Preprint BUW-IMACM 15/41

Simon Heybrock, Matthias Rottmann, Peter Georg, Tilo Wettig

Adaptive algebraic multigrid on SIMD architectures

December 14, 2015

PROCEEDINGS OF SCIENCE

Adaptive algebraic multigrid on SIMD architectures*

Simon Heybrock[a,b], **Matthias Rottmann**[c], **Peter Georg**[a], **Tilo Wettig**[†a]

[a] *Department of Physics, University of Regensburg, 93040 Regensburg, Germany*

[b] *Data Management and Software Centre, European Spallation Source, Universitetsparken 5, 2100 Copenhagen, Denmark*

[c] *Department of Mathematics, University of Wuppertal, 42097 Wuppertal, Germany*

E-mail: tilo.wettig@ur.de

We present details of our implementation of the Wuppertal adaptive algebraic multigrid code DD-αAMG on SIMD architectures, with particular emphasis on the Intel Xeon Phi processor (KNC) used in QPACE 2. As a smoother, the algorithm uses a domain-decomposition-based solver code previously developed for the KNC in Regensburg. We optimized the remaining parts of the multigrid code and conclude that it is a very good target for SIMD architectures. Some of the remaining bottlenecks can be eliminated by vectorizing over multiple test vectors in the setup, which is discussed in the contribution of Daniel Richtmann.

The 33nd International Symposium on Lattice Field Theory

*Work supported by the German Research Foundation (DFG) in the framework of SFB/TRR-55.

[†] Speaker.

1. Introduction

In order to solve the Dirac equation $Du = f$ on large lattices we usually use an iterative method and a preconditioner M with $M^{-1} \approx D^{-1}$. Writing $DM^{-1}Mu = DM^{-1}v = f$ we first solve for v with the preconditioned matrix DM^{-1}, which is much better behaved than D, and then obtain $u = M^{-1}v$. In each iteration the residual $r_n = f - DM^{-1}v_n$ gives an indication of how close we are to the true solution. Formally, the residual can be written as a linear combination of the eigenmodes of D. The art is to find a preconditioner that eliminates, or at least reduces, the contributions of these eigenmodes to the residual. Adaptive algebraic multigrid (MG) [1] is such a preconditioner, and here we focus on the Wuppertal version DD-αAMG [2]. It consists of two parts, a coarse-grid correction (CGC) that reduces the contributions of the low modes and a domain-decomposition (DD) based smoother that reduces the contributions of the high modes [3].

In this contribution we implement and optimize the Wuppertal code for SIMD architectures.[1] Our main target is the Intel Xeon Phi processor (KNC) used in QPACE 2, but our code can easily be adapted to other SIMD-based architectures. For the smoother we use the code already developed for the KNC in Regensburg [4]. Here we mainly focus on the remaining parts of the code.

2. Description of the algorithm

Some preliminaries: The lattice volume V is divided into N_{block} blocks. To implement γ_5-symmetry DD-αAMG defines, for each block, two aggregates that contain the left- and right-handed spinor components of the fields, respectively.[2] The outer Dirac solver is FGMRES [5]. The MG method consists of two parts: (1) the initial setup phase and (2) the application of the MG preconditioner (Alg. 1) in every FGMRES iteration. For pedagogical reasons we first explain the latter.

2.1 MG preconditioner

In the coarse-grid correction (Alg. 2), we first restrict the current iteration vector of the outer solver from a fine to a coarse grid using a restriction operator that approximately preserves the low modes of the Dirac operator. How this operator is constructed is explained in Sec. 2.2. The Dirac equation is then solved to low precision on the coarse grid, and the solution is lifted back to the fine grid. The main point is that this solution approximates the low-mode content of the true solution.

To also approximate the high-mode content of the true solution, a DD-based smoother (Alg. 3) is applied to the current iteration vector. The inverse block size acts as a cutoff for the low modes. Thus, for vectors y_{high} that do not contain modes below this cutoff, the smoother applied to y_{high} approximates $D^{-1}y_{\text{high}}$. If we use the solution from the coarse-grid correction (which approximates

Algorithm 1: MG preconditioner (V-cycle)

```
Input: right-hand side $y$
Output: approximate solution $x$ of $Dx = y$
apply coarse-grid correction to $y$ (Alg. 2)
apply smoother to $y$, with result from coarse-grid correction as starting guess (Alg. 3)
set $x$ to result of smoother
```

[1] We restrict ourselves to an MG V-cycle with two levels (coarse grid and fine grid), for reasons explained in Sec. 3.

