decision to stay away from assembly-tuning and cache-line optimization means that all the burden is on the compiler. Our strategy is to chose a suitable data layout, and to let the compiler know anything which eases the task of creating fast-performing code. For a modern compiler it is ideal if the trip counts

[8] This should not come as a surprise; one might think of the routine $v_{\rm dp} \leftarrow D_{\rm dp}[U_{\rm dp}] u_{\rm dp}$ as a routine which operates on a "copy" $U_{\rm dp}$ where all significant digits which are not present in $U_{\rm sp}$ are set to zero.

[9] An overview of the ancillary code distribution is given in the Supplementary Material.

[10] The user who prefers an unsmeared gauge field should set $n_{\rm stout} = 0$; this will establish $V_\mu(n) = U_\mu(n)$.

and the array extents are statically[11] known (via `parameter` in Fortran or `#define` in C). In the remainder of this paper we will explore whether this approach works in practice.

3 Norms, dot-products and multiply-adds

A Krylov-space solver such as CG and BiCGstab is designed to solve Eq. (1) via an iterative process [4–6]. The actual computer implementation (to be discussed in Sec. 9 below) involves two ingredients. On the one hand, a fast matrix-vector multiplication routine for the chosen operator $D \in \{D_{\mathrm{W}}, D_{\mathrm{B}}, D_{\mathrm{S}}\}$ on a given (possibly smeared) gauge background $V_\mu(n)$ is required. On the other hand, some linear algebra routines are needed, in particular one which determines the squared 2-norm $||v||_2^2$, and the dot-product[12] $\langle u, v\rangle$ between two vectors. The matrix-vector routines for the three Dirac operators will be presented in Secs. 5, 6 and 7 (along with timings). Here we provide similar information for the linear algebra routines mentioned.

For the first staggered layout (locally `NcNv`) the norm-square routine takes the form

```
function suv_normsqu_NcNv_sp(suv)
   implicit none
   complex(kind=sp),dimension(Nc,Nv,Nx,Ny,Nz,Nt),intent(in) :: suv
   real(kind=sp),dimension(Nv) :: suv_normsqu_NcNv_sp
   real(kind=dp),dimension(Nv) :: res !!! note: accumulation variable in "dp"
   integer :: x,y,z,t,rhs
   res(:)=0.0_dp
   !$OMP PARALLEL DO COLLAPSE(2) REDUCTION(+:res) SHARED(suv) ...
   do t=1,Nt
   do z=1,Nz
   do y=1,Ny
   do x=1,Nx
      !$OMP SIMD
      do rhs=1,Nv
         res(rhs)=res(rhs)+sum(myabssqu_sp(suv(:,rhs,x,y,z,t)))
      end do
   end do
   end do
   end do
   end do
   !$OMP END PARALLEL DO
   suv_normsqu_NcNv_sp=real(res,kind=sp)
end function suv_normsqu_NcNv_sp
```

where the `pure elemental` function `myabssqu_sp` returns $|z|^2$ in dp (for $z \in \mathbf{C}$ in sp), the relevant line being `myabssqu_sp=dble(real(z)**2)+dble(aimag(z)**2)`. This piece of code

[11] With such knowledge in hand, more informed vectorization cost-model decisions (whether or not to peel, unroll factors, etc.) are possible. Also compiler-based analysis of alignment for vectorization becomes more effective, and prefetch distances chosen by the compiler tend to be more adequate. Finally, the compiler is able to do more efficient outer-loop optimizations, for instance partial redundancy elimination (PRE) for address calculations, and partial dead store elimination (PDSE).

[12] Throughout the article we use the physics convention where $\langle u, v\rangle \in \mathbf{C}$ is linear in the *second* argument.

	single precision		double precision	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
NcNsNv	0.0090	0.0091	0.0174	0.0183
NsNcNv	0.0088	0.0091	0.0173	0.0183
NcNvNs	0.0089	0.0091	0.0174	0.0183
NvNcNs	0.0109	0.0092	0.0173	0.0182
NsNvNc	0.0188	0.0173	0.0177	0.0183
NvNsNc	0.0104	0.0093	0.0174	0.0182
NcNv	0.0045	0.0040	0.0044	0.0046
NvNc	0.0028	0.0023	0.0044	0.0046

Table 1: Time in seconds to compute all N_v norms of a multi-RHS vector on the 68-core KNL architecture with the array allocated in MCDRAM. The upper part is for Wilson-type vectors (spinor and color structure), the lower part for Susskind-type vectors (color structure only). The lattice size is $34^3 \times 68$ with parameters $N_c = 3$, $N_v = 12$. The rows refer to the array layout, the columns to the array precision and the number of OpenMP threads used. Regardless of the vector precision, the accumulation variable of the N_v norms is in double precision.

	single precision			double precision		
	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$
NcNsNv	0.0162	0.0134	0.0136	0.0278	0.0267	0.0270
NsNcNv	0.0163	0.0134	0.0135	0.0278	0.0266	0.0270
NcNvNs	0.0165	0.0135	0.0136	0.0277	0.0267	0.0270
NvNcNs	0.0172	0.0135	0.0135	0.0279	0.0266	0.0269
NsNvNc	0.0189	0.0136	0.0135	0.0285	0.0266	0.0269
NvNsNc	0.0172	0.0135	0.0135	0.0280	0.0267	0.0269
NcNv	0.0045	0.0035	0.0033	0.0072	0.0067	0.0069
NvNc	0.0042	0.0033	0.0033	0.0071	0.0068	0.0069

Table 2: Same as Tab. 1, but on the 2×24-core (dual socket) Skylake architecture.

illustrates important principles of the ancillary code distribution. Besides the (correctly ordered) loops over the space-time indices `x,y,z,t` there is a sum over color, and the loop over the RHS-index `rhs` is equipped with a pragma that instructs the compiler to use it for filling the SIMD pipeline. The construct `!$OMP PARALLEL DO` lets the compiler organize the work share among the $N_{\rm thr}$ threads which are spanned by the z and t loops, due to the clause `COLLAPSE(2)`. The variable `suv` is shared, and the clause `REDUCTION(+:res)` means that the results accumulated by individual threads are combined into the variable `res`. It is worth pointing out that all accumulation is done in dp, despite the input and output variables being in sp.

Such a routine needs to be written for each internal index ordering, i.e. two sp-routines for Susskind-type vectors and six for Wilson-type vectors. In addition, a slightly modified version of these eight routines needs to be provided for input/output variables in dp (here the accumulation variable is still in dp, not in quadruple precision). Overall, sixteen routines want to be tested, for various values of $N_{\rm thr}$. Depending on the hyperthreading capabilities of the architecture, the values are $N_{\rm thr} \in \{2N_{\rm hw}, 4N_{\rm hw}\}$ for four threads per hardware core (e.g. the KNL chip), or $N_{\rm thr} \in \{N_{\rm hw}, 2N_{\rm hw}\}$ for two threads per hardware core (e.g. the Skylake chip).

	single precision		double precision	
	$N_{\text{thr}} = 136$	$N_{\text{thr}} = 272$	$N_{\text{thr}} = 136$	$N_{\text{thr}} = 272$
`NcNsNv`	0.0392	0.0302	0.0575	0.0504
`NsNcNv`	0.0268	0.0277	0.0448	0.0500
`NcNvNs`	0.0296	0.0240	0.0426	0.0465
`NvNcNs`	0.0214	0.0221	0.0424	0.0445
`NsNvNc`	0.0214	0.0228	0.0433	0.0456
`NvNsNc`	0.0215	0.0221	0.0425	0.0447
`NcNv`	0.0068	0.0055	0.0107	0.0111
`NvNc`	0.0054	0.0057	0.0107	0.0112

Table 3: Time in seconds to perform all N_v increment operations $u^{(i)} = u^{(i)} + v^{(i)}\alpha^{(i)}$ with $\alpha^{(i)} \in \mathbf{C}$ and $i = 1, \ldots, N_v$ on the 68-core KNL architecture with the multi-RHS vectors u, v allocated in MCDRAM. The lattice size is $34^3 \times 68$ with parameters $N_c = 3$, $N_v = 12$.

	single precision			double precision		
	$N_{\text{thr}} = 24$	$N_{\text{thr}} = 48$	$N_{\text{thr}} = 96$	$N_{\text{thr}} = 24$	$N_{\text{thr}} = 48$	$N_{\text{thr}} = 96$
`NcNsNv`	0.0584	0.0484	0.0475	0.1399	0.1087	0.1034
`NsNcNv`	0.0507	0.0463	0.0479	0.1392	0.1091	0.1033
`NcNvNs`	0.0468	0.0466	0.0450	0.0932	0.0902	0.0906
`NvNcNs`	0.0471	0.0469	0.0451	0.0934	0.0925	0.0903
`NsNvNc`	0.0463	0.0457	0.0452	0.0943	0.0901	0.0905
`NvNsNc`	0.0470	0.0465	0.0451	0.0936	0.0930	0.0903
`NcNv`	0.0119	0.0115	0.0116	0.0238	0.0234	0.0233
`NvNc`	0.0118	0.0116	0.0115	0.0237	0.0231	0.0232

Table 4: Same as Tab. 3, but on the 2×24-core (dual socket) Skylake architecture.

The KNL data were obtained on a single-chip node with $N_{\text{hw}} = 68$ physical cores, the Skylake data on a dual-socket node with a total of $N_{\text{hw}} = 2 \times 24 = 48$ physical cores.

The timings of the routines {`vec`,`suv`}`_normsqu_`{`sp`,`dp`} are listed in Tabs. 1, 2 for the KNL and Skylake architectures, respectively. The figures give the time needed to compute all $N_v = 12$ norm-squares with $N_c = 3$ colors on a lattice with 2 672 672 sites. A peculiarity of the KNL architecture is that the lattice fits into the MCDRAM, and the vector is initialized with the same number of threads. This "first touch" policy is used in all subsequent timings. On the KNL the MCDRAM high-bandwidth memory has an aggregate bandwidth[13] of about 450 GB/s. In sp the N_v (complex) staggered vectors occupy $8N_cN_vN_xN_yN_zN_t = 769\,729\,536$ bytes in memory, hence transferring them to the registers takes 0.0017 s (assuming zero latency). In dp the bandwidth limit amounts to 0.0034 s, and for Wilson-type vectors these figures are four-fold increased to 0.0068 s and 0.0136 s, respectively. Comparing the actual entries in Tab. 1 to these lower bounds, we see that most of the timings are reasonably close to it, only the staggered `NcNv` layout in sp takes considerably longer. Regarding two-fold versus four-fold hyperthreading, there is no universal law on the KNL architecture; sometimes one option is faster, sometimes the other. The timings in Tab. 2 for the Skylake architecture are generally

[13] See e.g. `https://colfaxresearch.com/knl-mcdram`.

	single precision		double precision	
	$N_{\text{thr}} = 136$	$N_{\text{thr}} = 272$	$N_{\text{thr}} = 136$	$N_{\text{thr}} = 272$
`NcNsNv`	0.0162	0.0174	0.0241	0.0253
`NsNcNv`	0.0162	0.0176	0.0322	0.0346
`NcNvNs`	0.0143	0.0114	0.0205	0.0246
`NvNcNs`	0.0101	0.0105	0.0200	0.0205
`NsNvNc`	0.0162	0.0175	0.0322	0.0346
`NvNsNc`	0.0112	0.0120	0.0219	0.0233
`NcNv`	0.0027	0.0030	0.0050	0.0051
`NvNc`	0.0026	0.0027	0.0048	0.0049

Table 5: Time in seconds to apply a γ-matrix to all N_v columns of a multi-RHS vector on the 68-core KNL architecture with the array allocated in MCDRAM. The upper part is for γ_5 and Wilson-type vectors (spinor and color structure), the lower part for ϵ and Susskind-type vectors (color structure only). The lattice size is $34^3 \times 68$ with parameters $N_c = 3$, $N_v = 12$.

	single precision			double precision		
	$N_{\text{thr}} = 24$	$N_{\text{thr}} = 48$	$N_{\text{thr}} = 96$	$N_{\text{thr}} = 24$	$N_{\text{thr}} = 48$	$N_{\text{thr}} = 96$
`NcNsNv`	0.0326	0.0328	0.0325	0.0525	0.0505	0.0514
`NsNcNv`	0.0327	0.0330	0.0324	0.0658	0.0657	0.0651
`NcNvNs`	0.0226	0.0208	0.0205	0.0455	0.0429	0.0416
`NvNcNs`	0.0210	0.0205	0.0197	0.0454	0.0431	0.0414
`NsNvNc`	0.0326	0.0330	0.0325	0.0658	0.0658	0.0652
`NvNsNc`	0.0223	0.0217	0.0210	0.0470	0.0445	0.0435
`NcNv`	0.0061	0.0056	0.0055	0.0110	0.0104	0.0102
`NvNc`	0.0055	0.0055	0.0053	0.0109	0.0105	0.0102

Table 6: Same as Tab. 5, but on the 2×24-core (dual socket) Skylake architecture.

slower, but not dramatically so. On this architecture the ordering of the internal indices seems irrelevant, and the norm-square operation does not benefit from two-fold hyperthreading.

Changing the task from computing the squared norm $||v||_2^2$ to computing the dot-product $\langle u, v\rangle$ will double both the memory traffic, and the flop count, since $r = r + \text{Re}^2(v) + \text{Im}^2(v)$ takes four flops, while $r = r + \text{Re}(u)\text{Re}(v) - \text{Im}(u)\text{Im}(v), i = i + \text{Re}(u)\text{Im}(v) - \text{Im}(u)\text{Re}(v)$ takes eight flops. We thus expect that all timings (regardless of precision, and architecture) will double. A quick test reveals this is precisely what happens (tables not included).

Another important ingredient in an iterative solver is the vectorial multiply-add operation. Instead of the generic $w^{(i)} \leftarrow u^{(i)} + v^{(i)}\alpha^{(i)}$ for $i \in \{1, \ldots, N_v\}$, which is more demanding on memory bandwidth, we implement two multiply-add routines with overwrite

$$\texttt{incr}: \quad u^{(i)} \leftarrow u^{(i)} + v^{(i)}\alpha^{(i)} \qquad (i = 1, \ldots, N_v) \tag{2}$$

$$\texttt{anti}: \quad v^{(i)} \leftarrow u^{(i)} + v^{(i)}\alpha^{(i)} \qquad (i = 1, \ldots, N_v) \tag{3}$$

for $\alpha^{(i)}$ in $\mathbf{R}$ or $\mathbf{C}$. For the precision of the (complex) vectors there are three possibilities: (i) u, v both in sp, (ii) u, v both in dp, and (iii) u in dp and v in sp (relevant to the mixed-precision solvers mentioned in Sec. 9). Furthermore, each of the N_v vectors may have spinor degrees of freedom (length $4N_cN_xN_xN_zN_t$) or not (length $N_cN_xN_xN_zN_t$). Hence, the names of these

routines are {vec,suv}_{incr,anti}_{r,c}{sp,dp}, and the "r" or "c" indicates whether the $\alpha^{(i)}$ are real or complex. The respective timings on the KNL architecture are summarized in Tab. 3. In sp the layouts with the color degree of freedom first (i.e. NcNsNv and NcNvNs) take a little longer than the remaining four, and for the latter ones four-fold hyperthreading does not seem to bring any advantage over two-fold hyperthreading. Quite generally, there is factor two difference between sp-timings and dp-timings of (2) and (3). Analogous timing results on the Skylake architecture are listed in Tab. 4. In this case using one socket ($N_{\rm thr} = 24$) or both sockets without ($N_{\rm thr} = 48$) or with ($N_{\rm thr} = 96$) hyperthreading yields similar figures.

Another routine that proves relevant below is the left-multiplication with γ_5 for Wilson-type vectors and $\epsilon \equiv (-1)^{x_1+x_2+x_3+x_4} \doteq \gamma_5 \otimes \xi_5$ for Susskind-type vectors. Such results are collected in Tab. 5 for the KNL, and in Tab. 6 for the Skylake architecture. Once more, we find that the vector layout has a mild effect on the actual timing on the KNL, and essentially no effect on the Skylake node. And the effect of hyperthreading is negligible on both architectures.

Overall, the chosen vector layout (i.e. the order of the internal indices color/spinor/RHS) affects the timings of the linear algebra routines by just a few percent. As we shall see below, for the Wilson and Susskind operators the matrix-vector operation is about an order of magnitude more expensive than the linear algebra operations. And the Brillouin operator is almost two orders of magnitude more time consuming. In view of these forthcoming results, it is fair to say that the linear algebra routines have been optimized to the point where their CPU share is a subdominant part of the overall solver time (cf. Sec. 9 below).

4 Clover routine

The clover routine is a matrix-vector routine which is applied[14] in addition to the Wilson or Brillouin Dirac operator D. The operation is $v \leftarrow (D + C)u$ with $D \in \{D_{\rm W}, D_{\rm B}\}$ and

$$C(n,m) = -\frac{c_{\rm SW}}{2} \sum_{\mu<\nu} \sigma_{\mu\nu} F_{\mu\nu}(n) \delta_{n,m} \tag{4}$$

where n and m are positions in the four-dimensional lattice. It acts on the vector v non-trivially in color and spinor space, but as an identity in RHS space and position space. In other words, it acts locally in space-time, so $v(n)$ depends on $u(m)$ only for $n = m$. Depending on the layout (i.e. the ordering of the color/spinor/RHS indices) Eq. (4) is thus a shorthand for

$$C(n,m) = -\frac{c_{\rm SW}}{2} \sum_{\mu<\nu} \delta_{n,m} \otimes \begin{cases} I \otimes \sigma_{\mu\nu} \otimes F_{\mu\nu}(n) & \text{for} \quad \texttt{NcNsNv} \\ I \otimes F_{\mu\nu}(n) \otimes \sigma_{\mu\nu} & \text{for} \quad \texttt{NsNcNv} \\ \sigma_{\mu\nu} \otimes I \otimes F_{\mu\nu}(n) & \text{for} \quad \texttt{NcNvNs} \\ \sigma_{\mu\nu} \otimes F_{\mu\nu}(n) \otimes I & \text{for} \quad \texttt{NvNcNs} \\ F_{\mu\nu}(n) \otimes I \otimes \sigma_{\mu\nu} & \text{for} \quad \texttt{NsNvNc} \\ F_{\mu\nu}(n) \otimes \sigma_{\mu\nu} \otimes I & \text{for} \quad \texttt{NvNsNc} \end{cases} \tag{5}$$

where the sum[15] is over the six plaquette orientations with $\mu < \nu$. For each orientation the 4×4 matrix $\sigma_{\mu\nu} \equiv \frac{\rm i}{2}[\gamma_\mu, \gamma_\nu]$ is hermitean in spinor space, and the clover-leaf field-strength

[14] With $c_{\rm SW} = 0$ both $D_{\rm W}$ and $D_{\rm B}$ induce cut-off effects $O(a)$. With the tree-level value $c_{\rm SW} = 1$ the latter are mitigated to $O(\alpha a)$, and with the one-loop value to $O(\alpha^2 a)$, where $\alpha = g^2_{\rm ren}/(4\pi)$ is the strong coupling constant. With a non-perturbatively determined value $c^{\rm NP}_{\rm SW}$ cut-off effects can be lifted to $O(a^2)$ [39–41].

[15] Since $\sigma_{\mu\nu}$ and $F_{\mu\nu}$ are both anti-symmetric in $\mu \leftrightarrow \nu$, the sum may be written without the constraint among μ, ν, but with a prefactor $c_{\rm SW}/4$ instead of $c_{\rm SW}/2$.

	single precision		double precision	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
`NcNsNv`	0.0426	0.0375	0.0737	0.0642
`NsNcNv`	0.0424	0.0375	0.0728	0.0635
`NcNvNs`	0.0461	0.0447	0.0640	0.0595
`NvNcNs`	0.0349	0.0296	0.0617	0.0570
`NsNvNc`	0.0435	0.0370	0.0634	0.0589
`NvNsNc`	0.0340	0.0291	0.0547	0.0559

Table 7: Time in seconds to apply the clover term (4) to all N_v columns of a multi-RHS vector on the 68-core KNL architecture with the vector and the precomputed field-strength allocated in MCDRAM. The lattice size is $34^3 \times 68$ with parameters $N_c = 3$, $N_v = 12$. The best timings correspond to 2500 Gflop/s in sp, and 1300 Gflop/s in dp – see App. E for details.

	single precision			double precision		
	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$
`NcNsNv`	0.0606	0.0519	0.0519	0.1222	0.1004	0.0979
`NsNcNv`	0.0612	0.0519	0.0518	0.1212	0.0994	0.0982
`NcNvNs`	0.0621	0.0523	0.0519	0.1137	0.0977	0.0973
`NvNcNs`	0.0560	0.0516	0.0520	0.1148	0.0986	0.0975
`NsNvNc`	0.0605	0.0520	0.0521	0.1140	0.0971	0.0974
`NvNsNc`	0.0560	0.0513	0.0520	0.1168	0.0991	0.0976

Table 8: Same as Tab. 7, but on the 2×24-core (dual socket) Skylake architecture. The best timings correspond to 1450 Gflop/s in sp, and 750 Gflop/s in dp.

operator $F_{\mu\nu}(n)$ is hermitean in color space. In consequence the clover term (4) is a hermitean contribution to the combined matrix-vector operation $v \leftarrow (D + C)u$.

The field-strength operator $F_{\mu\nu}$ is based on the smeared ("fat-link") gauge field $V_\mu(n)$ which derives from the original ("thin-link") gauge field $U_\mu(n)$. Here, any type of smearing may be used; the code uses stout-smearing [38] which produces differentiable links (though this point is not relevant to this article). The field-strength is precomputed in a routine which takes the (possibly smeared) gauge field `V` as input; the result is stored in the rank-seven array

```
complex(kind=sp),dimension(Nc,Nc,6,Nx,Ny,Nz,Nt) :: F
```

since a solver requests dozens to millions of operations $v \leftarrow (D + C)u$ in which U (and thus V) stays unchanged. Since C depends on U only via F, it pays[16] to compute $F_{\mu\nu}$ only once. The six orientations stand for $(\mu, \nu) \in \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\}$.

The clover routine takes the field-strength array `F` as input, as well as the vector `old`, denoted u in the line above (4). The vector `new` at position n, i.e. $v(n)$ above (4), is incremented by $-\frac{1}{2} c_{\rm SW} \sum_{\mu<\nu} \sigma_{\mu\nu} F_{\mu\nu}(n)$ applied to $u(n)$. The operation is thus *site-diagonal*, and this means that the OpenMP thread-parallelization is easily achieved by declaring `SHARED(old,new,F)` in the `!$OMP PARALLEL DO` construct. A nice side-effect is that any read-collision or write-collision among the threads is excluded by construction, since each thread reads from (and writes to) a specified segment in memory, with no overlap among the segments (there is an

[16] With HPC architectures becoming increasingly memory bandwidth limited, this may change in the future.