[2] I.e., an aggregate contains $6V_{\text{block}}$ degrees of freedom, which we can enumerate with an index $n = 1, \ldots, 6V_{\text{block}}$.

Algorithm 2: Coarse-grid correction

```
Input: right-hand side $y$
Output: approximate solution $x$ of $Dx = y$
// Restrict (i.e., project):
restrict vector $y$ from fine to coarse grid:            // $\dim(R) = 2N_{\text{tv}}N_{\text{block}} \times 12V$

    $y_c = Ry$   with $R = \operatorname{diag}(R_1^\ell, R_1^r, \dots, R_{N_{\text{block}}}^\ell, R_{N_{\text{block}}}^r)$ and $R_i^{\ell/r}$ from Alg. 5      (2.1)

// Coarse-grid solve to low precision using FGMRES with even/odd preconditioning: $x_c \approx D_c^{-1} y_c$
repeat
    apply coarse-grid operator to current iteration vector
    BLAS-like linear algebra (mainly Gram-Schmidt)
until norm of residual $\lesssim 0.05$
// Prolongate (i.e., interpolate):
extend solution vector from coarse to fine grid: $x = Px_c$ with $P = R^\dagger$
```

Algorithm 3: Smoother (DD)

```
Input: right-hand side $y$, starting guess $x^0$
Output: approximate solution $x^{(N_{\text{smoother}})}$ of $Dx = y$
split lattice into blocks
write $D = B + Z$ with $B$ = couplings within blocks and $Z$ = couplings between blocks
for $n = 1$ to $N_{\text{smoother}}$ do
    $x^{(n)} = x^{(n-1)} + B^{-1}(y - Dx^{(n-1)})$            // simplified; in practice SAP [3] is used
```

the low modes of the true solution) as a starting guess x^0, the smoother works on $y_{\text{high}} = y - Dx^0$, i.e., the low modes have been approximately eliminated from y by the subtraction of Dx^0.

2.2 MG setup

In any MG method one needs to restrict from a fine to a coarse grid. In geometric MG, the restriction proceeds by simply averaging subsets of the fields on the fine grid. Algebraic MG is more sophisticated and includes nontrivial weight factors in the average. The art is to find good weight factors so that the low modes of the operator to be inverted are approximately preserved after the restriction.[3] The purpose of the MG setup phase is the computation of these weight factors. The main idea is to apply an iterative process through which more and more of the high-mode components are eliminated. To this end, we define a set of test vectors $\{v_j : j = 1, \dots, N_{tv}\}$ (each of dimension $12V$) that will, at the end of the iterative process, approximate the low-mode components. The iterative process is described in Alg. 4, see [2] for details. Alg. 5 describes how the restriction operator is constructed from the test vectors v_j and how D is restricted to the coarse grid.

3. Implementation details

The implementation of the DD-based smoother (Alg. 3) on a SIMD architecture is quite complicated and described in detail in [4]. In contrast, the remaining parts of the DD-αAMG code are

[3] This is the same principle as inexact deflation [6].

Algorithm 4: MG setup

```
// Initial setup:
set the $N_{\text{tv}}$ test vectors to random starting vectors
for $k = 1$ to 3 do
    update each test vector by applying smoother with $N_{\text{smoother}} = k$, with starting guess 0 (Alg. 3)
setup of restriction and coarse-grid operator (Alg. 5)
normalize the test vectors
// Iterative setup:
for $i = 1$ to $N_{\text{setup}}$ do
    for $j = 1$ to $N_{\text{tv}}$ do
        apply CGC to test vector $v_j$ (Alg. 2)
        apply smoother to test vector $v_j$, with result from CGC as starting guess (Alg. 3)
        replace test vector $v_j$ by result of smoother
    setup of restriction and coarse-grid operator (Alg. 5)
```

Algorithm 5: Setup of restriction and coarse-grid operator

```
Input: test vectors $\{v_j\}$
Output: restriction operator $R$ and coarse-grid operator $D_c$
// Setup of restriction operator:
for $i = 1$ to $N_{\text{block}}$ do
    foreach $h = \ell, r$ do
        set $R_i^h$ to $N_{\text{tv}} \times 6V_{\text{block}}$ matrix having in its rows the vectors $v_j^\dagger$ restricted to aggregate $A_i^h$
        run Gram-Schmidt on the rows of $R_i^h$
// Setup of coarse-grid operator by restriction:
for $i = 1$ to $N_{\text{block}}$ do
    foreach $j \in \{i$ and nearest neighbors of $i\}$ do                // optimize using (3.1)
        compute couplings between sites $i$ and $j$ on coarse grid:      // no sums over $i, j, \ell, r$

            $$\begin{pmatrix} D_c^{\ell\ell} & D_c^{\ell r} \\ D_c^{r\ell} & D_c^{rr} \end{pmatrix}_{ij} = \begin{pmatrix} R_i^\ell & 0 \\ 0 & R_i^r \end{pmatrix} \begin{pmatrix} D_{ij}^{\ell\ell} & D_{ij}^{\ell r} \\ D_{ij}^{r\ell} & D_{ij}^{rr} \end{pmatrix} \begin{pmatrix} P_j^\ell & 0 \\ 0 & P_j^r \end{pmatrix} \qquad (2.2)$$

// $\dim(D_c^{hh'})_{ij} = N_{\text{tv}}$, $\dim(D_{ij}^{hh'}) = 6V_{\text{block}}$
// (2.2) can be written as $(D_c)_{ij}^{hh'} = R_i^h D_{ij}^{hh'} P_j^{h'}$ or as $D_c = RDP$ with $R$ defined in (2.1) and $P = R^\dagger$.
```

easier to vectorize since the number of components that can be treated on the same footing contains a factor of N_{tv} (on the fine grid) or $2N_{\text{tv}}$ (on the coarse grid). If we choose this factor to be equal to an integer multiple of the SIMD length N_{SIMD} (i.e., the number of SIMD components) we achieve perfect use of the SIMD unit.[4] On the downside, DD-αAMG consists of many parts, most of which take a non-negligible part of the total execution time. Therefore we need to vectorize/optimize all

[4] For the KNC, $N_{\text{SIMD}} = 16$ in single precision. We use $N_{\text{tv}} = 16$ to 32. This choice is appropriate from an algorithmic point of view. Should the algorithmic performance require N_{tv} to be unequal to an integer multiple of N_{SIMD} we would use ceil($N_{\text{tv}}/N_{\text{SIMD}}$) SIMD vectors on the fine grid and pad the last of these (and the corresponding memory region) with zeros. Analogously for the coarse grid. E.g., for $N_{\text{tv}} = 24$ we need padding only on the fine grid.

Algorithm 6: SIMD implementation of Ry in (2.1)

```
for $i = 1$ to $N_{\text{block}}$ do
    foreach $h = \ell, r$ do
        set $(y_c)_i^h = 0$ in SIMD vectors (real and imaginary part)        // $\dim(y_c)_i^h = N_{\text{tv}}$
        for $n = 1$ to $6V_{\text{block}}$ do                                 // work on aggregate $A_i^h$
            load real and imaginary part of column $n$ of $R_i^h$ into SIMD vectors
            broadcast real and imaginary part of corresponding element of $y$ into SIMD vectors
            increase $(y_c)_i^h$ by complex fused multiply-add (corresponding to 4 real SIMD fmadds)
        write $(y_c)_i^h$ to memory
```

of them. Before going into detail we briefly summarize general aspects of our work.

We streamlined the original Wuppertal code and removed some redundant or unnecessary parts (e.g., some orthonormalizations). We reduced the memory consumption slightly by eliminating redundancies and temporary copies. We threaded the code by decomposing the lattice into pieces that are assigned to individual threads. Based on earlier microbenchmarks, we decided to use persistent threads with synchronization points, rather than a fork-join model. Our SIMD implementation is based on intrinsics for the Intel compiler. To facilitate efficient SIMD multiplication of complex numbers, the data layout of R and D_c is such that real and imaginary parts are not mixed in the same register. In this paper we only describe 2-level MG, where all KNCs work both on the fine and the coarse grid. Many parts carry over to 3-level (or higher), with two exceptions: (1) since the KNC has rather limited memory, more levels lead to more idle cores or even idle processors, and (2) we currently do not have an efficient implementation of the DD smoother on a coarse grid.[5]

We now describe how the key components that needed to be optimized were implemented.

Restriction (Alg. 2): We vectorize the matrix-vector multiplication (2.1) for each aggregate as described in Alg. 6. The vectorization is done such that the row index of $R_i^{\ell/r}$ runs in the SIMD vector, i.e., the latter contains a column of $R_i^{\ell/r}$ if $N_{\text{tv}} = N_{\text{SIMD}}$.[6]

Prolongation (Alg. 2): Similar to restriction but with $R \to P = R^\dagger$. Since the aspect ratio of the rectangular matrix is reversed, now the column index of $P_i^{\ell/r}$ (= row index of R) runs in the SIMD vector. At the end an additional sum over the elements in the SIMD vector is required, which makes prolongation somewhat less efficient than restriction.