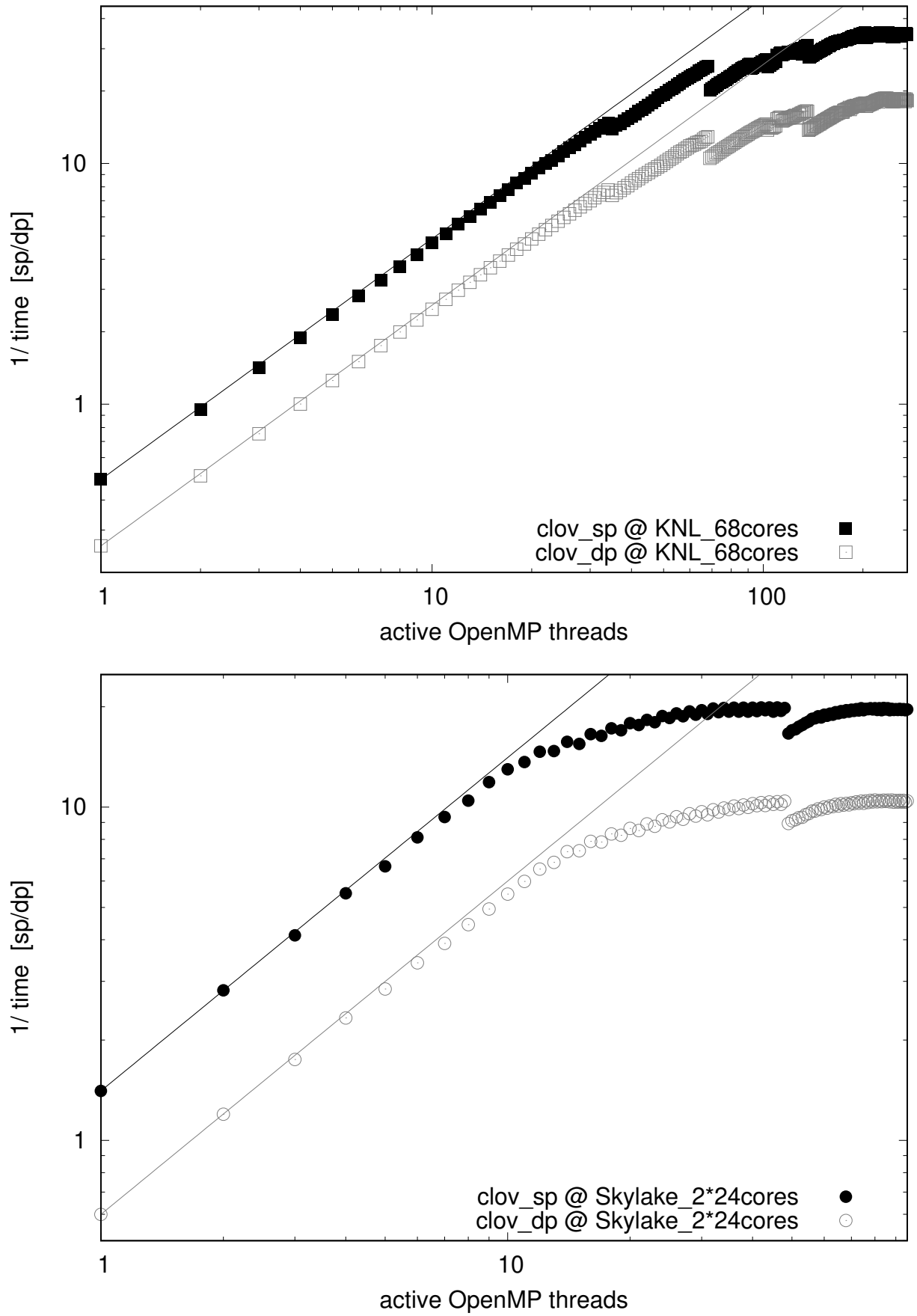


Figure 1: Clover term performance versus the number of active threads in sp and dp, for the KNL and the dual-socket Skylake architectures (same parameters as in Tab. 7).

implied barrier at the end of the `!$OMP PARALLEL DO` construct). The structure of the routine is thus similar to the function `suv_normsqu_sp` displayed in Sec. 3, with four nested do-loops to cover all space-time points. Since all site-index computations are handled by the compiler no large lookup-table is needed. The reader is reminded that the coding must be done for sp and dp vectors separately. Within sp or dp, also for each vector layout a dedicated routine is needed to achieve good performance (with a wrapper routine which presents them as a single routine to the outside world).

The timings of the clover routine (which is specific to Wilson-type vectors) are listed in Tabs. 7, 8 for the KNL and Skylake architectures, respectively. On the KNL, the layouts `NvNcNs` and `NvNsNc` are faster than the remaining four. This is unsurprising, since the SIMD operation, which is over the RHS-index `1:Nv`, should affect the fastest (in Fortran: first) index of the array. Obviously, this performance hierarchy among the six possible layouts will persist in the Wilson-type Dirac operators. In the sections on $D_{\rm W}, D_{\rm B}$ we shall thus restrict ourselves to these two layouts, plus the most naive `NcNsNv` for comparison. On the Skylake architecture, the performance of the clover term is essentially the same for all layout options. On the latter architecture the memory bandwidth is the limiting factor; so the AVX-512 instruction set is pointless for this routine (though it exists on the Skylake architecture).

The scaling of the clover routine (in sp and dp, for the `NvNsNc` layout, with $34^3 \times 68$ volume, and $N_c = 3$, $N_v = 12$) as a function of the number of active threads is shown in Fig. 1. On the KNL architecture the number of threads ranges from 1 to $4N_{\rm cpu} = 272$, where $N_{\rm cpu} = 68$ is the number of physical cores. On the Skylake architecture the number of threads ranges from 1 to $2N_{\rm cpu} = 96$, where $N_{\rm cpu} = 48$ is the number of physical cores in the full dual-socket node. On the KNL chip we find essentially perfect scaling behavior until every physical core hosts one thread. There is a reduced slope associated with the second thread on a physical core (from 69 to 136), and modest improvement with the third (from 137 to 204) and fourth (from 205 to 272) thread on a given core. On the Skylake architecture saturation effects set in at $O(10)$ threads; this is a clear sign of the process being limited by memory bandwidth.

A time of 30 ms for $N_v = 12$ in Tab. 7 amounts to the clover operator being applied to a single vector with $3 \cdot 4 \cdot 34^3 \cdot 68 = 32\,072\,064$ complex elements within 2.5 ms. As we shall see in Secs. 5 and 6, this is about one third of the time needed to apply the Wilson Dirac operator (6), and the Brillouin Dirac operator (11) takes an order of magnitude longer. In short, the clover routine has been optimized to the point where further optimization would speed up the solver routines (to be discussed in Sec. 9 below) only marginally.

5 Wilson Laplace and Dirac routines

For a given $V_\mu(n)$ the Wilson Dirac operator with optional clover term (4) is defined as [1]

$$D_{\rm W}(n,m) = \sum_\mu \gamma_\mu \nabla^{\rm std}_\mu(n,m) - \frac{ar}{2} \triangle^{\rm std}(n,m) + m_0 \delta_{n,m} + aC(n,m) \tag{6}$$

where a is the lattice spacing, $\nabla^{\rm std}_\mu$ is the 2-point discretization of the covariant derivative

$$a\nabla^{\rm std}_\mu(n,m) = \frac{1}{2}\left[V_\mu(n)\delta_{n+\hat\mu,m} - V^\dagger_\mu(n-\hat\mu)\delta_{n-\hat\mu,m}\right] \tag{7}$$

	single precision		double precision	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
`NcNsNv`	0.1387	0.1375	0.2094	0.2162
`NvNcNs`	0.1031	0.0973	0.1808	0.1727
`NvNsNc`	0.0749	0.0745	0.1485	0.1602

Table 9: Time in seconds per matrix times multi-RHS vector operation for the Wilson Laplace operator (8) on the 68-core KNL architecture, with all variables allocated in MCDRAM. The lattice size is $34^3 \times 68$ with parameters $N_c = 3$, $N_v = 12$. The best timings correspond to 1000 Gflop/s in sp, and 500 Gflop/s in dp – see App. E for details.

	single precision			double precision		
	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$
`NcNsNv`	0.1366	0.1148	0.1197	0.2581	0.2215	0.2450
`NvNcNs`	0.1262	0.1141	0.1190	0.2579	0.2216	0.2451
`NvNsNc`	0.1221	0.1141	0.1192	0.2390	0.2200	0.2443

Table 10: Same as Tab. 9, but on the 2×24-core (dual socket) Skylake architecture. The best timings correspond to 650 Gflop/s in sp, and 340 Gflop/s in dp.

and $\triangle^{\rm std}$ is the 9-point discretization of the covariant Laplacian

$$a^2 \triangle^{\rm std}(n,m) = -\,8\,\delta_{n,m} + \sum_\mu \left[V_\mu(n)\delta_{n+\hat\mu,m} + V_\mu^\dagger(n-\hat\mu)\delta_{n-\hat\mu,m} \right] . \tag{8}$$

The sums in (6, 8) extend over the positive Euclidean directions $\mu \in \{1,\ldots,4\}$, and the bare quark mass m_0 undergoes both additive and multiplicative renormalization. How the V-links in $\nabla_\mu^{\rm std}, \triangle^{\rm std}$ and C relate to the original U-links has been explained in Sec. 4. Note that the "standard derivative" (7) is anti-hermitean, while the "standard/Wilson Laplacian" 8 and the clover term are hermitean operators. The species-lifting parameter is typically set to $r = 1$, and the operator (6) is HPC friendly, since its stencil contains at most 1-hop terms.

The action of the Wilson operator (6) at $r = 1, c_{\rm SW} = 0$ on a Dirac vector ψ (spinor⊗color internal degrees of freedom) with periodic boundary conditions in all directions is given by

$$(D_{\rm W}\psi)(n) = \frac{1}{2}\sum_\mu \big\{ [(\gamma_\mu - 1)\otimes V_\mu(n)]\psi(n+\hat\mu) - [(\gamma_\mu+1)\otimes V_\mu^\dagger(n-\hat\mu)]\psi(n-\hat\mu) \big\} + (4+m_0)\psi(n) \tag{9}$$

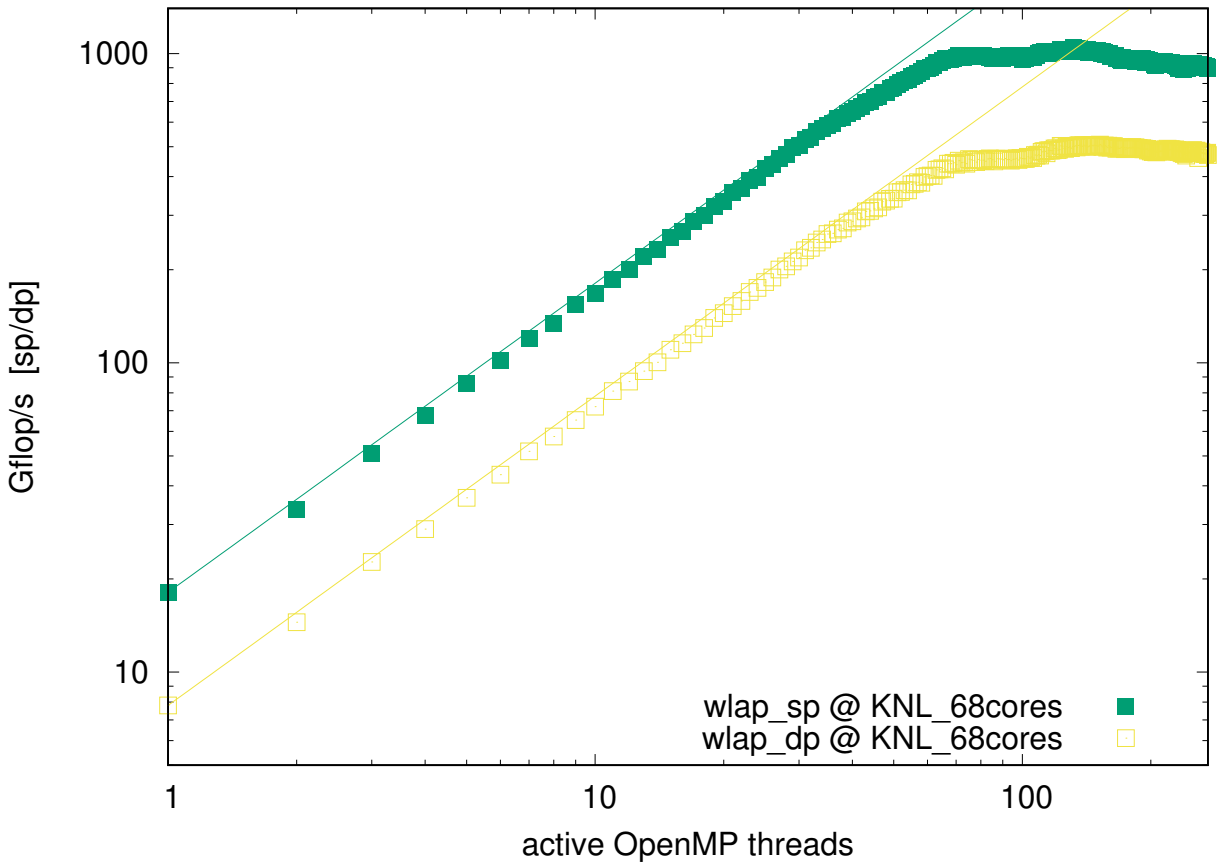
and our task is to implement a routine which performs this operation efficiently. The action of the embedded Laplace operator (8) alone (which we implement for comparison) is

$$\Big(-\frac{1}{2}\triangle^{\rm std} + \frac{m_0^2}{2} \Big)\psi(n) = \frac{1}{2}\sum_\mu \big[-V_\mu(n)\psi(n+\hat\mu) - V_\mu^\dagger(n-\hat\mu)\psi(n-\hat\mu) \big] + (4 + \frac{m_0^2}{2})\psi(n) \tag{10}$$

with the mass parameter m_0 replaced by $m_0^2/2$, in line with the standard boson propagator in quantum field theory. In (9) and (10) we have taken the liberty to set the lattice spacing $a = 1$.

From a coding viewpoint it is clear that the Wilson Laplace operator (10) is easier to implement than the Wilson Dirac operator (9), since it acts trivially[17] in spinor space, and we

[17]Hence one might let $\triangle^{\rm std}$ act on a Susskind-type vector (color structure only), and the ancillary code distribution contains such a multiplication routine under the label `app_wsuv_{sp,dp}`, see Supplementary Material.



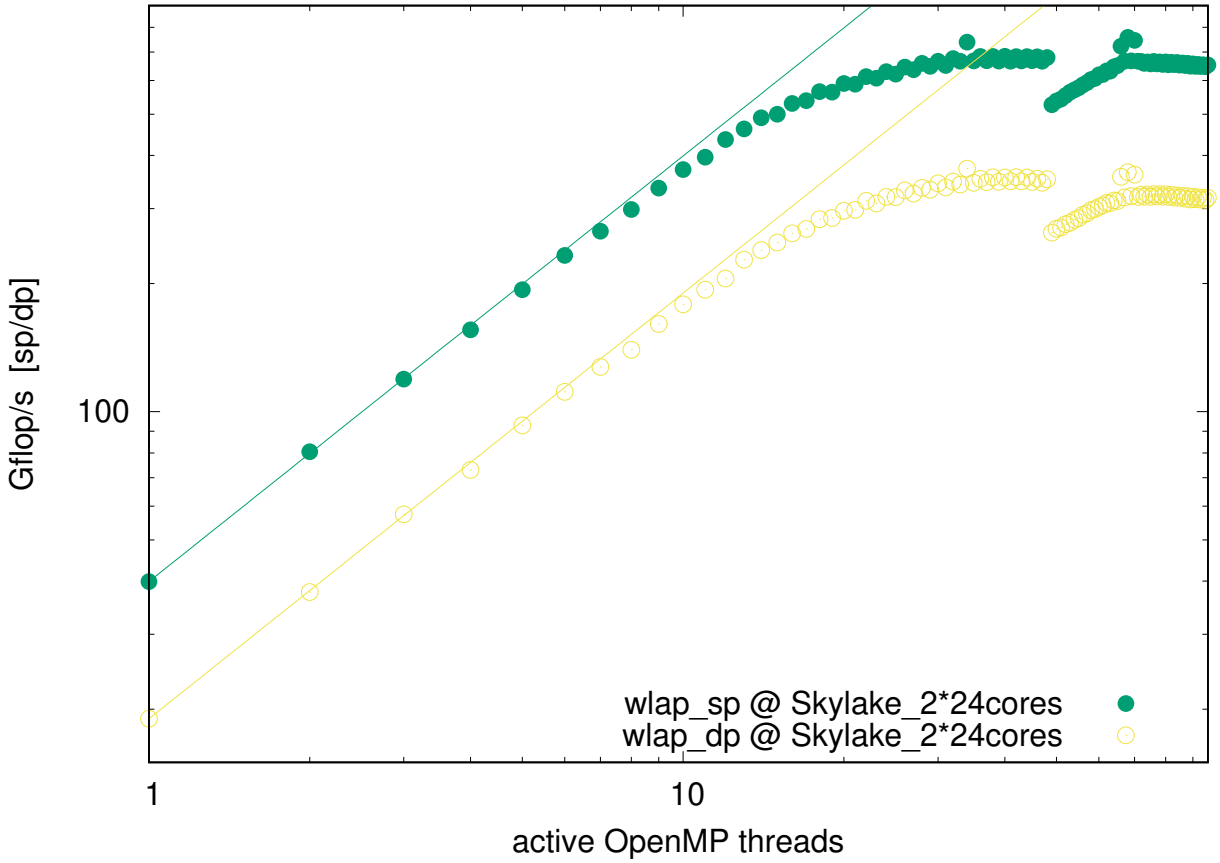


Figure 2: Wilson Laplace operator performance versus the number of threads in sp and dp, for the KNL and the dual-socket Skylake architectures (same parameters as in Tab. 9).

proceed with implementation details of this routine. The main difference to the clover routine discussed in the previous section is that the Wilson Laplacian (10) is *not* site diagonal; it has a 9-point stencil, since $\psi_{\rm new}(n)$ depends, besides $\psi_{\rm old}(n)$, on the eight points $\psi_{\rm old}(n \pm \hat{\mu})$ with $\mu \in \{1,2,3,4\}$. From the perspective of a given thread, the problem with SMP shared-memory parallelization is that these data may lay in the patch of memory which is associated[18] with another thread. Hence read-collisions (or even worse write-collisions) may arise, if different threads attempt to read from (or write to) one portion of memory at the same time. The good news is that there is – within the SMP-paradigm – a simple and effective strategy which bans write-collisions fully, and makes read-collisions very unlikely [36,37]. One accumulates, for each n, the nine contributions (from the central point and from the eight nearest neighbors) in a thread-private variable, e.g. `site(1:Nv,1:4,1:Nc)`, and writes it *once* to $\psi_{\rm new}(n)$. The design of the routine is thus governed by a set of four nested loops (over the x, y, z, t-directions, respectively) that generate the space-time point n of the out-vector $\psi_{\rm new}$, with a `SHARED(old,new,V)` clause in the `!$OMP PARALLEL DO` construct. In the accumulation process one uses `!$OMP SIMD` pragmas to vectorize the summation over the RHS-index, with explicit unrolling of color/spinor indices inside. The usual comments regarding separate implementations of the sp/dp-versions (and the various vector layouts) apply; for details see App. B.

The timings of the Wilson Laplace routine are listed in Tabs. 9, 10 for the KNL and Skylake architectures, respectively. On the KNL chip the vector layout is important; for good performance the SIMD index `rhs` must be first, and the option `NvNsNc` wins the contest. On the Skylake node all vector layouts deliver comparable speed. Most notably, already 24 threads yield almost maximal performance; hence for this routine the second socket (which is populated by threads $25-48$ and $73-96$ under full load) is essentially pointless. For the Wilson Laplacian the Skylake timing surplus (relative to KNL) is about a factor two.

The scaling of the Wilson Laplace routine (in sp and dp, for the `NvNsNc` layout) as a function of the number of active threads is shown in Fig. 2. The parameters, and the range over which the number of threads is varied, are the same as in Sec. 4. On the KNL architecture we find nearly perfect scaling behavior until every physical core hosts one thread. A second thread per core brings a tiny improvement, while a third and fourth thread tend to deteriorate performance. On the Skylake architecture the bottleneck in memory bandwidth is effective from $O(20)$ threads; and there is a local maximum at 68 threads.

The coding of the Wilson Dirac operator is similar to the Laplacian, except that extra operations with $\pm\gamma_\mu$ are involved. Hence, one would naively expect that the matrix-vector operation (9) takes twice as long as (10). Fortunately, actual timings are in the same ball-park (see below). The reason is that for every μ the operator $\frac{1}{2}(\gamma_\mu \mp 1)$ in (9) is a projector. As a result, for each μ the multiplication with $V_\mu(n)$ or $V_\mu^\dagger(n-\hat{\mu})$ in color space can be limited to an object which is *half in size*, see App. A for details. Otherwise the implementation follows the example of the Laplacian sibling routine, with an overall `!$OMP PARALLEL DO` construct on the set of nested space-time loops, and SIMD vectorization over the RHS-index in the part which increments the thread-private variable `site(1:Nv,1:4,1:Nc)`. This accumulation variable is eventually written into the memory block of $\psi_{\rm new}(n)$ as specified in App. B.

The timings of the Wilson Dirac routine at $c_{\rm SW} = 0$ are listed in Tabs. 11, 12 for the KNL and Skylake architectures, respectively. On the KNL chip good performance is achieved whenever the SIMD index `rhs` is in front, and the option `NvNsNc` wins the contest. On the

[18] No inconsistency may arise, but read/write-collisions among threads are detrimental to the performance.

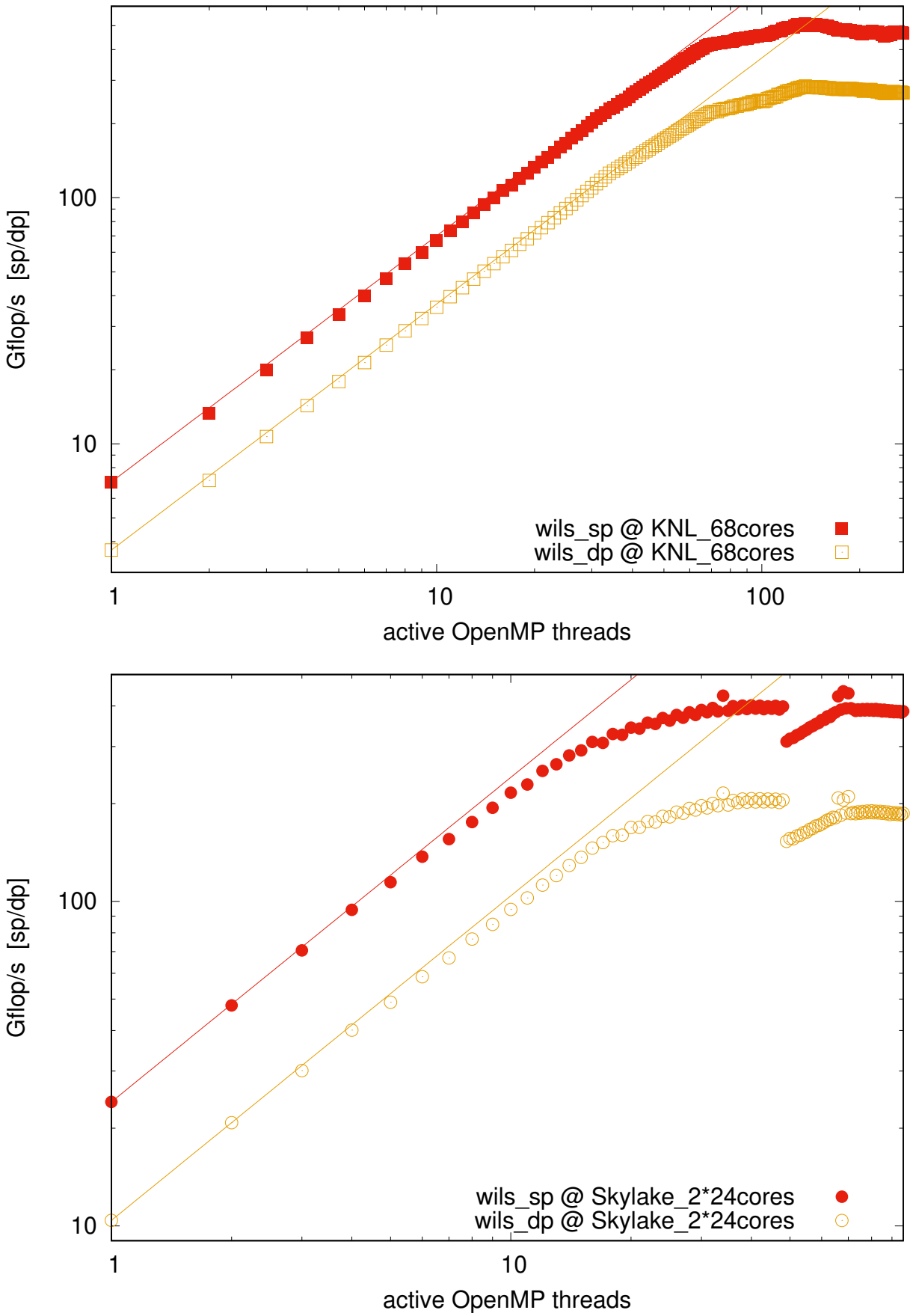


Figure 3: Wilson Dirac operator performance versus the number of threads in sp and dp, for the KNL and the dual-socket Skylake architectures (same parameters as in Tab. 11).

	single precision		double precision	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
`NcNsNv`	0.1269	0.1306	0.1953	0.2246
`NvNcNs`	0.1025	0.1013	0.2032	0.1992
`NvNsNc`	0.0938	0.0913	0.1654	0.1633

Table 11: Time in seconds per matrix times multi-RHS vector operation for the Wilson Dirac operator (6) on the 68-core KNL architecture, with all variables allocated in MCDRAM. The lattice size is $34^3 \times 68$ with parameters $N_c = 3$, $N_v = 12$, $c_{\rm SW} = 0$. The best timings correspond to 480 Gflop/s in sp, and 260 Gflop/s in dp – see App. E for details.

	single precision			double precision		
	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$
`NcNsNv`	0.1472	0.1148	0.1183	0.2768	0.2204	0.2460
`NvNcNs`	0.1274	0.1145	0.1194	0.3009	0.2378	0.2485
`NvNsNc`	0.1233	0.1138	0.1179	0.2466	0.2218	0.2448

Table 12: Same as Tab. 11 but for the 2×24-core (dual socket) Skylake architecture. The best timings correspond to 350 Gflop/s in sp, and 180 Gflop/s in dp.

Skylake node all memory layouts fare in the same league, and 48 threads (i.e. one per physical core on either socket) reach maximum performance.