Setup of coarse-grid operator (Alg. 5): In (2.2) we first compute $D_{ij}^{hh'} P_j^{h'}$, which corresponds to the application of the sparse matrix $D_{ij}^{hh'}$ to multiple vectors, i.e., the columns of $P_j^{h'}$. The vectorization of this operation is described in Alg. 7. The application of R to the result corresponds to a restriction with multiple right-hand sides, which can be optimized as described in the contribution of Daniel Richtmann [7]. As in the original Wuppertal code, we only compute (2.2) for $i = j$ and the forward neighbors. For the backward neighbors we use

$$(D_c)_{ji}^{hh} = (D_c)_{ij}^{hh\dagger} \quad \text{and} \quad (D_c)_{ji}^{hh'} = -(D_c)_{ij}^{h'h\dagger} \; (h \neq h') . \tag{3.1}$$

[5] For two levels this operator is not needed. For higher levels we cannot reuse our fine-grid DD smoother since it is for a different operator.

[6] For $N_{\text{tv}} \neq N_{\text{SIMD}}$ footnote 4 applies. The need for padding on the fine grid could be reduced by combining R_i^ℓ and R_i^r at the expense of a more complicated broadcast in Alg. 6. This is a tradeoff between memory bandwidth and instruction count that we have not explored yet.

Algorithm 7: SIMD implementation of $D^{hh'}_{ij} P^{h'}_j$ in (2.2).

```
for $x \in$ block $i$ do
    set output $= 0$ in SIMD vectors (real and imaginary parts)
    foreach $\mu \in \{\pm 1, \pm 2, \pm 3, \pm 4\}$ do
        if $x+\hat\mu \in$ block $j$ then
            load real and imag. parts of the 6 rows of $P^{h'}_j$ corresponding to $x+\hat\mu$ into SIMD vectors
            broadcast real and imag. parts of the 9 elements of SU(3) link $U_\mu(x)$ into SIMD vectors
            increase output by complex fmadd $(1+\gamma_\mu)^{hh'} U_\mu(x)^\dagger P^{h'}_j(x+\hat\mu)$
    if $i = j$ and $h = h'$ then
        load real and imaginary parts of the 6 rows of $P^h_i$ corresponding to $x$ into SIMD vectors
        broadcast real and imaginary parts of the clover matrix elements $C^{hh}(x)$ into SIMD vectors
        increase output by complex fmadd $C^{hh}(x) P^h_i(x)$
```

In contrast to the original code we also store $(D_c)_{ji}$ since a transpose is expensive in SIMD (i.e., we perform the transpose only when the operator is constructed, but not when it is applied). The coarse-grid operator can be stored in half precision to reduce working set and memory bandwidth requirements. We did not observe any negative effects of this on stability or iteration count and therefore made it the default.

Application of coarse-grid operator (Alg. 2): The spatial structure of D_c is similar to Wilson clover. In (2.2), $(D_c)_{ij} \neq 0$ only if i and j are equal or nearest neighbors. In that case $(D_c)_{ij}$ is dense and stored in memory. Therefore the vectorization can be done as in the restriction.

Gram-Schmidt on aggregrates (Alg. 5): We do not use modified Gram-Schmidt [5] since (1) classical Gram-Schmidt lends itself more easily to vectorization and needs fewer globals sums and (2) stability of the Gram-Schmidt process is not expected to become an issue in the preconditioner. To obtain better cache reuse and thus save memory bandwidth we use the block Gram-Schmidt method [8]. As usual, the vectorization is done by merging the same components of the N_{tv} test vectors in the SIMD vectors. The disadvantage of this strategy is that axpy operations and dot products then waste parts (on average one half) of the SIMD vectors.

BLAS-like linear algebra on coarse grid (Alg. 2): In order to utilize the SIMD unit we would need to change the data layout on the coarse grid. This change would propagate to other parts of the code. We have not yet made this change since it requires nontrivial programming efforts, while the impact on performance is not dominant. As a temporary workaround we sometimes de-interleave real and imaginary parts on the fly to do a SIMD computation.

4. Performance and conclusions

In Tab. 1 we show the speedup obtained by vectorization of the MG components for a single thread on a single KNC compared to the original Wuppertal code, for a lattice size of 8^4 that does not fit in cache. In Fig. 1 we show how these improvements affect the contributions of the various MG components to the total wall-clock time, for both setup and solve. We observe that the MG parts that have been optimized (see Tab. 1) no longer contribute significantly to the execution time. The smoother, which was optimized earlier, takes about 1/3 of the time. The main contribution

MG component	Restrict.	Prolong.	D_c setup	$(D_c)_{i \neq j}$	$(D_c)_{ii}$	GS on aggr.
SIMD speedup	14.1	8.6	19.7	20.2	19.5	10.8

Table 1: SIMD speedup for a single thread on a single KNC.