The scaling of the Wilson Dirac routine (in sp and dp, for the `NvNsNc` layout) as a function of the number of active threads is shown in Fig. 3. On the KNL architecture we find nearly perfect scaling behavior until every physical core hosts one thread, with mild improvement by a second thread per core, and flat or decreasing behavior after this point. On the Skylake architecture maximum performance is already reached with one thread per physical core on one socket. Again, there is a local maximum at 68 threads, and the figure looks like a carbon copy of Fig. 2. The computational intensity (flops per load from memory, see App. E) of the Wilson Dirac operator is so low that the second Skylake socket hardly boosts performance. This will be different with the Brillouin Laplace and Dirac operators.

6 Brillouin Laplace and Dirac routines

For a given $V_\mu(n)$ the Brillouin Dirac operator with optional clover term (4) is defined as [16,17]

$$D_{\rm B}(n,m) = \sum_\mu \gamma_\mu \nabla^{\rm iso}_\mu(n,m) - \frac{ar}{2} \triangle^{\rm bri}(n,m) + m_0 \delta_{n,m} + aC(n,m) \tag{11}$$

where the isotropic derivative $\nabla^{\rm iso}_\mu$ is the 54-point discretization of the covariant derivative

$$\begin{aligned} a\nabla^{\rm iso}_\mu(n,m) &= \rho_1 \left[W_\mu(n) \delta_{n+\hat\mu,m} - W_{-\mu}(n) \delta_{n-\hat\mu,m} \right] \\ &+ \rho_2 \sum\nolimits_{\neq(\nu;\mu)} \left[W_{\mu\nu}(n) \delta_{n+\hat\mu+\hat\nu,m} - (\mu \to -\mu) \right] \\ &+ \rho_3 \sum\nolimits_{\neq(\nu,\rho;\mu)} \left[W_{\mu\nu\rho}(n) \delta_{n+\hat\mu+\hat\nu+\hat\rho,m} - (\mu \to -\mu) \right] \\ &+ \rho_4 \sum\nolimits_{\neq(\nu,\rho,\sigma;\mu)} \left[W_{\mu\nu\rho\sigma}(n) \delta_{n+\hat\mu+\hat\nu+\hat\rho+\hat\sigma,m} - (\mu \to -\mu) \right] \end{aligned} \tag{12}$$

hops	#terms	#paths	formula
1	8	1	$W_\mu(n) = V_\mu(n)$ using smeared link with $\mu \in \{\pm1, \pm2, \pm3, \pm4\}$
2	24	2	$W_{\mu\nu}(n) = \frac{1}{2}[V_\mu(n)V_\nu(n+\hat{\mu}) + V_\nu(n)V_\mu(n+\hat{\nu})]$
3	32	6	$W_{\mu\nu\rho}(n) = \frac{1}{6}[V_\mu(n)V_\nu(n+\hat{\mu})V_\rho(n+\hat{\mu}+\hat{\nu}) + \text{perms}]$
4	16	24	$W_{\mu\nu\rho\sigma}(n) = \frac{1}{24}[V_\mu(n)V_\nu(n+\hat{\mu})V_\rho(n+\hat{\mu}+\hat{\nu})V_\sigma(n+\hat{\mu}+\hat{\nu}+\hat{\sigma}) + \text{perms}]$

Table 13: Summary of on/off-axis links $W_{\rm dir}(n)$, with lengths between 1 and 4 hops. The number of paths contributing to a given term matches the (total) number of permutations. Starting at site n, there are $1+8+24+32+16 = 81$ directions, but one is trivial ("no hop"), and the remaining 80 can be reduced to 40, based on $W_{-\rm dir}(n) = W^\dagger_{\rm dir}(n-{\rm dir})$. In the code W is precomputed and stored in the array `W(Nc,Nc,40,Nx,Ny,Nz,Nt)`. Note that for 36 of the 40 directions the entry is not special unitary; here gauge compression is not possible. Alternatively, the prefactors $\frac{1}{2}, \frac{1}{6}, \frac{1}{24}$ might be replaced by $P_{SU(N_c)}$; in this case gauge compression is possible.

and the Brillouin Laplacian $\triangle^{\rm bri}$ is the 81-point discretization of the covariant Laplacian

$$\begin{aligned} a^2\triangle^{\rm bri}(n,m) = \lambda_0\,\delta_{n,m} \;&+\; \lambda_1 \sum\nolimits_\mu W_\mu(n)\delta_{n+\hat{\mu},m} \\ &+\; \lambda_2 \sum\nolimits_{\neq(\mu,\nu)} W_{\mu\nu}(n)\delta_{n+\hat{\mu}+\hat{\nu},m} \\ &+\; \lambda_3 \sum\nolimits_{\neq(\mu,\nu,\rho)} W_{\mu\nu\rho}(n)\delta_{n+\hat{\mu}+\hat{\nu}+\hat{\rho},m} \\ &+\; \lambda_4 \sum\nolimits_{\neq(\mu,\nu,\rho,\sigma)} W_{\mu\nu\rho\sigma}(n)\delta_{n+\hat{\mu}+\hat{\nu}+\hat{\rho}+\hat{\sigma},m} \end{aligned} \tag{13}$$

with $(\rho_1,\rho_2,\rho_3,\rho_4) \equiv (64,16,4,1)/432$ and $(\lambda_0,\lambda_1,\lambda_2,\lambda_3,\lambda_4) \equiv (-240,8,4,2,1)/64$. The sum in (11) extends over the positive Euclidean directions, i.e. $\mu \in \{1,\ldots,4\}$, and the bare quark mass m_0 undergoes both additive and multiplicative renormalization. In Eq. (12) the last sum extends over (positive and negative) indices whose absolute values are orderd ($|\nu| < |\rho| < |\sigma|$) and different from μ (which is > 0). In Eq. (13) the last sum extends over indices whose absolute values are ordered ($|\mu| < |\nu| < |\rho| < |\sigma|$). Here $W_{\rm dir}(n)$ denotes a link in direction "dir" which may be on-axis (dir=μ) or off-axis with Euclidean length $\sqrt{2}$ (dir=$\mu\nu$), $\sqrt{3}$ (dir=$\mu\nu\rho$), $\sqrt{4}$ (dir=$\mu\nu\rho\sigma$). This $W_{\rm dir}(n)$ is defined as the average of all chains of V-links that connect n and $n+{\rm dir}$ with the minimum number of hops. How the V-links (contained in W and C) relate to the original U-links has been explained in Sec. 4. As a result, $W_{\rm dir}(n)$ is a legitimate parallel transporter from $n+{\rm dir}$ to n, see Tab. 13 for details. More details on the physics motivation and the free-field behavior of this operator are given in Refs. [16, 18].

From the viewpoint of computational expedience, it pays[19] to precompute the W-links, and to feed the routine that eventually implements (11) with $W_{\rm dir}(n)$ [and possibly $F_{\mu\nu}(n)$, if $c_{\rm SW} > 0$]. Similar as with the Wilson Dirac operator the "isotropic derivative" (12) is anti-hermitean, while the "Brillouin Laplacian" (13) and the clover term are hermitean operators. The species-lifting parameter is typically set to $r = 1$, and from a HPC viewpoint the challenge with the operator (11) is that its stencil contains up to 4-hop terms.

The action of the Brillouin operator (11) at $r = 1, c_{\rm SW} = 0$ on a Dirac vector ψ (spinor$\otimes$color

[19]This holds on current CPU architectures, including the KNL and Skylake chips. With the anticipated increase of the compute-to-bandwidth capacity ratio, this may change at some point in the future.

internal degrees of freedom) is given by [with $\gamma_{-\mu} \equiv -\gamma_\mu$ for $\mu > 0$]

$$\begin{aligned}
(D_{\rm B}\psi)(n) &= (m_0 - \frac{\lambda_0}{2})\psi(n) + \sum_\mu \Big[(\rho_1\gamma_\mu - \frac{\lambda_1}{2}) \otimes W_\mu(n)\Big]\psi(n+\hat\mu) \\
&+ \sum_{\neq(\mu,\nu)} \Big[(\rho_2\gamma_\mu - \frac{\lambda_2}{2}) \otimes W_{\mu\nu}(n)\Big]\psi(n+\hat\mu+\hat\nu) \\
&+ \sum_{\neq(\mu,\nu,\rho)} \Big[(\rho_3\gamma_\mu - \frac{\lambda_3}{2}) \otimes W_{\mu\nu\rho}(n)\Big]\psi(n+\hat\mu+\hat\nu+\hat\rho) \\
&+ \sum_{\neq(\mu,\nu,\rho,\sigma)} \Big[(\rho_4\gamma_\mu - \frac{\lambda_4}{2}) \otimes W_{\mu\nu\rho\sigma}(n)\Big]\psi(n+\hat\mu+\hat\nu+\hat\rho+\hat\sigma) \qquad (14)
\end{aligned}$$

where now even μ admits negative values, and our task is to implement a routine which performs this operation efficiently. The action of the embedded Laplace operator (13) is

$$\begin{aligned}
\Big(-\frac{1}{2}\triangle^{\rm bri} + \frac{m_0^2}{2}\Big)\psi(n) &= (\frac{m_0^2}{2} - \frac{\lambda_0}{2})\psi(n) - \frac{\lambda_1}{2}\sum_\mu W_\mu(n)\psi(n+\hat\mu) \\
&- \frac{\lambda_2}{2}\sum_{\neq(\mu,\nu)} W_{\mu\nu}(n)\psi(n+\hat\mu+\hat\nu) \\
&- \frac{\lambda_3}{2}\sum_{\neq(\mu,\nu,\rho)} W_{\mu\nu\rho}(n)\psi(n+\hat\mu+\hat\nu+\hat\rho) \\
&- \frac{\lambda_4}{2}\sum_{\neq(\mu,\nu,\rho,\sigma)} W_{\mu\nu\rho\sigma}(n)\psi(n+\hat\mu+\hat\nu+\hat\rho+\hat\sigma) \qquad (15)
\end{aligned}$$

where the same comment on μ applies, and the mass parameter reflects the usual choice for the Euclidean boson propagator. In both (14) and (15) the previous restrictions carry over, i.e. the first sum is over $8/1! = 8$ directions, the second one over $8 \cdot 6/2! = 24$ directions, the third sum is over $8 \cdot 6 \cdot 4/3! = 32$ directions, and the last one over $8 \cdot 6 \cdot 4 \cdot 2/4! = 16$ directions, see Tab. 13 for details. All together, we have 80 non-trivial directions, and this is exactly what we expect if the total sum extends over a 3^4 cube centered around the space-time point $n = (x, y, z, t)$. In (14) and (15) we have taken the liberty to set the lattice spacing $a = 1$.

Obviously the Brillouin Laplace operator (15) is easier to implement than the Brillouin Dirac operator (14), since it acts trivially[20] in spinor space, so we consider this routine first. At first sight (15) looks complicated, but the five terms can be combined in a convenient manner. Put differently, there is no need to organize the matrix-vector operation separately for 0-hop, 1-hop, ..., 4-hop terms. Given that the off-axis links $W_\mu(n)$, $W_{\mu\nu}(n)$, $W_{\mu\nu\rho}(n)$, $W_{\mu\nu\rho\sigma}(n)$ are contained in the single array `W(Nc,Nc,40,Nx,Ny,Nz,Nt)`, it is more convenient (and faster) to organize the routine through a nested set of four additional loops (the variables `go_x,go_y,go_z,go_t` take the values $-1, 0, +1$) which address the 81 positions m of $\psi_{\rm old}(m)$ in the 3^4 hypercube centered around n. To pick the right prefactor from the set $\{\frac{1}{2}m_0^2 - \frac{1}{2}\lambda_0, -\frac{1}{2}\lambda_1, \ldots, -\frac{1}{2}\lambda_4\}$, it suffices to know which distance (in "taxi-driver metric") the vector `[go_x,go_y,go_z,go_t]`

[20] Hence one might let $\triangle^{\rm bri}$ act on a Susskind-type vector (color structure only), and the ancillary code distribution contains such a multiplication routine under the label `app_bsuv_{sp,dp}`, see Supplementary Material.

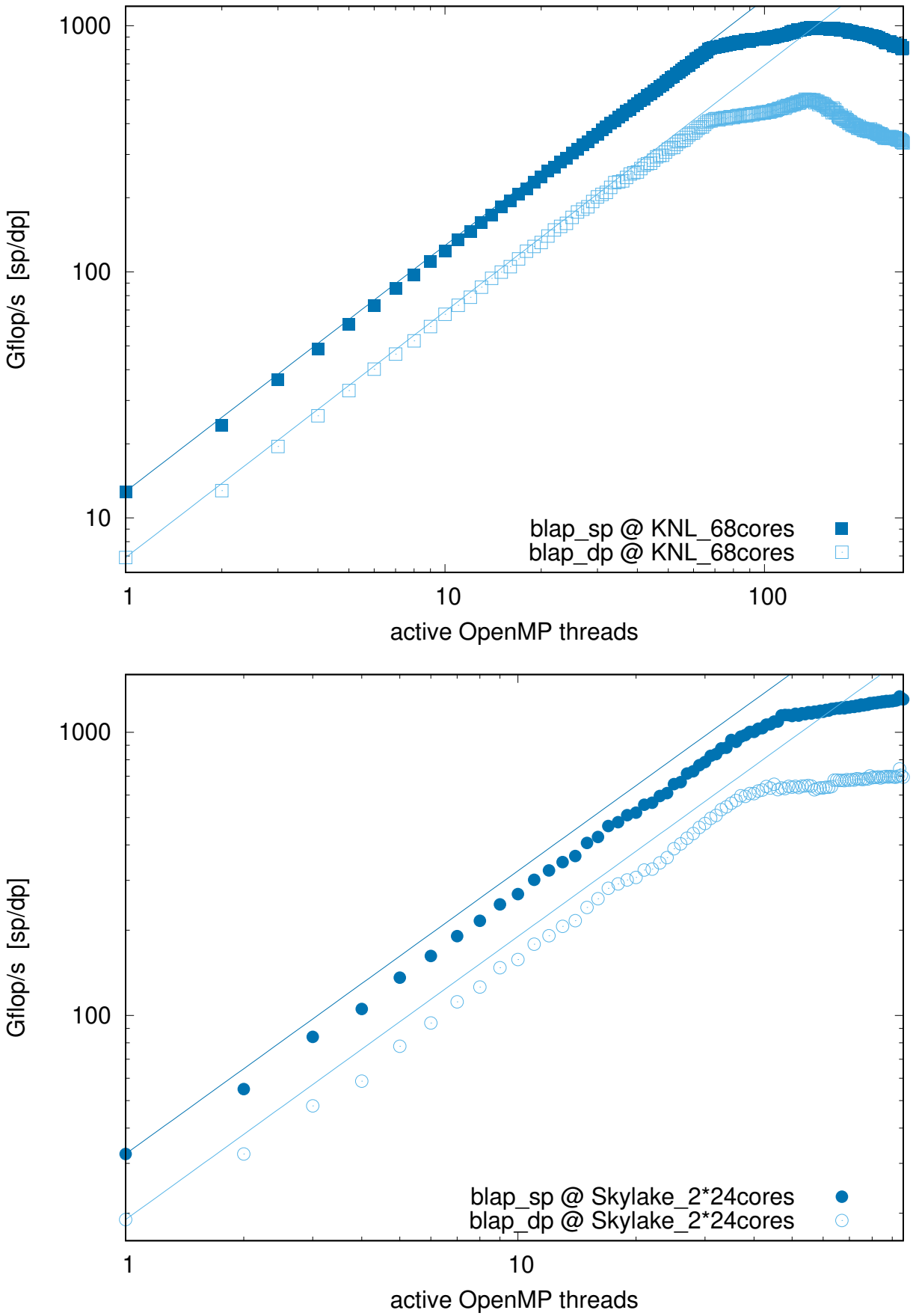


Figure 4: Brillouin Laplace operator performance versus the number of threads in sp and dp, for the KNL and the dual-socket Skylake architectures (same parameters as in Tab. 14).

	single precision		double precision	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
`NcNsNv`	0.9814	1.0363	1.6139	2.3746
`NvNcNs`	0.8146	0.8685	1.6750	2.4589
`NvNsNc`	0.6574	0.7611	1.4889	2.1980

Table 14: Time in seconds per matrix times multi-RHS vector operation for the Brillouin Laplace operator (13) on the 68-core KNL architecture, with variables allocated in MCDRAM. The lattice size is $34^3 \times 68$ with parameters $N_c = 3$, $N_v = 12$. The best timings correspond to 1220 Gflop/s in sp, and 540 Gflop/s in dp – see App. E for details.

	single precision			double precision		
	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$
`NcNsNv`	1.2716	0.6758	0.5580	2.0792	1.1825	1.0767
`NvNcNs`	1.1534	0.6266	0.5324	1.8679	1.0877	1.0182
`NvNsNc`	1.1746	0.6288	0.5355	1.9440	1.1489	1.0844

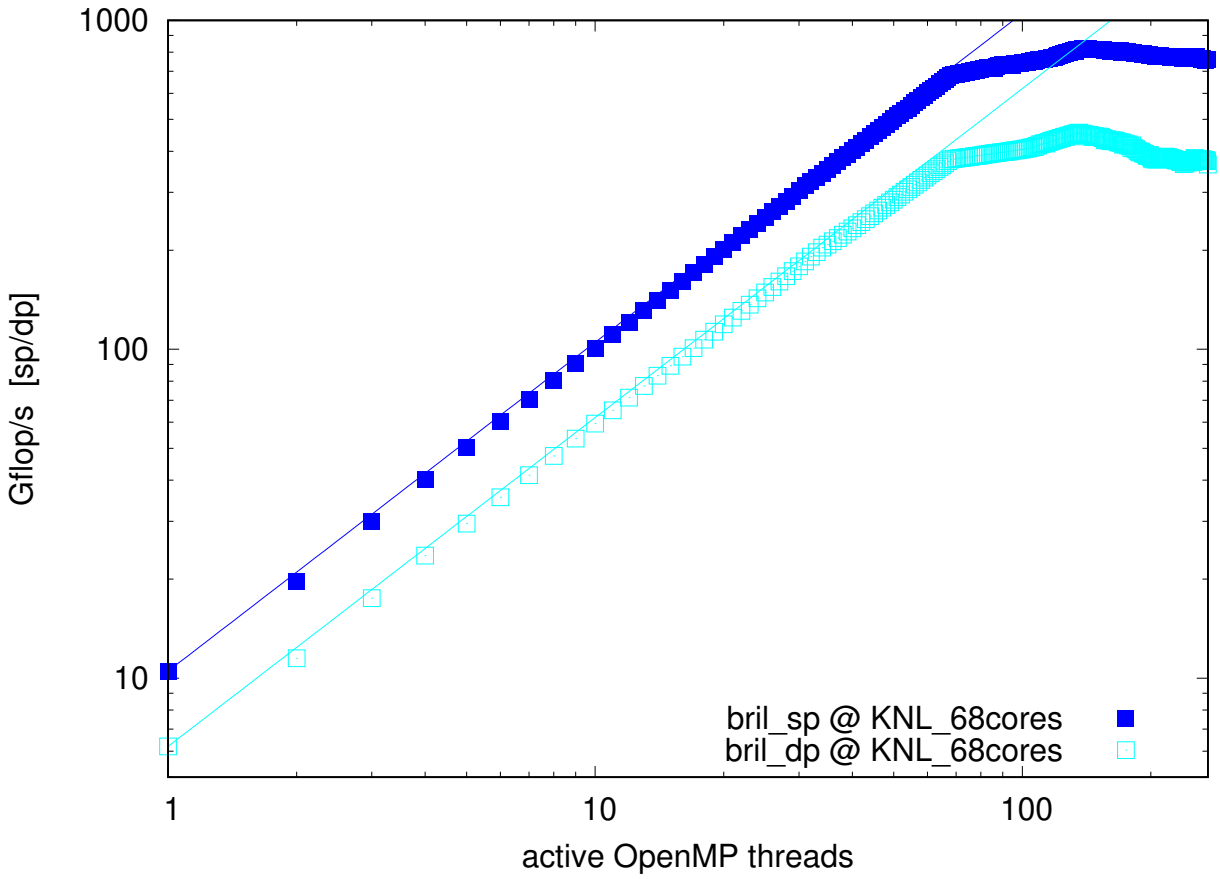
Table 15: Same as Tab. 14 but for the 2×24-core (dual socket) Skylake architecture. The best timings correspond to 1500 Gflop/s in sp, and 790 Gflop/s in dp.

would bridge. This is achieved through `count([go_x,go_y,go_z,go_t].ne.0)` or via a look-up table which uses `min(dir,82-dir)`, where `dir` is the direction count (from 1 to 81 in the nested `[go_x,go_y,go_z,go_t]` loop). Again, all contributions are accumulated in the thread-private variable `site(1:Nv,1:4,1:Nc)`, which is eventually written into the memory block of $\psi_{\rm new}(n)$. More details are provided in App. C.

The timings of the Brillouin Laplace routine are listed in Tabs. 14, 15 for the KNL and Skylake architectures, respectively. On the KNL chip the vector layout is important; the layout `NvNsNc` wins the contest. On the Skylake architecture the layouts `NvNcNs`, `NvNsNc` (with the SIMD index `rhs` in front) are just slightly better than `NcNsNv`. Unlike with the Wilson Laplace routine (see Tab. 10) more threads yield higher performance; the table culminates in a whopping 1500 Gflops (for sp vectors) with 96 threads. In other words, this is the first operator for which the dual-socket Skylake node delivers higher performance than a single KNL chip.

The scaling of the Brillouin Laplace routine (in sp and dp, for the `NvNsNc` layout) as a function of the number of active threads is shown in Fig. 4. The parameters, and the range over which the number of threads is varied, are the same as in Secs. 4, 5. On the KNL architecture we find nearly perfect scaling behavior until every physical core hosts one thread. A second thread per core brings modest improvement, while a third and fourth thread tend to deteriorate performance. By contrast, on the Skylake architecture performance increases (both in sp and dp) until the global maximum is reached near 96 threads.

The coding of the Brillouin Dirac operator (14) proceeds analogous to (15), except that it acts non-trivially in spinor space, too. These extra terms involve the "isotropic derivative" (12) which again leads to a 81-point stencil (with the contraction, for each μ it is fewer points). To maximize performance it pays to combine both gauge-field dependent terms in the Brillouin Dirac operator, so that each $W_{\rm dir}(n)$ is loaded once. Similar to the Brillouin Laplace sibling routine, one uses a set of four nested loops (with variables `go_x,go_y,go_z,go_t` taking the values $-1, 0, +1$ each) to visit the 81 points m of $\psi_{\rm old}(m)$ in the hypercube around n. Unlike



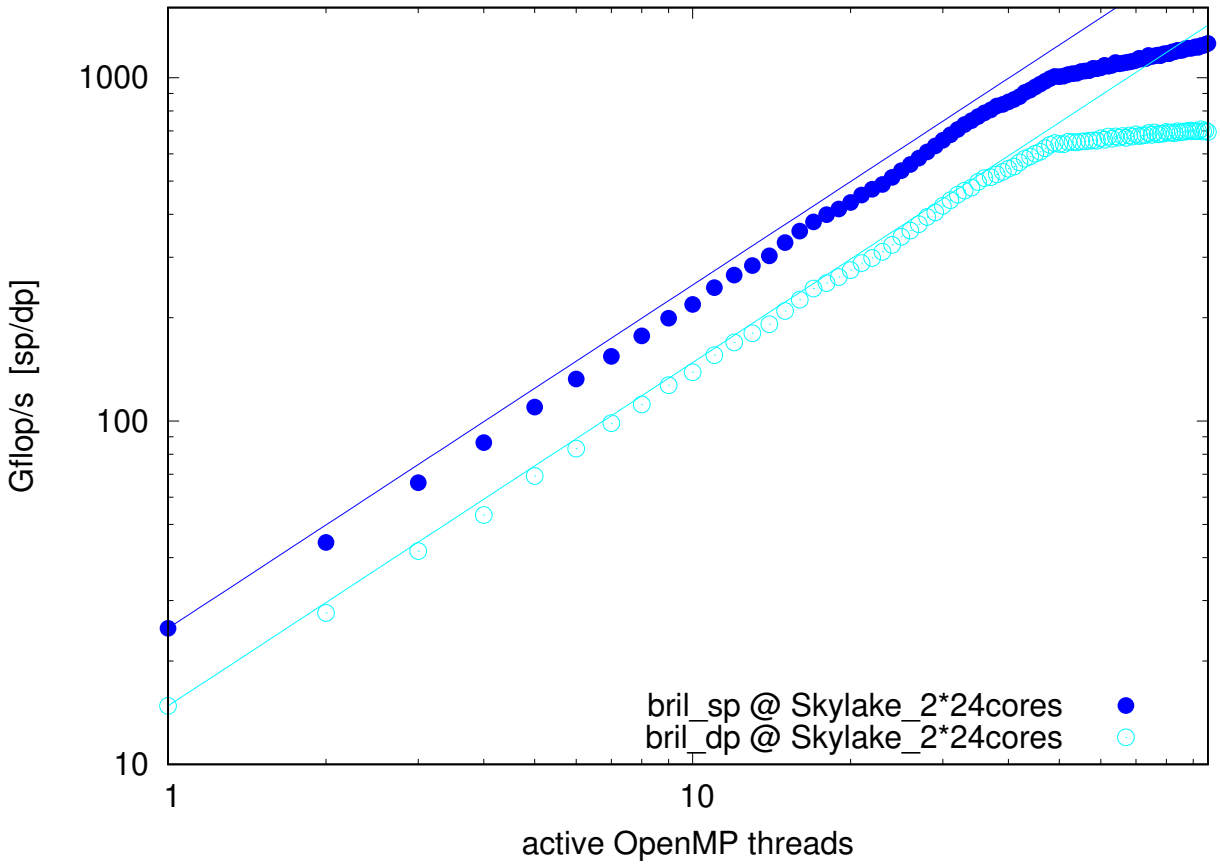


Figure 5: Brillouin Dirac operator performance versus the number of threads in sp and dp, for the KNL and the dual-socket Skylake architectures (same parameters as in Tab. 16).

	single precision		double precision	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
`NcNsNv`	1.3264	1.2675	2.0097	2.6167
`NvNcNs`	1.1326	1.0912	2.0275	2.6740
`NvNsNc`	1.1265	1.1102	2.0228	2.5207

Table 16: Time in seconds per matrix times multi-RHS vector operation for the Brillouin Dirac operator (11) on the 68-core KNL architecture, with variables allocated in MCDRAM. The lattice size is $34^3 \times 68$ with parameters $N_c = 3$, $N_v = 12$, and $c_{\rm SW} = 0$. The best timings correspond to 880 Gflop/s in sp, and 480 Gflop/s in dp – see App. E for details.

	single precision			double precision		
	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$
`NcNsNv`	1.6864	0.8700	0.7433	2.8651	1.5404	1.4205
`NvNcNs`	1.5488	0.8161	0.7262	2.7298	1.4811	1.3666
`NvNsNc`	1.7860	0.9200	0.7426	2.7952	1.5275	1.4288

Table 17: Same as Tab. 16 but for the 2×24-core (dual socket) Skylake architecture. The best timings correspond to 1320 Gflop/s in sp, and 700 Gflop/s in dp.

the Laplace term (which requires only knowledge abut the taxi-driver distance between m and n) the derivative term combines the details of $m - n$ with the detailed choice of the Dirac matrix (in the code the chiral representation specified in App. A is used). All contributions are accumulated in the thread-private variable `site(1:Nv,1:4,1:Nc)`, which is eventually written into the memory block of $\psi_{\rm new}(n)$. More details are provided in App. C.

The timings of the Brillouin Dirac routine at $c_{\rm SW} = 0$ are listed in Tabs. 16, 17 for the KNL and Skylake architectures, respectively. On the KNL chip the vector layouts `NvNcNs`, `NvNsNc` (with the SIMD index `rhs` in front) yield best performance, with $N_{\rm thr} = 136$ and $N_{\rm thr} = 272$ virtually on par. On the Skylake architecture the layout `NvNcNs` seems slightly better than the other two. The peak is at 96 threads, i.e. with all available threads on the dual-socket node. Also for this operator the maximum performance available on the Skylake dual-socket node (1320 Gflops) exceeds the best figure on the KNL node (880 Gflops) by 50%.

The scaling of the Brillouin Dirac routine (in sp and dp, for the `NvNsNc` layout) as a function of the number of active threads is shown in Fig. 5. On the KNL we find (again) nearly perfect scaling behavior until every physical core hosts one thread. The second thread still yields some improvement, while the third and fourth threads per physical core deteriorate performance. By contrast, on the Skylake architecture performance increases (both in sp and dp) until the global maximum is reached at 96 threads. Apart from an overall vertical shift, the entire figure looks like a carbon copy of Fig. 4 (a phenomenon encountered in Sec. 5, too).

The main lesson from this section is that the Brillouin (Laplace and Dirac) operators have a higher computational intensity than the Wilson (Laplace and Dirac) operators (see App. E). This lets the Brillouin operators (81-point stencil) benefit from the compute power of the second socket in the Skylake node, while the Wilson operators (9-point stencil) barely do so.

	single precision		double precision	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
`NcNv`	0.0331	0.0306	0.0558	0.0506
`NvNc`	0.0245	0.0237	0.0455	0.0429

Table 18: Time in seconds per matrix times multi-RHS vector operation for the staggered Dirac operator (16) on the 68-core KNL architecture, with all variables allocated in MCDRAM. The lattice size is $34^3 \times 68$ with parameters $N_c = 3$, $N_v = 12$. The best timings correspond to 780 Gflop/s in sp, and 440 Gflop/s in dp – see App. E for details.

	single precision			double precision		
	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$	$N_{\rm thr} = 24$	$N_{\rm thr} = 48$	$N_{\rm thr} = 96$
`NcNv`	0.0396	0.0342	0.0366	0.0676	0.0613	0.0633
`NvNc`	0.0342	0.0340	0.0363	0.0649	0.0612	0.0633

Table 19: Same as Tab. 18 but for the 2×24-core (dual socket) Skylake architecture. The best timings correspond to 550 Gflop/s in sp, and 300 Gflop/s in dp.

7 Susskind "staggered" Dirac routine

For a given $V_\mu(n)$ the Susskind ("staggered") Dirac operator is defined as [2, 3]

$$D_{\rm S}(n,m) = \sum_\mu \eta_\mu(n) \, \frac{1}{2} \left[V_\mu(n) \delta_{n+\hat{\mu},m} - V_\mu^\dagger(n-\hat{\mu}) \delta_{n-\hat{\mu},m} \right] + m_0 \delta_{n,m} \tag{16}$$

with $\eta_1(n) = 1$, $\eta_2(n) = (-1)^x$, $\eta_3(n) = (-1)^{x+y}$, $\eta_4(n) = (-1)^{x+y+z}$ and $n = (x, y, z, t)$. Here $V_\mu(n)$ represents a smeared version of the (original) gauge link $U_\mu(n)$, i.e. a gauge-covariant parallel transporter from $n + \hat{\mu}$ to n. Its main purpose is to reduce taste-symmetry breaking [42, 43], but there are more sophisticated alternatives with a larger stencil [44].

The main physics difference between the Susskind ("staggered") action and previously discussed fermion actions is that the operator (16) yields four species in the continuum, not just one. Furthermore, the bare quark mass m_0 gets multiplicatively renormalized only (for Wilson and Brillouin fermions there is also an additive shift).

The action of the operator (16) on a Susskind vector ϕ (internal color structure only) is

$$(D_{\rm S}\phi)(n) = \sum_\mu \eta_\mu(n) \, \frac{1}{2} \left[V_\mu(n) \phi(n+\hat{\mu}) - V_\mu^\dagger(n-\hat{\mu}) \phi(n-\hat{\mu}) \right] + m_0 \phi(n) \tag{17}$$

and our task is to implement a routine which performs this operation efficiently.

To prepare for his task, it helps to "downgrade" the Wilson Laplace routine (as discussed in Sec. 5) to the "staggered utility vector", i.e. to remove the spinorial degrees of freedom (which, in this case, were trivially acted on), and the resulting routine `app_wsuv_{sp,dp}` was mentioned in footnote 17. In a similar vein, the Brillouin Laplace operator (as discussed in Sec. 6) can be "downgraded" to act on a "staggered utility vector", and the resulting routine `app_bsuv_{sp,dp}` was mentioned in footnote 20. In either case, the six layout options (among the internal degrees of freedom `rhs,spi,col`) collapse into two layout options (among

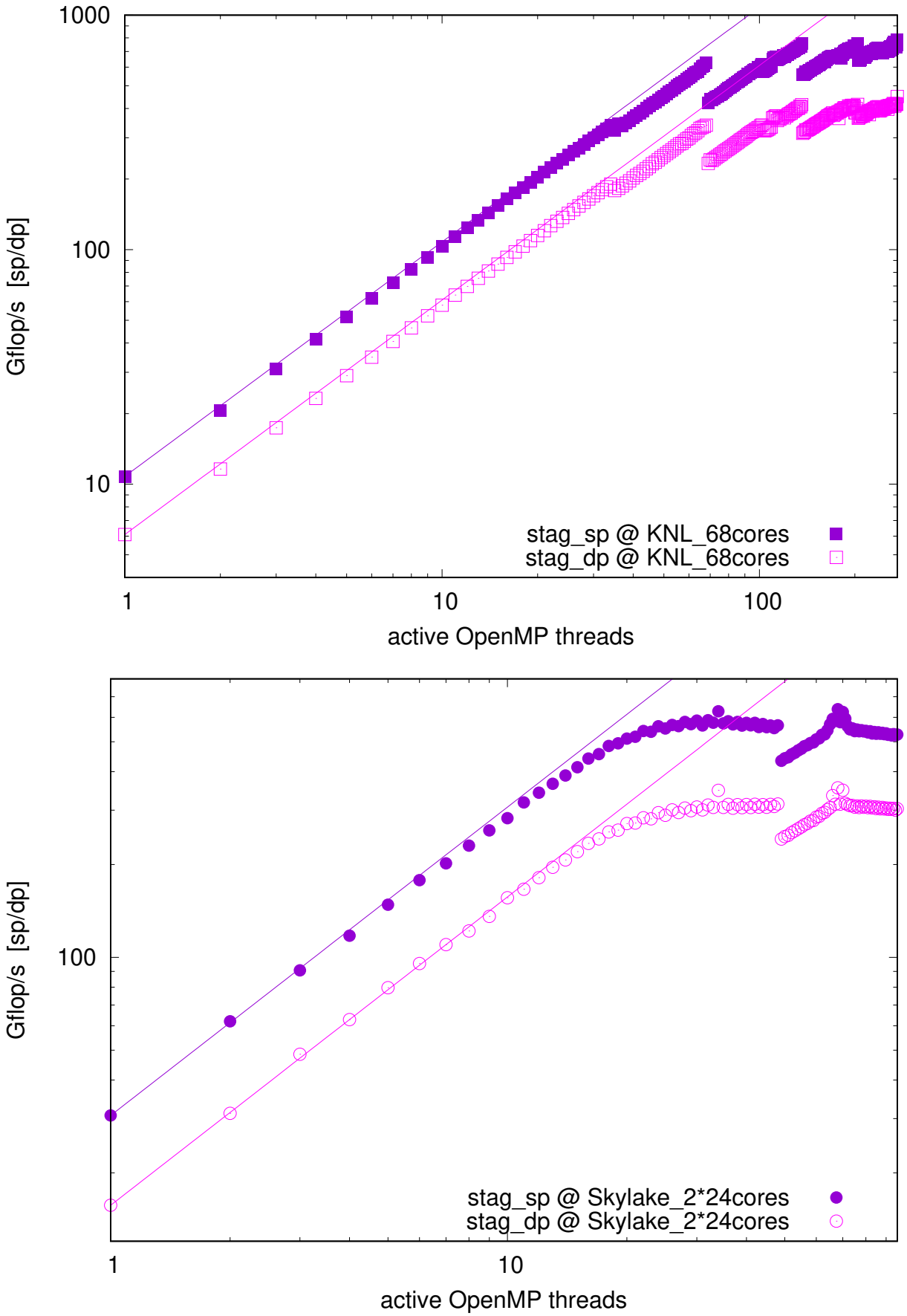


Figure 6: Staggered Dirac operator performance versus the number of threads in sp and dp, for the KNL and the dual-socket Skylake architectures (same parameters as in Tab. 18).

rhs,col). Starting from app_wsuv_{sp,dp}, it is easy to get the staggered routine; one just inserts[21] the extra phase factors η_2, η_3, η_4, where appropriate ($\eta_1 = 1$ is a constant).

The timings of the staggered Dirac routine are listed in Tabs. 18, 19 for the KNL and Skylake architectures, respectively. On the KNL chip the layout NvNc outpaces the reverse ordering, with a peak at 136 threads (both in sp and dp). On the Skylake node the difference between the two layouts is marginal. Here 24 threads (one thread per physical core on one socket) yield better performance than 48 or 96 threads. The relative strength of a single KNL versus a dual-socket Skylake node is opposite to what we have seen with the Brillouin operator; for the staggered operator the KNL chip outpaces the Skylake performance by about 40%.

The scaling of the staggered Dirac routine (in sp and dp, for the NvNc layout) as a function of the number of active threads is shown in Fig. 6. On the KNL we find (again) nearly perfect scaling behavior until every physical core hosts one thread. After a tiny dip, the second thread yields minor improvement, whereas the third and fourth thread barely warrant the extra scheduling cost (the dips in this figure reflect the OpenMP scheduling option static). By contrast, on the Skylake architecture a local extremum is reached at 68 threads (i.e. one thread per timeslice)). The flat structure after saturation at $O(20)$ threads suggests the memory bandwidth is the bottleneck on this architecture (similar to the Wilson case discussed in Sec. 5).

8 Dependence on compile-time parameters

The question behind this article is whether it is possible to write in a high-level language a piece of code that yields decent performance for an arbitrary number of colors, N_c, number of RHS, N_v, and lattice volume $N_x \times N_y \times N_z \times N_t$. In the following, we shall test whether this design goal has been met. We restrict ourselves to the three Dirac operators D_{W}, D_{B} (both at $c_{\mathrm{SW}} = 0$) and D_{S}, in sp and with the best-performing layout on the KNL architecture, i.e. NvNsNc and NvNc, respectively. The number of active threads is $N_{\mathrm{thr}} = 136$ or $N_{\mathrm{thr}} = 272$.

How the performance depends on the volume in lattice units, $N_x \times N_y \times N_z \times N_t$, is summarized in Tab. 20. For all three operators the performance seems largely independent of the volume, apart from a few minor dips (which may be influenced by some OS jitter).

In Tab. 21 the dependence on the number of RHS, N_v, is shown. The volume $24^3 \times 48$ is fixed, and N_v is an integer multiple of $N_c = 3$. For each Dirac operator the performance grows initially with N_v, reaching a maximum at $N_v = 24$. Beyond this point performance degrades[22] for D_{W} and D_{B}, while D_{S} stays more-or-less constant.

The dependence on N_c may be discussed in two settings (both of which keep the volume $24^3 \times 48$ fixed). In Tab. 22 $N_v = 24$ is kept fixed, and N_c is taken to be an integer divisor of 24. In Tab. 23 $N_v = 4N_c$ scales with N_c. In both cases[23] performance grows with N_c (apart from a few minor dips) up to $N_c = 6$, where it reaches a maximum for Wilson quarks, or $N_c = 8$, where the maximum for Brillouin and staggered fermions is found.

The main finding is that no "odd corners" with dramatically reduced performance are detected. Just limiting the number of RHS to small numbers (say $N_v \leq 4$) seems inadvisable;

[21] In some legacy lattice QCD codes, the gauge field $V_\mu(n)$ is dressed with these factors. On modern architectures, e.g. the KNL and Skylake processors used in this work, this does not buy an advantage any more.

[22] Recall that on the KNL chip the first 16 GB is allocated in MCDRAM, the remainder in DDR4 memory.

[23] The attentive reader may notice the $N_c = 6$ rows of Tabs. 22 and 23 refer to the same situation (though from different runs). Comparing them one finds that rounding results to multiples of 10 Gflop/s is reasonable.

	Wilson		Brillouin		Susskind	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
$L = 16$	460	440	800	710	630	620
$L = 20$	480	480	820	740	760	830
$L = 24$	490	470	850	770	790	780
$L = 32$	410	440	760	660	730	690
$L = 40$	470	470	830	810	730	710
$L = 48$	380	400	790	740	670	670

Table 20: Performance in Gflop/s of the Wilson, Brillouin and Susskind routines in sp on the KNL chip as a function of the volume $L^3 \times T$ with $T = 2L$, at fixed $N_c = 3$, $N_v = 12$.

	Wilson		Brillouin		Susskind	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
$N_v = 1N_c$	240	270	190	280	210	240
$N_v = 2N_c$	430	450	530	670	380	460
$N_v = 4N_c$	480	450	840	720	770	780
$N_v = 8N_c$	530	500	1050	710	1000	850
$N_v = 16N_c$	450	420	630	644	990	860
$N_v = 32N_c$	390	390	580	570	960	890

Table 21: Performance in Gflop/s of the Wilson, Brillouin and Susskind routines in sp on the KNL chip as a function of N_v, at fixed volume $24^4 \times 48$ and $N_c = 3$.

	Wilson		Brillouin		Susskind	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
$N_c = 2$	410	390	850	630	770	670
$N_c = 3$	520	500	1040	710	990	860
$N_c = 4$	540	530	910	830	1100	1010
$N_c = 6$	630	530	1030	1050	1160	1080
$N_c = 8$	610	400	1230	1210	1260	1090
$N_c = 12$	520	350	1280	910	720	870

Table 22: Performance in Gflop/s of the Wilson, Brillouin and Susskind routines in sp on the KNL chip as a function of N_c, at fixed volume $24^4 \times 48$ and $N_v = 24$.

	Wilson		Brillouin		Susskind	
	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$	$N_{\rm thr} = 136$	$N_{\rm thr} = 272$
$N_c = 2$	460	400	840	700	640	550
$N_c = 3$	470	470	840	720	770	780
$N_c = 4$	480	500	920	830	960	830
$N_c = 6$	630	530	1030	1060	1180	1080
$N_c = 8$	570	400	1210	960	1260	1210
$N_c = 12$	410	310	880	780	1090	1130

Table 23: Performance in Gflop/s of the Wilson, Brillouin and Susskind routines in sp on the KNL chip as a function of N_c, at fixed volume $24^4 \times 48$ with proportionality $N_v = 4N_c$.

this backs the arguments presented in Sec. 1 for pursuing a multi-RHS strategy. In summary, the code seems fairly robust against changes of the volume, the number of RHS, and the number of colors. It appears to be a useful tool for studying QCD at large N_c.

9 Krylov space inverters

We have all ingredients needed to compare the various vector layout options in an attempt to tackle Eq. (1). Krylov-space solvers are iterative procedures to solve the system $Au = b$, for a given RHS b, to a predefined tolerance ϵ. The solver is stopped, as soon as the norm of the residual $r \equiv b - Au$ satisfies $||r|| < \epsilon||b||$. In this article $||.||$ is taken to be the 2-norm, and the stringent tolerance $\epsilon = 10^{-12}$ (which can be reached in dp but not in sp) is used.

We aim to compare the `vec`-layouts for the Conjugate Gradient (CG) algorithm with one of the hermitean positive definite (HPD) operators $A = -\frac{1}{2}\triangle^{\rm std} + \frac{1}{2}m_0^2$, $A = D_{\rm W}^\dagger D_{\rm W}$, $A = -\frac{1}{2}\triangle^{\rm bri} + \frac{1}{2}m_0^2$ or $A = D_{\rm B}^\dagger D_{\rm B}$, as well as the BiCGstab algorithm with $A = D_{\rm W}$ or $A = D_{\rm B}$ (which are neither hermitean nor anti-hermitean but γ_5-hermitean). And we compare the `suv`-layouts for the CG algorithm with the HPD operator $A = D_{\rm S}^\dagger D_{\rm S}$. The bare mass $am_0 = 0.01$ is used for all operators; for $D_{\rm W}$ and $D_{\rm B}$ it is combined with $c_{\rm SW} = 1$. We use a quenched $24^3 \times 48$ configuration with $a \simeq 0.9\,{\rm fm}$, and $N_v = 12$. The gauge field $V_\mu(n)$ [from which $F_{\mu\nu}(n)$ derives] is constructed from $U_\mu(n)$ via three steps of $\rho = 0.12$ stout smearing [38].

The solvers (CG and BiCGstab) are written in a generic way, i.e. they operate on vectors with any of the six (two) layout options for Wilson-type (Susskind-type) Dirac operators. For the matrix times vector operation they call a "wrapper" routine which eventually calls the optimized routine for the specific layout (a similar statement holds w.r.t. the linear algebra routines). The residual is "recomputed" (i.e. $r = b - Au$ explicitly formed with u the current approximation) rather than "updated" (via a cheaper vector-only operation), if one of the following conditions is met: (*i*) the iteration count is an integer multiple of 20 in sp (50 in dp), (*ii*) the updated residual r suggests that $||r||/||b|| < \epsilon$ might hold for each RHS, (*iii*) the maximum iteration count occurs. The routine exits if either $||r||/||b|| < \epsilon$ for each RHS (based on the recomputed $r = b - Au$), or the discrepancy between the updated and actual residual norm exceeds 1%, or the maximum iteration count is reached. To avoid excessively long log files the relative norm is printed (and thus plotted) every ten iterations only. In this section half of the maximum number of OpenMP threads is used on either architecture, i.e. $N_{\rm thr} = 136$ on the KNL chip and $N_{\rm thr} = 48$ on the Skylake node.

The CG history of $A = -\frac{1}{2}\triangle^{\rm std} + \frac{1}{2}m_0^2$ is shown in Fig. 7. On the KNL chip the layout `NcNsNv` is slowest (+ symbol), `NvNcNs` is better (× symbol), and `NvNsNc` is best (filled boxes), both in sp and dp. It is worth mentioning that the physics content of the RHS vector b is the same for the three vector layouts, i.e. the memory content of b in `NvNsNc` is just a permuted version of `NcNsNv`. The x-axis shows the time per RHS, the y-axis the *worst* of the N_v relative residual norms. In dp the target precision $\epsilon = 10^{-12}$ is met after 109 iterations (for each layout option), while in sp after 60 iterations the algorithm notices that the updated residual norm $||r||/||b|| = 1.11 \times 10^{-7}$ differs from the recomputed residual norm $||r||/||b|| = 3.12 \times 10^{-7}$ by more than 1%, and thus stops[24]. On the Skylake architecture the difference among the layouts is gone, but the best overall time increases quite a bit (from 0.8 to 1.2 s in dp).

[24] If one were to continue beyond this point, the recomputed residual norm would stagnate or grow.

The CG history of $A = D_{\rm W}^\dagger D_{\rm W}$ is shown in Fig. 8. On the KNL chip the layout `NvNsNc` is the fasted one (both in sp and dp, plotted with filled symbols). In dp the target precision $\epsilon = 10^{-12}$ is met after 1858 iterations (for each layout option), while in sp after 620 iterations the algorithm notices that the updated residual norm $||r||/||b|| = 1.69 \times 10^{-5}$ differs from the recomputed residual norm $||r||/||b|| = 1.71 \times 10^{-5}$ by more than 1%, and thus stops[25]. On the Skylake architecture the difference among the layouts is almost gone, but the best overall execution time increases a bit (from 26 s to 32 s in dp).

The CG history of $A = -\frac{1}{2}\triangle^{\rm bri} + \frac{1}{2}m_0^2$ is shown in Fig. 9. On the KNL chip the layout `NvNsNc` is again the fastest one (both in sp and dp, plotted with filled symbols). In dp the target precision $\epsilon = 10^{-12}$ is met after 54 iterations (for each layout option), while in sp after 40 iterations the algorithm notices that the updated residual norm $||r||/||b|| = 5.46 \times 10^{-10}$ differs from the recomputed residual norm $||r||/||b|| = 1.40 \times 10^{-7}$ by more than 1%, and thus stops[26]. No signs of numerical instability are seen; the six symbols at a given precision and iteration count are just horizontally displaced from each other. On the Skylake architecture the difference among the layouts is gone, while the best overall execution time is a bit shorter than on the KNL (from 2.0 s to 1.7 s in dp).

The CG history of $A = D_{\rm B}^\dagger D_{\rm B}$ is shown in Fig. 10. In dp the target precision $\epsilon = 10^{-12}$ is met after 811 iterations (for each layout option), while in sp after 280 iterations the algorithm notices that the updated residual norm $||r||/||b|| = 9.25 \times 10^{-6}$ differs from the recomputed residual norm $||r||/||b|| = 9.44 \times 10^{-6}$ by more than 1%, and thus stops. Here the difference among the three layout options is mild, and out of the KNL and Skylake architectures the latter one fares significantly better (75 s versus 58 s in dp).

In Figs. 7–10 no sign of numerical imprecision is seen; the three symbols at a given iteration count (for either sp or dp) are just horizontally displaced. A second issue is worth mentioning. On the Skylake architecture the Brillouin operator *converges in about twice the time* of the Wilson operator. The additive mass shift of the two Dirac operators is roughly in the same[27] ballpark. Thus the timings of Sec. 5 and Sec. 6 (where $D_{\rm B}$ seemed about an order of magnitude more expensive than $D_{\rm W}$) do not represent the last word on the relative cost of these two Dirac operators. The reason is the more compact eigenvalue spectrum of $D_{\rm B}$ [reaching up to $\mathrm{Re}(z) = 2 + am_0$ in the free field case] in comparison to $D_{\rm W}$ [which extends to $\mathrm{Re}(z) = 8 + am_0$]. Hence at fixed pion mass, the matrix-vector cost explosion (in trading $D_{\rm W}$ for $D_{\rm B}$) is mitigated by a reduced condition number (see also the discussion in Refs. [16–18]).