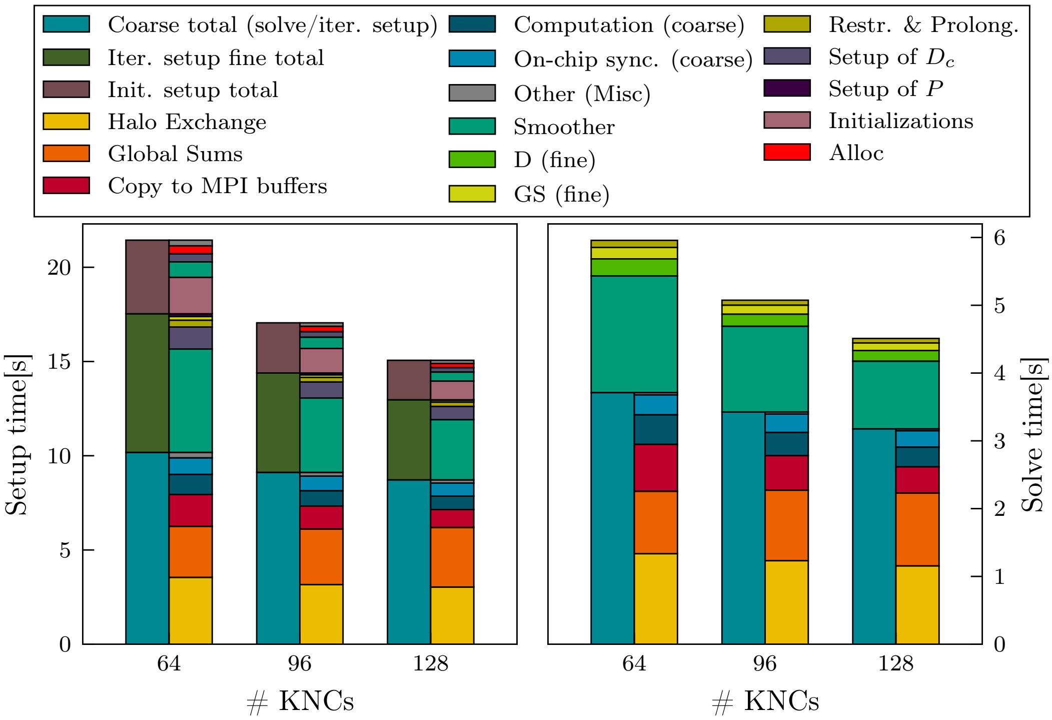

Figure 1: Strong-scaling plot of the contributions of various MG components to the execution time, for a $48^3 \times 96$ CLS lattice with $\beta = 3.4$, $m_\pi = 220$ MeV, and $a = 0.086$ fm. The algorithmic parameters are tuned to minimize the total wall-clock time for the 128 KNC run. The simulations were performed on QPACE 2.

now comes from the coarse-grid solve. After our optimizations, its computational part has become cheap so that off-chip communication (halo exchange and global sums) becomes dominant.

We conclude that DD-αAMG is a very good target for SIMD architectures. The run time is now dominated by the communications in the coarse-grid solve, which will be optimized next.

The code presented here will be made publicly available in the near future. We thank Andreas Frommer and Karsten Kahl for helpful discussions and Daniel Richtmann for producing the figures.

References

[1] R. Babich et al., *Phys. Rev. Lett.* **105** (2010) 201602 [arXiv:1005.3043].

[2] A. Frommer et al., *SIAM J.Sci.Comput.* **36** (2014) A1581 [arXiv:1303.1377].

[3] M. Lüscher, *Comput.Phys.Commun.* **156** (2004) 209 [hep-lat/0310048].

[4] S. Heybrock et al., *Proceedings of SC '14* (2014) 69 [arXiv:1412.2629].

[5] Y. Saad, *Iterative Methods for Sparse Linear Systems: Second Edition*. SIAM, 2003.

[6] M. Lüscher, *JHEP* **0707** (2007) 081 [arXiv:0706.2298].

[7] D. Richtmann, S. Heybrock, and T. Wettig, *PoS* **LATTICE2015** (2016) 035.

[8] W. Jalby and B. Philippe, *SIAM J.Sci.Stat.Comp.* **12** (1991) 1058.