The CG history of $A = D_{\rm S}^\dagger D_{\rm S}$ is shown[28] in Fig. 11 (as usual adjacent boxes are separated by ten iterations). On the KNL chip the layout `NvNc` (filled symbols) is faster than `NcNv` (open symbols); on the Skylake architecture the difference is marginal. In dp the target precision $\epsilon = 10^{-12}$ is met after 2489 iterations (for either layout), while in sp after 740 iterations the algorithm notices that the updated residual norm $||r||/||b|| = 6.95 \times 10^{-5}$ differs from the recomputed residual norm $||r||/||b|| = 7.06 \times 10^{-5}$ by more than 1%, and thus stops.

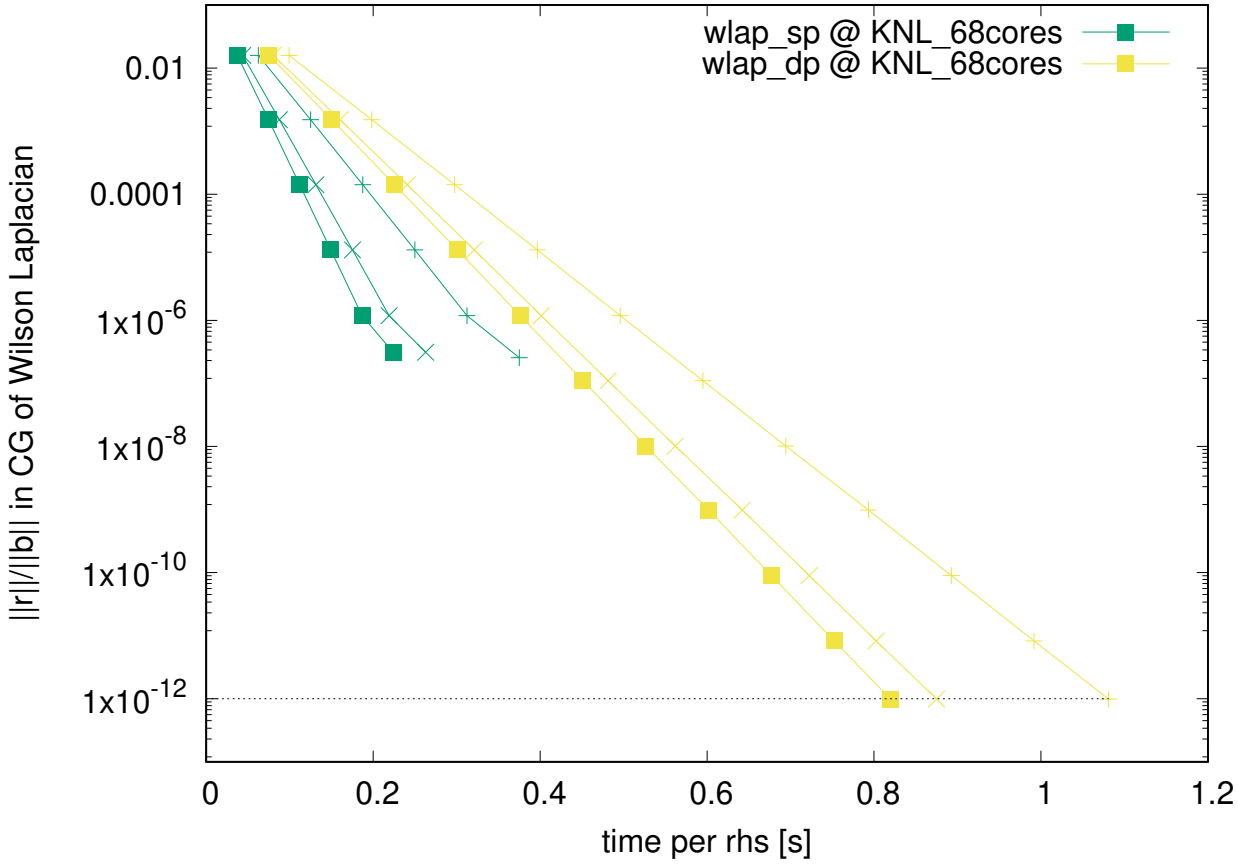
Fig. 12 shows the convergence history of the BiCGstab algorithm for $D_{\rm W} u = b$. The descent is "wigglier" than for the CG algorithm. The sensitivity of the BiCGstab algorithm to numerical

[25]The attentive reader may notice these figures are significantly larger than those mentioned in the discussion of the CG histories of $-\frac{1}{2}\triangle^{\rm std} + \frac{1}{2}m_0^2$. This suggests there is no universal tolerance that can be reached in sp; the minimum $||r||/||b||$ depends on the type of operator used and am_0.

[26]In sp the third (fourth) box shows the updated (recomputed) relative residual norm at iteration 30 (40).

[27]Preliminary spectroscopy on a handful of configurations suggests $M_\pi^{\rm wils} \simeq 760\,{\rm MeV}$ and $M_\pi^{\rm bril} \simeq 670\,{\rm MeV}$.

[28]Preliminary spectroscopy on a handful of configurations suggests $M_\pi^{\rm stag} \simeq 340\,{\rm MeV}$.



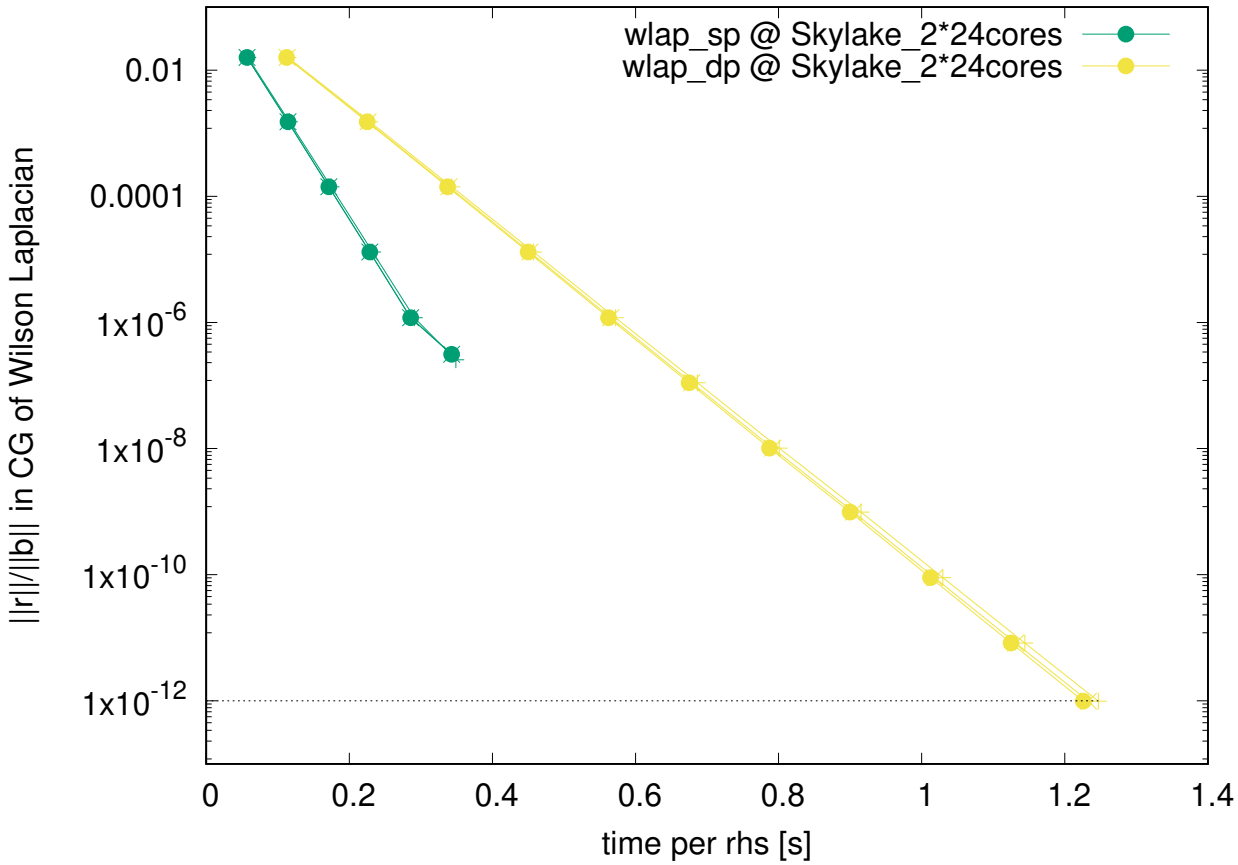
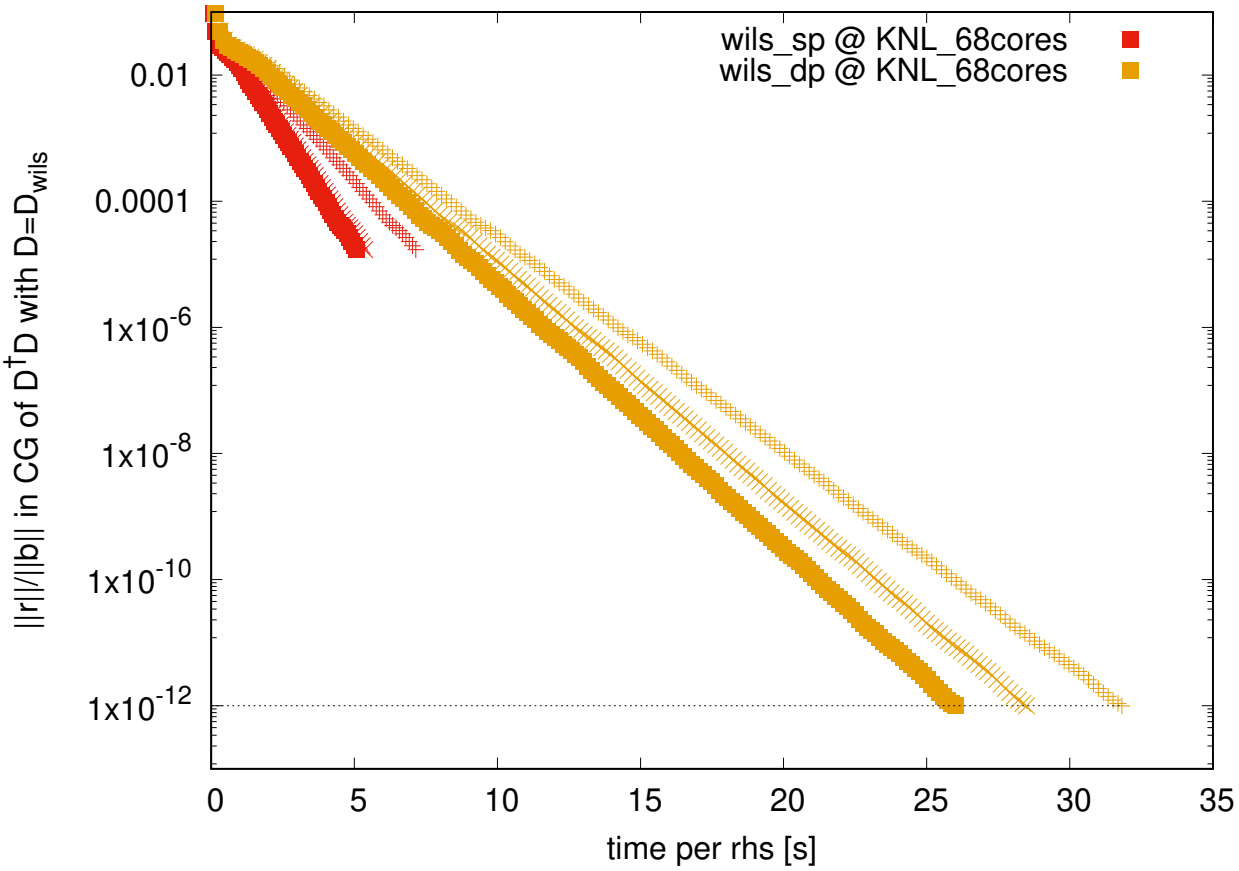


Figure 7: Relative residual norm versus time of the CG solver for the Wilson Laplacian at $am_0 = 0.01$ on a $24^3 \times 48$ lattice in sp/dp (layout `NvNsNc` filled, other $+, \times$) on the KNL and Skylake node. In all cases the solver exits after 60 (109) iterations in sp (dp).



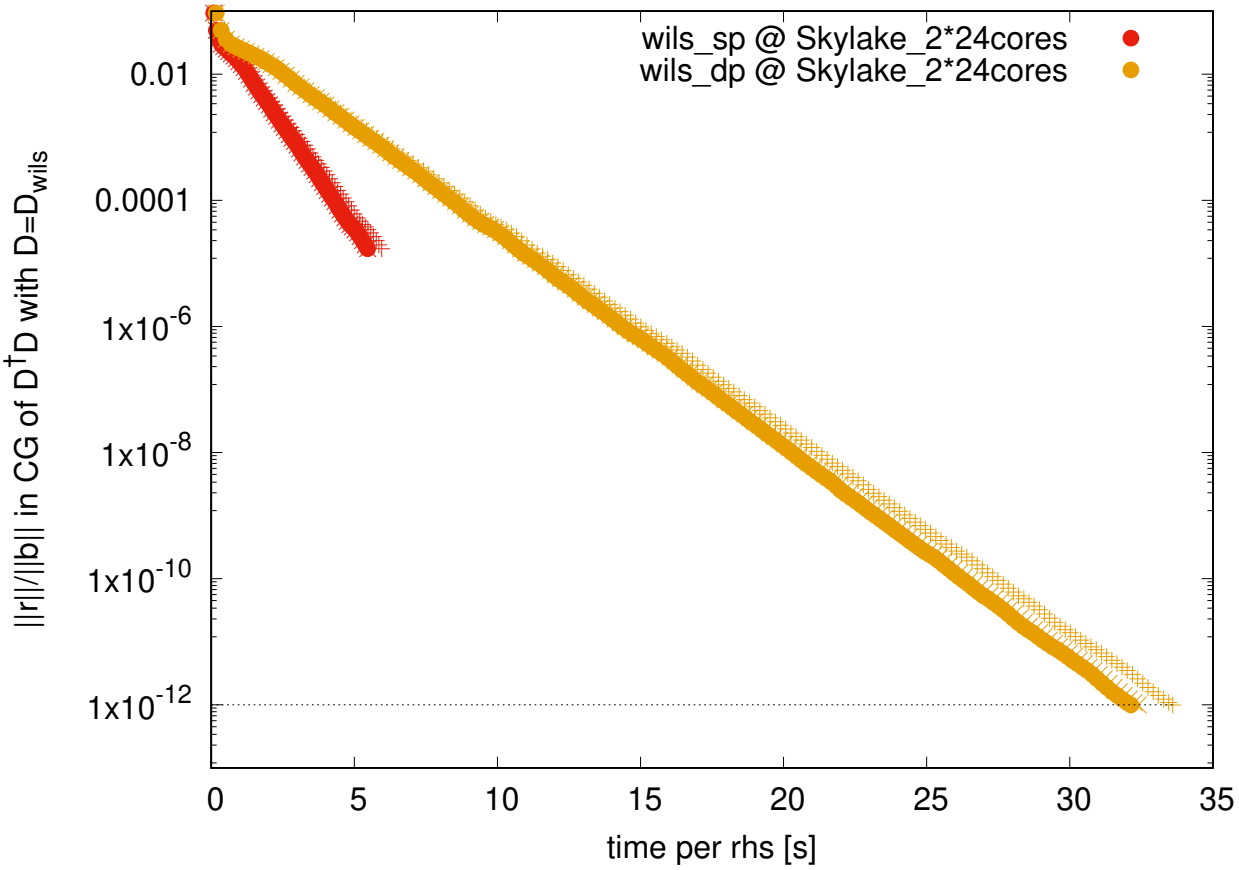
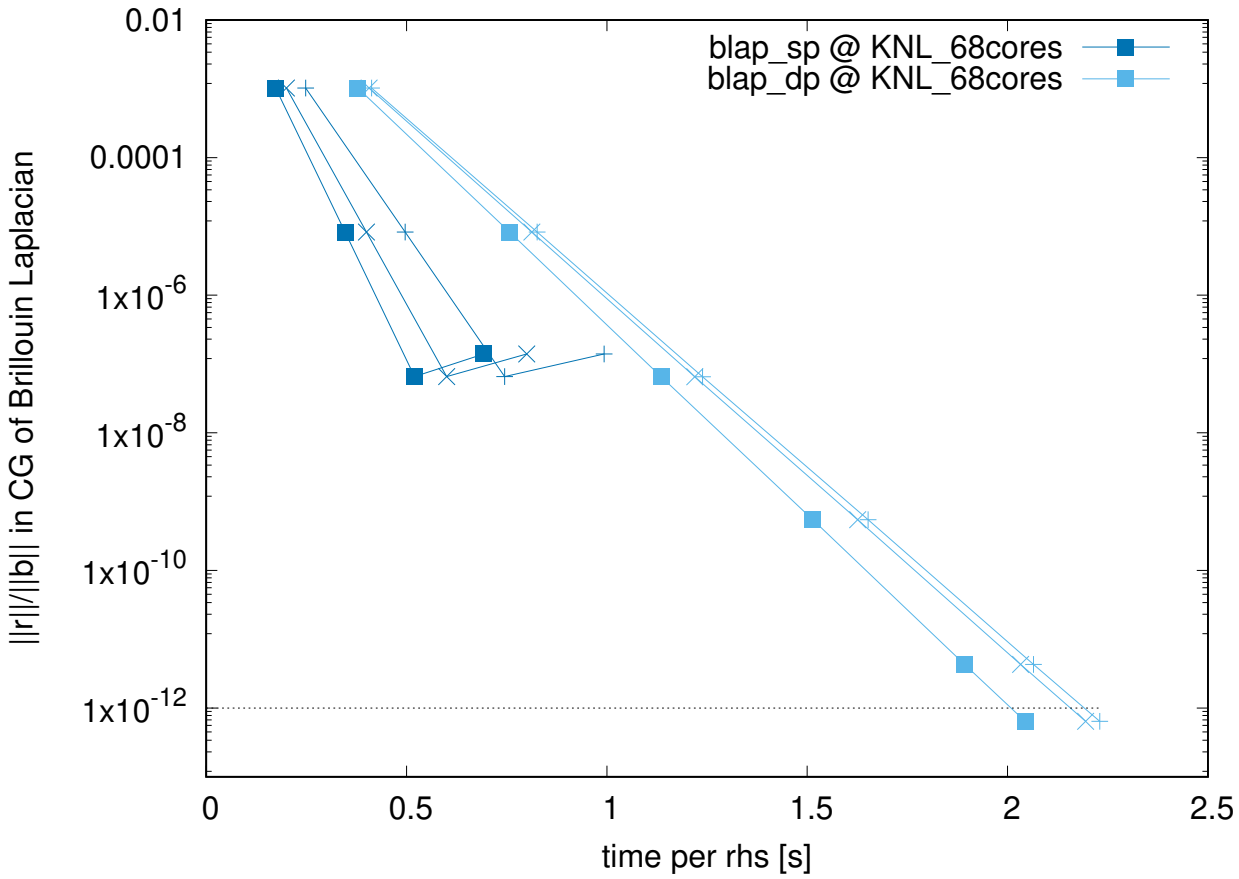


Figure 8: Relative residual norm versus time of the CG solver for the Wilson $D_{\mathrm{W}}^{\dagger} D_{\mathrm{W}}$ at $am_0 = 0.01$, $c_{\mathrm{SW}} = 1$ on a $24^3 \times 48$ lattice in sp/dp (layout `NvNsNc` filled, other $+, \times$) on the KNL and Skylake node. In all cases the solver exits after 620 (1858) iterations in sp (dp).



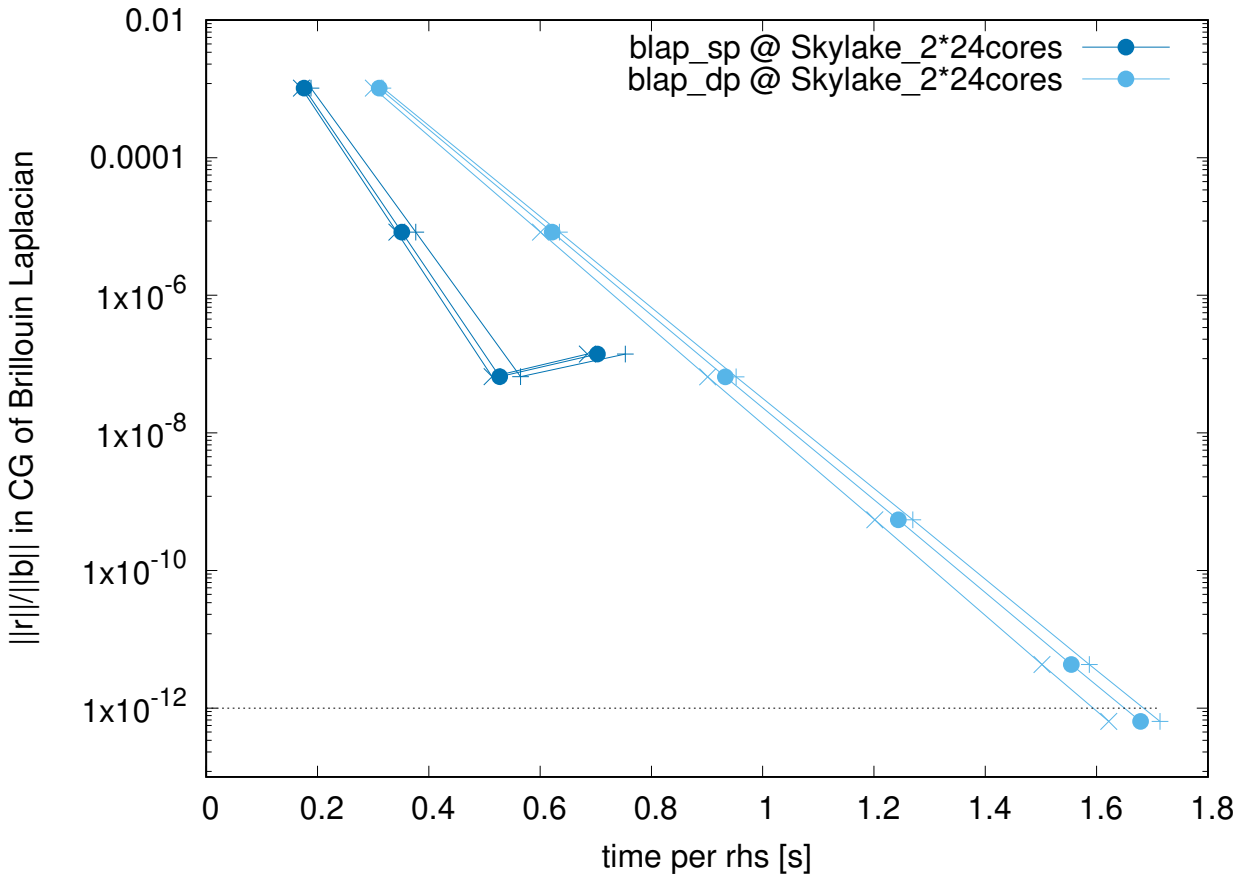
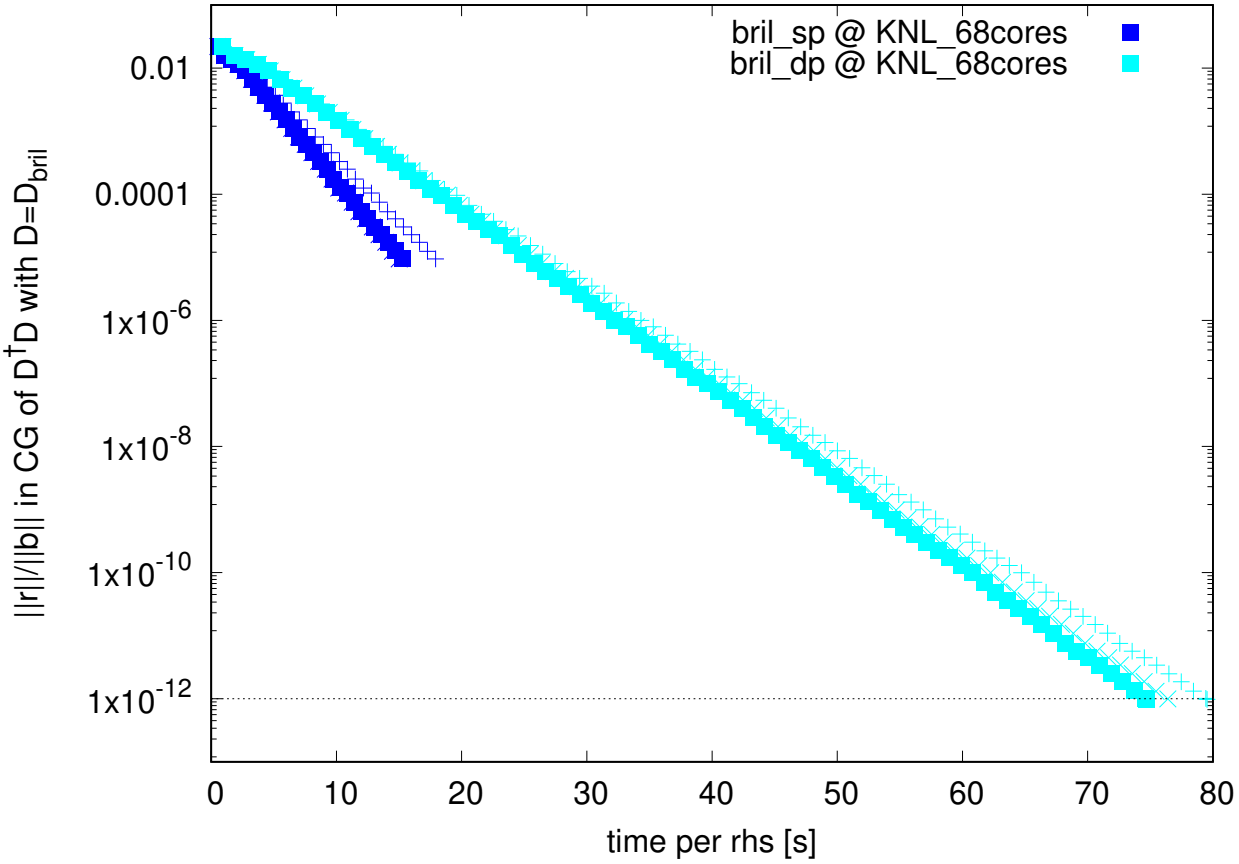


Figure 9: Relative residual norm versus time of the CG solver for the Brillouin Laplacian at $am_0 = 0.01$ on a $24^3 \times 48$ lattice in sp/dp (layout `NvNsNc` filled, other $+, \times$) on the KNL and Skylake node. In all cases the solver exits after 40 (54) iterations in sp (dp); cf. footnote 26.



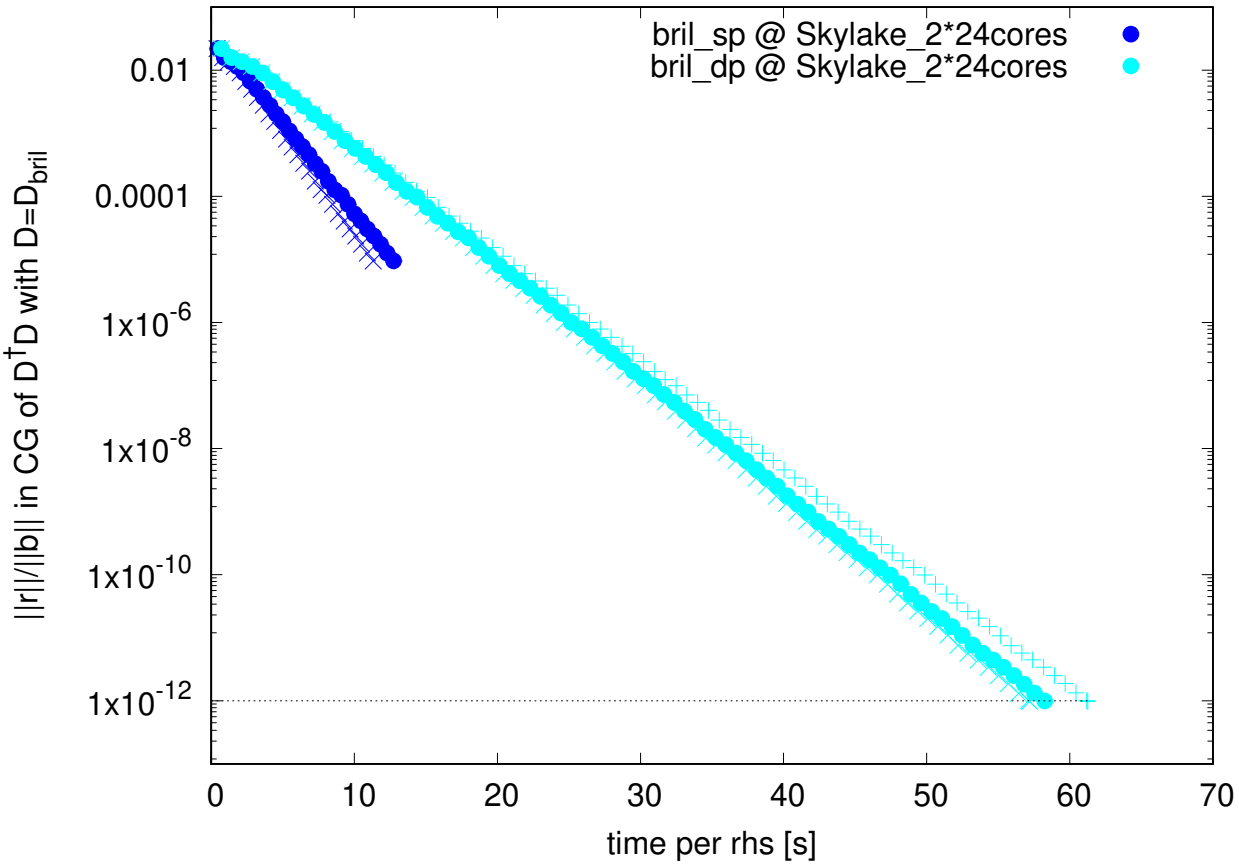
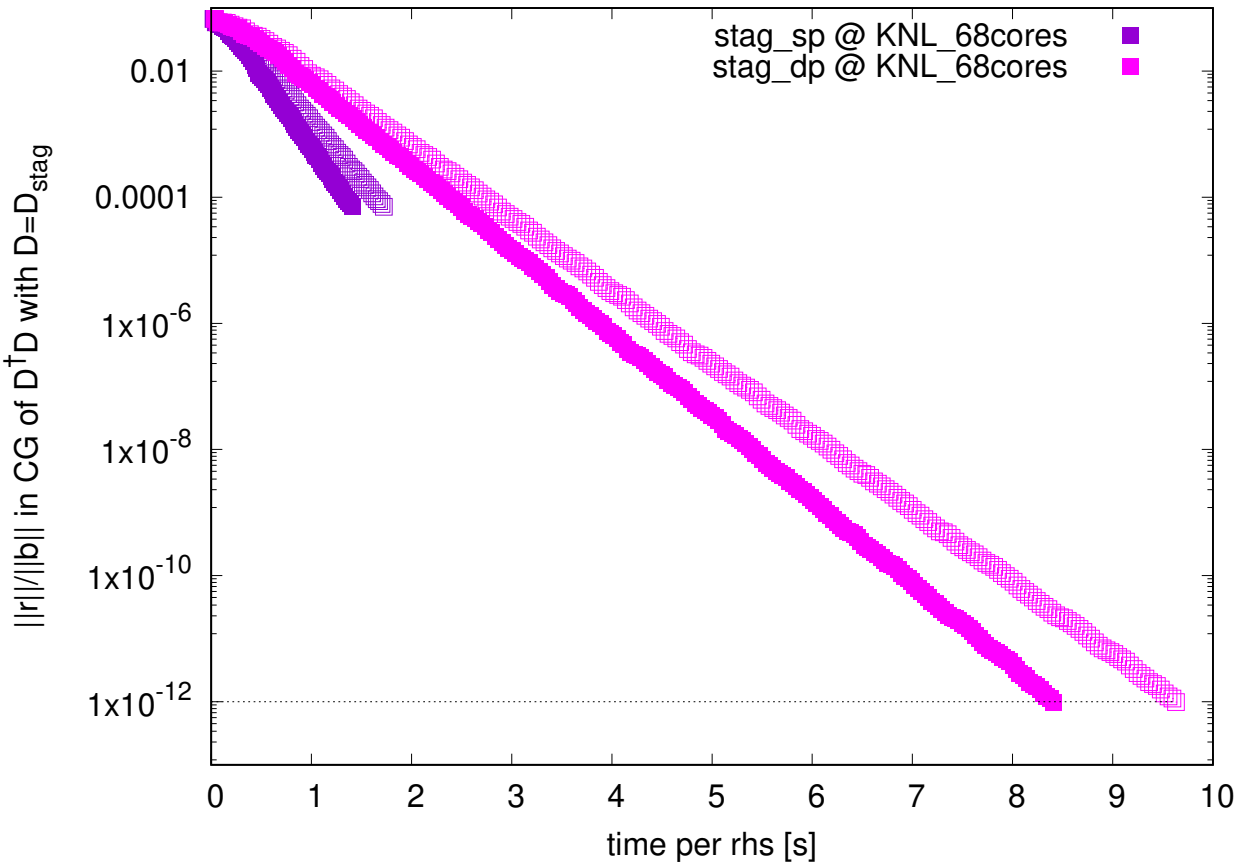


Figure 10: Residual residual norm versus time of the CG solver for the Brillouin $D_{\mathrm{B}}^{\dagger} D_{\mathrm{B}}$ at $am_0 = 0.01$, $c_{\mathrm{SW}} = 1$ on a $24^3 \times 48$ lattice in sp/dp (layout `NvNsNc` filled, other $+, \times$) on the KNL and Skylake node. In all cases the solver exits after 280 (811) iterations in sp (dp).



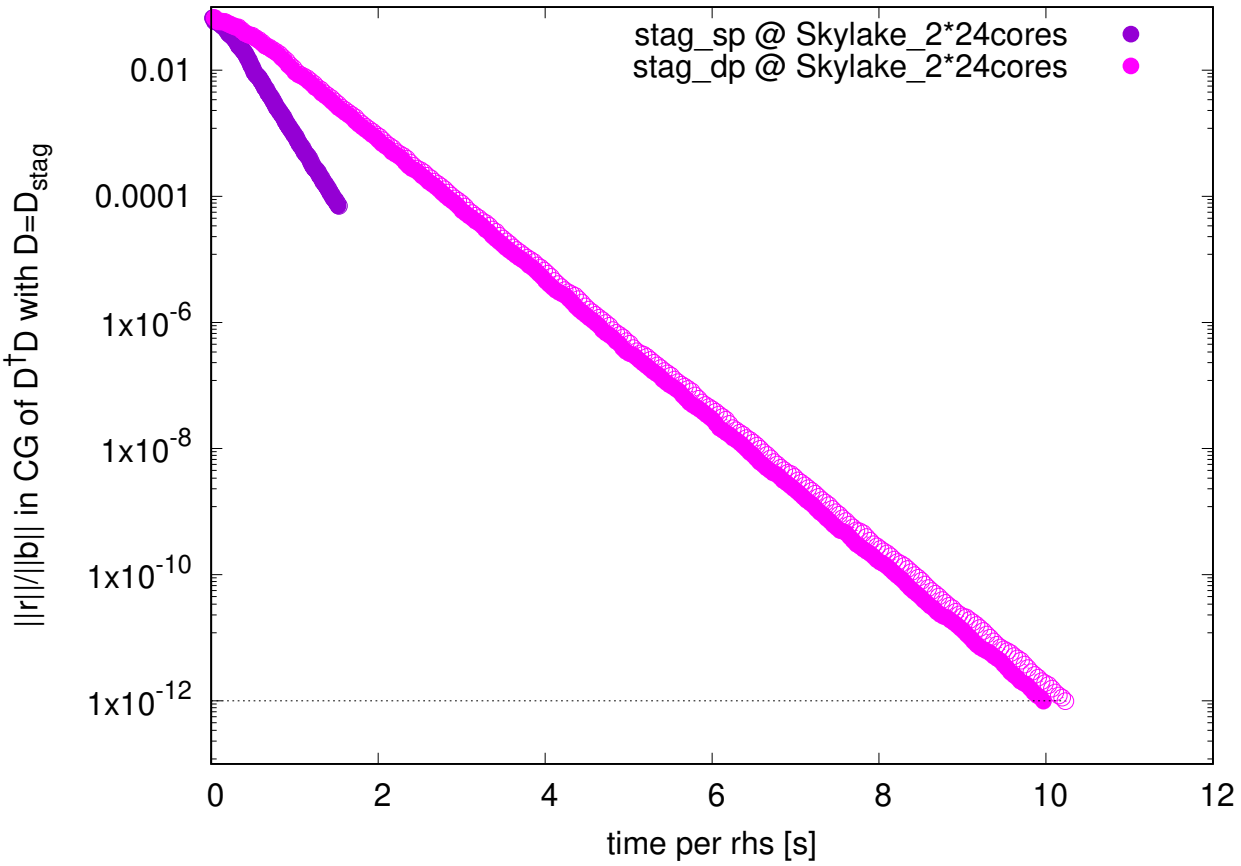


Figure 11: Relative residual norm versus time of the CG solver for the staggered $D_{\rm S}^\dagger D_{\rm S}$ at $am_0 = 0.01$ on a $24^3 \times 48$ lattice in sp/dp (layout `NvNc` filled, other open) on the KNL and Skylake node. In all cases the solver exits after 740 (2489) iterations in sp (dp).

inaccuracy is visible; the six convergence histories in dp are not "horizontally stretched carbon copies" of each other. As was true for the CG algorithm, with the Wilson kernel the convergence on the KNL chip is a bit faster than on the Skylake architecture.

Fig. 13 shows the convergence history of the BiCGstab algorithm for $D_{\mathrm{B}}u = b$. The overall characteristic is similar to the Wilson kernel, i.e. there are some wiggles, but they are not dramatic (at this heavy pion mass). The main difference to the previous plot is the performance ratio between the two architectures. As was true for the CG algorithm, with the Brillouin kernel the convergence on the Skylake architecture is significantly faster than on the KNL chip. This is good news for the suitability of the Brillouin Dirac operator in phenomenological studies on architectures which are limited by memory bandwidth (cf. Refs. [16–18]).

Overall, explicit Krylov space solve-for operations show that the `vec`-layout `NvNsNc` usually fares best in terms of the total run-time. And among the `suv`-layouts `NvNc` beats `NcNv`. These statements hold for to the KNL chip; on the Skylake node the time differences are marginal. This matches the observations made in Secs. 5, 6, 7 for D_{W}, D_{B}, D_{S}, respectively. The linear algebra rankings established in Sec. 3 are not crucial to the overall solver performance. Accordingly, if one were to upgrade the code into a distribution with hybrid parallelization (MPI and OpenMP), one would restrict oneself to the `NvNsNc` and `NvNc` layouts, respectively.

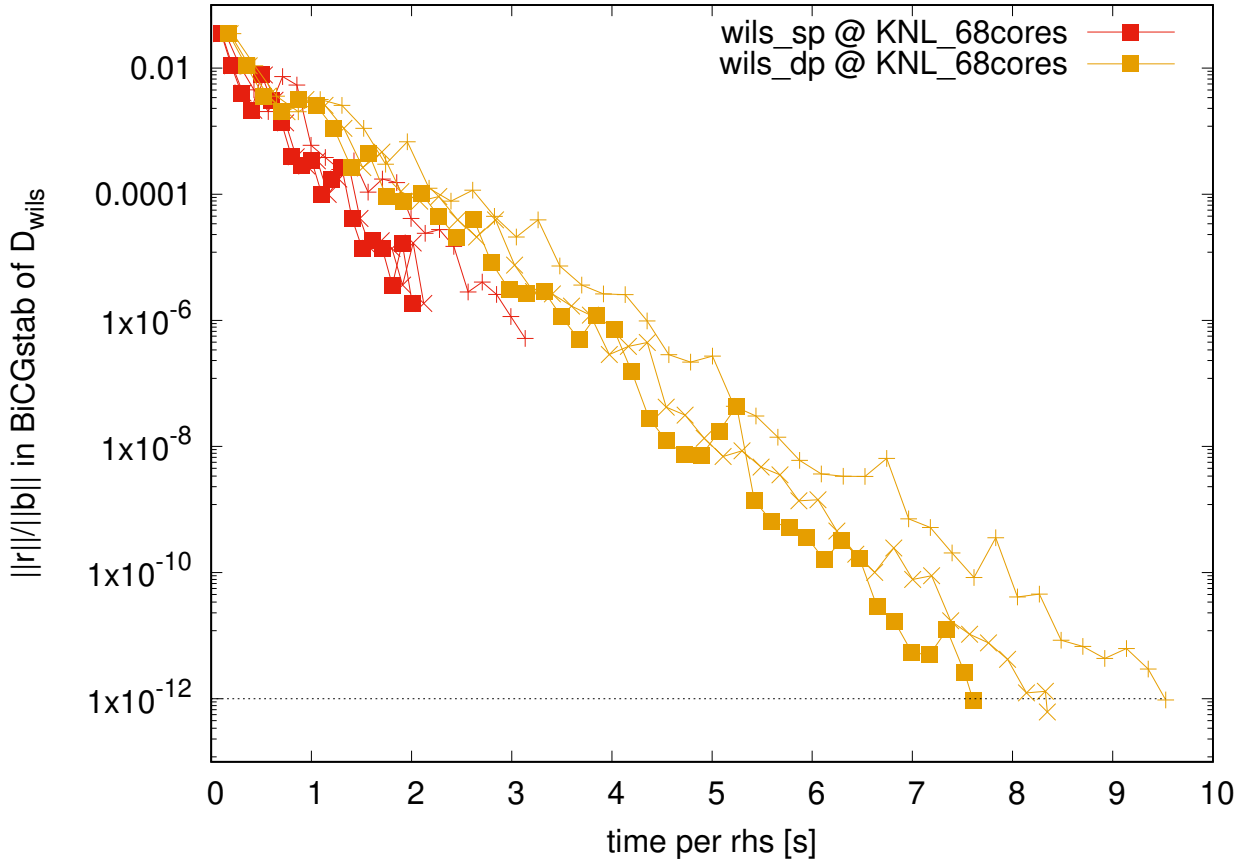
As is well known [45–48] the descent of $||r||$ in the sp-solver can be extended beyond the sp-"limit" $\epsilon \simeq 10^{-6}$ by wrapping the sp-solver into a simple dp-updater (e.g. Richardson iteration). Such "mixed precision solvers" are standard in the lattice community, and the ancillary code distribution contains a "mixed precision" version of each solver presented in this section.

10 Summary

The goal of this article has been to explore whether a "traditional" strategy of implementing the Dirac-matrix times vector operation yields acceptable performance figures on many-core architectures such as the KNL or modern successors. Here "traditional" means that the implementation uses a high-level language (e.g. Fortran 2008, also C/C++ could have done the job), without assembly-tuning, and without cache-line optimization; only OpenMP shared-memory parallelization and SIMD pragmas are used. Furthermore, the freedom to choose the data layout is deliberately limited to the ordering of the internal degrees of freedom (color/spinor/RHS-index). In this way, full portability of the code on CPU architectures is ensured.

On the KNL processor the SIMD-index must be the fastest (in Fortran: first) one, so the layouts `NvNcNs` and `NvNsNc` are preferable for Wilson-type, and `NvNc` for Susskind-type vectors. For the three Dirac operators studied (Wilson, Brillouin and Susskind) acceptable performance figures are found. In sp, they are about 480 Gflop/s, 880 Gflop/s, and 780 Gflop/s, respectively, on one 68-core KNL chip. On the 24-core Skylake architecture (tested in a dual-socket node) the ordering of the internal degrees of freedom seems almost irrelevant, and the sp performance figures for the three operators read 350 Gflop/s, 1320 Gflop/s, and 550 Gflop/s, respectively. In dp these performance figures are roughly halved, but in practice the sp performance is more relevant, since it determines the speed of a mixed-precision solver [45–48]. As an aside it was demonstrated that the relative residual norm $\epsilon = 10^{-12}$ can be reached with dp vectors even if the underlying gauge field is in sp, i.e. if the matrix-vector operation is $v_{\mathrm{dp}} \leftarrow D_{\mathrm{dp}}[U_{\mathrm{sp}}]u_{\mathrm{dp}}$.

Comparing the time-to-solution of a typical CG or BiCGstab inverter on the KNL and on the Skylake node reveals an important difference between the Dirac operators considered. For



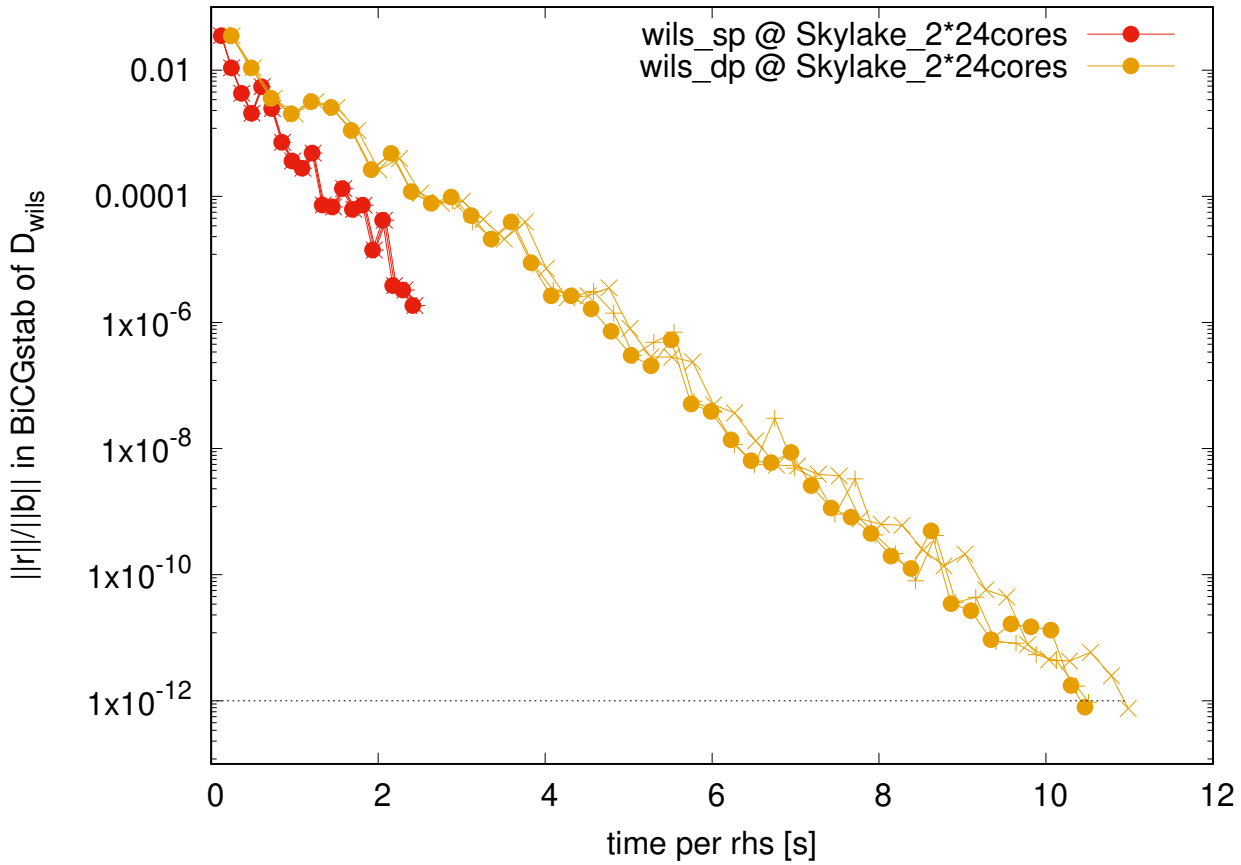
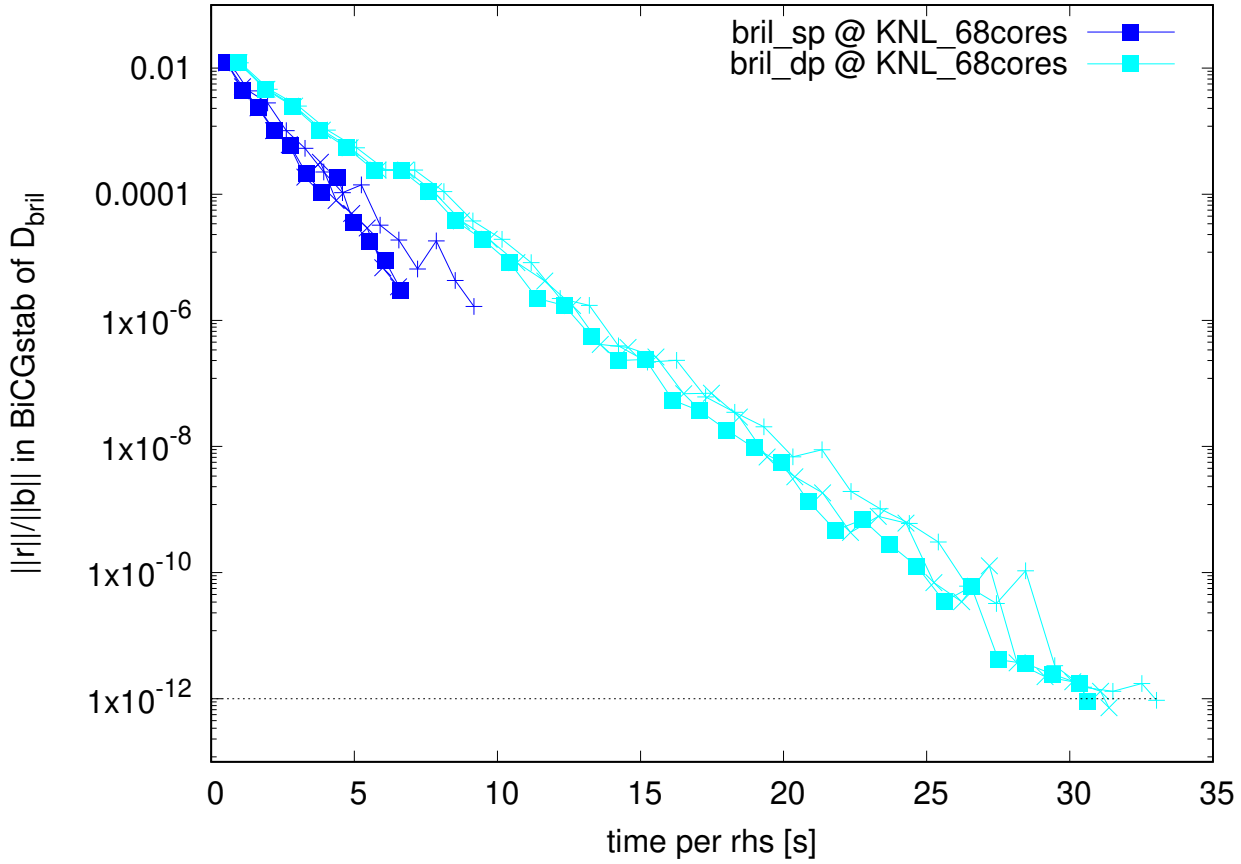


Figure 12: Relative residual norm versus time of the BiCGstab solver for D_{W} at $am_0 = 0.01$, $c_{\mathrm{SW}} = 1$ on a $24^3 \times 48$ lattice in sp/dp (layout `NvNsNc` filled, other $+, \times$) on the KNL and Skylake node. In all cases the solver exits after 200 (~ 435) iterations in sp (dp).



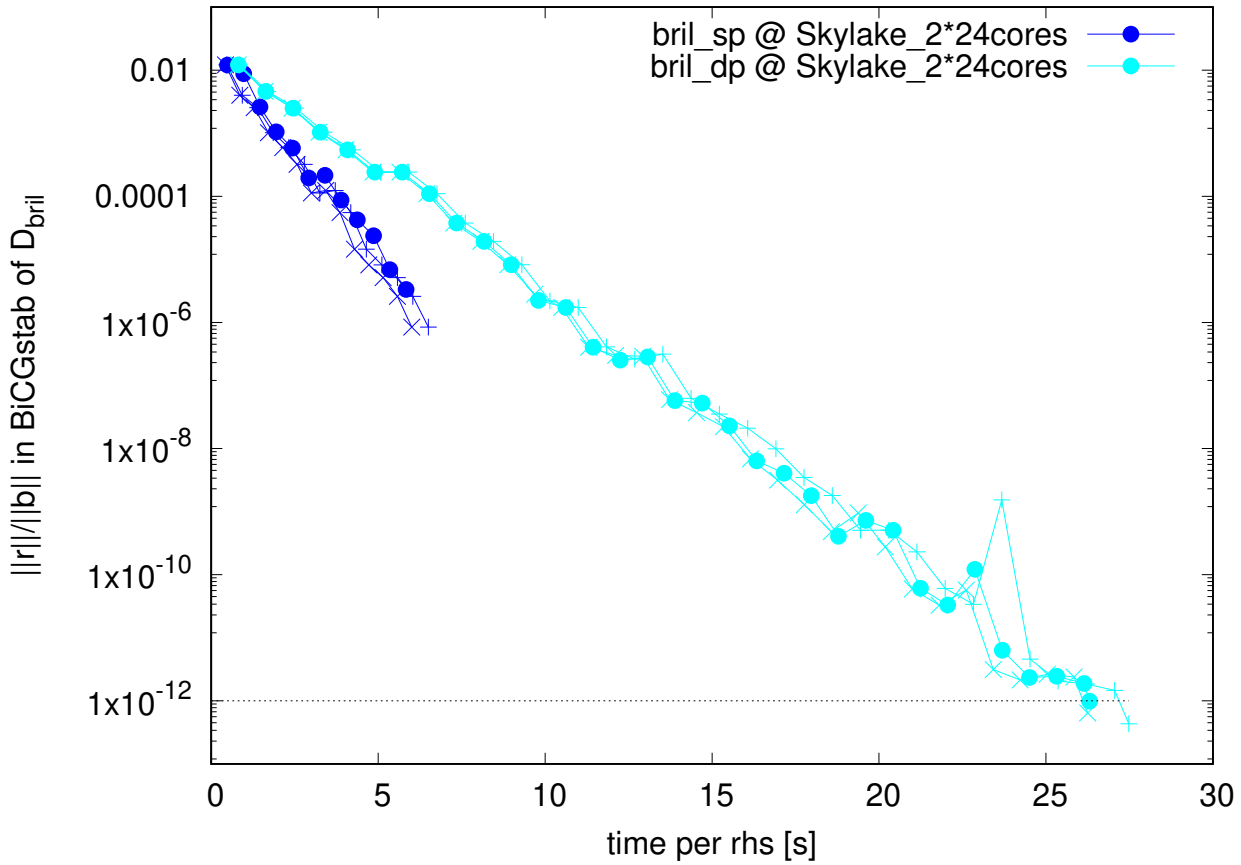


Figure 13: Relative residual norm versus time of the BiCGstab solver for D_{B} at $am_0 = 0.01$, $c_{\mathrm{SW}} = 1$ on a $24^3 \times 48$ lattice in sp/dp (layout `NvNsNc` filled, other $+, \times$) on the KNL and Skylake node. In all cases the solver exits after 140 (~ 322) iterations in sp (dp).

the Wilson and the staggered Dirac operators (and the Wilson Laplacian) the KNL chip turns out to be faster, for the Brillouin Dirac operator (and the Brillouin Laplacian) the Skylake node is faster. This difference is linked to the *computational intensity* of these operators. The Wilson operator uses between 0.89 and 1.53 flops/byte for $N_c = 3$ (depending on the number of RHS, see App. E), the staggered numbers are similar. For Brillouin fermions these landmarks are lifted to 2.21 and 3.83 flops/byte respectively (see App. E). Hence, for operators with a large enough computational intensity the Skylake architecture provides an advantage over KNL. The Brillouin operator (11) is in this category. The future will show whether it can take advantage of the even higher compute-to-bandwidth ratios that future architectures will come with.

The code presented is limited in scope (restriction to shared-memory CPU environments), yet it has unique features (like simple portability), owing to its structural simplicity and design choices which aim to delegate all optimization work to the compiler. The author hopes that it is useful to independent researchers in lattice QCD who want to run it, in farming mode, on small institute clusters. An upgrade into a code base that is suitable to fill large allocations at present and future HPC facilities requires major extensions (both offload capabilities to accelerators like GPUs and multi-node parallelization via MPI need to be added). The author is determined to explore wheter his can be done with the same standards of structural simplicity and hardware independence, and – if successful – to release the result in a future publication.

Acknowledgements

Code development and computations were carried out on DEEP, as well as JURECA and JUWELS at IAS/JSC of the Forschungszentrum Jülich. Access to DEEP was granted through the DEEP-EST Early Access Programme. The author received partial support from the DFG via the collaborative research programme SFB/TRR-55. Useful discussions with Eric Gregory from IAS/JSC of the Forschungszentrum Jülich are gratefully acknowledged.

A Gamma matrices and Wilson projection trick

We employ the "chiral" or Weyl representation of the Euclidean γ-matrices

$$\gamma_1 = \begin{pmatrix} 0 & 0 & 0 & -\mathrm{i} \\ 0 & 0 & -\mathrm{i} & 0 \\ 0 & \mathrm{i} & 0 & 0 \\ \mathrm{i} & 0 & 0 & 0 \end{pmatrix} = \sigma_2 \otimes \sigma_1 \ , \quad \gamma_2 = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} = \sigma_2 \otimes \sigma_2 \ ,$$

$$\gamma_3 = \begin{pmatrix} 0 & 0 & -\mathrm{i} & 0 \\ 0 & 0 & 0 & \mathrm{i} \\ \mathrm{i} & 0 & 0 & 0 \\ 0 & -\mathrm{i} & 0 & 0 \end{pmatrix} = \sigma_2 \otimes \sigma_3 \ , \quad \gamma_4 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} = \sigma_1 \otimes \sigma_0 \ , \tag{18}$$

where the tensor-product notation uses the Pauli matrices

$$\sigma_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \ , \quad \sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \ , \quad \sigma_2 = \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix} \ , \quad \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \ , \tag{19}$$

and the main feature of this representation is the diagonal form of

$$\gamma_5 = \gamma_1\gamma_2\gamma_3\gamma_4 = (\sigma_2\sigma_2\sigma_2\sigma_1) \otimes (\sigma_1\sigma_2\sigma_3\sigma_0) = (-\mathrm{i}\sigma_3) \otimes (\mathrm{i}\sigma_0) = \mathrm{diag}(1, 1, -1, -1) \ . \tag{20}$$

The Wilson Dirac operator (6) multiplies the contribution to the out-vector at site n that comes from $n \pm \hat{\mu}$ with $V_{\pm\mu}(n)$ in color space and with $\frac{1}{2}(1 \pm \gamma_\mu)$ in spinor space. A vector of length $4N_c$ (with the color index moving fastest) is thus multiplied by $\frac{1}{2}(1 \pm \gamma_\mu) \otimes V_{\pm\mu}(n)$, and this is equivalent to the following procedure. Reshape the vector into a $N_c \times 4$ matrix, multiply it with $V_{\pm\mu}(n)$ from the left and with $\frac{1}{2}(1 \pm \gamma_\mu)^{\mathrm{trsp}}$ from the right, and reshape the result back into a column vector. In the right-multiplication we exploit that $\frac{1}{2}(1 \pm \gamma_\mu)$ is a projector, and that either associate eigenspace (to the eigenvalues 0 and 1, respectively) has dimension two. This holds regardless of the representation used, and for each direction μ.

Specifically, for the chiral representation the eigenvector decomposition takes the form

$$\frac{1}{2}(1+\gamma_1) = PP^\dagger,\ \frac{1}{2}(1-\gamma_1) = QQ^\dagger \quad \text{with} \quad P = \frac{1}{\sqrt{2}} \begin{pmatrix} -\mathrm{i} & 0 \\ 0 & 1 \\ 0 & \mathrm{i} \\ 1 & 0 \end{pmatrix},\ Q = \frac{1}{\sqrt{2}} \begin{pmatrix} \mathrm{i} & 0 \\ 0 & 1 \\ 0 & -\mathrm{i} \\ 1 & 0 \end{pmatrix}, \quad (21)$$

$$\frac{1}{2}(1+\gamma_2) = PP^\dagger,\ \frac{1}{2}(1-\gamma_2) = QQ^\dagger \quad \text{with} \quad P = \frac{1}{\sqrt{2}} \begin{pmatrix} -1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix},\ Q = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & 0 \end{pmatrix}, \quad (22)$$

$$\frac{1}{2}(1+\gamma_3) = PP^\dagger,\ \frac{1}{2}(1-\gamma_3) = QQ^\dagger \quad \text{with} \quad P = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 1 \\ \mathrm{i} & 0 \\ 0 & \mathrm{i} \\ 1 & 0 \end{pmatrix},\ Q = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 1 \\ -\mathrm{i} & 0 \\ 0 & -\mathrm{i} \\ 1 & 0 \end{pmatrix}, \quad (23)$$

$$\frac{1}{2}(1+\gamma_4) = PP^\dagger,\ \frac{1}{2}(1-\gamma_4) = QQ^\dagger \quad \text{with} \quad P = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & -1 \\ 1 & 0 \\ 0 & -1 \\ 1 & 0 \end{pmatrix},\ Q = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & -1 \\ -1 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad (24)$$

where P and Q are unique up to arbitrary phases. Hence, the following order of operations saves CPU time. First act with $\sqrt{2}P^*$ from the right, next multiply the resulting $N_c \times 2$ matrix with $\frac{1}{2}V_\mu(n)$ from the left, and finally right-multiply with $\sqrt{2}P^{\mathrm{trsp}}$. This holds for any $\mu > 0$; for a negative μ simply replace P by the respective Q.

Lattice practitioners refer to this procedure as the "spin projection trick" or "shrink expand trick", but it is sparsely documented in the literature (notable exceptions include Refs. [49,50]). On a distributed memory machine, there is an obvious lemma. In case the spinor is part of a halo component which is to be communicated to another MPI rank, the first right-multiplication (by a 4×2 matrix) is done prior to sending, the second one (by a 2×4 matrix) after receipt. This way the number of bits to be communicated is reduced by a factor two (applicable, again, for all directions $\pm\mu$). Finally, the left-multiplication with the color-matrix $V_\mu(n)$ is done in the node where the latter resides (i.e. prior to sending for $\mu < 0$, and after receipt for $\mu > 0$).

Using the tensor notation (18) it is easy to write down the matrices $\sigma_{\mu\nu} = \frac{\mathrm{i}}{2}[\gamma_\mu, \gamma_\nu]$, viz.

$$\begin{aligned}
\sigma_{12} &= \frac{\mathrm{i}}{2}[\sigma_2 \otimes \sigma_1, \sigma_2 \otimes \sigma_2] = \frac{\mathrm{i}}{2}\sigma_2^2 \otimes [\sigma_1, \sigma_2] = \frac{\mathrm{i}}{2}\sigma_0 \otimes +2\mathrm{i}\sigma_3 = -\sigma_0 \otimes \sigma_3 \ , \\
\sigma_{13} &= \frac{\mathrm{i}}{2}[\sigma_2 \otimes \sigma_1, \sigma_2 \otimes \sigma_3] = \frac{\mathrm{i}}{2}\sigma_2^2 \otimes [\sigma_1, \sigma_3] = \frac{\mathrm{i}}{2}\sigma_0 \otimes -2\mathrm{i}\sigma_2 = +\sigma_0 \otimes \sigma_2 \ , \\
\sigma_{14} &= \frac{\mathrm{i}}{2}[\sigma_2 \otimes \sigma_1, \sigma_1 \otimes \sigma_0] = \frac{\mathrm{i}}{2}[\sigma_2\sigma_1 \otimes \sigma_1\sigma_0 - \sigma_1\sigma_2 \otimes \sigma_0\sigma_1] = +\sigma_3 \otimes \sigma_1 \ , \\
\sigma_{23} &= \frac{\mathrm{i}}{2}[\sigma_2 \otimes \sigma_2, \sigma_2 \otimes \sigma_3] = \frac{\mathrm{i}}{2}\sigma_2^2 \otimes [\sigma_2, \sigma_3] = \frac{\mathrm{i}}{2}\sigma_0 \otimes +2\mathrm{i}\sigma_1 = -\sigma_0 \otimes \sigma_1 \ , \\
\sigma_{24} &= \frac{\mathrm{i}}{2}[\sigma_2 \otimes \sigma_2, \sigma_1 \otimes \sigma_0] = \frac{\mathrm{i}}{2}[\sigma_2\sigma_1 \otimes \sigma_2\sigma_0 - \sigma_1\sigma_2 \otimes \sigma_0\sigma_2] = +\sigma_3 \otimes \sigma_2 \ , \\
\sigma_{34} &= \frac{\mathrm{i}}{2}[\sigma_2 \otimes \sigma_3, \sigma_1 \otimes \sigma_0] = \frac{\mathrm{i}}{2}[\sigma_2\sigma_1 \otimes \sigma_3\sigma_0 - \sigma_1\sigma_2 \otimes \sigma_0\sigma_3] = +\sigma_3 \otimes \sigma_3 \ ,
\end{aligned} \tag{25}$$

and this explicit form is used in the clover routine, see Sec. 4.

B Details of the Wilson Laplace and Dirac routines

The structure of the Wilson Laplace routine (8) for the vector layout `NvNsNc` is

```
!$OMP PARALLEL DO COLLAPSE(2) DEFAULT(private) FIRSTPRIVATE(mass) SHARED(old,new,V)
do t=1,Nt
do z=1,Nz; z_min=... ; z_plu=... ; t_min=modulo(t-2,Nt)+1; t_plu=modulo(t,Nt)+1
do y=1,Ny; y_min=... ; y_plu=...
do x=1,Nx; x_min=... ; x_plu=...
   !!! direction 0 gets mass term
   do concurrent(col=1:Nc,spi=1:4,rhs=1:Nv)
      site(rhs,spi,col)=(8.0+mass**2)*old(rhs,spi,col,x,y,z,t)
   end do
   !!! add contributions from -4 and +4 directions
   do concurrent(ccc=1:Nc,spi=1:4)
      !$OMP SIMD
      do rhs=1,Nv
         site(rhs,spi,:)-=conjg(V(ccc,:,4,x,y,z,t_min))*old(rhs,spi,ccc,x,y,z,t_min)
         site(rhs,spi,:)-=      V(:,ccc,4,x,y,z,t    ) *old(rhs,spi,ccc,x,y,z,t_plu)
      end do
   end do
   ...
   !!! plug site into new vector
   forall(col=1:Nc,spi=1:4,rhs=1:Nv) new(rhs,spi,col,x,y,z,t)=0.5*site(rhs,spi,col)
end do ! x=1,Nx
end do ! y=1,Ny
end do ! z=1,Nz
end do ! t=1,Nt
!$OMP END PARALLEL DO
```

where we use a Fortran inspired notation. The variables `old,new,V` are shared among the threads, while `mass` is copied from the master thread at the bifurcation point. In a serial code

the variables `t_min`, `t_plu` would be computed in the first line within the t-loop. The clause `COLLAPSE(2)` forces us to transfer these statements into the z-loop. In the same line the clause `SCHEDULE(static)` may be added to enforce compile-time thread scheduling. In Fortran the notation `site-=...` is wishful thinking; it is `site=site-...` properly spelled out. Within the SIMD loop the stride notation establishes an implicit `forall(col=1:Nc) ...` construct, which is a one-line version of `do concurrent(col=1:Nc); ...; end do`. Taking everything together we thus have eight nested loops. The line with dots only indicates that the other six directions are implemented analogously. In Fortran this routine must be implemented separately for each layout `NcNsNv`, `NsNcNv`, `NcNvNs`, `NvNcNs`, `NsNvNc`, `NvNsNc`, and for `old,new` being sp or dp. Finally, a "wrapper routine" may be written to call them in a more convenient way.

The structure of the Wilson Dirac routine (6) for the vector layout `NvNsNc` is the same, except minor differences. The initialization of `site` uses the factor `(8.0+2.0*mass)`, since there is a factor 0.5 in the end. The most significant change is due to the "Wilson projection trick" discussed in App. A. For the ∓ 4 directions the respective lines above are replaced by

```
do ccc=1,Nc
   !$OMP SIMD PRIVATE(red,obj)
   do rhs=1,Nv
      red(1)=old(rhs,1,ccc,x,y,z,t_min)+     old(rhs,3,ccc,x,y,z,t_min)
      red(2)=old(rhs,2,ccc,x,y,z,t_min)+     old(rhs,4,ccc,x,y,z,t_min)
      red(3)=old(rhs,1,ccc,x,y,z,t_plu)-     old(rhs,3,ccc,x,y,z,t_plu)
      red(4)=old(rhs,2,ccc,x,y,z,t_plu)-     old(rhs,4,ccc,x,y,z,t_plu)
      forall(col=1:Nc,spi=1:2) obj(spi,col)=red(spi)*conjg(V(ccc,col,4,x,y,z,t_min))
      forall(col=1:Nc,spi=3:4) obj(spi,col)=red(spi)*      V(col,ccc,4,x,y,z,t    )
      site(rhs,1,:)=site(rhs,1,:)-     (obj(1,:)+obj(3,:))
      site(rhs,2,:)=site(rhs,2,:)-     (obj(2,:)+obj(4,:))
      site(rhs,3,:)=site(rhs,3,:)-     (obj(1,:)-obj(3,:))
      site(rhs,4,:)=site(rhs,4,:)-     (obj(2,:)-obj(4,:))
   end do
end do
```

and similarly for the other directions. The blank spaces mark positions where factors `i_sp*` would show up for some of the other directions (with details depending on the choice of the Dirac matrices, see App. A). Beyond this there is no change.

Overall, these implementations are compellingly simple, and it is perhaps a little surprising to the expert that such simple routines can yield the timings reported in Sec. 5.

C Details of the Brillouin Laplace and Dirac routines

The structure of the Brillouin Laplace routine (13) for the vector layout `NvNsNc` is

```
!$OMP PARALLEL DO COLLAPSE(3) DEFAULT(private) FIRSTPRIVATE(mass) SHARED(old,new,W)
do t=1,Nt
do z=1,Nz
do y=1,Ny
do x=1,Nx
   site(:,:,:)=cmplx(0.0,kind=sp)
   !!! add laplacian parts to site
```

```
   dir=0
   do go_t=-1,1; tsh=modulo(t+go_t-1,Nt)+1
   do go_z=-1,1; zsh=modulo(z+go_z-1,Nz)+1
   do go_y=-1,1; ysh=modulo(y+go_y-1,Ny)+1
   do go_x=-1,1; xsh=modulo(x+go_x-1,Nx)+1
      dir=dir+1 !!! note: dir=(go_t+1)*27+(go_z+1)*9+(go_y+1)*3+(go_x+2)
      if (dir<41) then
         tmp(:,:)=mask_lap(   dir)*W(:,:,dir,x,y,z,t)
      else if (dir>41) then
         tmp(:,:)=mask_lap(82-dir)*conjg(transpose(W(:,:,82-dir,xsh,ysh,zsh,tsh)))
      else
         tmp(:,:)=(1.875+0.5*mass**2)*eye(:,:)
      end if
      do ccc=1,Nc
         !$OMP SIMD PRIVATE(help)
         do rhs=1,Nv
            help(:)=old(rhs,:,ccc,xsh,ysh,zsh,tsh)
            do col=1,Nc
               site(rhs,:,col)=site(rhs,:,col)-help(:)*tmp(col,ccc)
            end do
         end do
      end do
   end do ! go_x=-1,1
   end do ! go_y=-1,1
   end do ! go_z=-1,1
   end do ! go_t=-1,1
   !!! plug site into new vector
   forall(col=1:Nc,spi=1:04,rhs=1:Nv) new(rhs,spi,col,x,y,z,t)=site(rhs,spi,col)
end do ! x=1,Nx
end do ! y=1,Ny
end do ! z=1,Nz
end do ! t=1,Nt
!$OMP END PARALLEL DO
```

where the variables `old,new,W` are shared among the threads, while `mass` is copied from the master thread at the bifurcation point. The clause `SCHEDULE(dynamic)` may be added to this line to enforce run-time thread scheduling. The four nested `go` loops organize the harvesting within the 3^4 hypercube around the point $n = (x, y, z, t)$ of $\psi_{\rm new}(n)$. The factors $(\frac{1}{2}\lambda_1, \frac{1}{2}\lambda_2, \frac{1}{2}\lambda_3, \frac{1}{2}\lambda_4)$ are stored in `mask_lap(1:40)`; this is faster than evaluating `count([go_x,go_y,go_z,go_t].ne.0)` and accessing a table with just four elements. The factor $\frac{1}{2}\lambda_0 + \frac{1}{2}m_0^2$ multiplies the $N_c \times N_c$ identity matrix stored in `eye(1:Nc,1:Nc)`. Within the SIMD loop there is an explicit color-loop, and the stride-notation establishes an implicit `forall(spi=1:04)` construct. Taking everything together we thus have twelve nested loops.

The structure of the Brillouin Dirac routine (11) for the vector layout `NvNsNc` is the same, except minor differences. The factor `mask_lap(min(dir,82-dir))` is not included in `tmp` but stored in the variable `lap`. The variable `absgo_xyzt` is loaded from an array `mask_absgo(1:40)`, since this is faster than evaluating `count([go_x,go_y,go_z,go_t].ne.0)`. In fact, with this variable in hand, one can take `lap` from an array `mask_lap(0:4)`. Similarly, the derivatives are put together from an array `mask_der(-1:1,0:4)`. Hence, the `ccc` loop above is replaced by

```
lap=mask_lap(absgo_xyzt)
der_tz=cmplx(mask_der(go_t,absgo_xyzt),mask_der(go_z,absgo_xyzt),kind=sp)
der_yx=cmplx(mask_der(go_y,absgo_xyzt),mask_der(go_x,absgo_xyzt),kind=sp)
do ccc=1,Nc
   !$OMP SIMD PRIVATE(myold,help)
   do rhs=1,Nv
      myold(:)=old(rhs,:,ccc,xsh,ysh,zsh,tsh)
      help(1)=-lap*myold(1)+conjg(der_tz)*myold(3)-      der_yx *myold(4)
      help(2)=-lap*myold(2)+conjg(der_yx)*myold(3)+      der_tz *myold(4)
      help(3)=-lap*myold(3)+      der_tz *myold(1)+      der_yx *myold(2)
      help(4)=-lap*myold(4)-conjg(der_yx)*myold(1)+conjg(der_tz)*myold(2)
      do col=1,Nc
         site(rhs,:,col)=site(rhs,:,col)+help(:)*tmp(col,ccc)
      end do
   end do
end do
```

and taking everything together, we end up with a set of twelve nested loops.

Overall, these implementations are fairly straightforward. The timings of these routines, for $N_c = 3$ and $N_v = 12$ on a $34^3 \times 68$ lattice, are discussed in Sec. 6.

D Details of the Susskind "staggered" Dirac routine

The structure of the staggered Dirac routine for the layout `NvNc` is

```
!$OMP PARALLEL DO DEFAULT(private) FIRSTPRIVATE(mass) SHARED(old,new,V)
do t=1,Nt; t_min=...; t_plu=...; eta4=1.000
do z=1,Nz; z_min=...; z_plu=...; eta4=-eta4; eta3=1.000
do y=1,Ny; y_min=...; y_plu=...; eta4=-eta4; eta3=-eta3; eta2=1.000
do x=1,Nx; x_min=...; x_plu=...; eta4=-eta4; eta3=-eta3; eta2=-eta2
   !!! direction 0 gets mass term
   forall(col=1:Nc,rhs=1:Nv) site(rhs,col)=(2.0*mass)*old(rhs,col,x,y,z,t)
   !!! add non-trivial directions
   do ccc=1,Nc
      !$OMP SIMD
      do rhs=1,Nv
         site(rhs,:)-=eta4*conjg(V(ccc,:,4,x,y,z,t_min))*old(rhs,ccc,x,y,z,t_min)
         site(rhs,:)+=eta4*      V(:,ccc,4,x,y,z,t    ) *old(rhs,ccc,x,y,z,t_plu)
      end do
   end do
   ...
   !!! plug site into new vector
   forall(col=1:Nc,rhs=1:Nv) new(rhs,col,x,y,z,t)=0.5_sp*site(rhs,col)
end do ! x=1,Nx
end do ! y=1,Ny
end do ! z=1,Nz
end do ! t=1,Nt
!$OMP END PARALLEL DO
```

where the variables `old,new,V` are shared among the threads, while `mass` is copied from the master thread at the bifurcation point. In the actual code there is the clause `COLLAPSE(3)` in the `PARALLEL DO` construct, and this delays the initialization of `eta3,eta4` by one or two loops, respectively. In the same line the clause `SCHEDULE(static)` is added to enforce compile-time thread scheduling. The line with dots only indicates that the other six directions are implemented analogously. In the end, the variable `site(1:Nv,1:Nc)` is written, with a factor $\frac{1}{2}$, into the respective field of `new`. Within the SIMD loop the stride notation establishes an implicit `forall(col=1:Nc)` construct. In total there are thus seven nested loops.

E Flop count and memory traffic details

It is common practice to count the number of additions and multiplications, coined "floating-point operations" (flop). In view of the capabilities of modern processors it would make sense to count fused-multiply-add operations, but for backward compatibility everyone adopts the traditional counting rule. Hence a complex-plus-complex addition takes 2 flops, a real-times-complex multiplication takes 2 flops, and a complex-times-complex multiplication takes 6 flops. Accordingly, a $SU(N_c)$ left-multiplication of a $N_c \times 4$ Dirac spinor takes $N_c^2 \times 4$ multiplications and $N_c(N_c-1) \times 4$ additions, which amounts to $[8N_c^2 - 2N_c] \times 4$ flops. Based on this we arrive at the following flop counts for the operators considered in this article.

E.1 Wilson Laplace operator

A matrix-vector operation with the hpd operator $-\frac{1}{2}\triangle^{\mathrm{std}} + \frac{1}{2}m_0^2$ proceeds as follows:

(i) $SU(N_c)$-multiply the $N_c \times 4$ block for each direction. In this step we assume the prefactor $-\frac{1}{2}$ is included into the local copy of $V_\mu(n)$ and thus consider it for free. Overall, this takes $[8N_c^2 - 2N_c] \cdot 4 \cdot 8$ flops.

(ii) Accumulate these 8 terms, as well as the 0-hop contribution which is multiplied with the precomputed factor $(4 + \frac{1}{2}m_0^2)$. Overall, this takes $2N_c \cdot 4 \cdot 9$ flops.

All together we have a grand total of $256N_c^2 + 8N_c$ flops per site, or 2328 flops for $SU(3)$.

Here we assume the vector has the same spinor⊗color structure per lattice site as for the Wilson Dirac operator (see E.2). In the event the vector has no spinor degree of freedom, the flop count is four-fold reduced, i.e. $64N_c^2 + 2N_c$ flops per site, or 582 for $SU(3)$.

In addition, the following memory access operations are due (per site, for one RHS):

(i) Read one color-spinor object for each point of the 9-point stencil from the "in" vector; this requires $2N_c \cdot 4 \cdot 9$ floats (doubles) in sp (dp).

(ii) Read one $SU(N_c)$ matrix[29] (in this code suite always in sp, cf. the discussion in Sec. 2) per direction; this requires $2N_c^2 \cdot 8$ floats.

(iii) Write one color-spinor object to the "out" vector; this requires $2N_c \cdot 4$ floats (doubles) in sp (dp).

[29] Based on unitarity, one could omit the last row or column, and reconstruct it "on the fly"; this reduces the load to $2N_c(N_c-1) \cdot 8$ floats. This does usually pay off for $N_c = 3$ but barely so for higher N_c, and this is why no gauge compression is used in this code. Using the matrix "exp" and "log" functions, one could even reduce this number to $(N_c^2 - 1) \cdot 8$ floats, but the latter function is tricky to implement (for arbitrary N_c).

With several RHS, the memory footprint in (i) and (iii) is multiplied by N_v, while (ii) stays unchanged. All together we thus arrive at $80N_cN_v + 16N_c^2$ floats of traffic per site in sp (the number in dp follows by replacing $N_v \to 2N_v$). For $N_c = 3, N_v = 1$ the 2328 flops and 384 floats per site amount to a computational intensity of 1.52 flops/byte in sp. For $N_c = 3, N_v = 12$ the 27936 flops and 3024 floats per site yield 2.31 flops/byte in sp. Increasing N_v increases the computational intensity, but there is an asymptotic bound of $(256N_c^2+8N_c)/(320N_c)$ flops/byte in sp for $N_v \to \infty$, which evaluates to 2.42 flops/byte for $N_c = 3$.

E.2 Wilson Dirac operator

A matrix-vector operation with the Wilson Dirac operator D_{W} proceeds as follows:

(i) Spin project (from 4 to 2 components) the $N_c \times 4$ matrix for each direction. Overall, this takes $N_c \cdot 4 \cdot 8$ flops.

(ii) $SU(N_c)$-multiply the $N_c \times 2$ block for each direction, and expand it back to the $N_c \times 4$ format (for free). Overall, this takes $[8N_c^2 - 2N_c] \cdot 2 \cdot 8$ flops.

(iii) Accumulate these 8 terms, as well as the 0-hop contribution which uses the precomputed factor $(4 + m_0)$. Overall, this takes $2N_c \cdot 4 \cdot 9$ flops.

All together we have a grand total of $128N_c^2 + 72N_c$ flops per site, or 1368 flops for $SU(3)$.

In the older literature a different normalization of the Dirac operator was used, where the factor $(4 + m_0)$ is absent. In this case only $2N_c \cdot 4 \cdot 8$ flops are spent in step (iii), and the grand total amounts to $128N_c^2 + 64N_c$ flops, or 1344 flops for $SU(3)$. Sometimes the Dirac operator *without* the 0-hop contribution is considered. In this case step (iii) uses only $2N_c \cdot 4 \cdot 7$ flops, and the grand total amounts to $128N_c^2 + 56N_c$ flops, or 1320 flops for $SU(3)$.

In addition, for a matrix-vector operation we must perform the same memory operations as for the Wilson Laplace operator, see E.1. This was $80N_cN_v + 16N_c^2$ floats of traffic per site in sp (in dp the first term doubles). For $N_c = 3, N_v = 1$ the 1368 flops and 384 floats per site amount to a computational intensity of 0.89 flops/byte in sp. For $N_c = 3, N_v = 12$ the 16416 flops and 3024 floats per site yield 1.36 flops/byte in sp. Again increasing N_v increases the computational intensity, but there is an asymptotic bound of $(128N_c^2 + 72N_c)/(320N_c)$ flops/byte in sp for $N_v \to \infty$, which evaluates to 1.53 flops/byte for $N_c = 3$.

E.3 Brillouin Laplace operator

A matrix-vector operation with the hpd operator $-\frac{1}{2}\triangle^{\mathrm{bri}} + \frac{1}{2}m_0^2$ proceeds as follows:

(i) $SU(N_c)$-multiply the $N_c \times 4$ block for each of the 80 non-trivial directions. Overall, this takes $[8N_c^2 - 2N_c] \cdot 4 \cdot 80$ flops.

(ii) Multiply the resulting $N_c \times 4$ matrix for 81 directions with the weight factor as given by the Brillouin Laplacian. This weight factor is real, and the mass term may be incorporated into the 0-hop contribution of the Laplacian. Overall, this takes $N_c \cdot 4 \cdot 2 \cdot 81$ flops.

(iii) Accumulate the 81 contributions to the out-spinor. Overall, this takes $2N_c \cdot 4 \cdot 80$ flops.

All together we have a grand total of $2560N_c^2 + 648N_c$ flops per site, or 24984 flops for $SU(3)$.

Here we assume the vector has the same spinor⊗color structure per lattice site as for the Brillouin Dirac operator (see E.4). In the event the vector has no spinor degree of freedom, the flop count is four-fold reduced, i.e. $640N_c^2 + 162N_c$ flops per site, or 6246 for $SU(3)$.

In addition, for a matrix-vector operation we must (per site, for one RHS):

(i) Read one color-spinor object for each point of the 81-point stencil from the "in" vector; this requires $2N_c \cdot 4 \cdot 81$ floats (doubles) in sp (dp).

(ii) Read one complex $N_c \times N_c$ matrix[30] (in this code suite always in sp, cf. the discussion in Sec. 2) per non-trivial direction; this requires $2 \cdot N_c^2 \cdot 80$ floats.

(iii) Write one color-spinor object to the "out" vector; this requires $2N_c \cdot 4$ floats (doubles) in sp (dp).

With several RHS, the memory footprint in (i) and (iii) is multiplied by N_v, while (ii) stays unchanged. All together we thus arrive at $656N_cN_v + 160N_c^2$ floats of traffic per site in sp (the number in dp follows by replacing $N_v \to 2N_v$). For $N_c = 3, N_v = 1$ the 24984 flops and 3408 floats per site amount to a computational intensity of 1.83 flops/byte in sp. For $N_c = 3, N_v = 12$ the 299808 flops and 25056 floats per site yield 2.99 flops/byte in sp. Again increasing N_v increases the computational intensity, but there is an asymptotic bound of $(2560N_c^2 + 648N_c)/(2624N_c)$ flops/byte in sp for $N_v \to \infty$, which evaluates to 3.17 flops/byte for $N_c = 3$.

E.4 Brillouin Dirac operator

A matrix-vector operation with the Brillouin Dirac operator D_{B} proceeds as follows:

(i) $SU(N_c)$-multiply the $N_c \times 4$ block for each of the 80 non-trivial directions. Overall, this takes $[8N_c^2 - 2N_c] \cdot 4 \cdot 80$ flops.

(ii) Multiply the resulting $N_c \times 4$ matrix with the weight factors as given by the isotropic derivatives and the Brillouin Laplacian. These weight factors are either real or purely imaginary, and for each $\nabla_\mu^{\mathrm{iso}}$ non-zero only for 54 out of the 81 directions. The mass term may be incorporated into the 0-hop contribution of the Laplacian. Overall, this takes $2N_c \cdot 4 \cdot (4 \cdot 54 + 81)$ flops.

(iii) Accumulate the 81 contributions to the out-spinor. Overall, this takes $2N_c \cdot 4 \cdot 80$ flops.

All together we have a grand total of $2560N_c^2 + 2376N_c$ flops per site, or 30168 flops for $SU(3)$.

In addition, for a matrix-vector operation we must perform the same memory operations as for the Brillouin Laplace operator, see E.3. This was $656N_cN_v + 160N_c^2$ floats of traffic per site in sp (in dp the first term doubles). For $N_c = 3, N_v = 1$ the 30168 flops and 3408 floats per site amount to a computational intensity of 2.21 flops/byte in sp. For $N_c = 3, N_v = 12$ the 362016 flops and 25056 floats per site yield 3.61 flops/byte in sp. Again increasing N_v increases the computational intensity, but there is an asymptotic bound of $(2560N_c^2 + 2376N_c)/(2624N_c)$ flops/byte in sp for $N_v \to \infty$, which evaluates to 3.83 flops/byte for $N_c = 3$.

[30] For 36 of the 40 stored directions this matrix is *not unitary*; compare the caption of Tab. 13.

E.5 Susskind "staggered" Dirac operator

In terms of the flop count a matrix-vector operation with the "staggered" Dirac operator D_{S} mimics an application of the Wilson Laplacian on a scalar argument (see E.1). The staggering brings factors of -1, which are considered for free, and the center-point of the stencil is multiplied by m_0 rather than by $4 + \frac{1}{2}m_0^2$, but this does not affect the flop count either. The final result is thus still $64N_c^2 + 2N_c$ flops per site, or 582 flops for $SU(3)$.

Also the memory requirement is a quarter of the number given in E.1, i.e. $20N_cN_v + 4N_c^2$ floats of traffic per site in sp (the first term doubles in dp) with N_v RHS. The computational intensity is thus $(64N_c^2N_v + 2N_cN_v)/(80N_cN_v + 16N_c^2)$ flops/byte in sp. For $N_c = 3, N_v = 1$ this yields $(64N_c + 2)/(80 + 16N_c) \simeq 1.52$ flops/byte in sp. For $N_c = 3, N_v = 12$ this yields $(768N_c + 24)/(960 + 16N_c) \simeq 2.31$ flops/byte in sp. The asymptotic bound for $N_v \to \infty$ is again $(64N_c^2 + 2N_c)/(80N_c)$ flops/byte in sp, which evaluates to 2.42 flops/byte for $N_c = 3$.

E.6 Clover improvement operator

A matrix-vector operation with (4, 5), using the precomputed $F_{\mu\nu}(n)$, proceeds as follows:

(i) $SU(N_c)$-multiply the $N_c \times 4$ block for any of the 6 clover orientations (specified by $1 \leq \mu < \nu \leq 4$) at the given space-time position. Overall, this takes $[8N_c^2 - 2N_c] \cdot 4 \cdot 6$ flops.

(ii) Add the 6 contributions per color and spinor index to the out-vector (here we use that each $\sigma_{\mu\nu}$ has one non-zero entry in spinor space per row or column; factors of i are considered for free). Overall, this takes $2N_c \cdot 4 \cdot 6$ flops.

All together we have a grand total of $192N_c^2$ flops per site, or 1728 flops for $SU(3)$.

In the interest of speed the special properties of the $\sigma_{\mu\nu}$ matrices can be exploited (see App. A). Ignoring the numerical effort to form the linear combinations of the $F_{\mu\nu}$ matrices (justified for large enough N_c), the effective number of $SU(N_c)$-multiplications is reduced to two. In this case one ends up with $[8N_c^2 - 2N_c] \cdot 4 \cdot 2$ flops under (i), and $2N_c \cdot 4 \cdot 2$ flops under (ii). All together this yields a grand total of $64N_c^2$ flops per site, or 576 flops for $SU(3)$. The ancillary code distribution uses this trick, and hence the smaller number when a timing information is converted to a Gflop/s figure for the clover routine or for $D_{\mathrm{W}}, D_{\mathrm{B}}$ at $c_{\mathrm{SW}} > 0$.

References

[1] K. G. Wilson, Phys. Rev. D **10**, 2445-2459 (1974), doi:10.1103/PhysRevD.10.2445.

[2] J. B. Kogut and L. Susskind, Phys. Rev. D **11**, 395-408 (1975), doi:10.1103/PhysRevD.11.395.

[3] L. Susskind, Phys. Rev. D **16**, 3031-3039 (1977), doi:10.1103/PhysRevD.16.3031.

[4] M. R. Hestenes and E. Stiefel, J. Res. Nat. Bur. Stand. **49**, 409-436 (1952), doi:10.6028/jres.049.044.

[5] H. A. Van der Vorst, SIAM J. Sci. Stat. Comput. **13**, 631-644 (1992), doi:10.1137/0913035.

[6] Y. Saad, "Iterative Methods for Sparse Linear Systems" (2nd ed.), SIAM (1992/2003), doi:10.2277/0898715342.

[7] M. Luscher, [arXiv:1002.4232 [hep-lat]].

[8] I. Dasgupta, A. Ruben Levi, V. Lubicz and C. Rebbi, Comput. Phys. Commun. **98**, 365-397 (1996), doi:10.1016/0010-4655(96)00103-8 [arXiv:hep-lat/9605012 [hep-lat]].

[9] L. Del Debbio, L. Giusti, M. Luscher, R. Petronzio and N. Tantalo, JHEP **02**, 056 (2007), doi:10.1088/1126-6708/2007/02/056 [arXiv:hep-lat/0610059 [hep-lat]].

[10] L. Del Debbio, L. Giusti, M. Luscher, R. Petronzio and N. Tantalo, JHEP **02**, 082 (2007), doi:10.1088/1126-6708/2007/02/082 [arXiv:hep-lat/0701009 [hep-lat]].

[11] I. Campos *et al.* [RC*], Eur. Phys. J. C **80**, no.3, 195 (2020), doi:10.1140/epjc/s10052-020-7617-3 [arXiv:1908.11673 [hep-lat]].

[12] M. Bach, V. Lindenstruth, O. Philipsen and C. Pinke, Comput. Phys. Commun. **184**, 2042-2052 (2013) doi:10.1016/j.cpc.2013.03.020 [arXiv:1209.5942 [hep-lat]].

[13] A. Bazavov *et al.* [Fermilab Lattice and MILC], Phys. Rev. D **90**, no.7, 074509 (2014), doi:10.1103/PhysRevD.90.074509 [arXiv:1407.3772 [hep-lat]].

[14] P. Boyle, A. Yamaguchi, G. Cossu and A. Portelli, [arXiv:1512.03487 [hep-lat]].

[15] L. Altenkort, D. Bollweg, D. A. Clarke, O. Kaczmarek, L. Mazur, C. Schmidt, P. Scior and H. T. Shu, [arXiv:2111.10354 [hep-lat]].

[16] S. Durr and G. Koutsou, Phys. Rev. D **83**, 114512 (2011), doi:10.1103/PhysRevD.83.114512 [arXiv:1012.3615 [hep-lat]].

[17] S. Durr, G. Koutsou and T. Lippert, Phys. Rev. D **86**, 114514 (2012), doi:10.1103/PhysRevD.86.114514 [arXiv:1208.6270 [hep-lat]].

[18] S. Durr and G. Koutsou, [arXiv:1701.00726 [hep-lat]].

[19] P. Majumdar, J. Phys. Conf. Ser. **759**, no. 1, 012070 (2016), doi:10.1088/1742-6596/759/1/012070.

[20] P. A. Boyle, PoS LATTICE **2016**, 013 (2017), doi:10.22323/1.256.0013 [arXiv:1702.00208 [hep-lat]].

[21] A. Rago, EPJ Web Conf. **175**, 01021 (2018), doi:10.1051/epjconf/201817501021 [arXiv:1711.01182 [hep-lat]].

[22] M. Lin, plenary talk at Lattice 2018 (unpublished).

[23] E. Gregory, PoS LATTICE **2019**, 205 (2020), doi:10.22323/1.363.0205.

[24] P. Arts *et al.*, PoS LATTICE **2014**, 021 (2015), doi:10.22323/1.214.0021 [arXiv:1502.04025 [cs.DC]].

[25] S. Heybrock, M. Rottmann, P. Georg and T. Wettig, PoS **LATTICE2015**, 036 (2016), doi:10.22323/1.251.0036 [arXiv:1512.04506 [physics.comp-ph]].

[26] D. Richtmann, S. Heybrock and T. Wettig, PoS **LATTICE2015**, 035 (2016), doi:10.22323/1.251.0035 [arXiv:1601.03184 [hep-lat]].

[27] H. Kobayashi, Y. Nakamura, S. Takeda and Y. Kuramashi, PoS LATTICE **2015**, 029 (2016), doi:10.22323/1.251.0029.

[28] T. Boku, K. I. Ishikawa, Y. Kuramashi, L. Meadows, M. D'Mello, M. Troute and R. Vemuri, PoS LATTICE **2016**, 261 (2016), doi:10.22323/1.256.0261 [arXiv:1612.06556 [hep-lat]].

[29] C. DeTar, D. Doerfler, S. Gottlieb, A. Jha, D. Kalamkar, R. Li and D. Toussaint, PoS LATTICE **2016**, 270 (2016), doi:10.22323/1.256.0270 [arXiv:1611.00728 [hep-lat]].

[30] C. DeTar, S. Gottlieb, R. Li and D. Toussaint, EPJ Web Conf. **175**, 02009 (2018), doi:10.1051/epjconf/201817502009 [arXiv:1712.00143 [hep-lat]].

[31] I. Kanamori and H. Matsufuru, [arXiv:1712.01505 [hep-lat]].

[32] I. Kanamori and H. Matsufuru, LNCS 10962, 456-471 (2018), doi:10.1007/978-3-319-95168-3_31 [arXiv:1811.00893 [hep-lat]].

[33] C. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein and T. Wettig, [arXiv:2103.03013 [cs.PF]].

[34] K. I. Ishikawa, I. Kanamori, H. Matsufuru, I. Miyoshi, Y. Mukai, Y. Nakamura, K. Nitadori and M. Tsuji, [arXiv:2109.10687 [hep-lat]].

[35] Y. Akahoshi, S. Aoki, T. Aoyama, I. Kanamori, K. Kanaya, H. Matsufuru, Y. Namekawa, H. Nemura and Y. Taniguchi, [arXiv:2111.04457 [hep-lat]].

[36] S. Durr, EPJ Web Conf. **175**, 02001 (2018), doi:10.1051/epjconf/201817502001 [arXiv:1709.01828 [hep-lat]].

[37] S. Durr, PoS LATTICE **2018**, 033 (2018), doi:10.22323/1.334.0033 [arXiv:1808.05506 [hep-lat]].

[38] C. Morningstar and M. J. Peardon, Phys. Rev. D **69**, 054501 (2004), doi:10.1103/PhysRevD.69.054501 [arXiv:hep-lat/0311018 [hep-lat]].

[39] B. Sheikholeslami and R. Wohlert, Nucl. Phys. B **259**, 572 (1985) doi:10.1016/0550-3213(85)90002-1

[40] S. Aoki and Y. Kuramashi, Phys. Rev. D **68**, 094019 (2003) doi:10.1103/PhysRevD.68.094019 [arXiv:hep-lat/0306015 [hep-lat]].

[41] M. Luscher, S. Sint, R. Sommer and P. Weisz, Nucl. Phys. B **478**, 365-400 (1996) doi:10.1016/0550-3213(96)00378-1 [arXiv:hep-lat/9605038 [hep-lat]].

[42] T. A. DeGrand [MILC], Phys. Rev. D **60**, 094501 (1999) doi:10.1103/PhysRevD.60.094501 [arXiv:hep-lat/9903006 [hep-lat]].

[43] K. Orginos *et al.* [MILC], Phys. Rev. D **60**, 054503 (1999) doi:10.1103/PhysRevD.60.054503 [arXiv:hep-lat/9903032 [hep-lat]].

[44] G. P. Lepage, Phys. Rev. D **59**, 074502 (1999) doi:10.1103/PhysRevD.59.074502

[45] L. Giusti, C. Hoelbling, M. Luscher and H. Wittig, Comput. Phys. Commun. **153**, 31-51 (2003), doi:10.1016/S0010-4655(02)00874-3 [arXiv:hep-lat/0212012 [hep-lat]].

[46] S. Durr, Z. Fodor, C. Hoelbling, R. Hoffmann, S. D. Katz, S. Krieg, T. Kurth, L. Lellouch, T. Lippert and K. K. Szabo, *et al.* Phys. Rev. D **79**, 014501 (2009), doi:10.1103/PhysRevD.79.014501 [arXiv:0802.2706 [hep-lat]].

[47] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek and S. Tomov, Computer Physics Communications, Volume 180, Issue 12, Pages 2526-2533, (2009), doi.org/10.1016/j.cpc.2008.11.005.

[48] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra and N. J. Higham, Proc. R. Soc. A.476, 20200110, doi.org/10.1098/rspa.2020.0110.

[49] W. Kamleh, Lect. Notes Phys. **663**, 65 (2005), doi:10.1007/11356462_3 [hep-lat/0209154].

[50] A. Alexandru, C. Pelissier, B. Gamari and F. Lee, J. Comput. Phys. **231**, 1866-1878 (2012), doi:10.1016/j.jcp.2011.11.003 [arXiv:1103.5103 [hep-lat]].